Николай Прохоренок
Владимир Дронов

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Python 3 и PyQt 5 Разработка приложений

2-е издание

- Описание языка Python
- Объектно-ориентированное программирование
- Работа с файлами и каталогами
- Создание оконных приложений
- Работа с базами данных
- Мультимедиа
- Печать и экспорт в формат PDF
- Взаимодействие с Windows
- Сохранение настроек приложений
- Работающий пример: приложение «Судоку»

Материалы на www.bhv.ru

bhv®

Николай Прохоренок

Владимир Дронов

Python 3 и PyQt 5 Разработка приложений

2-е издание

Санкт-Петербург

«БХВ-Петербург»

2018

УДК 004.43
ББК 32.973.26-018.1
П84

Прохоренок, Н. А.

П84 Python 3 и PyQt 5. Разработка приложений. — 2-е изд., перераб. и доп. / Н. А. Прохоренок, В. А. Дронов. — СПб.: БХВ-Петербург, 2018. — 832 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-3978-4

Описан язык Python 3: типы данных, операторы, условия, циклы, регулярные выражения, функции, инструменты объектно-ориентированного программирования, работа с файлами и каталогами, модули стандартной библиотеки. Особое внимание уделено библиотеке PyQt, позволяющей создавать приложения с графическим интерфейсом. Рассмотрены средства для обработки сигналов и событий, управления свойствами окна, разработки многопоточных приложений, описаны основные компоненты (кнопки, поля и др.), инструменты для работы с базами данных, мультимедиа, печати документов и их экспорта. На сайте издательства приведены примеры из книги.

Во втором издании описаны актуальные версии Python 3.6.3 и PyQt 5.9.2, средства взаимодействия с Windows и сохранения настроек приложений, рассмотрен процесс разработки полнофункционального приложения.

Для программистов

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Марины Дамбиевой*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-3978-4

© ООО "БХВ", 2018
© Оформление. ООО "БХВ-Петербург", 2018

Оглавление

Введение ... 15

ЧАСТЬ I. ОСНОВЫ ЯЗЫКА PYTHON ... 17

Глава 1. Первые шаги ... 19
1.1. Установка Python ... 19
1.1.1. Установка нескольких интерпретаторов Python ... 23
1.1.2. Запуск программы с помощью разных версий Python ... 25
1.2. Первая программа на Python ... 26
1.3. Структура программы ... 28
1.4. Комментарии ... 31
1.5. Дополнительные возможности IDLE ... 31
1.6. Вывод результатов работы программы ... 33
1.7. Ввод данных ... 35
1.8. Доступ к документации ... 36

Глава 2. Переменные ... 40
2.1. Именование переменных ... 40
2.2. Типы данных ... 41
2.3. Присваивание значения переменным ... 44
2.4. Проверка типа данных ... 47
2.5. Преобразование типов данных ... 48
2.6. Удаление переменных ... 50

Глава 3. Операторы ... 52
3.1. Математические операторы ... 52
3.2. Двоичные операторы ... 54
3.3. Операторы для работы с последовательностями ... 55
3.4. Операторы присваивания ... 56
3.5. Приоритет выполнения операторов ... 57

Глава 4. Условные операторы и циклы ... 59
4.1. Операторы сравнения ... 60
4.2. Оператор ветвления *if...else* ... 62

4.3. Цикл *for* 65
4.4. Функции *range()* и *enumerate()* 67
4.5. Цикл *while* 70
4.6. Оператор *continue*: переход на следующую итерацию цикла 71
4.7. Оператор *break*: прерывание цикла 71

Глава 5. Числа 73
5.1. Встроенные функции и методы для работы с числами 75
5.2. Модуль *math*: математические функции 77
5.3. Модуль *random*: генерирование случайных чисел 79

Глава 6. Строки и двоичные данные 82
6.1. Создание строки 83
6.2. Специальные символы 86
6.3. Операции над строками 87
6.4. Форматирование строк 90
6.5. Метод *format()* 95
    6.5.1. Форматируемые строки 99
6.6. Функции и методы для работы со строками 100
6.7. Настройка локали 103
6.8. Изменение регистра символов 104
6.9. Функции для работы с символами 105
6.10. Поиск и замена в строке 105
6.11. Проверка типа содержимого строки 109
6.12. Тип данных *bytes* 111
6.13. Тип данных *bytearray* 116
6.14. Преобразование объекта в последовательность байтов 119
6.15. Шифрование строк 120

Глава 7. Регулярные выражения 122
7.1. Синтаксис регулярных выражений 122
7.2. Поиск первого совпадения с шаблоном 131
7.3. Поиск всех совпадений с шаблоном 136
7.4. Замена в строке 137
7.5. Прочие функции и методы 139

Глава 8. Списки, кортежи, множества и диапазоны 141
8.1. Создание списка 142
8.2. Операции над списками 145
8.3. Многомерные списки 148
8.4. Перебор элементов списка 148
8.5. Генераторы списков и выражения-генераторы 149
8.6. Функции *map()*, *zip()*, *filter()* и *reduce()* 150
8.7. Добавление и удаление элементов списка 153
8.8. Поиск элемента в списке и получение сведений о значениях, входящих в список 156
8.9. Переворачивание и перемешивание списка 157
8.10. Выбор элементов случайным образом 157
8.11. Сортировка списка 158
8.12. Заполнение списка числами 159
8.13. Преобразование списка в строку 160

8.14. Кортежи ... 160
8.15. Множества ... 162
8.16. Диапазоны ... 167
8.17. Модуль *itertools* ... 168
8.17.1. Генерирование неопределенного количества значений ... 168
8.17.2. Генерирование комбинаций значений ... 169
8.17.3. Фильтрация элементов последовательности ... 171
8.17.4. Прочие функции ... 172

Глава 9. Словари ... 175
9.1. Создание словаря ... 175
9.2. Операции над словарями ... 177
9.3. Перебор элементов словаря ... 179
9.4. Методы для работы со словарями ... 180
9.5. Генераторы словарей ... 183

Глава 10. Работа с датой и временем ... 184
10.1. Получение текущих даты и времени ... 184
10.2. Форматирование даты и времени ... 186
10.3. «Засыпание» скрипта ... 188
10.4. Модуль *datetime*: манипуляции датой и временем ... 188
10.4.1. Класс *timedelta* ... 189
10.4.2. Класс *date* ... 191
10.4.3. Класс *time* ... 194
10.4.4. Класс *datetime* ... 196
10.5. Модуль *calendar*: вывод календаря ... 201
10.5.1. Методы классов *TextCalendar* и *LocaleTextCalendar* ... 202
10.5.2. Методы классов *HTMLCalendar* и *LocaleHTMLCalendar* ... 204
10.5.3. Другие полезные функции ... 205
10.6. Измерение времени выполнения фрагментов кода ... 208

Глава 11. Пользовательские функции ... 210
11.1. Определение функции и ее вызов ... 210
11.2. Расположение определений функций ... 213
11.3. Необязательные параметры и сопоставление по ключам ... 214
11.4. Переменное число параметров в функции ... 216
11.5. Анонимные функции ... 218
11.6. Функции-генераторы ... 219
11.7. Декораторы функций ... 221
11.8. Рекурсия. Вычисление факториала ... 223
11.9. Глобальные и локальные переменные ... 224
11.10. Вложенные функции ... 228
11.11. Аннотации функций ... 229

Глава 12. Модули и пакеты ... 231
12.1. Инструкция *import* ... 231
12.2. Инструкция *from* ... 234
12.3. Пути поиска модулей ... 237
12.4. Повторная загрузка модулей ... 238
12.5. Пакеты ... 239

Глава 13. Объектно-ориентированное программирование 244
13.1. Определение класса и создание экземпляра класса 244
13.2. Методы *__init__()* и *__del__()* 247
13.3. Наследование 248
13.4. Множественное наследование 250
13.4.1. Примеси и их использование 252
13.5. Специальные методы 253
13.6. Перегрузка операторов 255
13.7. Статические методы и методы класса 258
13.8. Абстрактные методы 259
13.9. Ограничение доступа к идентификаторам внутри класса 260
13.10. Свойства класса 261
13.11. Декораторы классов 263

Глава 14. Обработка исключений 264
14.1. Инструкция *try...except...else...finally* 265
14.2. Инструкция *with...as* 269
14.3. Классы встроенных исключений 271
14.4. Пользовательские исключения 273

Глава 15. Итераторы, контейнеры и перечисления 277
15.1. Итераторы 278
15.2. Контейнеры 279
15.2.1. Контейнеры-последовательности 279
15.2.2. Контейнеры-словари 281
15.3. Перечисления 282

Глава 16. Работа с файлами и каталогами 287
16.1. Открытие файла 287
16.2. Методы для работы с файлами 293
16.3. Доступ к файлам с помощью модуля *os* 299
16.4. Классы *StringIO* и *BytesIO* 301
16.5. Права доступа к файлам и каталогам 305
16.6. Функции для манипулирования файлами 307
16.7. Преобразование пути к файлу или каталогу 310
16.8. Перенаправление ввода/вывода 312
16.9. Сохранение объектов в файл 315
16.10. Функции для работы с каталогами 319
16.10.1. Функция *scandir()* 322
16.11. Исключения, возбуждаемые файловыми операциями 324

ЧАСТЬ II. БИБЛИОТЕКА PYQT 5 325

Глава 17. Знакомство с PyQt 5 327
17.1. Установка PyQt 5 327
17.2. Первая программа 328
17.3. Структура PyQt-программы 329
17.4. ООП-стиль создания окна 331
17.5. Создание окна с помощью программы Qt Designer 335
17.5.1. Создание формы 335

17.5.2. Использование UI-файла в программе 337
17.5.3. Преобразование UI-файла в PY-файл 339
17.6. Модули PyQt 5 340
17.7. Типы данных в PyQt 341
17.8. Управление основным циклом приложения 342
17.9. Многопоточные приложения 344
17.9.1. Класс *QThread*: создание потока 344
17.9.2. Управление циклом внутри потока 347
17.9.3. Модуль *queue*: создание очереди заданий 351
17.9.4. Классы *QMutex* и *QMutexLocker* 354
17.10. Вывод заставки 359
17.11. Доступ к документации 361

Глава 18. Управление окном приложения 362
18.1. Создание и отображение окна 362
18.2. Указание типа окна 363
18.3. Изменение и получение размеров окна 365
18.4. Местоположение окна на экране и управление им 367
18.5. Указание координат и размеров 370
18.5.1. Класс *QPoint*: координаты точки 370
18.5.2. Класс *QSize*: размеры прямоугольной области 372
18.5.3. Класс *QRect*: координаты и размеры прямоугольной области 374
18.6. Разворачивание и сворачивание окна 379
18.7. Управление прозрачностью окна 381
18.8. Модальные окна 381
18.9. Смена значка в заголовке окна 383
18.10. Изменение цвета фона окна 384
18.11. Вывод изображения в качестве фона 385
18.12. Создание окна произвольной формы 387
18.13. Всплывающие подсказки 388
18.14. Программное закрытие окна 390
18.15. Использование таблиц стилей CSS для оформления окон 390

Глава 19. Обработка сигналов и событий 395
19.1. Назначение обработчиков сигналов 395
19.2. Блокировка и удаление обработчика 399
19.3. Генерация сигналов 401
19.4. Передача данных в обработчик 403
19.5. Использование таймеров 404
19.6. Перехват всех событий 407
19.7. События окна 410
19.7.1. Изменение состояния окна 410
19.7.2. Изменение положения и размеров окна 411
19.7.3. Перерисовка окна или его части 412
19.7.4. Предотвращение закрытия окна 413
19.8. События клавиатуры 414
19.8.1. Установка фокуса ввода 414
19.8.2. Назначение клавиш быстрого доступа 417
19.8.3. Нажатие и отпускание клавиши на клавиатуре 419

19.9. События мыши 420
19.9.1. Нажатие и отпускание кнопки мыши 420
19.9.2. Перемещение указателя мыши 422
19.9.3. Наведение и увод указателя 423
19.9.4. Прокрутка колесика мыши 423
19.9.5. Изменение внешнего вида указателя мыши 424
19.10. Технология drag & drop 425
19.10.1. Запуск перетаскивания 425
19.10.2. Класс *QMimeData* 427
19.10.3. Обработка сброса 429
19.11. Работа с буфером обмена 430
19.12. Фильтрация событий 431
19.13. Искусственные события 432

Глава 20. Размещение компонентов в окнах 434
20.1. Абсолютное позиционирование 434
20.2. Горизонтальное и вертикальное выравнивание 435
20.3. Выравнивание по сетке 438
20.4. Выравнивание компонентов формы 441
20.5. Классы *QStackedLayout* и *QStackedWidget* 443
20.6. Класс *QSizePolicy* 444
20.7. Объединение компонентов в группу 445
20.8. Панель с рамкой 447
20.9. Панель с вкладками 448
20.10. Компонент «аккордеон» 452
20.11. Панели с изменяемым размером 454
20.12. Область с полосами прокрутки 456

Глава 21. Основные компоненты 458
21.1. Надпись 458
21.2. Командная кнопка 461
21.3. Переключатель 463
21.4. Флажок 463
21.5. Однострочное текстовое поле 464
21.5.1. Основные методы и сигналы 464
21.5.2. Ввод данных по маске 467
21.5.3. Контроль ввода 468
21.6. Многострочное текстовое поле 469
21.6.1. Основные методы и сигналы 469
21.6.2. Изменение параметров поля 471
21.6.3. Указание параметров текста и фона 473
21.6.4. Класс *QTextDocument* 474
21.6.5. Класс *QTextCursor* 477
21.7. Текстовый браузер 480
21.8. Поля для ввода целых и вещественных чисел 481
21.9. Поля для ввода даты и времени 483
21.10. Календарь 486
21.11. Электронный индикатор 488
21.12. Индикатор хода процесса 489

21.13. Шкала с ползунком 490
21.14. Круговая шкала с ползунком 492
21.15. Полоса прокрутки 492
21.16. Веб-браузер 493

Глава 22. Списки и таблицы 498
22.1. Раскрывающийся список 498
22.1.1. Добавление, изменение и удаление элементов 498
22.1.2. Изменение параметров списка 499
22.1.3. Поиск элементов 500
22.1.4. Сигналы 501
22.2. Список для выбора шрифта 501
22.3. Роли элементов 502
22.4. Модели 503
22.4.1. Доступ к данным внутри модели 503
22.4.2. Класс *QStringListModel* 504
22.4.3. Класс *QStandardItemModel* 506
22.4.4. Класс *QStandardItem* 509
22.5. Представления 513
22.5.1. Класс *QAbstractItemView* 513
22.5.2. Простой список 516
22.5.3. Таблица 518
22.5.4. Иерархический список 520
22.5.5. Управление заголовками строк и столбцов 522
22.6. Управление выделением элементов 525
22.7. Промежуточные модели 526
22.8. Использование делегатов 528

Глава 23. Работа с базами данных 532
23.1. Соединение с базой данных 532
23.2. Получение сведений о структуре таблицы 535
23.2.1. Получение сведений о таблице 535
23.2.2. Получение сведений об отдельном поле 536
23.2.3. Получение сведений об индексе 537
23.2.4. Получение сведений об ошибке 537
23.3. Выполнение SQL-запросов и получение их результатов 538
23.3.1. Выполнение запросов 538
23.3.2. Обработка результатов выполнения запросов 541
23.3.3. Очистка запроса 543
23.3.4. Получение служебных сведений о запросе 543
23.4. Модели, связанные с данными 544
23.4.1. Модель, связанная с SQL-запросом 544
23.4.2. Модель, связанная с таблицей 545
23.4.3. Модель, поддерживающая межтабличные связи 551
23.4.4. Использование связанных делегатов 553

Глава 24. Работа с графикой 555
24.1. Вспомогательные классы 555
24.1.1. Класс *QColor*: цвет 556
24.1.2. Класс *QPen*: перо 559

24.1.3. Класс *QBrush*: кисть 561
24.1.4. Класс *QLine*: линия 562
24.1.5. Класс *QPolygon*: многоугольник 562
24.1.6. Класс *QFont*: шрифт 564
24.2. Класс *QPainter* 566
24.2.1. Рисование линий и фигур 567
24.2.2. Вывод текста 570
24.2.3. Вывод изображения 571
24.2.4. Преобразование систем координат 572
24.2.5. Сохранение команд рисования в файл 573
24.3. Работа с изображениями 574
24.3.1. Класс *QPixmap* 575
24.3.2. Класс *QBitmap* 577
24.3.3. Класс *QImage* 578
24.3.4. Класс *QIcon* 581

Глава 25. Графическая сцена 583
25.1. Класс *QGraphicsScene*: сцена 583
25.1.1. Настройка сцены 584
25.1.2. Добавление и удаление графических объектов 584
25.1.3. Добавление компонентов на сцену 585
25.1.4. Поиск объектов 586
25.1.5. Управление фокусом ввода 587
25.1.6. Управление выделением объектов 588
25.1.7. Прочие методы и сигналы 588
25.2. Класс *QGraphicsView*: представление 590
25.2.1. Настройка представления 590
25.2.2. Преобразования между координатами представления и сцены 592
25.2.3. Поиск объектов 592
25.2.4. Преобразование системы координат 593
25.2.5. Прочие методы 593
25.3. Класс *QGraphicsItem*: базовый класс для графических объектов 594
25.3.1. Настройка объекта 595
25.3.2. Выполнение преобразований 597
25.3.3. Прочие методы 597
25.4. Графические объекты 598
25.4.1. Линия 598
25.4.2. Класс *QAbstractGraphicsShapeItem* 599
25.4.3. Прямоугольник 599
25.4.4. Многоугольник 599
25.4.5. Эллипс 600
25.4.6. Изображение 600
25.4.7. Простой текст 601
25.4.8. Форматированный текст 602
25.5. Группировка объектов 603
25.6. Эффекты 603
25.6.1. Класс *QGraphicsEffect* 604
25.6.2. Тень 604

25.6.3. Размытие 605
25.6.4. Изменение цвета 605
25.6.5. Изменение прозрачности 606
25.7. Обработка событий..606
25.7.1. События клавиатуры 607
25.7.2. События мыши 607
25.7.3. Обработка перетаскивания и сброса..610
25.7.4. Фильтрация событий..611
25.7.5. Обработка изменения состояния объекта..612

Глава 26. Диалоговые окна 614
26.1. Пользовательские диалоговые окна..614
26.2. Класс *QDialogButtonBox* 616
26.3. Класс *QMessageBox* 619
26.3.1. Основные методы и сигналы..620
26.3.2. Окно информационного сообщения 622
26.3.3. Окно подтверждения..623
26.3.4. Окно предупреждающего сообщения..624
26.3.5. Окно критического сообщения 624
26.3.6. Окно сведений о программе 625
26.3.7. Окно сведений о библиотеке Qt 625
26.4. Класс *QInputDialog*..626
26.4.1. Основные методы и сигналы..626
26.4.2. Окно для ввода строки 628
26.4.3. Окно для ввода целого числа..629
26.4.4. Окно для ввода вещественного числа..630
26.4.5. Окно для выбора пункта из списка 630
26.4.6. Окно для ввода большого текста..631
26.5. Класс *QFileDialog*..632
26.5.1. Основные методы и сигналы..632
26.5.2. Окно для выбора каталога 634
26.5.3. Окно для открытия файлов 635
26.5.4. Окно для сохранения файла..637
26.6. Окно для выбора цвета..638
26.7. Окно для выбора шрифта..639
26.8. Окно для вывода сообщения об ошибке..640
26.9. Окно с индикатором хода процесса 641
26.10. Создание многостраничного мастера 642
26.10.1. Класс *QWizard*..642
26.10.2. Класс *QWizardPage* 646

Глава 27. Создание SDI- и MDI-приложений..648
27.1. Главное окно приложения..648
27.2. Меню..653
27.2.1. Класс *QMenuBar* 653
27.2.2. Класс *QMenu* 654
27.2.3. Контекстное меню компонента..657
27.2.4. Класс *QAction*..658
27.2.5. Объединение переключателей в группу 661

27.3. Панели инструментов 662
27.3.1. Класс *QToolBar* 662
27.3.2. Класс *QToolButton* 664
27.4. Прикрепляемые панели 665
27.5. Управление строкой состояния 667
27.6. MDI-приложения 668
27.6.1. Класс *QMdiArea* 668
27.6.2. Класс *QMdiSubWindow* 671
27.7. Добавление значка приложения в область уведомлений 672

Глава 28. Мультимедиа 674
28.1. Класс *QMediaPlayer* 674
28.2. Класс *QVideoWidget* 683
28.3. Класс *QMediaPlaylist* 686
28.4. Запись звука 689
28.4.1. Класс *QAudioRecorder* 689
28.4.2. Класс *QAudioEncoderSettings* 692
28.5. Класс *QSoundEffect* 695

Глава 29. Печать документов 699
29.1. Основные средства печати 699
29.1.1. Класс *QPrinter* 699
29.1.2. Вывод на печать 703
29.1.3. Служебные классы 708
29.1.3.1. Класс *QPageSize* 708
29.1.3.2. Класс *QPageLayout* 710
29.2. Задание параметров принтера и страницы 712
29.2.1. Класс *QPrintDialog* 712
29.2.2. Класс *QPageSetupDialog* 714
29.3. Предварительный просмотр документов перед печатью 716
29.3.1. Класс *QPrintPreviewDialog* 716
29.3.2. Класс *QPrintPreviewWidget* 719
29.4. Класс *QPrinterInfo*: получение сведений о принтере 721
29.5. Класс *QPdfWriter*: экспорт в формат PDF 723

Глава 30. Взаимодействие с Windows 725
30.1. Управление кнопкой в панели задач 725
30.1.1. Класс *QWinTaskbarButton* 725
30.1.2. Класс *QWinTaskbarProgress* 727
30.2. Списки быстрого доступа 730
30.2.1. Класс *QWinJumpList* 730
30.2.2. Класс *QWinJumpListCategory* 731
30.2.3. Класс *QWinJumpListItem* 732
30.3. Панели инструментов, выводящиеся на миниатюрах 735
30.3.1. Класс *QWinThumbnailToolBar* 735
30.3.2. Класс *QWinThumbnailToolButton* 736
30.4. Дополнительные инструменты по управлению окнами 739
30.5. Получение сведений об операционной системе 741
30.6. Получение путей к системным каталогам 742

Глава 31. Сохранение настроек приложений 743
31.1. Создание экземпляра класса *QSettings* 743
31.2. Запись и чтение данных 745
31.2.1. Базовые средства записи и чтения данных 745
31.2.2. Группировка сохраняемых значений. Ключи 746
31.2.3. Запись списков 748
31.3. Вспомогательные методы класса *QSettings* 750
31.4. Где хранятся настройки? 750

Глава 32. Приложение «Судоку» 752
32.1. Правила судоку 752
32.2. Описание приложения «Судоку» 753
32.3. Программирование приложения 755
32.3.1. Подготовительные действия 755
32.3.2. Класс *MyLabel*: ячейка поля судоку 755
32.3.3. Класс *Widget*: поле судоку 759
32.3.3.1. Конструктор класса *Widget* 760
32.3.3.2. Прочие методы класса *Widget* 762
32.3.4. Класс *MainWindow*: основное окно приложения 766
32.3.4.1. Конструктор класса *MainWindow* 767
32.3.4.2. Остальные методы класса *MainWindow* 770
32.3.5. Запускающий модуль 770
32.3.6. Копирование и вставка головоломок 771
32.3.6.1. Форматы данных 771
32.3.6.2. Реализация копирования и вставки в классе *Widget* 772
32.3.6.3. Реализация копирования и вставки в классе *MainWindow* 774
32.3.7. Сохранение и загрузка данных 778
32.3.8. Печать и предварительный просмотр 781
32.3.8.1. Реализация печати в классе *Widget* 781
32.3.8.2. Класс *PreviewDialog*: диалоговое окно предварительного просмотра 782
32.3.8.3. Реализация печати в классе *MainWindow* 785

Заключение 787

Приложение. Описание электронного архива 789

Предметный указатель 791

Введение

Добро пожаловать в мир Python!

В *первой части* книги мы рассмотрим собственно *Python* — высокоуровневый, объектно-ориентированный, тьюринг-полный, интерпретируемый язык программирования, предназначенный для решения самого широкого круга задач. С его помощью можно обрабатывать числовую и текстовую информацию, создавать изображения, работать с базами данных, разрабатывать веб-сайты и приложения с графическим интерфейсом. Python — язык кросс-платформенный, он позволяет создавать программы, которые будут работать во всех операционных системах. В этой книге мы изучим базовые возможности Windows-редакции Python версии 3.6.3.

Согласно официальной версии, название языка произошло вовсе не от змеи. Создатель языка, Гвидо ван Россум (Guido van Rossum), назвал свое творение в честь британского комедийного телешоу Би-Би-Си «Летающий цирк Монти Пайтона» (Monty Python's Flying Circus). Поэтому правильно название этого замечательного языка должно звучать как «Пайтон».

Программа на языке Python представляет собой обычный текстовый файл с расширением py (консольная программа) или pyw (программа с графическим интерфейсом). Все инструкции из этого файла выполняются интерпретатором построчно. Для ускорения работы при первом импорте модуля создается промежуточный байт-код, который сохраняется в одноименном файле с расширением pyc. При последующих запусках, если модуль не был изменен, исполняется именно байт-код. Для выполнения низкоуровневых операций и задач, требующих высокой скорости работы, можно написать модуль на языке C или C++, скомпилировать его, а затем подключить к основной программе.

Поскольку Python, как было только что отмечено, является языком объектно-ориентированным, практически все данные в нем представляются объектами — даже значения, относящиеся к элементарным типам данных, наподобие чисел и строк, а также сами типы данных. При этом в переменной всегда сохраняется только ссылка на объект, а не сам объект. Например, можно создать функцию, сохранить ссылку на нее в переменной, а затем вызвать функцию через эту переменную. Такое обстоятельство делает язык Python идеальным инструментом для создания программ, использующих функции обратного вызова, — например, при разработке графического интерфейса. Тот факт, что Python относится к категории языков объектно-ориентированных, отнюдь не означает, что и объектно-ориентированный стиль программирования (ООП) является при его использовании обязательным: на языке Python можно писать программы как в стиле ООП, так и в процедурном стиле, — как того требует конкретная ситуация или как предпочитает программист.

Python — самый стильный язык программирования в мире, он не допускает двоякого написания кода. Так, языку Perl, например, присущи зависимость от контекста и множественность синтаксиса, и часто два программиста, пишущие на Perl, просто не понимают код друг друга. В Python же отсутствуют лишние конструкции, и код можно написать только одним способом. Все программисты, работающие с языком Python, должны придерживаться стандарта PEP-8, описанного в документе **https://www.python.org/dev/peps/pep-0008/**. Соответственно, более читаемого кода нет ни в одном ином языке программирования.

Синтаксис языка Python вызывает много нареканий у программистов, знакомых с другими языками программирования. На первый взгляд может показаться, что отсутствие ограничительных символов для выделения блоков (фигурных скобок или конструкции `begin...end`) и использование для их формирования пробелов могут приводить к ошибкам. Однако это только первое и неправильное впечатление. Хороший стиль программирования в любом языке обязывает выделять инструкции внутри блока одинаковым количеством пробелов. В этой ситуации ограничительные символы просто ни к чему. Бытует мнение, что программа будет по-разному смотреться в разных редакторах. И это неверно — согласно стандарту, для выделения блоков необходимо использовать *четыре пробела*, а четыре пробела в любом редакторе смотрятся одинаково. При этом, если количество пробелов внутри блока окажется разным, интерпретатор выведет сообщение о фатальной ошибке, и программа будет остановлена. Таким образом язык Python приучает программистов писать красивый и понятный код, и, если в другом языке вас не приучили к хорошему стилю программирования, язык Python быстро это исправит.

Поскольку программа на языке Python представляет собой обычный текстовый файл, его можно редактировать с помощью любого текстового редактора — например, того же Notepad++. Можно использовать и другие, более специализированные программы такого рода: PyScripter, PythonWin, UliPad, Eclipse с установленным модулем PyDev, Netbeans и др. (полный список приемлемых редакторов можно найти на странице **https://wiki.python.org/moin/PythonEditors**). Мы же в процессе изложения материала этой книги будем пользоваться интерактивным интерпретатором IDLE, который входит в состав стандартной поставки Python в Windows, — он идеально подходит для изучения Python.

Вторая часть книги посвящена рассмотрению версии 5.9.2 библиотеки PyQt, позволяющей создавать кроссплатформенные приложения с графическим интерфейсом. Библиотека очень проста в использовании и идеально подходит для разработки весьма серьезных оконных приложений. Пользуясь исключительно ее средствами, мы можем выводить на экран графику практически любой сложности, работать с базами данных наиболее распространенных форматов, воспроизводить мультимедийные файлы, выводить документы на печать и экспортировать их в популярный формат Adobe PDF.

А в самом конце мы с вами самостоятельно напишем на языке Python с применением библиотеки PyQt полнофункциональное приложение «Судоку», предназначенное для создания и решения одноименных головоломок.

Все листинги из этой книги вы найдете в файлах Listings.doc (листинги из глав книги), PyQt.doc (дополнительные листинги с примерами на PyQt) и в папке sudoku (листинги приложения «Судоку»), электронный архив с которыми можно загрузить с FTP-сервера издательства «БХВ-Петербург» по ссылке: **ftp://ftp.bhv.ru/9785977539784.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

Авторы книги желают вам приятного чтения и надеются, что она станет верным спутником в вашей грядущей карьере программиста! Тем не менее, не забывайте, что книги по программированию нужно не только читать, — весьма желательно выполнять все имеющиеся в них примеры, а также экспериментировать, что-либо в этих примерах изменяя.

ЧАСТЬ I

Основы языка Python

Глава 1. Первые шаги

Глава 2. Переменные

Глава 3. Операторы

Глава 4. Условные операторы и циклы

Глава 5. Числа

Глава 6. Строки и двоичные данные

Глава 7. Регулярные выражения

Глава 8. Списки, кортежи, множества и диапазоны

Глава 9. Словари

Глава 10. Работа с датой и временем

Глава 11. Пользовательские функции

Глава 12. Модули и пакеты

Глава 13. Объектно-ориентированное программирование

Глава 14. Обработка исключений

Глава 15. Итераторы, контейнеры и перечисления

Глава 16. Работа с файлами и каталогами

ГЛАВА 1

Первые шаги

Прежде чем мы начнем рассматривать синтаксис языка, необходимо сделать два замечания. Во-первых, как уже было отмечено во *введении*, не забывайте, что книги по программированию нужно не только читать, — весьма желательно выполнять все имеющиеся в них примеры, а также экспериментировать, что-нибудь в этих примерах изменяя. Поэтому, если вы удобно устроились на диване и настроились просто читать, у вас практически нет шансов изучить язык. Чем больше вы будете делать самостоятельно, тем большему научитесь.

Ну что, приступим к изучению языка? Python достоин того, чтобы его знал каждый программист!

> ***Внимание!***
>
> Начиная с версии 3.5, Python более не поддерживает Windows XP. В связи с этим в книге не будут описываться моменты, касающиеся его применения под этой версией операционной системы.

1.1. Установка Python

Вначале необходимо установить на компьютер *интерпретатор* Python (его также называют *исполняющей средой*).

1. Для загрузки дистрибутива заходим на страницу **https://www.python.org/downloads/** и в списке доступных версий щелкаем на гиперссылке **Python 3.6.3** (эта версия является самой актуальной из стабильных версий на момент подготовки книги). На открывшейся странице находим раздел **Files** и щелкаем на гиперссылке **Windows x86 executable installer** (32-разрядная редакция интерпретатора) или **Windows x86-64 executable installer** (его 64-разрядная редакция) — в зависимости от версии вашей операционной системы. В результате на наш компьютер будет загружен файл python-3.6.3.exe или python-3.6.3-amd64.exe соответственно. Затем запускаем загруженный файл двойным щелчком на нем.
2. В открывшемся окне (рис. 1.1) проверяем, установлен ли флажок **Install launcher for all users (recommended)** (Установить исполняющую среду для всех пользователей), устанавливаем флажок **Add Python 3.6 to PATH** (Добавить Python 3.6 в список путей переменной PATH) и нажимаем кнопку **Customize installation** (Настроить установку).
3. В следующем диалоговом окне (рис. 1.2) нам предлагается выбрать устанавливаемые компоненты. Оставляем установленными все флажки, представляющие эти компоненты, и нажимаем кнопку **Next**.

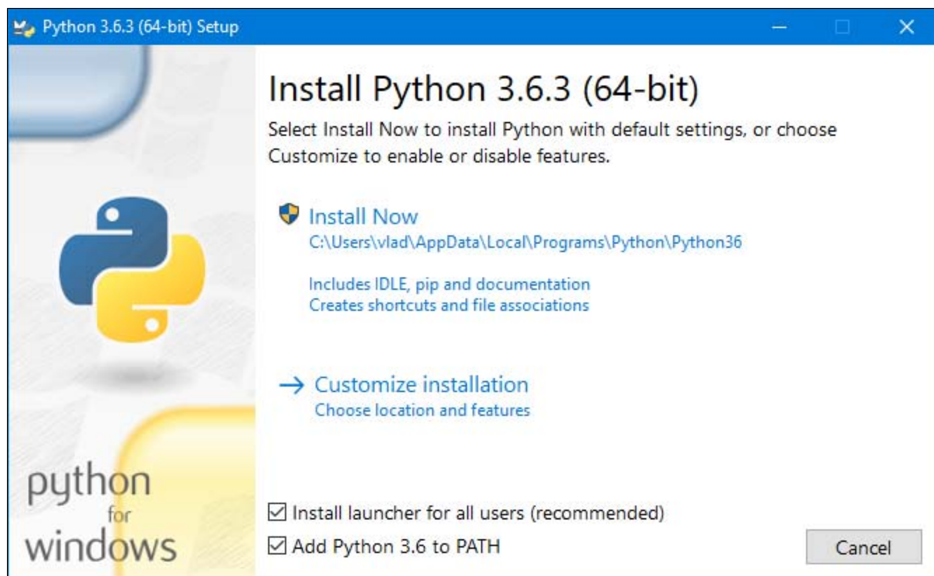

Рис. 1.1. Установка Python. Шаг 1

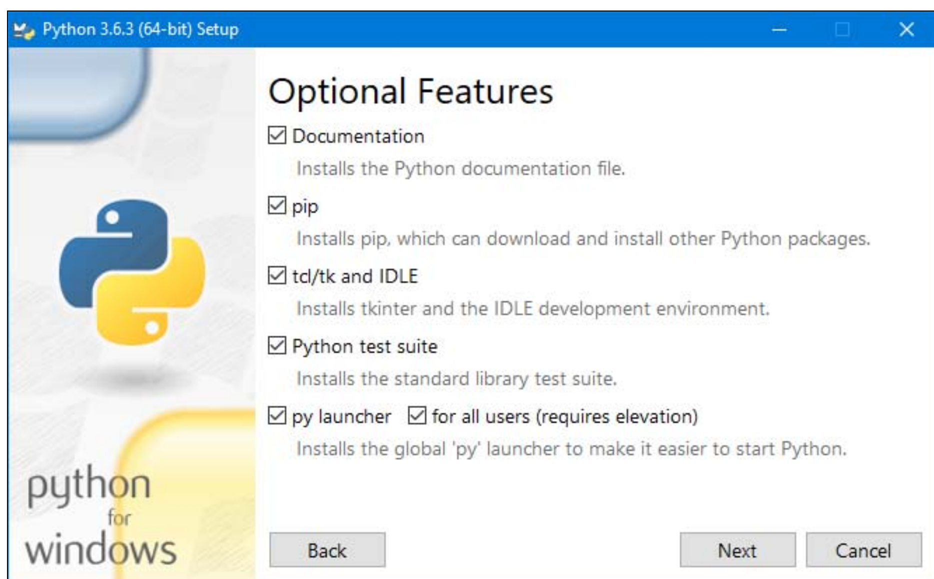

Рис. 1.2. Установка Python. Шаг 2

4. На следующем шаге (рис. 1.3) мы зададим некоторые дополнительные настройки и выберем путь установки. Проверим, установлены ли флажки **Associate files with Python (requires the py launcher)** (Ассоциировать файлы с Python), **Create shortcuts for installed applications** (Создать ярлыки для установленных приложений), **Add Python to environment variables** (Добавить Python в переменные окружения) и **Precompile standard library** (Предварительно откомпилировать стандартную библиотеку), и установим флажок **Install for all users** (Установить для всех пользователей).

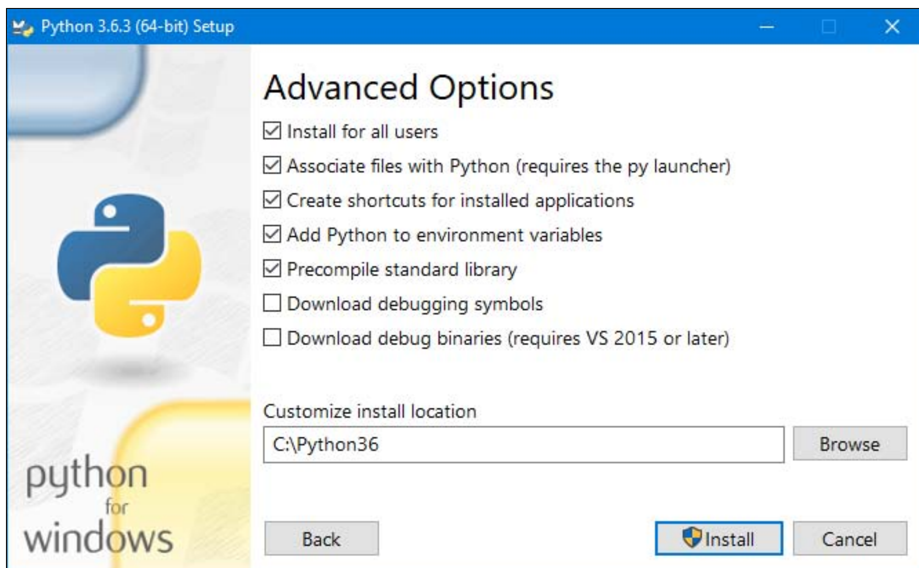

Рис. 1.3. Установка Python. Шаг 3

Внимание!

Некоторые параметры при установке Python приходится задавать по нескольку раз на разных шагах. Вероятно, это недоработка разработчиков инсталлятора.

Теперь уточним путь, по которому будет установлен Python. Изначально нам предлагается установить интерпретатор по пути c:\Program Files\Python36. Можно сделать и так, но тогда при установке любой дополнительной библиотеки понадобится запускать командную строку с повышенными правами, иначе библиотека не установится.

Авторы книги рекомендуют установить Python по пути c:\Python36, т. е. непосредственно в корень диска (см. рис. 1.3). В этом случае мы избежим проблем при установке дополнительных библиотек.

Задав все необходимые параметры, нажимаем кнопку **Install** и положительно отвечаем на появившееся на экране предупреждение UAC.

5. После завершения установки откроется окно, изображенное на рис. 1.4. Нажимаем в нем кнопку **Close** для выхода из программы установки.

В результате установки исходные файлы интерпретатора будут скопированы в папку C:\Python36. В этой папке вы найдете два исполняемых файла: python.exe и pythonw.exe. Файл python.exe предназначен для выполнения консольных приложений. Именно эта программа запускается при двойном щелчке на файле с расширением py. Файл pythonw.exe служит для запуска оконных приложений (при двойном щелчке на файле с расширением pyw) — в этом случае окно консоли выводиться не будет.

Итак, если выполнить двойной щелчок на файле python.exe, то интерактивная оболочка запустится в окне консоли (рис. 1.5). Символы `>>>` в этом окне означают приглашение для ввода инструкций на языке Python. Если после этих символов ввести, например, `2 + 2` и нажать клавишу <Enter>, то на следующей строке сразу будет выведен результат выполнения, а затем опять приглашение для ввода новой инструкции. Таким образом, это окно можно использовать в качестве калькулятора, а также для изучения языка.

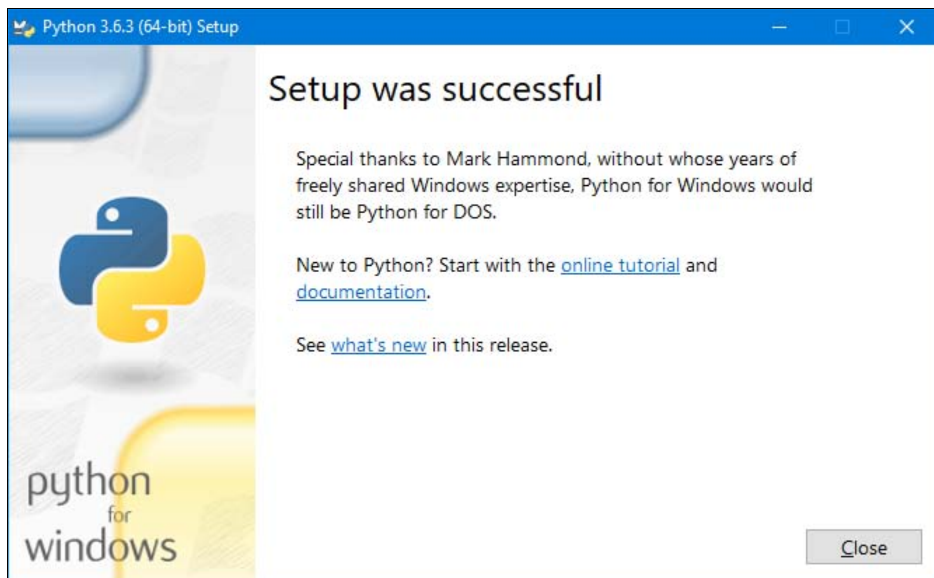

Рис. 1.4. Установка Python. Шаг 4

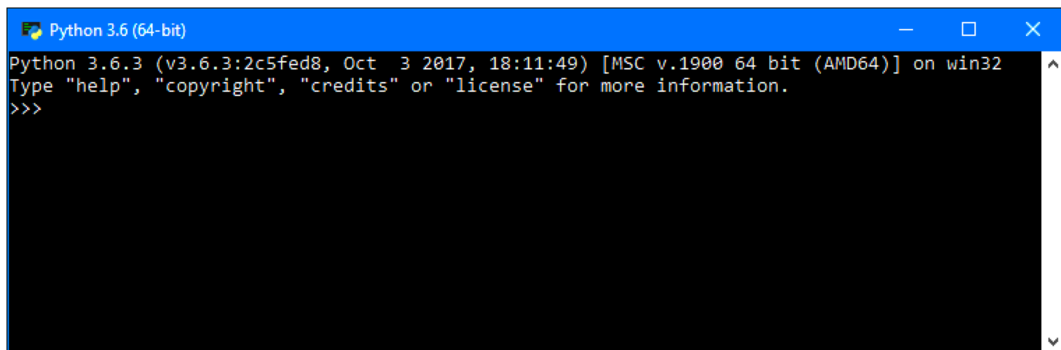

Рис. 1.5. Интерактивная оболочка

Открыть это же окно можно, выбрав пункт **Python 3.6 (32-bit)** или **Python 3.6 (64-bit)** в меню **Пуск | Программы (Все программы) | Python 3.6**.

Однако для изучения языка, а также для создания и редактирования файлов с программами, лучше пользоваться редактором IDLE, который входит в состав установленных компонентов. Для запуска этого редактора в меню **Пуск | Программы (Все программы) | Python 3.6** выбираем пункт **IDLE (Python 3.6 32-bit)** или **IDLE (Python 3.6 64-bit)**. В результате откроется окно **Python Shell** (рис. 1.6), которое выполняет все функции интерактивной оболочки, но дополнительно производит подсветку синтаксиса, выводит подсказки и др. Именно этим редактором мы будем пользоваться в процессе изучения материала книги. Более подробно редактор IDLE мы рассмотрим немного позже.

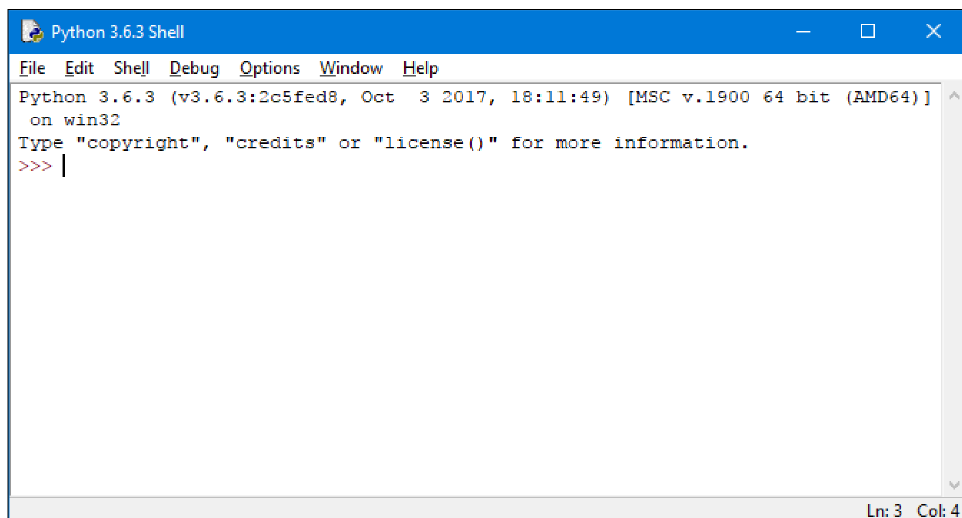

Рис. 1.6. Окно **Python Shell** редактора IDLE

1.1.1. Установка нескольких интерпретаторов Python

Версии языка Python выпускаются с завидной регулярностью, но, к сожалению, сторонние разработчики не успевают за такой скоростью и не столь часто обновляют свои модули. Поэтому иногда приходится при наличии версии Python 3 использовать на практике также и версию Python 2. Как же быть, если установлена версия 3.6, а необходимо запустить модуль для версии 2.7? В этом случае удалять версию 3.6 с компьютера не нужно. Все программы установки позволяют выбрать устанавливаемые компоненты. Существует также возможность задать ассоциацию запускаемой версии с файловым расширением — так вот эту возможность необходимо отключить при установке.

В качестве примера мы дополнительно установим на компьютер версию 2.7.13.2716, но вместо программы установки с сайта **https://www.python.org/** выберем альтернативный дистрибутив от компании ActiveState.

Итак, переходим на страницу **http://www.activestate.com/activepython/downloads/** и скачиваем дистрибутив. Последовательность запуска нескольких программ установки от компании ActiveState имеет значение, поскольку в контекстное меню добавляется пункт **Edit with Pythonwin**. С помощью этого пункта запускается редактор PythonWin, который можно использовать вместо IDLE. Соответственно, из контекстного меню будет открываться версия PythonWin, установленная последней. Установку программы производим в каталог по умолчанию (C:\Python27).

> ***Внимание!***
>
> При установке в окне **Choose Setup Type** (рис. 1.7) необходимо нажать кнопку **Custom**, а в окне **Choose Setup Options** (рис. 1.8) — сбросить флажки **Add Python to the PATH environment variable** и **Create Python file extension associations**. Не забудьте это сделать, иначе Python 3.6.3 перестанет быть текущей версией.

В состав ActivePython, кроме редактора PythonWin, входит также редактор IDLE. Однако в меню **Пуск** нет пункта, с помощью которого можно его запустить. Чтобы это исправить, создадим файл IDLE27.cmd со следующим содержимым:

```
@echo off
start C:\Python27\pythonw.exe C:\Python27\Lib\idlelib\idle.pyw
```

С помощью двойного щелчка на этом файле можно будет запускать редактор IDLE для версии Python 2.7.

Ну, а запуск IDLE для версии Python 3.6 будет по-прежнему осуществляться так же, как и предлагалось ранее, — выбором в меню **Пуск | Программы (Все программы) | Python 3.6** пункта **IDLE (Python 3.6 32-bit)** или **IDLE (Python 3.6 64-bit)**.

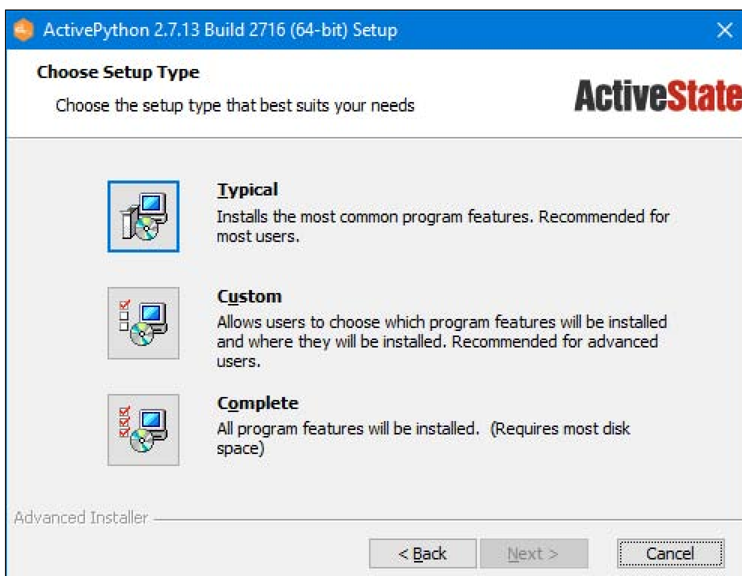

Рис. 1.7. Окно **Choose Setup Type**

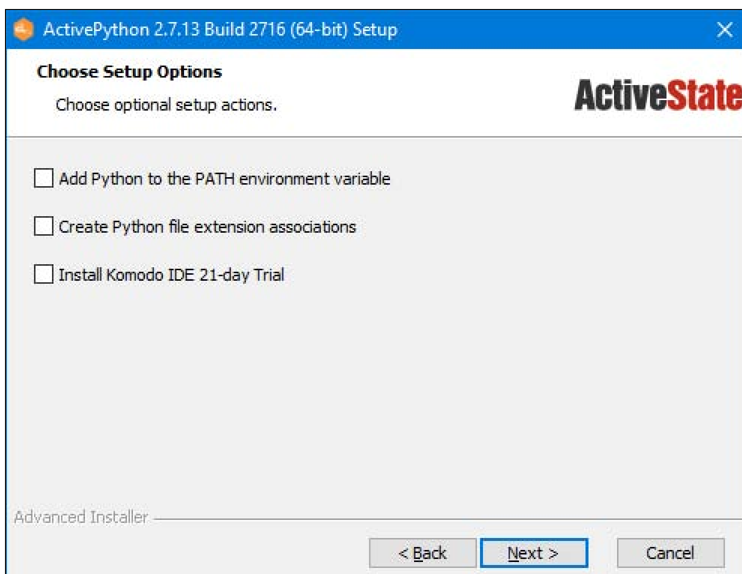

Рис. 1.8. Окно **Choose Setup Options**

1.1.2. Запуск программы с помощью разных версий Python

Теперь рассмотрим запуск программы с помощью разных версий Python. По умолчанию при двойном щелчке на значке файла запускается Python 3.6. Чтобы запустить Python-программу с помощью другой версии этого языка, щелкаем правой кнопкой мыши на значке файла с программой и в контекстном меню находим пункт **Открыть с помощью**.

На экране откроется небольшое окно выбора альтернативной программы для запуска файла (рис. 1.9). Сразу же сбросим флажок **Всегда использовать это приложение для открытия .py файлов** (подпись у этого флажка различается в разных версиях Windows) и щелкнем на гиперссылке **Еще приложения**. В окне появится список установленных на вашем компьютере программ, но нужного нам приложения Python 2.7 в нем не будет. Поэтому щелкнем на ссылке **Найти другое приложение на этом компьютере**, находящейся под списком. На экране откроется стандартное диалоговое окно открытия файла, в котором мы выберем программу python.exe, python2.exe или python2.7.exe из папки C:\Python27.

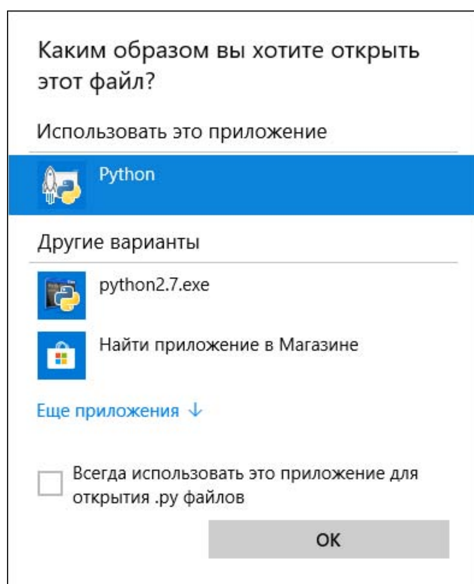

Рис. 1.9. Диалоговое окно выбора альтернативной программы для запуска файла

В Windows Vista, 7, 8 и 8.1 выбранная нами программа появится в подменю, открывающемся при выборе пункта **Открыть с помощью** (рис. 1.10), — здесь Python 2.7 представлен как **Python Launcher for Windows (Console)**. А в Windows 10 она будет присутствовать в списке, что выводится в диалоговом окне выбора альтернативной программы (см. рис. 1.9).

Для проверки установки создайте файл test.py с помощью любого текстового редактора — например, Блокнота. Содержимое файла приведено в листинге 1.1.

Листинг 1.1. Проверка установки

```
import sys
print (tuple(sys.version_info))
try:
    raw_input()          # Python 2
except NameError:
    input()              # Python 3
```

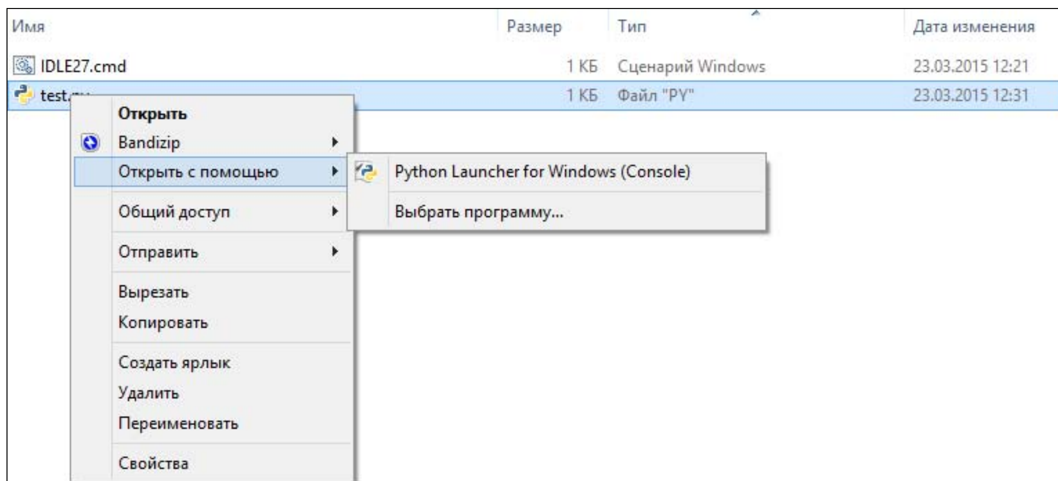

Рис. 1.10. Варианты запуска программы разными версиями Python

Затем запустите программу с помощью двойного щелчка на значке файла. Если результат выполнения: `(3, 6, 3, 'final', 0)`, то установка прошла нормально, а если `(2, 7, 13, 'final', 0)`, то вы не сбросили флажки **Add Python to the PATH environment variable** и **Create Python file extension associations** в окне **Choose Setup Options** (см. рис. 1.8).

Для изучения материала этой книги по умолчанию должна запускаться версия Python 3.6.

1.2. Первая программа на Python

Изучение языков программирования принято начинать с программы, выводящей надпись «Привет, мир!» Не будем нарушать традиции и продемонстрируем, как это будет выглядеть на Python (листинг 1.2).

Листинг 1.2. Первая программа на Python

```
# Выводим надпись с помощью функции print()
print("Привет, мир!")
```

Для запуска программы в меню **Пуск | Программы (Все программы) | Python 3.6** выбираем пункт **IDLE (Python 3.6 32-bit)** или **IDLE (Python 3.6 64-bit)**. В результате откроется окно **Python Shell**, в котором символы >>> означают приглашение ввести команду (см. рис. 1.6). Вводим сначала первую строку из листинга 1.2, а затем вторую. После ввода каждой строки нажимаем клавишу <Enter>. На следующей строке сразу отобразится результат, а далее — приглашение для ввода новой команды. Последовательность выполнения нашей программы такова:

```
>>> # Выводим надпись с помощью функции print()
>>> print("Привет, мир!")
Привет, мир!
>>>
```

> ***Примечание***
>
> Символы >>> вводить не нужно, они вставляются автоматически.

Для создания файла с программой в меню **File** выбираем пункт **New File** или нажимаем комбинацию клавиш <Ctrl>+<N>. В открывшемся окне набираем код из листинга 1.2, а затем сохраняем его под именем hello_world.py, выбрав пункт меню **File | Save** (комбинация клавиш <Ctrl>+<S>). При этом редактор сохранит файл в кодировке UTF-8 без BOM (Byte Order Mark, метка порядка байтов). Именно кодировка UTF-8 является кодировкой по умолчанию в Python 3. Если файл содержит инструкции в другой кодировке, то необходимо в первой или второй строке указать кодировку с помощью инструкции:

```
# -*- coding: <Кодировка> -*-
```

Например, для кодировки Windows-1251 инструкция будет выглядеть так:

```
# -*- coding: cp1251 -*-
```

Редактор IDLE учитывает указанную кодировку и автоматически производит перекодирование при сохранении файла. При использовании других редакторов следует проконтролировать соответствие указанной кодировки и реальной кодировки файла. Если кодировки не совпадают, то данные будут преобразованы некорректно, или во время преобразования произойдет ошибка.

Запустить программу на выполнение можно, выбрав пункт меню **Run | Run Module** или нажав клавишу <F5>. Результат выполнения программы будет отображен в окне **Python Shell**.

Запустить программу также можно с помощью двойного щелчка мыши на значке файла. В этом случае результат выполнения будет отображен в консоли Windows. Следует учитывать, что после вывода результата окно консоли сразу закрывается. Чтобы предотвратить закрытие окна, необходимо добавить вызов функции `input()`, которая станет ожидать нажатия клавиши <Enter> и не позволит окну сразу закрыться. С учетом сказанного наша программа будет выглядеть так, как показано в листинге 1.3.

Листинг 1.3. Программа для запуска с помощью двойного щелчка мыши

```
# -*- coding: utf-8 -*-
print("Привет, мир!")          # Выводим строку
input()                        # Ожидаем нажатия клавиши <Enter>
```

> ***Примечание***
>
> Если до выполнения функции `input()` возникнет ошибка, то сообщение о ней будет выведено в консоль, но сама консоль после этого сразу закроется, и вы не сможете прочитать сообщение об ошибке. Попав в подобную ситуацию, запустите программу из командной строки или с помощью редактора IDLE, и вы сможете прочитать сообщение об ошибке.

В языке Python 3 строки по умолчанию хранятся в кодировке Unicode. При выводе кодировка Unicode автоматически преобразуется в кодировку терминала. Поэтому русские буквы отображаются корректно, хотя в окне консоли в Windows по умолчанию используется кодировка cp866, а файл с программой у нас в кодировке UTF-8.

Чтобы отредактировать уже созданный файл, запустим IDLE, выполним команду меню **File | Open** (комбинация клавиш <Ctrl>+<O>) и выберем нужный файл, который будет открыт в другом окне.

> ***Напоминание***
>
> Поскольку программа на языке Python представляет собой обычный текстовый файл, сохраненный с расширением *py* или *pyw*, его можно редактировать с помощью других про-

грамм, например Notepad++. Можно также воспользоваться специализированными редакторами, скажем, PyScripter.

Когда интерпретатор Python начинает выполнение программы, хранящейся в файле, он сначала компилирует ее в особое внутреннее представление, — это делается с целью увеличить производительность кода. Файл с откомпилированным кодом хранится в папке __pycache__, вложенной в папку, где хранится сам файл программы, а его имя имеет следующий вид:

<имя файла с исходным, неоткомпилированным кодом>.cpython-<первые две цифры номера версии Python>.pyc

Так, при запуске на исполнение файла test4.py будет создан файл откомпилированного кода с именем test4.cpython-36.pyc.

При последующем запуске на выполнение того же файла будет исполняться именно откомпилированный код. Если же мы исправим исходный код, программа его автоматически перекомпилирует. При необходимости мы можем удалить файлы с откомпилированным кодом или даже саму папку __pycache__ — впоследствии интерпретатор сформирует их заново.

1.3. Структура программы

Как вы уже знаете, программа на языке Python представляет собой обычный текстовый файл с инструкциями. Каждая инструкция располагается на отдельной строке. Если инструкция не является вложенной, она должна начинаться с начала строки, иначе будет выведено сообщение об ошибке:

```
>>>  import sys

SyntaxError: unexpected indent
>>>
```

В этом случае перед инструкцией `import` расположен один лишний пробел, который привел к выводу сообщения об ошибке.

Если программа предназначена для исполнения в операционной системе UNIX, то в первой строке необходимо указать путь к интерпретатору Python:

```
#!/usr/bin/python
```

В некоторых операционных системах путь к интерпретатору выглядит по-другому:

```
#!/usr/local/bin/python
```

Иногда можно не указывать точный путь к интерпретатору, а передать название языка программе `env`:

```
#!/usr/bin/env python
```

В этом случае программа `env` произведет поиск интерпретатора Python в соответствии с настройками путей поиска.

Помимо указания пути к интерпретатору Python, необходимо, чтобы в правах доступа к файлу был установлен бит на выполнение. Кроме того, следует помнить, что перевод строки в операционной системе Windows состоит из последовательности символов `\r` (перевод каретки) и `\n` (перевод строки). В операционной системе UNIX перевод строки осу-

ществляется только одним символом `\n`. Если загрузить файл программы по протоколу FTP в бинарном режиме, то символ `\r` вызовет фатальную ошибку. По этой причине файлы по протоколу FTP следует загружать только в текстовом режиме (режим ASCII). В этом режиме символ `\r` будет удален автоматически.

После загрузки файла следует установить права на выполнение. Для исполнения скриптов на Python устанавливаем права в 755 (`-rwxr-xr-x`).

Во второй строке (для ОС Windows в первой строке) следует указать кодировку. Если кодировка не указана, то предполагается, что файл сохранен в кодировке UTF-8. Для кодировки Windows-1251 строка будет выглядеть так:

```
# -*- coding: cp1251 -*-
```

Редактор IDLE учитывает указанную кодировку и автоматически производит перекодирование при сохранении файла. Получить полный список поддерживаемых кодировок и их псевдонимы позволяет код, приведенный в листинге 1.4.

Листинг 1.4. Вывод списка поддерживаемых кодировок

```
# -*- coding: utf-8 -*-
import encodings.aliases
arr = encodings.aliases.aliases
keys = list( arr.keys() )
keys.sort()
for key in keys:
    print("%s => %s" % (key, arr[key]))
```

Во многих языках программирования (например, в PHP, Perl и др.) каждая инструкция должна завершаться точкой с запятой. В языке Python в конце инструкции также можно поставить точку с запятой, но это не обязательно. Более того, в отличие от языка JavaScript, где рекомендуется завершать инструкции точкой с запятой, в языке Python точку с запятой ставить *не рекомендуется*. Концом инструкции является конец строки. Тем не менее, если необходимо разместить несколько инструкций на одной строке, точку с запятой *следует указать*:

```
>>> x = 5; y = 10; z = x + y # Три инструкции на одной строке
>>> print(z)
15
```

Еще одной отличительной особенностью языка Python является отсутствие ограничительных символов для выделения инструкций внутри блока. Например, в языке PHP инструкции внутри цикла `while` выделяются фигурными скобками:

```
$i = 1;
while ($i < 11) {
   echo $i . "\n";
   $i++;
}
echo "Конец программы";
```

В языке Python тот же код будет выглядеть по-другому (листинг 1.5).

Листинг 1.5. Выделение инструкций внутри блока

```
i = 1
while i < 11:
    print(i)
    i += 1
print("Конец программы")
```

Обратите внимание, что перед всеми инструкциями внутри блока расположено одинаковое количество пробелов. Именно так в языке Python выделяются *блоки*. Инструкции, перед которыми расположено одинаковое количество пробелов, являются *телом блока*. В нашем примере две инструкции выполняются десять раз. Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В нашем случае это функция `print()`, которая выводит строку `"Конец программы"`. Если количество пробелов внутри блока окажется разным, то интерпретатор выведет сообщение о фатальной ошибке, и программа будет остановлена. Так язык Python приучает программистов писать красивый и понятный код.

> ***Примечание***
>
> В языке Python принято использовать четыре пробела для выделения инструкций внутри блока.

Если блок состоит из одной инструкции, то допустимо разместить ее на одной строке с основной инструкцией. Например, код:

```
for i in range(1, 11):
    print(i)
print("Конец программы")
```

можно записать так:

```
for i in range(1, 11): print(i)
print("Конец программы")
```

Если инструкция является слишком длинной, то ее можно перенести на следующую строку, например, так:

- в конце строки поставить символ `\`, после которого должен следовать перевод строки. Другие символы (в том числе и комментарии) недопустимы. Пример:

  ```
  x = 15 + 20 \
      + 30
  print(x)
  ```

- поместить выражение внутри круглых скобок. Этот способ лучше, т. к. внутри круглых скобок можно разместить любое выражение. Пример:

  ```
  x = (15 + 20                 # Это комментарий
      + 30)
  print(x)
  ```

- определение списка и словаря можно разместить на нескольких строках, т. к. при этом используются квадратные и фигурные скобки соответственно. Пример определения списка:

  ```
  arr = [15, 20,               # Это комментарий
         30]
  print(arr)
  ```

Пример определения словаря:

```
arr = {"x": 15, "y": 20,    # Это комментарий
       "z": 30}
print(arr)
```

1.4. Комментарии

Комментарии предназначены для вставки пояснений в текст программы, интерпретатор полностью их игнорирует. Внутри комментария может располагаться любой текст, включая инструкции, которые выполнять не следует.

> **Совет**
>
> Помните — комментарии нужны программисту, а не интерпретатору Python. Вставка комментариев в код позволит через некоторое время быстро вспомнить предназначение фрагмента кода.

В языке Python присутствует только *однострочный комментарий*. Он начинается с символа `#`:

```
# Это комментарий
```

Однострочный комментарий может начинаться не только с начала строки, но и располагаться после инструкции. Например, в следующем примере комментарий расположен после инструкции, предписывающей вывести надпись `"Привет, мир!"`:

```
print("Привет, мир!") # Выводим надпись с помощью функции print()
```

Если же символ комментария разместить перед инструкцией, то она не будет выполнена:

```
# print("Привет, мир!") Эта инструкция выполнена не будет
```

Если символ `#` расположен внутри кавычек или апострофов, то он не является символом комментария:

```
print("# Это НЕ комментарий")
```

Так как в языке Python нет многострочного комментария, то комментируемый фрагмент часто размещают внутри утроенных кавычек (или утроенных апострофов):

```
"""
Эта инструкция выполнена не будет
print("Привет, мир!")
"""
```

Следует заметить, что этот фрагмент кода не игнорируется интерпретатором, т. к. он не является комментарием. В результате выполнения фрагмента будет создан объект строкового типа. Тем не менее, инструкции внутри утроенных кавычек выполнены не будут, поскольку интерпретатор сочтет их простым текстом. Такие строки являются строками документирования, а не комментариями.

1.5. Дополнительные возможности IDLE

Поскольку в процессе изучения материала этой книги в качестве редактора мы будем использовать IDLE, рассмотрим дополнительные возможности этой среды разработки.

Как вы уже знаете, в окне **Python Shell** символы >>> означают приглашение ввести команду. Введя команду, нажимаем клавишу <Enter> — на следующей строке сразу отобразится ре-

зультат (при условии, что инструкция возвращает значение), а далее — приглашение для ввода новой команды. При вводе многострочной команды после нажатия клавиши <Enter> редактор автоматически вставит отступ и будет ожидать дальнейшего ввода. Чтобы сообщить редактору о конце ввода команды, необходимо дважды нажать клавишу <Enter>:

```
>>> for n in range(1, 3):
        print(n)


1
2
>>>
```

В предыдущем разделе мы выводили строку `"Привет, мир!"` с помощью функции `print()`. В окне **Python Shell** это делать не обязательно — мы можем просто ввести строку и нажать клавишу <Enter> для получения результата:

```
>>> "Привет, мир!"
'Привет, мир!'
>>>
```

Обратите внимание на то, что строки выводятся в апострофах. Этого не произойдет, если выводить строку с помощью функции `print()`:

```
>>> print("Привет, мир!")
Привет, мир!
>>>
```

Учитывая возможность получить результат сразу после ввода команды, окно **Python Shell** можно использовать для изучения команд, а также в качестве многофункционального калькулятора:

```
>>> 12 * 32 + 54
438
>>>
```

Результат вычисления последней инструкции сохраняется в переменной _ (одно подчеркивание). Это позволяет производить дальнейшие расчеты без ввода предыдущего результата. Вместо него достаточно ввести символ подчеркивания:

```
>>> 125 * 3              # Умножение
375
>>> _ + 50               # Сложение. Эквивалентно 375 + 50
425
>>> _ / 5                # Деление. Эквивалентно 425 / 5
85.0
>>>
```

При вводе команды можно воспользоваться комбинацией клавиш <Ctrl>+<Пробел>. В результате будет отображен список, из которого можно выбрать нужный идентификатор. Если при открытом списке вводить буквы, то показываться будут идентификаторы, начинающиеся с этих букв. Выбирать идентификатор необходимо с помощью клавиш <↑> и <↓>. После выбора не следует нажимать клавишу <Enter>, иначе это приведет к выполнению инструкции, — просто вводите инструкцию дальше, а список закроется. Такой же список будет автоматически появляться (с некоторой задержкой) при обращении к атрибутам объекта или модуля после ввода точки. Для автоматического завершения идентификатора

после ввода первых букв можно воспользоваться комбинацией клавиш <Alt>+</>. При каждом последующем нажатии этой комбинации будет вставляться следующий идентификатор. Эти две комбинации клавиш очень удобны, если вы забыли, как пишется слово, или хотите, чтобы редактор закончил его за вас.

При необходимости повторно выполнить ранее введенную инструкцию ее приходится набирать заново. Можно, конечно, скопировать инструкцию, а затем вставить, но как вы можете сами убедиться, в контекстном меню нет пунктов **Copy** (Копировать) и **Paste** (Вставить). Они расположены в меню **Edit**. Постоянно выбирать пункты из этого меню очень неудобно. Одним из решений проблемы является использование комбинации клавиш быстрого доступа <Ctrl>+<C> (Копировать) и <Ctrl>+<V> (Вставить). Комбинации стандартны для Windows, и вы наверняка их уже использовали ранее. Но опять-таки, прежде чем скопировать инструкцию, ее предварительно необходимо выделить. Редактор IDLE избавляет нас от лишних действий и предоставляет комбинацию клавиш <Alt>+<N> для вставки первой введенной инструкции, а также комбинацию <Alt>+<P> для вставки последней инструкции. Каждое последующее нажатие этих клавиш будет вставлять следующую (или предыдущую) инструкцию. Для еще более быстрого повторного ввода инструкции следует предварительно ввести ее первые буквы. В этом случае перебираться будут только инструкции, начинающиеся с этих букв.

1.6. Вывод результатов работы программы

Вывести результаты работы программы можно с помощью функции `print()`. Функция имеет следующий формат:

```
print([<Объекты>][, sep=' '][, end='\n'][, file=sys.stdout][, flush=False])
```

Функция `print()` преобразует объект в строку и посылает ее в стандартный вывод `stdout`. С помощью параметра `file` можно перенаправить вывод в другое место — например, в файл. При этом, если параметр `flush` имеет значение `False`, выводимые значения будут принудительно записаны в файл. Перенаправление вывода мы подробно рассмотрим при изучении файлов.

После вывода строки автоматически добавляется символ перевода строки:

```
print("Строка 1")
print("Строка 2")
```

Результат:

```
Строка 1
Строка 2
```

Если необходимо вывести результат на той же строке, то в функции `print()` данные указываются через запятую в первом параметре:

```
print("Строка 1", "Строка 2")
```

Результат:

```
Строка 1 Строка 2
```

Как видно из примера, между выводимыми строками автоматически вставляется пробел. С помощью параметра `sep` можно указать другой символ. Например, выведем строки без пробела между ними:

```
print("Строка1", "Строка2", sep="")
```

Результат:

```
Строка 1Строка 2
```

После вывода объектов в конце добавляется символ перевода строки. Если необходимо произвести дальнейший вывод на той же строке, то в параметре `end` следует указать другой символ:

```
print("Строка 1", "Строка 2", end=" ")
print("Строка 3")
# Выведет: Строка 1 Строка 2 Строка 3
```

Если, наоборот, необходимо вставить символ перевода строки, то функция `print()` указывается без параметров:

```
for n in range(1, 5):
    print(n, end=" ")
print()
print("Это текст на новой строке")
```

Результат выполнения:

```
1 2 3 4
Это текст на новой строке
```

Здесь мы использовали цикл `for`, который позволяет последовательно перебирать элементы. На каждой итерации цикла переменной `n` присваивается новое число, которое мы выводим с помощью функции `print()`, расположенной на следующей строке.

Обратите внимание, что перед функцией мы добавили четыре пробела. Как уже отмечалось ранее, таким образом в языке Python выделяются блоки. При этом инструкции, перед которыми расположено одинаковое количество пробелов, представляют собой тело цикла. Все эти инструкции выполняются определенное количество раз. Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В нашем случае это функция `print()` без параметров, которая вставляет символ перевода строки.

Если необходимо вывести большой блок текста, его следует разместить между утроенными кавычками или утроенными апострофами. В этом случае текст сохраняет свое форматирование:

```
print("""Строка 1
Строка 2
Строка 3""")
```

В результате выполнения этого примера мы получим три строки:

```
Строка 1
Строка 2
Строка 3
```

Для вывода результатов работы программы вместо функции `print()` можно использовать метод `write()` объекта `sys.stdout`:

```
import sys                    # Подключаем модуль sys
sys.stdout.write("Строка")    # Выводим строку
```

В первой строке с помощью оператора `import` мы подключаем модуль `sys`, в котором объявлен объект `stdout`. Далее с помощью метода `write()` этого объекта выводим строку. Сле-

дует заметить, что метод не вставляет символ перевода строки, поэтому при необходимости следует добавить его самим с помощью символа `\n`:

```
import sys
sys.stdout.write("Строка 1\n")
sys.stdout.write("Строка 2")
```

1.7. Ввод данных

Для ввода данных в Python 3 предназначена функция `input()`, которая получает данные со стандартного ввода `stdin`. Функция имеет следующий формат:

```
[<Значение> = ] input([<Сообщение>])
```

Для примера переделаем нашу первую программу так, чтобы она здоровалась не со всем миром, а только с нами (листинг 1.6).

Листинг 1.6. Пример использования функции `input()`

```
# -*- coding: utf-8 -*-
name = input("Введите ваше имя: ")
print("Привет,", name)
input("Нажмите <Enter> для закрытия окна")
```

Чтобы окно сразу не закрылось, в конце программы указан еще один вызов функции `input()`. В этом случае окно не закроется, пока не будет нажата клавиша <Enter>.

Вводим код и сохраняем файл, например, под именем test2.py, а затем запускаем программу на выполнение с помощью двойного щелчка на значке файла. Откроется черное окно, в котором мы увидим надпись: **Введите ваше имя:**. Вводим свое имя, например `Николай`, и нажимаем клавишу <Enter>. В результате будет выведено приветствие: **Привет, Николай**.

При использовании функции `input()` следует учитывать, что при достижении конца файла или при нажатии комбинации клавиш <Ctrl>+<Z>, а затем клавиши <Enter> генерируется исключение `EOFError`. Если не предусмотреть обработку исключения, то программа аварийно завершится. Обработать исключение можно следующим образом:

```
try:
    s = input("Введите данные: ")
    print(s)
except EOFError:
    print("Обработали исключение EOFError")
```

Если внутри блока `try` возникнет исключение `EOFError`, то управление будет передано в блок `except`. После исполнения инструкций в блоке `except` программа нормально продолжит работу.

В Python 2 для ввода данных применялись две функции: `raw_input()` и `input()`. Функция `raw_input()` просто возвращала введенные данные, а функция `input()` предварительно обрабатывала данные с помощью функции `eval()` и затем возвращала результат ее выполнения. В Python 3 функция `raw_input()` была переименована в `input()`, а прежняя функция `input()` — удалена. Чтобы в Python 3 вернуться к поведению функции `input()` в стиле Python 2, необходимо передать значение в функцию `eval()` явным образом:

```
# -*- coding: utf-8 -*-
result = eval(input("Введите инструкцию: ")) # Вводим:  2 + 2
print("Результат:", result)                  # Выведет: 4
input()
```

> **Внимание!**
>
> Функция `eval()` выполнит любую введенную инструкцию. Никогда не используйте этот код, если не доверяете пользователю.

Передать данные можно в командной строке, указав их после имени файла программы. Такие данные доступны через список `argv` модуля `sys`. Первый элемент списка `argv` будет содержать название файла запущенной программы, а последующие элементы — переданные данные. Для примера создадим файл test3.py в папке C:\book. Содержимое файла приведено в листинге 1.7.

Листинг 1.7. Получение данных из командной строки

```
# -*- coding: utf-8 -*-
import sys
arr = sys.argv[:]
for n in arr:
    print(n)
```

Теперь запустим программу на выполнение из командной строки и передадим ей данные. Для этого вызовем командную строку: выберем в меню **Пуск** пункт **Выполнить**, в открывшемся окне наберем команду `cmd` и нажмем кнопку **OK** — откроется черное окно командной строки с приглашением для ввода команд. Перейдем в папку C:\book, набрав команду:

```
cd C:\book
```

В командной строке должно появиться приглашение:

```
C:\book>
```

Для запуска нашей программы вводим команду:

```
C:\Python36\python.exe test3.py -uNik -p123
```

В этой команде мы передаем имя файла (`test3.py`) и некоторые данные (`-uNik` и `-p123`). Результат выполнения программы будет выглядеть так:

```
test3.py
-uNik
-p123
```

1.8. Доступ к документации

При установке Python на компьютер помимо собственно интерпретатора копируется документация по этому языку в формате CHM. Чтобы открыть ее, в меню **Пуск | Программы (Все программы) | Python 3.6** нужно выбрать пункт **Python 3.6 Manuals (32-bit)** или **Python 3.6 Manuals (64-bit)**.

Если в меню **Пуск | Программы (Все программы) | Python 3.6** выбрать пункт **Python 3.6 Module Docs (32-bit)** или **Python 3.6 Module Docs (64-bit)**, запустится сервер документов

pydoc (рис. 1.11). Он представляет собой написанную на самом Python программу веб-сервера, выводящую результаты своей работы в веб-браузере.

Сразу после запуска pydoc откроется веб-браузер, в котором будет выведен список всех стандартных модулей, поставляющихся в составе Python. Щелкнув на названии модуля, представляющем собой гиперссылку, мы откроем страницу с описанием всех классов, функций и констант, объявленных в этом модуле.

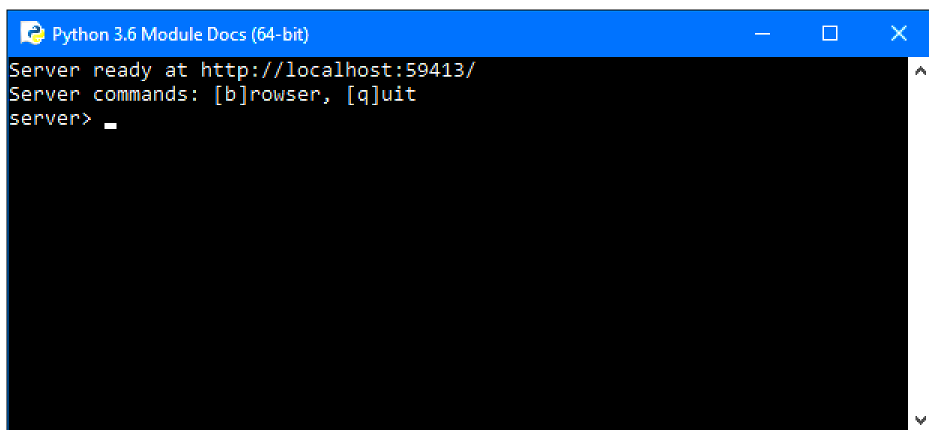

Рис. 1.11. Окно pydoc

Чтобы завершить работу pydoc, следует переключиться в его окно (см. рис. 1.11), ввести в нем команду `q` (от quit, выйти) и нажать клавишу <Enter> — окно при этом автоматически закроется. А введенная там команда `b` (от browser, браузер) повторно выведет в браузере страницу со списком модулей.

В окне **Python Shell** также можно отобразить документацию. Для этого предназначена функция `help()`. В качестве примера отобразим документацию по встроенной функции `input()`:

```
>>> help(input)
```

Результат выполнения:

```
Help on built-in function input in module builtins:

input(prompt=None, /)
    Read a string from standard input.  The trailing newline is stripped.

    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.

    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
    On *nix systems, readline is used if available.
```

С помощью функции `help()` можно получить документацию не только по конкретной функции, но и по всему модулю сразу. Для этого предварительно необходимо подключить модуль. Например, подключим модуль `builtins`, содержащий определения всех встроенных функций и классов, а затем выведем документацию по этому модулю:

```
>>> import builtins
>>> help(builtins)
```

При рассмотрении комментариев мы говорили, что часто для комментирования большого фрагмента кода используются утроенные кавычки или утроенные апострофы. Такие строки не являются комментариями в полном смысле этого слова. Вместо комментирования фрагмента создается объект строкового типа, который сохраняется в атрибуте `__doc__`. Функция `help()` при составлении документации получает информацию из этого атрибута. Такие строки называются *строками документирования*.

В качестве примера создадим файл test4.py, содержимое которого показано в листинге 1.8.

Листинг 1.8. Тестовый модуль test4.py

```
# -*- coding: utf-8 -*-
""" Это описание нашего модуля """
def func():
    """ Это описание функции"""
    pass
```

Теперь подключим этот модуль и выведем содержимое строк документирования. Все эти действия выполняет код из листинга 1.9.

Листинг 1.9. Вывод строк документирования посредством функции `help()`

```
# -*- coding: utf-8 -*-
import test4                          # Подключаем файл test4.py
help(test4)
input()
```

Запустим эту программу из среды Python Shell. (Если запустить ее щелчком мыши, вывод будет выполнен в окне интерактивной оболочки, и результат окажется нечитаемым. Вероятно, это происходит вследствие ошибки в интерпретаторе.) Вот что мы увидим:

```
Help on module test4:

NAME
    test4 - Это описание нашего модуля

FUNCTIONS
    func()
        Это описание функции

FILE
    d:\data\документы\работа\книги\python 3 и pyqt 5 разработка приложений
ii\примеры\1\test4.py
```

Теперь получим содержимое строк документирования с помощью атрибута `__doc__`. Как это делается, показывает код из листинга 1.10.

Листинг 1.10. Вывод строк документирования посредством атрибута __doc__

```
# -*- coding: utf-8 -*-
import test4                        # Подключаем файл test4.py
print(test4.__doc__)
print(test4.func.__doc__)
input()
```

Результат выполнения:

```
Это описание нашего модуля
Это описание функции
```

Атрибут `__doc__` можно использовать вместо функции `help()`. В качестве примера получим документацию по функции `input()`:

```
>>> print(input.__doc__)
```

Результат выполнения:

```
Read a string from standard input.  The trailing newline is stripped.

The prompt string, if given, is printed to standard output without a
trailing newline before reading input.

If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
On *nix systems, readline is used if available.
```

Получить список всех идентификаторов внутри модуля позволяет функция `dir()`. Пример ее использования показывает код из листинга 1.11.

Листинг 1.11. Получение списка идентификаторов

```
# -*- coding: utf-8 -*-
import test4                        # Подключаем файл test4.py
print(dir(test4))
input()
```

Результат выполнения:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'func']
```

Теперь получим список всех встроенных идентификаторов:

```
>>> import builtins
>>> print(dir(builtins))
```

Функция `dir()` может не принимать параметров вообще. В этом случае возвращается список идентификаторов текущего модуля:

```
>>> print(dir())
```

Результат выполнения:

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

ГЛАВА 2

Переменные

Все данные в языке Python представлены *объектами*. Каждый объект имеет тип данных и значение. Для доступа к объекту предназначены *переменные*. При инициализации в переменной сохраняется *ссылка* на объект (адрес объекта в памяти компьютера). Благодаря этой ссылке можно в дальнейшем изменять объект из программы.

2.1. Именование переменных

Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знаков подчеркивания, причем имя переменной не может начинаться с цифры. Кроме того, следует избегать указания символа подчеркивания в начале имени, поскольку идентификаторам с таким символом определено специальное назначение. Например, имена, начинающиеся с символа подчеркивания, не импортируются из модуля с помощью инструкции `from module import *`, а имена, включающие по два символа подчеркивания — в начале и в конце, для интерпретатора имеют особый смысл.

В качестве имени переменной нельзя использовать *ключевые слова*. Получить список всех ключевых слов позволяет такой код:

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

Помимо ключевых слов, следует избегать совпадений со встроенными идентификаторами. Дело в том, что, в отличие от ключевых слов, встроенные идентификаторы можно переопределять, но дальнейший результат может стать для вас неожиданным:

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.

>>> help = 10
>>> help
10
```

```
>>> help(abs)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    help(abs)
TypeError: 'int' object is not callable
```

В этом примере мы с помощью встроенной функции `help()` получаем справку по функции `abs()`. Далее переменной `help` присваиваем число 10. После переопределения идентификатора мы больше не можем пользоваться функцией `help()`, т. к. это приведет к выводу сообщения об ошибке. По этой причине лучше избегать имен, совпадающих со встроенными идентификаторами. Очень часто подобная ошибка возникает при попытке назвать переменную, в которой предполагается хранение строки, именем `str`. Вроде бы логично, но `str` является часто используемым встроенным идентификатором и после такого переопределения поведение программы становится непредсказуемым. В редакторе IDLE встроенные идентификаторы подсвечиваются фиолетовым цветом. Обращайте внимание на цвет переменной — он должен быть черным. Если вы заметили, что переменная подсвечена, то название переменной следует обязательно изменить. Получить полный список встроенных идентификаторов позволяет следующий код:

```
>>> import builtins
>>> print(dir(builtins))
```

Правильные имена переменных: `x`, `y1`, `strName`, `str_name`.

Неправильные имена переменных: `1y`, `ИмяПеременной`.

Последнее имя неправильное, т. к. в нем используются русские буквы. Хотя на самом деле такой вариант также будет работать, но лучше русские буквы все же не применять:

```
>>> ИмяПеременной = 10              # Лучше так не делать!!!
>>> ИмяПеременной
10
```

При указании имени переменной важно учитывать регистр букв: `x` и `X` — разные переменные:

```
>>> x = 10; X = 20
>>> x, X
(10, 20)
```

2.2. Типы данных

В Python 3 объекты могут иметь следующие типы данных:

- `bool` — логический тип данных. Может содержать значения `True` или `False`, которые ведут себя как числа `1` и `0` соответственно:

  ```
  >>> type(True), type(False)
  (<class 'bool'>, <class 'bool'>)
  >>> int(True), int(False)
  (1, 0)
  ```

- `NoneType` — объект со значением `None` (обозначает отсутствие значения):

  ```
  >>> type(None)
  <class 'NoneType'>
  ```

В логическом контексте значение `None` интерпретируется как `False`:

```
>>> bool(None)
False
```

- `int` — целые числа. Размер числа ограничен лишь объемом оперативной памяти:

```
>>> type(2147483647), type(999999999999999999999999)
(<class 'int'>, <class 'int'>)
```

- `float` — вещественные числа:

```
>>> type(5.1), type(8.5e-3)
(<class 'float'>, <class 'float'>)
```

- `complex` — комплексные числа:

```
>>> type(2+2j)
<class 'complex'>
```

- `str` — Unicode-строки:

```
>>> type("Строка")
<class 'str'>
```

- `bytes` — неизменяемая последовательность байтов:

```
>>> type(bytes("Строка", "utf-8"))
<class 'bytes'>
```

- `bytearray` — изменяемая последовательность байтов:

```
>>> type(bytearray("Строка", "utf-8"))
<class 'bytearray'>
```

- `list` — списки. Тип данных `list` аналогичен массивам в других языках программирования:

```
>>> type( [1, 2, 3] )
<class 'list'>
```

- `tuple` — кортежи:

```
>>> type( (1, 2, 3) )
<class 'tuple'>
```

- `range` — диапазоны:

```
>>> type( range(1, 10) )
<class 'range'>
```

- `dict` — словари. Тип данных `dict` аналогичен ассоциативным массивам в других языках программирования:

```
>>> type( {"x": 5, "y": 20} )
<class 'dict'>
```

- `set` — множества (коллекции уникальных объектов):

```
>>> type( {"a", "b", "c"} )
<class 'set'>
```

♦ `frozenset` — неизменяемые множества:

```
>>> type(frozenset(["a", "b", "c"]))
<class 'frozenset'>
```

♦ `ellipsis` — обозначается в виде трех точек или слова `Ellipsis`. Тип `ellipsis` используется в расширенном синтаксисе получения среза:

```
>>> type(...), ..., ... is Ellipsis
(<class 'ellipsis'>, Ellipsis, True)
>>> class C():
        def __getitem__(self, obj): return obj

>>> c = C()
>>> c[..., 1:5, 0:9:1, 0]
(Ellipsis, slice(1, 5, None), slice(0, 9, 1), 0)
```

♦ `function` — функции:

```
>>> def func(): pass

>>> type(func)
<class 'function'>
```

♦ `module` — модули:

```
>>> import sys
>>> type(sys)
<class 'module'>
```

♦ `type` — классы и типы данных. Не удивляйтесь! Все данные в языке Python являются объектами, даже сами типы данных!

```
>>> class C: pass

>>> type(C)
<class 'type'>
>>> type(type(""))
<class 'type'>
```

Основные типы данных делятся на *изменяемые* и *неизменяемые*. К изменяемым типам относятся списки, словари и тип `bytearray`. Пример изменения элемента списка:

```
>>> arr = [1, 2, 3]
>>> arr[0] = 0                # Изменяем первый элемент списка
>>> arr
[0, 2, 3]
```

К неизменяемым типам относятся числа, строки, кортежи, диапазоны и тип `bytes`. Например, чтобы получить строку из двух других строк, необходимо использовать операцию *конкатенации*, а ссылку на новый объект присвоить переменной:

```
>>> str1 = "авто"
>>> str2 = "транспорт"
>>> str3 = str1 + str2        # Конкатенация
>>> print(str3)
автотранспорт
```

Кроме того, типы данных делятся на *последовательности* и *отображения*. К последовательностям относятся строки, списки, кортежи, диапазоны, типы `bytes` и `bytearray`, а к отображениям — словари.

Последовательности и отображения поддерживают механизм итераторов, позволяющий произвести обход всех элементов с помощью метода `__next__()` или функции `next()`. Например, вывести элементы списка можно так:

```
>>> arr = [1, 2]
>>> i = iter(arr)
>>> i.__next__()              # Метод __next__()
1
>>> next(i)                   # Функция next()
2
```

Если используется словарь, то на каждой итерации возвращается ключ:

```
>>> d = {"x": 1, "y": 2}
>>> i = iter(d)
>>> i.__next__()              # Возвращается ключ
'y'
>>> d[i.__next__()]           # Получаем значение по ключу
1
```

На практике подобным способом не пользуются. Вместо него применяется цикл `for`, который использует механизм итераторов незаметно для нас. Например, вывести элементы списка можно так:

```
>>> for i in [1, 2]:
        print(i)
```

Перебрать слово по буквам можно точно так же. Для примера вставим тире после каждой буквы:

```
>>> for i in "Строка":
        print(i + " -", end=" ")
```

Результат:

```
С — т — р — о — к — а -
```

Пример перебора элементов словаря:

```
>>> d = {"x": 1, "y": 2}
>>> for key in d:
        print( d[key] )
```

Последовательности поддерживают также обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`) и проверку на вхождение (оператор `in`). Все эти операции мы будем подробно рассматривать по мере изучения языка.

2.3. Присваивание значения переменным

Присваивание — это занесение в переменную какого-либо значения (при этом значение, хранившееся в переменной ранее, теряется). Присваивание выполняется с помощью оператора `=` (знак равенства) таким образом:

```
>>> x = 7           # Тип int
>>> y = 7.8         # Тип float
>>> s1 = "Строка"   # Переменной s1 присвоено значение Строка
>>> s2 = 'Строка'   # Переменной s2 также присвоено значение Строка
>>> b = True        # Переменной b присвоено логическое значение True
```

В одной строке можно присвоить значение сразу нескольким переменным:

```
>>> x = y = 10      # Переменным x и y присвоено число 10
>>> x, y
(10, 10)
```

После присваивания значения в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при *групповом присваивании*. Групповое присваивание можно использовать для чисел, строк и кортежей, но для изменяемых объектов этого делать нельзя. Пример:

```
>>> x = y = [1, 2]          # Якобы создали два объекта
>>> x, y
([1, 2], [1, 2])
```

В этом примере мы создали список из двух элементов, присвоили его переменным `x` и `y` и теперь полагаем, что эти переменные хранят две разные копии упомянутого списка. Теперь попробуем изменить значение одного из элементов списка, что хранится в переменной `y`:

```
>>> y[1] = 100              # Изменяем второй элемент списка
>>> x, y
([1, 100], [1, 100])
```

Как видно из примера, изменение значения элемента списка из переменной `y` привело также к изменению значения того же элемента списка из переменной `x`. То есть, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> y[1] = 100              # Изменяем второй элемент
>>> x, y
([1, 2], [1, 100])
```

Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор `is`. Если переменные ссылаются на один и тот же объект, оператор `is` возвращает значение `True`:

```
>>> x = y = [1, 2]          # Один объект
>>> x is y
True
>>> x = [1, 2]              # Разные объекты
>>> y = [1, 2]              # Разные объекты
>>> x is y
False
```

Следует заметить, что в целях повышения эффективности кода интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что если ста переменным присвоено число 2, то, скорее всего, в этих переменных будет сохранена ссылка на один и тот же объект. Пример:

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
(True, True)
```

Посмотреть количество ссылок на объект позволяет метод `getrefcount()` из модуля `sys`:

```
>>> import sys               # Подключаем модуль sys
>>> sys.getrefcount(2)
304
```

Когда число ссылок на объект становится равно нулю, объект автоматически удаляется из оперативной памяти. Исключением являются объекты, которые подлежат кэшированию.

Помимо группового, Python поддерживает *позиционное присваивание*. В этом случае переменные записываются через запятую слева от оператора `=`, а значения — через запятую справа. Пример позиционного присваивания:

```
>>> x, y, z = 1, 2, 3
>>> x, y, z
(1, 2, 3)
```

С помощью позиционного присваивания можно поменять значения переменных местами. Пример:

```
>>> x, y = 1, 2
>>> x, y
(1, 2)
>>> x, y = y, x
>>> x, y
(2, 1)
```

По обе стороны оператора `=` могут быть указаны последовательности, к каковым относятся строки, списки, кортежи, диапазоны, типы `bytes` и `bytearray`:

```
>>> x, y, z = "123"              # Строка
>>> x, y, z
('1', '2', '3')
>>> x, y, z = [1, 2, 3]          # Список
>>> x, y, z
(1, 2, 3)
>>> x, y, z = (1, 2, 3)          # Кортеж
>>> x, y, z
(1, 2, 3)
>>> [x, y, z] = (1, 2, 3)        # Список слева, кортеж справа
>>> x, y, z
(1, 2, 3)
```

Обратите внимание на то, что количество элементов справа и слева от оператора `=` должно совпадать, иначе будет выведено сообщение об ошибке:

```
>>> x, y, z = (1, 2, 3, 4)
Traceback (most recent call last):
  File "<pyshell#130>", line 1, in <module>
    x, y, z = (1, 2, 3, 4)
ValueError: too many values to unpack (expected 3)
```

Python 3 при несоответствии количества элементов справа и слева от оператора = позволяет сохранить в переменной список, состоящий из лишних элементов. Для этого перед именем переменной указывается звездочка (*):

```
>>> x, y, *z = (1, 2, 3, 4)
>>> x, y, z
(1, 2, [3, 4])
>>> x, *y, z = (1, 2, 3, 4)
>>> x, y, z
(1, [2, 3], 4)
>>> *x, y, z = (1, 2, 3, 4)
>>> x, y, z
([1, 2], 3, 4)
>>> x, y, *z = (1, 2, 3)
>>> x, y, z
(1, 2, [3])
>>> x, y, *z = (1, 2)
>>> x, y, z
(1, 2, [])
```

Как видно из примера, переменная, перед которой указана звездочка, всегда получает в качестве значения список. Если для этой переменной не хватило значений, то ей присваивается пустой список. Следует помнить, что звездочку можно указать только перед одной переменной, в противном случае возникнет неоднозначность и интерпретатор выведет сообщение об ошибке:

```
>>> *x, y, *z = (1, 2, 3, 4)
SyntaxError: two starred expressions in assignment
```

2.4. Проверка типа данных

Во многих языках программирования при создании переменной нужно указывать тип данных, к которому должны относиться значения, присваиваемые этой переменной. Но в Python этого делать не нужно, поскольку любая переменная может хранить значения любого типа.

Выяснить тип данных, к которому относится хранящееся в переменной значение, позволяет функция `type(<Имя переменной>)`:

```
>>> type(a)
<class 'int'>
```

Проверить тип данных у значения, хранящегося в переменной, можно следующими способами:

- сравнить значение, возвращаемое функцией `type()`, с названием типа данных:

  ```
  >>> x = 10
  >>> if type(x) == int:
          print("Это целое число (тип int)")
  ```

- проверить тип с помощью функции `isinstance()`:

  ```
  >>> s = "Строка"
  >>> if isinstance(s, str):
          print("Это строка (тип str)")
  ```

2.5. Преобразование типов данных

Над значением, относящимся к определенному типу, можно производить лишь операции, допустимые для этого типа данных. Например, можно складывать друг с другом числа, но строку сложить с числом нельзя — это приведет к выводу сообщения об ошибке:

```
>>> 2 + "25"
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    2 + "25"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Для преобразования значения из одного типа данных в другой предназначены следующие функции:

- `bool([<Объект>])` — преобразует объект в логический тип данных:

  ```
  >>> bool(0), bool(1), bool(""), bool("Строка"), bool([1, 2]), bool([])
  (False, True, False, True, True, False)
  ```

- `int([<Объект>[, <Система счисления>]])` — преобразует объект в число. Во втором параметре можно указать систему счисления (значение по умолчанию — 10). Примеры:

  ```
  >>> int(7.5), int("71")
  (7, 71)
  >>> int("71", 10), int("71", 8), int("0o71", 8), int("A", 16)
  (71, 57, 57, 10)
  ```

  Если преобразование невозможно, то генерируется исключение:

  ```
  >>> int("71s")
  Traceback (most recent call last):
    File "<pyshell#9>", line 1, in <module>
      int("71s")
  ValueError: invalid literal for int() with base 10: '71s'
  ```

- `float([<Число или строка>])` — преобразует целое число или строку в вещественное число:

  ```
  >>> float(7), float("7.1")
  (7.0, 7.1)
  >>> float("Infinity"), float("-inf")
  (inf, -inf)
  >>> float("Infinity") + float("-inf")
  nan
  ```

- `str([<Объект>])` — преобразует объект в строку:

  ```
  >>> str(125), str([1, 2, 3])
  ('125', '[1, 2, 3]')
  >>> str((1, 2, 3)), str({"x": 5, "y": 10})
  ('(1, 2, 3)', "{'y': 10, 'x': 5}")
  >>> str(bytes("строка", "utf-8"))
  "b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0\\xba\\xd0
  \\xb0'"
  >>> str(bytearray("строка", "utf-8"))
  ```

```
"bytearray(b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0
\\xba\\xd0\\xb0')"
```

- `str(<Объект>[, <Кодировка>[, <Обработка ошибок>]])` — преобразует объект типа `bytes` или `bytearray` в строку. В третьем параметре можно задать значение `"strict"` (при ошибке возбуждается исключение `UnicodeDecodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом, имеющим код `\uFFFD`) или `"ignore"` (неизвестные символы игнорируются). Примеры:

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    str(obj1, "ascii", "strict")
UnicodeDecodeError: 'ascii' codec can't decode byte
0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
'1'
```

- `bytes(<Строка>, <Кодировка>[, <Обработка ошибок>])` — преобразует строку в объект типа `bytes`. В третьем параметре могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`. Примеры:

```
>>> bytes("строка", "cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка123", "ascii", "ignore")
b'123'
```

- `bytes(<Последовательность>)` — преобразует последовательность целых чисел от 0 до 255 в объект типа `bytes`. Если число не попадает в диапазон, возбуждается исключение `ValueError`:

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

- `bytearray(<Строка>, <Кодировка>[, <Обработка ошибок>])` — преобразует строку в объект типа `bytearray`. В третьем параметре могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`:

```
>>> bytearray("строка", "cp1251")
bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
```

- `bytearray(<Последовательность>)` — преобразует последовательность целых чисел от 0 до 255 в объект типа `bytearray`. Если число не попадает в диапазон, возбуждается исключение `ValueError`:

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
```

```
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

- `list(<Последовательность>)` — преобразует элементы последовательности в список:

```
>>> list("12345")              # Преобразование строки
['1', '2', '3', '4', '5']
>>> list((1, 2, 3, 4, 5))      # Преобразование кортежа
[1, 2, 3, 4, 5]
```

- `tuple(<Последовательность>)` — преобразует элементы последовательности в кортеж:

```
>>> tuple("123456")            # Преобразование строки
('1', '2', '3', '4', '5', '6')
>>> tuple([1, 2, 3, 4, 5])     # Преобразование списка
(1, 2, 3, 4, 5)
```

В качестве примера рассмотрим возможность сложения двух чисел, введенных пользователем. Как вы уже знаете, вводить данные позволяет функция `input()`. Воспользуемся этой функцией для получения чисел от пользователя (листинг 2.1).

Листинг 2.1. Получение данных от пользователя

```
# -*- coding: utf-8 -*-
x = input("x = ")            # Вводим 5
y = input("y = ")            # Вводим 12
print(x + y)
input()
```

Результатом выполнения этого скрипта будет не число, а строка `512`. Таким образом, следует запомнить, что функция `input()` возвращает результат в виде строки. Чтобы просуммировать два числа, необходимо преобразовать строку в число (листинг 2.2).

Листинг 2.2. Преобразование строки в число

```
# -*- coding: utf-8 -*-
x = int(input("x = "))       # Вводим 5
y = int(input("y = "))       # Вводим 12
print(x + y)
input()
```

В этом случае мы получим число 17, как и должно быть. Однако если пользователь вместо числа введет строку, то программа завершится с фатальной ошибкой. Как обработать ошибку, мы разберемся по мере изучения языка.

2.6. Удаление переменных

Удалить переменную можно с помощью инструкции `del`:

```
del <Переменная1>[, ..., <ПеременнаяN>]
```

Пример удаления одной переменной:

```
>>> x = 10; x
10
>>> del x; x
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    del x; x
NameError: name 'x' is not defined
```

Пример удаления нескольких переменных:

```
>>> x, y = 10, 20
>>> del x, y
```

ГЛАВА 3

Операторы

Операторы позволяют произвести с данными определенные действия. Например, операторы присваивания служат для сохранения данных в переменной, математические операторы позволяют выполнить арифметические вычисления, а оператор конкатенации строк служит для соединения двух строк в одну. Рассмотрим операторы, доступные в Python 3, подробно.

3.1. Математические операторы

Производить операции над числами позволяют математические операторы:

- `+` — сложение:

```
>>> 10 + 5              # Целые числа
15
>>> 12.4 + 5.2          # Вещественные числа
17.6
>>> 10 + 12.4           # Целые и вещественные числа
22.4
```

- `-` — вычитание:

```
>>> 10 - 5              # Целые числа
5
>>> 12.4 - 5.2          # Вещественные числа
7.2
>>> 12 - 5.2            # Целые и вещественные числа
6.8
```

- `*` — умножение:

```
>>> 10 * 5              # Целые числа
50
>>> 12.4 * 5.2          # Вещественные числа
64.48
>>> 10 * 5.2            # Целые и вещественные числа
52.0
```

- `/` — деление. Результатом деления всегда является вещественное число, даже если производится деление целых чисел. Обратите внимание на эту особенность, если вы раньше

программировали на Python 2. В Python 2 при делении целых чисел остаток отбрасывался и возвращалось целое число, в Python 3 поведение оператора изменилось:

```
>>> 10 / 5              # Деление целых чисел без остатка
2.0
>>> 10 / 3              # Деление целых чисел с остатком
3.3333333333333335
>>> 10.0 / 5.0          # Деление вещественных чисел
2.0
>>> 10.0 / 3.0          # Деление вещественных чисел
3.3333333333333335
>>> 10 / 5.0            # Деление целого числа на вещественное
2.0
>>> 10.0 / 5            # Деление вещественного числа на целое
2.0
```

- `//` — деление с округлением вниз. Вне зависимости от типа чисел остаток отбрасывается:

```
>>> 10 // 5           # Деление целых чисел без остатка
2
>>> 10 // 3           # Деление целых чисел с остатком
3
>>> 10.0 // 5.0       # Деление вещественных чисел
2.0
>>> 10.0 // 3.0       # Деление вещественных чисел
3.0
>>> 10 // 5.0         # Деление целого числа на вещественное
2.0
>>> 10 // 3.0         # Деление целого числа на вещественное
3.0
>>> 10.0 // 5         # Деление вещественного числа на целое
2.0
>>> 10.0 // 3         # Деление вещественного числа на целое
3.0
```

- `%` — остаток от деления:

```
>>> 10 % 5            # Деление целых чисел без остатка
0
>>> 10 % 3            # Деление целых чисел с остатком
1
>>> 10.0 % 5.0        # Операция над вещественными числами
0.0
>>> 10.0 % 3.0        # Операция над вещественными числами
1.0
>>> 10 % 5.0  # Операция над целыми и вещественными числами
0.0
>>> 10 % 3.0  # Операция над целыми и вещественными числами
1.0
>>> 10.0 % 5  # Операция над целыми и вещественными числами
0.0
>>> 10.0 % 3  # Операция над целыми и вещественными числами
1.0
```

♦ `**` — возведение в степень:

```
>>> 10 ** 2, 10.0 ** 2
(100, 100.0)
```

♦ унарный минус (–) и унарный плюс (+):

```
>>> +10, +10.0, -10, -10.0, -(-10), -(-10.0)
(10, 10.0, -10, -10.0, 10, 10.0)
```

Как видно из примеров, операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое число и вещественное, то целое число будет автоматически преобразовано в вещественное число, затем будет произведена операция над вещественными числами, а результатом станет вещественное число.

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 - 0.1 - 0.1 - 0.1
-2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат. Если необходимо производить операции с фиксированной точностью, следует использовать модуль `decimal`:

```
>>> from decimal import Decimal
>>> Decimal("0.3") - Decimal("0.1") - Decimal("0.1") - Decimal("0.1")
Decimal('0.0')
```

3.2. Двоичные операторы

Двоичные операторы предназначены для манипуляции отдельными битами. Язык Python поддерживает следующие двоичные операторы:

♦ `~` — двоичная инверсия. Значение каждого бита заменяется на противоположное:

```
>>> x = 100                    # 01100100
>>> x = ~x                     # 10011011
```

♦ `&` — двоичное И:

```
>>> x = 100                    # 01100100
>>> y = 75                     # 01001011
>>> z = x & y                  # 01000000
>>> "{0:b} & {1:b} = {2:b}".format(x, y, z)
'1100100 & 1001011 = 1000000'
```

♦ `|` — двоичное ИЛИ:

```
>>> x = 100                    # 01100100
>>> y = 75                     # 01001011
>>> z = x | y                  # 01101111
>>> "{0:b} | {1:b} = {2:b}".format(x, y, z)
'1100100 | 1001011 = 1101111'
```

♦ `^` — двоичное исключающее ИЛИ:

```
>>> x = 100                    # 01100100
>>> y = 250                    # 11111010
>>> z = x ^ y                  # 10011110
>>> "{0:b} ^ {1:b} = {2:b}".format(x, y, z)
'1100100 ^ 11111010 = 10011110'
```

♦ `<<` — сдвиг влево — сдвигает двоичное представление числа влево на один или более разрядов и заполняет разряды справа нулями:

```
>>> x = 100                    # 01100100
>>> y = x << 1                 # 11001000
>>> z = y << 1                 # 10010000
>>> k = z << 2                 # 01000000
```

♦ `>>` — сдвиг вправо — сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, если число положительное:

```
>>> x = 100                    # 01100100
>>> y = x >> 1                 # 00110010
>>> z = y >> 1                 # 00011001
>>> k = z >> 2                 # 00000110
```

Если число отрицательное, то разряды слева заполняются единицами:

```
>>> x = -127                   # 10000001
>>> y = x >> 1                 # 11000000
>>> z = y >> 2                 # 11110000
>>> k = z << 1                 # 11100000
>>> m = k >> 1                 # 11110000
```

3.3. Операторы для работы с последовательностями

Для работы с последовательностями предназначены следующие операторы:

♦ `+` — конкатенация:

```
>>> print("Строка1" + "Строка2")   # Конкатенация строк
Строка1Строка2
>>> [1, 2, 3] + [4, 5, 6]          # Списки
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)          # Кортежи
(1, 2, 3, 4, 5, 6)
```

♦ `*` — повторение:

```
>>> "s" * 20                       # Строки
'ssssssssssssssssssss'
>>> [1, 2] * 3                     # Списки
[1, 2, 1, 2, 1, 2]
>>> (1, 2) * 3                     # Кортежи
(1, 2, 1, 2, 1, 2)
```

♦ `in` — проверка на вхождение. Если элемент входит в последовательность, то возвращается логическое значение `True`:

```
>>> "Строка" in "Строка для поиска"   # Строки
True
>>> "Строка2" in "Строка для поиска"  # Строки
False
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]    # Списки
(True, False)
>>> 2 in (1, 2, 3), 6 in (1, 2, 3)    # Кортежи
(True, False)
```

♦ `not in` — проверка на невхождение. Если элемент не входит в последовательность, возвращается `True`:

```
>>> "Строка" not in "Строка для поиска"   # Строки
False
>>> "Строка2" not in "Строка для поиска"  # Строки
True
>>> 2 not in [1, 2, 3], 4 not in [1, 2, 3]    # Списки
(False, True)
>>> 2 not in (1, 2, 3), 6 not in (1, 2, 3)    # Кортежи
(False, True)
```

3.4. Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной. Приведем перечень операторов присваивания, доступных в языке Python:

♦ `=` — присваивает переменной значение:

```
>>> x = 5; x
5
```

♦ `+=` — увеличивает значение переменной на указанную величину:

```
>>> x = 5; x += 10            # Эквивалентно x = x + 10
>>> x
15
```

Для последовательностей оператор `+=` производит конкатенацию:

```
>>> s = "Стр"; s += "ока"
>>> print(s)
Строка
```

♦ `-=` — уменьшает значение переменной на указанную величину:

```
>>> x = 10; x -= 5            # Эквивалентно x = x — 5
>>> x
5
```

♦ `*=` — умножает значение переменной на указанную величину:

```
>>> x = 10; x *= 5            # Эквивалентно x = x * 5
>>> x
50
```

Для последовательностей оператор `*=` производит повторение:

```
>>> s = "*"; s *= 20
>>> s
'********************'
```

- `/=` — делит значение переменной на указанную величину:

```
>>> x = 10; x /= 3            # Эквивалентно x = x / 3
>>> x
3.3333333333333335
>>> y = 10.0; y /= 3.0        # Эквивалентно y = y / 3.0
>>> y
3.3333333333333335
```

- `//=` — деление с округлением вниз и присваиванием:

```
>>> x = 10; x //= 3           # Эквивалентно x = x // 3
>>> x
3
>>> y = 10.0; y //= 3.0       # Эквивалентно y = y // 3.0
>>> y
3.0
```

- `%=` — деление по модулю и присваивание:

```
>>> x = 10; x %= 2            # Эквивалентно x = x % 2
>>> x
0
>>> y = 10; y %= 3            # Эквивалентно y = y % 3
>>> y
1
```

- `**=` — возведение в степень и присваивание:

```
>>> x = 10; x **= 2           # Эквивалентно x = x ** 2
>>> x
100
```

3.5. Приоритет выполнения операторов

В какой последовательности будет вычисляться приведенное далее выражение?

```
x = 5 + 10 * 3 / 2
```

Это зависит от приоритета выполнения операторов. В данном случае последовательность вычисления выражения будет такой:

1. Число 10 будет умножено на 3, т. к. приоритет оператора умножения выше приоритета оператора сложения.
2. Полученное значение будет поделено на 2, т. к. приоритет оператора деления равен приоритету оператора умножения (а операторы с равными приоритетами выполняются слева направо), но выше, чем у оператора сложения.

3. К полученному значению будет прибавлено число 5, т. к. оператор присваивания = имеет наименьший приоритет.
4. Значение будет присвоено переменной x.

```
>>> x = 5 + 10 * 3 / 2
>>> x
20.0
```

С помощью скобок можно изменить последовательность вычисления выражения:

```
x = (5 + 10) * 3 / 2
```

Теперь порядок вычислений станет иным:

1. К числу 5 будет прибавлено 10.
2. Полученное значение будет умножено на 3.
3. Полученное значение будет поделено на 2.
4. Значение будет присвоено переменной x.

```
>>> x = (5 + 10) * 3 / 2
>>> x
22.5
```

Перечислим операторы в порядке убывания приоритета:

1. -x, +x, ~x, ** — унарный минус, унарный плюс, двоичная инверсия, возведение в степень. Если унарные операторы расположены слева от оператора **, то возведение в степень имеет больший приоритет, а если справа — то меньший. Например, выражение:

   ```
   -10 ** -2
   ```

   эквивалентно следующей расстановке скобок:

   ```
   -(10 ** (-2))
   ```

2. *, %, /, // — умножение (повторение), остаток от деления, деление, деление с округлением вниз.
3. +, - — сложение (конкатенация), вычитание.
4. <<, >> — двоичные сдвиги.
5. & — двоичное И.
6. ^ — двоичное исключающее ИЛИ.
7. | — двоичное ИЛИ.
8. =, +=, -=, *=, /=, //=, %=, **= — присваивание.

ГЛАВА 4

Условные операторы и циклы

Условные операторы позволяют в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнить его. Логические выражения возвращают только два значения: `True` (истина) или `False` (ложь), которые ведут себя как целые числа `1` и `0` соответственно:

```
>>> True + 2                    # Эквивалентно 1 + 2
3
>>> False + 2                   # Эквивалентно 0 + 2
2
```

Логическое значение можно сохранить в переменной:

```
>>> x = True; y = False
>>> x, y
(True, False)
```

Любой объект в логическом контексте может интерпретироваться как истина (`True`) или как ложь (`False`). Для определения логического значения можно использовать функцию `bool()`.

Значение `True` возвращает следующие объекты:

- любое число, не равное нулю:

  ```
  >>> bool(1), bool(20), bool(-20)
  (True, True, True)
  >>> bool(1.0), bool(0.1), bool(-20.0)
  (True, True, True)
  ```

- не пустой объект:

  ```
  >>> bool("0"), bool([0, None]), bool((None,)), bool({"x": 5})
  (True, True, True, True)
  ```

Следующие объекты интерпретируются как `False`:

- число, равное нулю:

  ```
  >>> bool(0), bool(0.0)
  (False, False)
  ```

- пустой объект:

  ```
  >>> bool(""), bool([]), bool(())
  (False, False, False)
  ```

- значение `None`:

```
>>> bool(None)
False
```

4.1. Операторы сравнения

Операторы сравнения используются в логических выражениях. Приведем их перечень:

- `==` — равно:

```
>>> 1 == 1, 1 == 5
(True, False)
```

- `!=` — не равно:

```
>>> 1 != 5, 1 != 1
(True, False)
```

- `<` — меньше:

```
>>> 1 < 5, 1 < 0
(True, False)
```

- `>` — больше:

```
>>> 1 > 0, 1 > 5
(True, False)
```

- `<=` — меньше или равно:

```
>>> 1 <= 5, 1 <= 0, 1 <= 1
(True, False, True)
```

- `>=` — больше или равно:

```
>>> 1 >= 0, 1 >= 5, 1 >= 1
(True, False, True)
```

- `in` — проверка на вхождение в последовательность:

```
>>> "Строка" in "Строка для поиска" # Строки
True
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]  # Списки
(True, False)
>>> 2 in (1, 2, 3), 4 in (1, 2, 3)  # Кортежи
(True, False)
```

  Оператор `in` можно также использовать для проверки существования ключа словаря:

```
>>> "x" in {"x": 1, "y": 2}, "z" in {"x": 1, "y": 2}
(True, False)
```

- `not in` — проверка на невхождение в последовательность:

```
>>> "Строка" not in "Строка для поиска" # Строки
False
>>> 2 not in [1, 2, 3], 4 not in [1, 2, 3]  # Списки
(False, True)
```

```
>>> 2 not in (1, 2, 3), 4 not in (1, 2, 3)  # Кортежи
(False, True)
```

♦ `is` — проверяет, ссылаются ли две переменные на один и тот же объект. Если переменные ссылаются на один и тот же объект, оператор `is` возвращает значение `True`:

```
>>> x = y = [1, 2]
>>> x is y
True
>>> x = [1, 2]; y = [1, 2]
>>> x is y
False
```

Следует заметить, что в целях повышения эффективности интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что если ста переменным присвоено число 2, то в этих переменных, скорее всего, будет сохранена ссылка на один и тот же объект:

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
(True, True)
```

♦ `is not` — проверяет, ссылаются ли две переменные на разные объекты. Если это так, возвращается значение `True`:

```
>>> x = y = [1, 2]
>>> x is not y
False
>>> x = [1, 2]; y = [1, 2]
>>> x is not y
True
```

Значение логического выражения можно инвертировать с помощью оператора `not`:

```
>>> x = 1; y = 1
>>> x == y
True
>>> not (x == y), not x == y
(False, False)
```

Если переменные `x` и `y` равны, возвращается значение `True`, но так как перед выражением стоит оператор `not`, выражение вернет `False`. Круглые скобки можно не указывать, поскольку оператор `not` имеет более низкий приоритет выполнения, чем операторы сравнения.

В логическом выражении можно указывать сразу несколько условий:

```
>>> x = 10
>>> 1 < x < 20, 11 < x < 20
(True, False)
```

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

♦ `and` — логическое И. Если `x` в выражении `x and y` интерпретируется как `False`, то возвращается `x`, в противном случае — `y`. Примеры:

```
>>> 1 < 5 and 2 < 5                # True and True == True
True
```

```
>>> 1 < 5 and 2 > 5              # True and False == False
False
>>> 1 > 5 and 2 < 5              # False and True == False
False
>>> 10 and 20, 0 and 20, 10 and 0
(20, 0, 0)
```

♦ `or` — логическое ИЛИ. Если `x` в выражении `x or y` интерпретируется как `False`, то возвращается `y`, в противном случае — `x`. Примеры:

```
>>> 1 < 5 or 2 < 5              # True or True == True
True
>>> 1 < 5 or 2 > 5              # True or False == True
True
>>> 1 > 5 or 2 < 5              # False or True == True
True
>>> 1 > 5 or 2 > 5              # False or False == False
False
>>> 10 or 20, 0 or 20, 10 or 0
(10, 20, 10)
>>> 0 or "" or None or [] or "s"
's'
```

Следующее выражение вернет `True` только в том случае, если оба выражения вернут `True`:

```
x1 == x2 and x2 != x3
```

А это выражение вернет `True`, если хотя бы одно из выражений вернет `True`:

```
x1 == x2 or x3 == x4
```

Перечислим операторы сравнения в порядке убывания приоритета:

1. <, >, <=, >=, ==, `!=`, <>, `is`, `is not`, `in`, `not in`.
2. `not` — логическое отрицание.
3. `and` — логическое И.
4. `or` — логическое ИЛИ.

4.2. Оператор ветвления *if...else*

Оператор ветвления `if...else` позволяет в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнить его. Оператор имеет следующий формат:

```
if <Логическое выражение>:
    <Блок, выполняемый, если условие истинно>
[elif <Логическое выражение>:
    <Блок, выполняемый, если условие истинно>
]
[else:
    <Блок, выполняемый, если все условия ложны>
]
```

Как вы уже знаете, блоки внутри составной инструкции выделяются одинаковым количеством пробелов (обычно четырьмя). Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В некоторых языках программирования логическое выражение заключается в круглые скобки. В языке Python это делать необязательно, но можно, т. к. любое выражение может быть расположено внутри круглых скобок. Тем не менее, круглые скобки следует использовать только при необходимости разместить условие на нескольких строках.

Для примера напишем программу, которая проверяет, является введенное пользователем число четным или нет (листинг 4.1). После проверки выводится соответствующее сообщение.

Листинг 4.1. Проверка числа на четность

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0:
    print(x, " - четное число")
else:
    print(x, " - нечетное число")
input()
```

Если блок состоит из одной инструкции, эту инструкцию можно разместить на одной строке с заголовком:

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0: print(x, " - четное число")
else: print(x, " - нечетное число")
input()
```

В этом случае концом блока является конец строки. Это означает, что можно разместить сразу несколько инструкций на одной строке, разделяя их точкой с запятой:

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0: print(x, end=" "); print("- четное число")
else: print(x, end=" "); print("- нечетное число")
input()
```

> ***Совет***
>
> Знайте, что так сделать можно, но никогда на практике не пользуйтесь этим способом, поскольку подобная конструкция нарушает стройность кода и ухудшает его сопровождение в дальнейшем. Всегда размещайте инструкцию на отдельной строке, даже если блок содержит только одну инструкцию.

Согласитесь, что следующий код читается намного проще, чем предыдущий:

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0:
    print(x, end=" ")
    print("- четное число")
```

```
else:
    print(x, end=" ")
    print("- нечетное число")
input()
```

Оператор ветвления `if...else` позволяет проверить сразу несколько условий. Рассмотрим это на примере (листинг 4.2).

Листинг 4.2. Проверка нескольких условий

```
# -*- coding: utf-8 -*-
print("""Какой операционной системой вы пользуетесь?
1 — Windows 10
2 — Windows 8.1
3 — Windows 8
4 — Windows 7
5 — Windows Vista
6 — Другая""")
os = input("Введите число, соответствующее ответу: ")
if   os == "1":
    print("Вы выбрали: Windows 10")
elif os == "2":
    print("Вы выбрали: Windows 8.1")
elif os == "3":
    print("Вы выбрали: Windows 8")
elif os == "4":
    print("Вы выбрали: Windows 7")
elif os == "5":
    print("Вы выбрали: Windows Vista")
elif os == "6":
    print("Вы выбрали: другая")
elif not os:
    print("Вы не ввели число")
else:
    print("Мы не смогли определить вашу операционную систему")
input()
```

С помощью инструкции `elif` мы можем определить выбранное значение и вывести соответствующее сообщение. Обратите внимание на то, что логическое выражение не содержит операторов сравнения:

```
elif not os:
```

Такая запись эквивалентна следующей:

```
elif os == "":
```

Проверка на равенство выражения значению `True` выполняется по умолчанию. Поскольку пустая строка интерпретируется как `False`, мы инвертируем возвращаемое значение с помощью оператора `not`.

Один условный оператор можно вложить в другой. В этом случае отступ вложенной инструкции должен быть в два раза больше (листинг 4.3).

Листинг 4.3. Вложенные инструкции

```
# -*- coding: utf-8 -*-
print("""Какой операционной системой вы пользуетесь?
1 — Windows 10
2 — Windows 8.1
3 — Windows 8
4 — Windows 7
5 — Windows Vista
6 — Другая""")
os = input("Введите число, соответствующее ответу: ")
if os != "":
    if   os == "1":
        print("Вы выбрали: Windows 10")
    elif os == "2":
        print("Вы выбрали: Windows 8.1")
    elif os == "3":
        print("Вы выбрали: Windows 8")
    elif os == "4":
        print("Вы выбрали: Windows 7")
    elif os == "5":
        print("Вы выбрали: Windows Vista")
    elif os == "6":
        print("Вы выбрали: другая")
    else:
        print("Мы не смогли определить вашу операционную систему")
else:
    print("Вы не ввели число")
input()
```

Оператор ветвления `if...else` имеет еще один формат:

```
<Переменная> = <Если истина> if <Условие> else <Если ложь>
```

Пример:

```
>>> print("Yes" if 10 % 2 == 0 else "No")
Yes
>>> s = "Yes" if 10 % 2 == 0 else "No"
>>> s
'Yes'
>>> s = "Yes" if 11 % 2 == 0 else "No"
>>> s
'No'
```

4.3. Цикл *for*

Предположим, нужно вывести все числа от 1 до 100 по одному на строке. Обычным способом пришлось бы писать 100 строк кода:

```
print(1)
print(2)
...
print(100)
```

При помощи *цикла* то же действие можно выполнить одной строкой:

```
for x in range(1, 101): print(x)
```

Иными словами, циклы позволяют выполнить одни и те же инструкции многократно.

Цикл `for` применяется для перебора элементов последовательности и имеет такой формат:

```
for <Текущий элемент> in <Последовательность>:
    <Инструкции внутри цикла>
[else:
    <Блок, выполняемый, если не использовался оператор break>
]
```

Здесь присутствуют следующие конструкции:

- `<Последовательность>` — объект, поддерживающий механизм итерации: строка, список, кортеж, диапазон, словарь и др.;
- `<Текущий элемент>` — на каждой итерации через эту переменную доступен очередной элемент последовательности или ключ словаря;
- `<Инструкции внутри цикла>` — блок, который будет многократно выполняться;
- если внутри цикла не использовался оператор `break`, то после завершения выполнения цикла будет выполнен блок в инструкции `else`. Этот блок не является обязательным.

Пример перебора букв в слове приведен в листинге 4.4.

Листинг 4.4. Перебор букв в слове

```
for s in "str":
    print(s, end=" ")
else:
    print("\nЦикл выполнен")
```

Результат выполнения:

```
s t r
Цикл выполнен
```

Теперь выведем каждый элемент списка и кортежа на отдельной строке (листинг 4.5).

Листинг 4.5. Перебор списка и кортежа

```
for x in [1, 2, 3]:
    print(x)
for y in (1, 2, 3):
    print(y)
```

Цикл `for` позволяет также перебрать элементы словарей, хотя словари и не являются последовательностями. В качестве примера выведем элементы словаря двумя способами. Первый способ использует метод `keys()`, возвращающий объект `dict_keys`, который содержит все ключи словаря:

```
>>> arr = {"x": 1, "y": 2, "z": 3}
>>> arr.keys()
```

```
dict_keys(['y', 'x', 'z'])
>>> for key in arr.keys():
        print(key, arr[key])

y 2
x 1
z 3
```

Во втором способе мы просто указываем словарь в качестве параметра — на каждой итерации цикла будет возвращаться ключ, с помощью которого внутри цикла можно получить значение, соответствующее этому ключу:

```
>>> for key in arr:
        print(key, arr[key])

y 2
x 1
z 3
```

Обратите внимание на то, что элементы словаря выводятся в произвольном порядке, а не в порядке, в котором они были указаны при создании объекта. Чтобы вывести элементы в алфавитном порядке, следует отсортировать ключи с помощью функции `sorted()`:

```
>>> arr = {"x": 1, "y": 2, "z": 3}
>>> for key in sorted(arr):
        print(key, arr[key])

x 1
y 2
z 3
```

С помощью цикла `for` можно перебирать сложные структуры данных. В качестве примера выведем элементы списка кортежей:

```
>>> arr = [(1, 2), (3, 4)]          # Список кортежей
>>> for a, b in arr:
        print(a, b)

1 2
3 4
```

4.4. Функции *range()* и *enumerate()*

До сих пор мы только выводили элементы последовательностей. Теперь попробуем умножить каждый элемент списка на 2:

```
>>> arr = [1, 2, 3]
>>> for i in arr:
        i = i * 2

>>> print(arr)
[1, 2, 3]
```

Как видно из примера, список не изменился. Переменная `i` на каждой итерации цикла содержит лишь копию значения текущего элемента списка, поэтому изменить таким способом элементы списка нельзя. Чтобы получить доступ к каждому элементу, можно, например, воспользоваться функцией `range()` для генерации индексов. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если параметр `<Начало>` не указан, то по умолчанию используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемые значения. Если параметр `<Шаг>` не указан, то используется значение `1`. Функция возвращает *диапазон* — особый объект, поддерживающий итерационный протокол. С помощью диапазона внутри цикла `for` можно получить значение текущего элемента. В качестве примера умножим каждый элемент списка на 2:

```
>>> arr = [1, 2, 3]
>>>for i in range(len(arr)):
       arr[i] *= 2


>>> print(arr)
[2, 4, 6]
```

В этом примере мы получаем количество элементов списка с помощью функции `len()` и передаем результат в функцию `range()`. В итоге последняя вернет диапазон значений от `0` до `len(arr) - 1`. На каждой итерации цикла через переменную `i` доступен текущий элемент из диапазона индексов. Чтобы получить доступ к элементу списка, указываем индекс внутри квадратных скобок. Умножаем каждый элемент списка на 2, а затем выводим результат с помощью функции `print()`.

Рассмотрим несколько примеров использования функции `range()`:

- Выведем числа от 1 до 100:

  ```
  for i in range(1, 101): print(i)
  ```

- Можно не только увеличивать значение, но и уменьшать его. Выведем все числа от 100 до 1:

  ```
  for i in range(100, 0, -1): print(i)
  ```

- Можно также изменять значение не только на единицу. Выведем все четные числа от 1 до 100:

  ```
  for i in range(2, 101, 2): print(i)
  ```

В Python 2 функция `range()` возвращала список чисел. В Python 3 поведение функции изменилось — теперь она возвращает диапазон. Чтобы получить список чисел, следует передать диапазон, возвращенный функцией `range()`, в функцию `list()`:

```
>>> obj = range(len([1, 2, 3]))
>>> obj
range(0, 3)
>>> obj[0], obj[1], obj[2]      # Доступ по индексу
(0, 1, 2)
>>> obj[0:2]                    # Получение среза
range(0, 2)
```

```
>>> i = iter(obj)
>>> next(i), next(i), next(i)   # Доступ с помощью итераторов
(0, 1, 2)
>>> list(obj)                   # Преобразование диапазона в список
[0, 1, 2]
>>> 1 in obj, 7 in obj          # Проверка вхождения значения
(True, False)
```

Диапазон поддерживает два полезных метода:

- `index(<Значение>)` — возвращает индекс элемента, имеющего указанное значение. Если значение не входит в диапазон, возбуждается исключение `ValueError`:

```
>>> obj = range(1, 5)
>>> obj.index(1), obj.index(4)
(0, 3)
>>> obj.index(5)
... Фрагмент опущен ...
ValueError: 5 is not in range
```

- `count(<Значение>)` — возвращает количество элементов с указанным значением. Если элемент не входит в диапазон, возвращается значение `0`:

```
>>> obj = range(1, 5)
>>> obj.count(1), obj.count(10)
(1, 0)
```

Функция `enumerate(<Объект>[, start=0])` на каждой итерации цикла `for` возвращает кортеж из индекса и значения текущего элемента. С помощью необязательного параметра `start` можно задать начальное значение индекса. В качестве примера умножим на 2 каждый элемент списка, который содержит четное число (листинг 4.6).

Листинг 4.6. Пример использования функции `enumerate()`

```
arr = [1, 2, 3, 4, 5, 6]
for i, elem in enumerate(arr):
    if elem % 2 == 0:
        arr[i] *= 2
print(arr)                 # Результат выполнения: [1, 4, 3, 8, 5, 12]
```

Функция `enumerate()` не создает список, а возвращает итератор. С помощью функции `next()` можно обойти всю последовательность. Когда перебор будет закончен, возбуждается исключение `StopIteration`:

```
>>> arr = [1, 2]
>>> obj = enumerate(arr, start=2)
>>> next(obj)
(2, 1)
>>> next(obj)
(3, 2)
>>> next(obj)
```

```
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(obj)
StopIteration
```

Кстати, цикл `for` при работе активно использует итераторы, но делает это незаметно для нас.

4.5. Цикл *while*

Выполнение инструкций в цикле `while` продолжается до тех пор, пока логическое выражение истинно. Цикл `while` имеет следующий формат:

```
<Задание начального значения для переменной-счетчика>
while <Условие>:
    <Инструкции>
    <Приращение значения в переменной-счетчике>
[else:
    <Блок, выполняемый, если не использовался оператор break>
]
```

Последовательность работы цикла `while`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие, и если оно истинно, то выполняются инструкции внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре `<Приращение>`.
4. Переход к пункту 2.
5. Если внутри цикла не использовался оператор `break`, то после завершения выполнения цикла будет выполнен блок в инструкции `else`. Этот блок не является обязательным.

Выведем все числа от 1 до 100, используя цикл `while` (листинг 4.7).

Листинг 4.7. Вывод чисел от 1 до 100

```
i = 1                    # <Начальное значение>
while i < 101:           # <Условие>
    print(i)             # <Инструкции>
    i += 1               # <Приращение>
```

> **ВНИМАНИЕ!**
>
> Если `<Приращение>` не указано, цикл будет выполняться бесконечно. Чтобы прервать бесконечный цикл, следует нажать комбинацию клавиш <Ctrl>+<C>. В результате генерируется исключение `KeyboardInterrupt`, и выполнение программы останавливается. Следует учитывать, что прервать таким образом можно только цикл, который выводит данные.

Выведем все числа от 100 до 1 (листинг 4.8).

Листинг 4.8. Вывод чисел от 100 до 1

```
i = 100
while i:
    print(i)
    i -= 1
```

Обратите внимание на условие — оно не содержит операторов сравнения. На каждой итерации цикла мы вычитаем единицу из значения переменной-счетчика. Как только значение будет равно `0`, цикл остановится, поскольку число `0` в логическом контексте эквивалентно значению `False`.

С помощью цикла `while` можно перебирать и элементы различных структур. Но в этом случае следует помнить, что цикл `while` работает медленнее цикла `for`. В качестве примера умножим каждый элемент списка на 2 (листинг 4.9).

Листинг 4.9. Перебор элементов списка

```
arr = [1, 2, 3]
i, count = 0, len(arr)
while i < count:
    arr[i] *= 2
    i += 1
print(arr)              # Результат выполнения: [2, 4, 6]
```

4.6. Оператор *continue*: переход на следующую итерацию цикла

Оператор `continue` позволяет перейти к следующей итерации цикла до завершения выполнения всех инструкций внутри цикла. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно (листинг 4.10).

Листинг 4.10. Оператор `continue`

```
for i in range(1, 101):
    if 4 < i < 11:
        continue        # Переходим на следующую итерацию цикла
    print(i)
```

4.7. Оператор *break*: прерывание цикла

Оператор `break` позволяет прервать выполнение цикла досрочно. Для примера выведем все числа от 1 до 100 еще одним способом (листинг 4.11).

Листинг 4.11. Оператор `break`

```
i = 1
while True:
    if i > 100: break   # Прерываем цикл
    print(i)
    i += 1
```

Здесь мы в условии указали значение `True`. В этом случае выражения внутри цикла станут выполняться бесконечно. Однако использование оператора `break` прерывает выполнение цикла, как только он будет выполнен 100 раз.

> ***Внимание!***
>
> Оператор `break` прерывает выполнение цикла, а не программы, т. е. далее будет выполнена инструкция, следующая сразу за циклом.

Цикл `while` совместно с оператором `break` удобно использовать для получения не определенного заранее количества данных от пользователя. В качестве примера просуммируем произвольное количество чисел (листинг 4.12).

Листинг 4.12. Суммирование не определенного заранее количества чисел

```
# -*- coding: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    if x == "stop":
        break                        # Выход из цикла
    x = int(x)                       # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода трех чисел и получения суммы выглядит так (значения, введенные пользователем, здесь выделены полужирным шрифтом):

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: 20
Введите число: 30
Введите число: stop
Сумма чисел равна: 60
```

ГЛАВА 5

Числа

Язык Python 3 поддерживает следующие *числовые типы*:

- `int` — целые числа. Размер числа ограничен лишь объемом оперативной памяти;
- `float` — вещественные числа;
- `complex` — комплексные числа.

Операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое число и вещественное, то целое число будет автоматически преобразовано в вещественное число, а затем произведена операция над вещественными числами. Результатом этой операции будет вещественное число.

Создать объект целочисленного типа в десятичной системе счисления можно обычным способом:

```
>>> x = 0; y = 10; z = -80
>>> x, y, z
(0, 10, -80)
```

Кроме того, можно записать число в двоичной, восьмеричной или шестнадцатеричной форме. Такие числа будут автоматически преобразованы в десятичные целые числа.

- Двоичные числа начинаются с комбинации символов `0b` (или `0B`) и содержат цифры `0` или `1`:

```
>>> 0b11111111, 0b101101
(255, 45)
```

- Восьмеричные числа начинаются с нуля и следующей за ним латинской буквы `o` (регистр не имеет значения) и содержат цифры от `0` до `7`:

```
>>> 0o7, 0o12, 0o777, 0O7, 0O12, 0O777
(7, 10, 511, 7, 10, 511)
```

- Шестнадцатеричные числа начинаются с комбинации символов `0x` (или `0X`) и могут содержать цифры от `0` до `9` и буквы от `A` до `F` (регистр букв не имеет значения):

```
>>> 0X9,0xA, 0x10, 0xFFF, 0xfff
(9, 10, 16, 4095, 4095)
```

- Вещественное число может содержать точку и (или) быть представлено в экспоненциальной форме с буквой `E` (регистр не имеет значения):

```
>>> 10., .14, 3.14, 11E20, 2.5e-12
(10.0, 0.14, 3.14, 1.1e+21, 2.5e-12)
```

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 - 0.1 - 0.1 - 0.1
-2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другое значение. Если необходимо производить операции с фиксированной точностью, то следует использовать модуль `decimal`:

```
>>> from decimal import Decimal
>>> Decimal("0.3") - Decimal("0.1") - Decimal("0.1") - Decimal("0.1")
Decimal('0.0')
```

Кроме того, можно использовать дроби, поддержка которых реализована в модуле `fractions`. При создании дроби можно указать как два числа: числитель и знаменатель, так и одно число или строку, содержащую число, которое будет преобразовано в дробь.

Для примера создадим несколько дробей. Вот так формируется дробь $^4/_5$:

```
>>> from fractions import Fraction
>>> Fraction(4, 5)
Fraction(4, 5)
```

А вот так — дробь $^1/_2$, причем можно сделать это тремя способами:

```
>>> Fraction(1, 2)
Fraction(1, 2)
>>> Fraction("0.5")
Fraction(1, 2)
>>> Fraction(0.5)
Fraction(1, 2)
```

Над дробями можно производить арифметические операции, как и над обычными числами:

```
>>> Fraction(9, 5) - Fraction(2, 3)
Fraction(17, 15)
>>> Fraction("0.3") - Fraction("0.1") - Fraction("0.1") - Fraction("0.1")
Fraction(0, 1)
>>> float(Fraction(0, 1))
0.0
```

Комплексные числа записываются в формате:

```
<Вещественная часть>+<Мнимая часть>J
```

Здесь буква `J` может стоять в любом регистре. Примеры комплексных чисел:

```
>>> 2+5J, 8j
((2+5j), 8j)
```

Подробное рассмотрение модулей `decimal` и `fractions`, а также комплексных чисел выходит за рамки нашей книги. За подробной информацией обращайтесь к документации по языку Python.

В Python 3.6 появилась возможность для улучшения читаемости кода вставлять в число, записанное в десятичной, двоичной, восьмеричной и шестнадцатеричной системах счисления, символы подчеркивания:

```
>>> 1_000_000
1000000
>>> 0b1111_1111
255
>>> 0o2_777
1535
>>> 0xab_cd
43981
```

5.1. Встроенные функции и методы для работы с числами

Для работы с числами предназначены следующие встроенные функции:

- `int([<Объект>[, <Система счисления>]])` — преобразует объект в целое число. Во втором параметре можно указать систему счисления преобразуемого числа (значение по умолчанию `10`):

```
>>> int(7.5), int("71", 10), int("0o71", 8), int("0xA", 16)
(7, 71, 57, 10)
>>> int(), int("0b11111111", 2)
(0, 255)
```

- `float([<Число или строка>])` — преобразует целое число или строку в вещественное число:

```
>>> float(7), float("7.1"), float("12.")
(7.0, 7.1, 12.0)
>>> float("inf"), float("-Infinity"), float("nan")
(inf, -inf, nan)
>>> float()
0.0
```

- `bin(<Число>)` — преобразует десятичное число в двоичное. Возвращает строковое представление числа:

```
>>> bin(255), bin(1), bin(-45)
('0b11111111', '0b1', '-0b101101')
```

- `oct(<Число>)` — преобразует десятичное число в восьмеричное. Возвращает строковое представление числа:

```
>>> oct(7), oct(8), oct(64)
('0o7', '0o10', '0o100')
```

- `hex(<Число>)` — преобразует десятичное число в шестнадцатеричное. Возвращает строковое представление числа:

```
>>> hex(10), hex(16), hex(255)
('0xa', '0x10', '0xff')
```

- `round(<Число>[, <Количество знаков после точки>])` — для чисел с дробной частью, меньшей `0.5`, возвращает число, округленное до ближайшего меньшего целого, а для чисел с дробной частью, большей `0.5`, возвращает число, округленное до ближайшего большего целого. Если дробная часть равна `0.5`, то округление производится до ближайшего четного числа:

```
>>> round(0.49), round(0.50), round(0.51)
(0, 0, 1)
>>> round(1.49), round(1.50), round(1.51)
(1, 2, 2)
>>> round(2.49), round(2.50), round(2.51)
(2, 2, 3)
>>> round(3.49), round(3.50), round(3.51)
(3, 4, 4)
```

  Во втором параметре можно указать желаемое количество знаков после запятой. Если оно не указано, используется значение `0` (т. е. число будет округлено до целого):

```
>>> round(1.524, 2), round(1.525, 2), round(1.5555, 3)
(1.52, 1.52, 1.556)
```

- `abs(<Число>)` — возвращает абсолютное значение:

```
>>> abs(-10), abs(10), abs(-12.5)
(10, 10, 12.5)
```

- `pow(<Число>, <Степень>[, <Делитель>])` — **возводит** `<Число>` **в** `<Степень>`:

```
>>> pow(10, 2), 10 ** 2, pow(3, 3), 3 ** 3
(100, 100, 27, 27)
```

  Если указан третий параметр, возвращается остаток от деления полученного результата на значение этого параметра:

```
>>> pow(10, 2, 2), (10 ** 2) % 2, pow(3, 3, 2), (3 ** 3) % 2
(0, 0, 1, 1)
```

- `max(<Список чисел через запятую>)` — максимальное значение из списка:

```
>>> max(1, 2, 3), max(3, 2, 3, 1), max(1, 1.0), max(1.0, 1)
(3, 3, 1, 1.0)
```

- `min(<Список чисел через запятую>)` — минимальное значение из списка:

```
>>> min(1, 2, 3), min(3, 2, 3, 1), min(1, 1.0), min(1.0, 1)
(1, 1, 1, 1.0)
```

- `sum(<Последовательность>[, <Начальное значение>])` — возвращает сумму значений элементов последовательности (списка, кортежа и пр.) плюс `<Начальное значение>`. Если второй параметр не указан, начальное значение принимается равным `0`. Если последовательность пустая, возвращается значение второго параметра:

```
>>> sum((10, 20, 30, 40)), sum([10, 20, 30, 40])
(100, 100)
>>> sum([10, 20, 30, 40], 2), sum([], 2)
(102, 2)
```

♦ `divmod(x, y)` — возвращает кортеж из двух значений `(x // y, x % y)`:

```
>>> divmod(13, 2)          # 13 == 6 * 2 + 1
(6, 1)
>>> 13 // 2, 13 % 2
(6, 1)
>>> divmod(13.5, 2.0)      # 13.5 == 6.0 * 2.0 + 1.5
(6.0, 1.5)
>>> 13.5 // 2.0, 13.5 % 2.0
(6.0, 1.5)
```

Следует понимать, что все типы данных, поддерживаемые Python, представляют собой классы. Класс `float`, представляющий вещественные числа, поддерживает следующие полезные методы:

♦ `is_integer()` — возвращает `True`, если заданное вещественное число не содержит дробной части, т. е. фактически представляет собой целое число:

```
>>> (2.0).is_integer()
True
>>> (2.3).is_integer()
False
```

♦ `as_integer_ratio()` — возвращает кортеж из двух целых чисел, представляющих собой числитель и знаменатель дроби, которая соответствует заданному числу:

```
>>> (0.5).as_integer_ratio()
(1, 2)
>>> (2.3).as_integer_ratio()
(2589569785738035, 1125899906842624)
```

5.2. Модуль *math*: математические функции

Модуль `math` предоставляет дополнительные функции для работы с числами, а также стандартные константы. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import math
```

> **ПРИМЕЧАНИЕ**
>
> Для работы с комплексными числами необходимо использовать модуль `cmath`.

Модуль `math` предоставляет следующие стандартные константы:

♦ `pi` — число π:

```
>>> import math
>>> math.pi
3.141592653589793
```

♦ `e` — константа *e*:

```
>>> math.e
2.718281828459045
```

Приведем перечень основных функций для работы с числами:

♦ `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус, косинус, тангенс). Значение указывается в радианах;

- `asin()`, `acos()`, `atan()` — обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах;
- `degrees()` — преобразует радианы в градусы:

```
>>> math.degrees(math.pi)
180.0
```

- `radians()` — преобразует градусы в радианы:

```
>>> math.radians(180.0)
3.141592653589793
```

- `exp()` — экспонента;
- `log(<Число>[, <База>])` — логарифм по заданной базе. Если база не указана, вычисляется натуральный логарифм (по базе *e*);
- `log10()` — десятичный логарифм;
- `log2()` — логарифм по базе 2;
- `sqrt()` — квадратный корень:

```
>>> math.sqrt(100), math.sqrt(25)
(10.0, 5.0)
```

- `ceil()` — значение, округленное до ближайшего большего целого:

```
>>> math.ceil(5.49), math.ceil(5.50), math.ceil(5.51)
(6, 6, 6)
```

- `floor()` — значение, округленное до ближайшего меньшего целого:

```
>>> math.floor(5.49), math.floor(5.50), math.floor(5.51)
(5, 5, 5)
```

- `pow(<Число>, <Степень>)` — возводит число в степень:

```
>>> math.pow(10, 2), 10 ** 2, math.pow(3, 3), 3 ** 3
(100.0, 100, 27.0, 27)
```

- `fabs()` — абсолютное значение:

```
>>> math.fabs(10), math.fabs(-10), math.fabs(-12.5)
(10.0, 10.0, 12.5)
```

- `fmod()` — остаток от деления:

```
>>> math.fmod(10, 5), 10 % 5, math.fmod(10, 3), 10 % 3
(0.0, 0, 1.0, 1)
```

- `factorial()` — факториал числа:

```
>>> math.factorial(5), math.factorial(6)
(120, 720)
```

- `fsum(<Список чисел>)` — возвращает точную сумму чисел из заданного списка:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

ПРИМЕЧАНИЕ

В этом разделе мы рассмотрели только основные функции. Чтобы получить полный список функций, обращайтесь к документации по модулю `math`.

5.3. Модуль *random*: генерирование случайных чисел

Модуль `random` позволяет генерировать случайные числа. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import random
```

Приведем перечень его основных функций:

- `random()` — возвращает псевдослучайное число от `0.0` до `1.0`:

```
>>> import random
>>> random.random()
0.9753144027290991
>>> random.random()
0.5468390487484339
>>> random.random()
0.13015058054767736
```

- `seed([<Параметр>][, version=2])` — настраивает генератор случайных чисел на новую последовательность. Если первый параметр не указан, в качестве базы для случайных чисел будет использовано системное время. При одинаковых значениях первого параметра будет генерироваться одинаковая последовательность чисел:

```
>>> random.seed(10)
>>> random.random()
0.5714025946899135
>>> random.seed(10)
>>> random.random()
0.5714025946899135
```

- `uniform(<Начало>, <Конец>)` — возвращает псевдослучайное вещественное число в диапазоне от `<Начало>` до `<Конец>`:

```
>>> random.uniform(0, 10)
9.965569925394552
>>> random.uniform(0, 10)
0.4455638245043303
```

- `randint(<Начало>, <Конец>)` — возвращает псевдослучайное целое число в диапазоне от `<Начало>` до `<Конец>`:

```
>>> random.randint(0, 10)
4
>>> random.randint(0, 10)
10
```

- `randrange([<Начало>, ]<Конец>[, <Шаг>])` — возвращает случайный элемент из создаваемого «за кадром» диапазона. Параметры аналогичны параметрам функции `range()`:

```
>>> random.randrange(10)
5
>>> random.randrange(0, 10)
2
>>> random.randrange(0, 10, 2)
6
```

♦ `choice(<Последовательность>)` — возвращает случайный элемент из заданной последовательности (строки, списка, кортежа):

```
>>> random.choice("string")          # Строка
'i'
>>> random.choice(["s", "t", "r"])   # Список
'r'
>>> random.choice(("s", "t", "r"))   # Кортеж
't'
```

♦ `shuffle(<Список>[, <Число от 0.0 до 1.0>])` — перемешивает элементы списка случайным образом. Если второй параметр не указан, то используется значение, возвращаемое функцией `random()`. Никакого результата при этом не возвращается:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)
>>> arr
[8, 6, 9, 5, 3, 7, 2, 4, 10, 1]
```

♦ `sample(<Последовательность>, <Количество элементов>)` — возвращает список из указанного количества элементов, которые будут выбраны случайным образом из заданной последовательности. В качестве таковой можно указать любые объекты, поддерживающие итерации:

```
>>> random.sample("string", 2)
['i', 'r']
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
[7, 10]
>>> arr # Сам список не изменяется
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample((1, 2, 3, 4, 5, 6, 7), 3)
[6, 3, 5]
>>> random.sample(range(300), 5)
[126, 194, 272, 46, 71]
```

Для примера напишем функцию-генератор паролей произвольной длины (листинг 5.1). Для этого добавим в список `arr` все разрешенные символы, а далее в цикле будем получать случайный элемент с помощью функции `choice()`. По умолчанию будет выдаваться пароль из 8 символов.

Листинг 5.1. Генератор паролей

```
# -*- coding: utf-8 -*-
import random # Подключаем модуль random
def passw_generator(count_char=8):
```

```
    arr = ['a','b','c','d','e','f','g','h','i','j','k','l','m',
    'n','o','p','q','r','s','t','u','v','w','x','y','z',
    'A','B','C','D','E','F','G','H','I','J','K','L',
    'M','N','O','P','Q','R','S','T','U','V', 'W',
    'X','Y','Z','1','2','3','4','5','6','7','8','9','0']
    passw = []
    for i in range(count_char):
        passw.append(random.choice(arr))
    return "".join(passw)
```

Испытаем эту функцию в действии:

```
print(passw_generator(10))  # Выведет что-то вроде ngODHE8J8x
print(passw_generator())    # Выведет что-то вроде ZxcpkF5O
```

ГЛАВА 6

Строки и двоичные данные

Строки представляют собой последовательности символов. Длина строки ограничена лишь объемом оперативной памяти компьютера. Как и все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор +), повторение (оператор *), проверку на вхождение (операторы `in` и `not in`).

Кроме того, строки относятся к неизменяемым типам данных. Поэтому практически все строковые методы в качестве значения возвращают новую строку. (При использовании небольших строк это не приводит к каким-либо проблемам, но при работе с большими строками можно столкнуться с проблемой нехватки памяти.) Иными словами, можно получить символ по индексу, но изменить его будет нельзя:

```
>>> s = "Python"
>>> s[0]                    # Можно получить символ по индексу
'P'
>>> s[0] = "J"              # Изменить строку нельзя
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    s[0] = "J"              # Изменить строку нельзя
TypeError: 'str' object does not support item assignment
```

В некоторых языках программирования строка должна заканчиваться нулевым символом. В языке Python нулевой символ может быть расположен внутри строки:

```
>>> "string\x00string" # Нулевой символ — это НЕ конец строки
'string\x00string'
```

Python поддерживает следующие строковые типы:

♦ `str` — Unicode-строка. Обратите внимание, конкретная кодировка: UTF-8, UTF-16 или UTF-32 — здесь не указана. Рассматривайте такие строки, как строки в некой абстрактной кодировке, позволяющие хранить символы Unicode и производить манипуляции с ними. При выводе Unicode-строку необходимо преобразовать в последовательность байтов в какой-либо кодировке:

  ```
  >>> type("строка")
  <class 'str'>
  >>> "строка".encode(encoding="cp1251")
  b'\xf1\xf2\xf0\xee\xea\xe0'
  >>> "строка".encode(encoding="utf-8")
  b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
  ```

- bytes — неизменяемая последовательность байтов. Каждый элемент последовательности может хранить целое число от 0 до 255, которое обозначает код символа. Объект типа bytes поддерживает большинство строковых методов и, если это возможно, выводится как последовательность символов. Однако доступ по индексу возвращает целое число, а не символ:

```
>>> s = bytes("стр str", "cp1251")
>>> s[0], s[5], s[0:3], s[4:7]
(241, 116, b'\xf1\xf2\xf0', b'str')
>>> s
b'\xf1\xf2\xf0 str'
```

  Объект типа bytes может содержать как однобайтовые, так и многобайтовые символы. Обратите внимание на то, что функции и методы строк некорректно работают с многобайтовыми кодировками, — например, функция len() вернет количество байтов, а не символов:

```
>>> len("строка")
6
>>> len(bytes("строка", "cp1251"))
6
>>> len(bytes("строка", "utf-8"))
12
```

- bytearray — изменяемая последовательность байтов. Тип bytearray аналогичен типу bytes, но позволяет изменять элементы по индексу и содержит дополнительные методы, дающие возможность добавлять и удалять элементы:

```
>>> s = bytearray("str", "cp1251")
>>> s[0] = 49; s            # Можно изменить символ
bytearray(b'1tr')
>>> s.append(55); s         # Можно добавить символ
bytearray(b'1tr7')
```

Во всех случаях, когда речь идет о текстовых данных, следует использовать тип str. Именно этот тип мы будем называть словом «строка». Типы bytes и bytearray следует задействовать для записи двоичных данных (например, изображений) и промежуточного хранения строк. Более подробно типы bytes и bytearray мы рассмотрим в конце этой главы.

6.1. Создание строки

Создать строку можно следующими способами:

- с помощью функции str([<Объект>[, <Кодировка>[, <Обработка ошибок>]]]). Если указан только первый параметр, функция возвращает строковое представление любого объекта. Если параметры не указаны вообще, возвращается пустая строка:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0")
"b'\\xf1\\xf2\\xf0\\xee\\xea\\xe0'"
```

  Обратите внимание на преобразование объекта типа bytes. Мы получили строковое представление объекта, а не нормальную строку. Чтобы получить из объектов типа bytes и bytearray именно строку, следует указать кодировку во втором параметре:

```
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0", "cp1251")
'строка'
```

В третьем параметре могут быть указаны значения `"strict"` (при ошибке возбуждается исключение `UnicodeDecodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом, имеющим код `\uFFFD`) или `"ignore"` (неизвестные символы игнорируются):

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    str(obj1, "ascii", "strict")
UnicodeDecodeError: 'ascii' codec can't decode byte
0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
'1'
```

- указав строку между апострофами или двойными кавычками:

```
>>> 'строка', "строка", '"x": 5', "'x': 5"
('строка', 'строка', '"x": 5', "'x': 5")
>>> print('Строка1\nСтрока2')
Строка1
Строка2
>>> print("Строка1\nСтрока2")
Строка1
Строка2
```

В некоторых языках программирования (например, в PHP) строка в апострофах отличается от строки в кавычках тем, что внутри апострофов специальные символы выводятся как есть, а внутри кавычек они интерпретируются. В языке Python никакого отличия между строкой в апострофах и строкой в кавычках нет. Если строка содержит кавычки, то ее лучше заключить в апострофы, и наоборот. Все специальные символы в таких строках интерпретируются — например, последовательность символов `\n` преобразуется в символ новой строки. Чтобы специальный символ выводился как есть, его необходимо экранировать с помощью слэша:

```
>>> print("Строка1\\nСтрока2")
Строка1\nСтрока2
>>> print('Строка1\\nСтрока2')
Строка1\nСтрока2
```

Кавычку внутри строки в кавычках и апостроф внутри строки в апострофах также необходимо экранировать с помощью защитного слэша:

```
>>> "\"x\": 5", '\'x\': 5'
('"x": 5', "'x': 5")
```

Следует также заметить, что заключить объект в одинарные кавычки (или апострофы) на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку:

```
>>> "string
SyntaxError: EOL while scanning string literal
```

Чтобы расположить объект на нескольких строках, следует перед символом перевода строки указать символ \, поместить две строки внутри скобок или использовать конкатенацию внутри скобок:

```
>>> "string1\
string2"              # После \ не должно быть никаких символов
'string1string2'
>>> ("string1"
"string2")            # Неявная конкатенация строк
'string1string2'
>>> ("string1" +
"string2")            # Явная конкатенация строк
'string1string2'
```

Кроме того, если в конце строки расположен символ \, то его необходимо экранировать, иначе будет выведено сообщение об ошибке:

```
>>> print("string\")

SyntaxError: EOL while scanning string literal
>>> print("string\\")
string\
```

- указав строку между утроенными апострофами или утроенными кавычками. Такие объекты можно разместить на нескольких строках. Допускается также одновременно использовать и кавычки, и апострофы без необходимости их экранировать. В остальном такие объекты эквивалентны строкам в апострофах и кавычках. Все специальные символы в таких строках интерпретируются:

```
>>> print('''Строка1
Строка2''')
Строка1
Строка2
>>> print("""Строка1
Строка2""")
Строка1
Строка2
```

Если строка не присваивается переменной, то она считается *строкой документирования*. Такая строка сохраняется в атрибуте `__doc__` того объекта, в котором расположена. В качестве примера создадим функцию со строкой документирования, а затем выведем содержимое строки:

```
>>> def test():
        """Это описание функции"""
        pass

>>> print(test.__doc__)
Это описание функции
```

Поскольку выражения внутри таких строк не выполняются, то утроенные кавычки (или утроенные апострофы) очень часто используются для комментирования больших фрагментов кода на этапе отладки программы.

Если перед строкой разместить модификатор `r`, то специальные символы внутри строки выводятся как есть. Например, символ `\n` не будет преобразован в символ перевода строки. Иными словами, он будет считаться последовательностью символов `\` и `n`:

```
>>> print("Строка1\nСтрока2")
Строка1
Строка2
>>> print(r"Строка1\nСтрока2")
Строка1\nСтрока2
>>> print(r"""Строка1\nСтрока2""")
Строка1\nСтрока2
```

Такие неформатированные строки удобно использовать в шаблонах регулярных выражений, а также при указании пути к файлу или каталогу:

```
>>> print(r"C:\Python36\lib\site-packages")
C:\Python36\lib\site-packages
```

Если модификатор не указать, то все слэши при указании пути необходимо экранировать:

```
>>> print("C:\\Python36\\lib\\site-packages")
C:\Python36\lib\site-packages
```

Если в конце неформатированной строки расположен слэш, то его необходимо экранировать. Однако следует учитывать, что этот слэш будет добавлен в исходную строку:

```
>>> print(r"C:\Python36\lib\site-packages\")

SyntaxError: EOL while scanning string literal
>>> print(r"C:\Python36\lib\site-packages\\")
C:\Python36\lib\site-packages\\
```

Чтобы избавиться от лишнего слэша, можно использовать операцию конкатенации строк, обычные строки или удалить слэш явным образом:

```
>>> print(r"C:\Python36\lib\site-packages" + "\\") # Конкатенация
C:\Python36\lib\site-packages\
>>> print("C:\\Python36\\lib\\site-packages\\")    # Обычная строка
C:\Python36\lib\site-packages\
>>> print(r"C:\Python36\lib\site-packages\\"[:-1]) # Удаление слэша
C:\Python36\lib\site-packages\
```

6.2. Специальные символы

Специальные символы — это комбинации знаков, обозначающих служебные или непечатаемые символы, которые невозможно вставить обычным способом. Приведем перечень специальных символов, допустимых внутри строки, перед которой нет модификатора `r`:

- `\n` — перевод строки;
- `\r` — возврат каретки;
- `\t` — знак табуляции;
- `\v` — вертикальная табуляция;
- `\a` — звонок;

- `\b` — забой;
- `\f` — перевод формата;
- `\0` — нулевой символ (не является концом строки);
- `\"` — кавычка;
- `\'` — апостроф;
- `\N` — символ с восьмеричным кодом `N`. Например, `\74` соответствует символу `<`;
- `\xN` — символ с шестнадцатеричным кодом `N`. Например, `\x6a` соответствует символу `j`;
- `\\` — обратный слэш;
- `\uxxxx` — 16-битный символ Unicode. Например, `\u043a` соответствует русской букве `к`;
- `\Uxxxxxxxx` — 32-битный символ Unicode.

Если после слэша не стоит символ, который вместе со слэшем интерпретируется как спецсимвол, то слэш сохраняется в составе строки:

```
>>> print("Этот символ \не специальный")
Этот символ \не специальный
```

Тем не менее, лучше экранировать слэш явным образом:

```
>>> print("Этот символ \\не специальный")
Этот символ \не специальный
```

6.3. Операции над строками

Как вы уже знаете, строки относятся к последовательностям. Как и все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение. Рассмотрим эти операции подробно.

К любому символу строки можно обратиться как к элементу списка — достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля:

```
>>> s = "Python"
>>> s[0], s[1], s[2], s[3], s[4], s[5]
('P', 'y', 't', 'h', 'o', 'n')
```

Если символ, соответствующий указанному индексу, отсутствует в строке, возбуждается исключение `IndexError`:

```
>>> s = "Python"
>>> s[10]
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    s[10]
IndexError: string index out of range
```

В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца строки, а точнее — чтобы получить положительный индекс, значение вычитается из длины строки:

```
>>> s = "Python"
>>> s[-1], s[len(s)-1]
('n', 'n')
```

Так как строки относятся к неизменяемым типам данных, то изменить символ по индексу нельзя:

```
>>> s = "Python"
>>> s[0] = "J"                          # Изменить строку нельзя
Traceback (most recent call last):
  File "<pyshell#94>", line 1, in <module>
    s[0] = "J"                          # Изменить строку нельзя
TypeError: 'str' object does not support item assignment
```

Чтобы выполнить изменение, можно воспользоваться операцией извлечения среза, которая возвращает указанный фрагмент строки. Формат операции:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры здесь не являются обязательными. Если параметр `<Начало>` не указан, то используется значение `0`. Если параметр `<Конец>` не указан, то возвращается фрагмент до конца строки. Следует также заметить, что символ с индексом, указанным в этом параметре, не входит в возвращаемый фрагмент. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве значения параметров можно указать отрицательные значения.

Рассмотрим несколько примеров:

- сначала получим копию строки:

```
>>> s = "Python"
>>> s[:] # Возвращается фрагмент от позиции 0 до конца строки
'Python'
```

- теперь выведем символы в обратном порядке:

```
>>> s[::-1] # Указываем отрицательное значение в параметре <Шаг>
'nohtyP'
```

- заменим первый символ в строке:

```
>>> "J" + s[1:] # Извлекаем фрагмент от символа 1 до конца строки
'Jython'
```

- удалим последний символ:

```
>>> s[:-1] # Возвращается фрагмент от 0 до len(s)-1
'Pytho'
```

- получим первый символ в строке:

```
>>> s[0:1] # Символ с индексом 1 не входит в диапазон
'P'
```

- а теперь получим последний символ:

```
>>> s[-1:] # Получаем фрагмент от len(s)-1 до конца строки
'n'
```

- и, наконец, выведем символы с индексами 2, 3 и 4:

```
>>> s[2:5] # Возвращаются символы с индексами 2, 3 и 4
'tho'
```

Узнать количество символов в строке (ее длину) позволяет функция `len()`:

```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
(6, 3, 6)
```

Теперь, когда мы знаем количество символов, можно перебрать все символы с помощью цикла `for`:

```
>>> s = "Python"
>>> for i in range(len(s)): print(s[i], end=" ")
```

Результат выполнения:

```
P y t h o n
```

Так как строки поддерживают итерации, мы можем просто указать строку в качестве параметра цикла:

```
>>> s = "Python"
>>> for i in s: print(i, end=" ")
```

Результат выполнения будет таким же:

```
P y t h o n
```

Соединить две строки в одну строку (выполнить их конкатенацию) позволяет оператор `+`:

```
>>> print("Строка1" + "Строка2")
Строка1Строка2
```

Кроме того, можно выполнить неявную конкатенацию строк. В этом случае две строки указываются рядом без оператора между ними:

```
>>> print("Строка1" "Строка2")
Строка1Строка2
```

Обратите внимание на то, что если между строками указать запятую, то мы получим кортеж, а не строку:

```
>>> s = "Строка1", "Строка2"
>>> type(s)                      # Получаем кортеж, а не строку
<class 'tuple'>
```

Если соединяются, например, переменная и строка, то следует обязательно указывать символ конкатенации строк, иначе будет выведено сообщение об ошибке:

```
>>> s = "Строка1"
>>> print(s + "Строка2")         # Нормально
Строка1Строка2
>>> print(s "Строка2")           # Ошибка
SyntaxError: invalid syntax
```

При необходимости соединить строку со значением другого типа (например, с числом) следует произвести явное преобразование типов с помощью функции `str()`:

```
>>> "string" + str(10)
'string10'
```

Кроме рассмотренных операций, строки поддерживают операцию повторения, проверки на вхождение и невхождение. Повторить строку указанное количество раз можно с помощью оператора `*`, выполнить проверку на вхождение фрагмента в строку позволяет оператор `in`, а проверить на невхождение — оператор `not in`:

```
>>> "-" * 20
'--------------------'
>>> "yt" in "Python"             # Найдено
True
```

```
>>> "yt" in "Perl"                  # Не найдено
False
>>> "PHP" not in "Python"           # Не найдено
True
```

6.4. Форматирование строк

Вместо соединения строк с помощью оператора + лучше использовать форматирование. Эта операция позволяет соединять строки со значениями любых других типов и выполняется быстрее конкатенации.

> ***Примечание***
>
> В последующих версиях Python оператор форматирования `%` может быть удален. Вместо этого оператора в новом коде следует использовать метод `format()`, который рассматривается в следующем разделе.

Операция форматирования записывается следующим образом:

```
<Строка специального формата> % <Значения>
```

Внутри параметра `<Строка специального формата>` могут быть указаны спецификаторы, имеющие следующий синтаксис:

```
%[(<Ключ>)][<Флаг>][<Ширина>][.<Точность>]<Тип преобразования>
```

Количество спецификаторов внутри строки должно быть равно количеству элементов в параметре `<Значения>`. Если используется только один спецификатор, то параметр `<Значения>` может содержать одно значение, в противном случае необходимо указать значения через запятую внутри круглых скобок, создавая тем самым кортеж:

```
>>> "%s" % 10                          # Один элемент
'10'
>>> "%s — %s — %s" % (10, 20, 30)      # Несколько элементов
'10 — 20 — 30'
```

Параметры внутри спецификатора имеют следующий смысл:

- `<Ключ>` — ключ словаря. Если задан ключ, то в параметре `<Значения>` необходимо указать словарь, а не кортеж:

  ```
  >>> "%(name)s — %(year)s" % {"year": 1978, "name": "Nik"}
  'Nik — 1978'
  ```

- `<Флаг>` — флаг преобразования. Может содержать следующие значения:

  - `#` — для восьмеричных значений добавляет в начало комбинацию символов `0o`, для шестнадцатеричных значений — комбинацию символов `0x` (если используется тип `x`) или `0X` (если используется тип `X`), для вещественных чисел предписывает всегда выводить дробную точку, даже если задано значение `0` в параметре `<Точность>`:

    ```
    >>> print("%#o %#o %#o" % (0o77, 10, 10.5))
    0o77 0o12 0o12
    >>> print("%#x %#x %#x" % (0xff, 10, 10.5))
    0xff 0xa 0xa
    >>> print("%#X %#X %#X" % (0xff, 10, 10.5))
    0XFF 0XA 0XA
    ```

```
>>> print("%#.0F %.0F" % (300, 300))
300. 300
```

- 0 — задает наличие ведущих нулей для числового значения:

  ```
  >>> "'%d' — '%05d'" % (3, 3) # 5 — ширина поля
  "'3' — '00003'"
  ```

- - — задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе. Если флаг указан одновременно с флагом 0, то действие флага 0 будет отменено:

  ```
  >>> "'%5d' - '%-5d'" % (3, 3) # 5 — ширина поля
  "'    3' - '3    '"
  >>> "'%05d' - '%-05d'" % (3, 3)
  "'00003' - '3    '"
  ```

- пробел — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус:

  ```
  >>> "'% d' - '% d'" % (-3, 3)
  "'-3' - ' 3'"
  ```

- + — задает обязательный вывод знака как для отрицательных, так и для положительных чисел. Если флаг + указан одновременно с флагом пробел, то действие флага пробел будет отменено:

  ```
  >>> "'%+d' - '%+d'" % (-3, 3)
  "'-3' - '+3'"
  ```

- <Ширина> — минимальная ширина поля. Если строка не помещается в указанную ширину, значение игнорируется, и строка выводится полностью:

  ```
  >>> "'%10d' - '%-10d'" % (3, 3)
  "'         3' - '3         '"
  >>> "'%3s''%10s'" % ("string", "string")
  "'string''    string'"
  ```

  Вместо значения можно указать символ «*». В этом случае значение следует задать внутри кортежа:

  ```
  >>> "'%*s''%10s'" % (10, "string", "str")
  "'    string''       str'"
  ```

- <Точность> — количество знаков после точки для вещественных чисел. Перед этим параметром обязательно должна стоять точка:

  ```
  >>> import math
  >>> "%s %f %.2f" % (math.pi, math.pi, math.pi)
  '3.141592653589793 3.141593 3.14'
  ```

  Вместо значения можно указать символ «*». В этом случае значение следует задать внутри кортежа:

  ```
  >>> "'%*.*f'" % (8, 5, math.pi)
  "' 3.14159'"
  ```

- <Тип преобразования> — задает тип преобразования. Параметр является обязательным.

В параметре `<Тип преобразования>` могут быть указаны следующие символы:

* `s` — преобразует любой объект в строку с помощью функции `str()`:

```
>>> print("%s" % ("Обычная строка"))
Обычная строка
>>> print("%s %s %s" % (10, 10.52, [1, 2, 3]))
10 10.52 [1, 2, 3]
```

* `r` — преобразует любой объект в строку с помощью функции `repr()`:

```
>>> print("%r" % ("Обычная строка"))
'Обычная строка'
```

* `a` — преобразует объект в строку с помощью функции `ascii()`:

```
>>> print("%a" % ("строка"))
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

* `c` — выводит одиночный символ или преобразует числовое значение в символ. В качестве примера выведем числовое значение и соответствующий этому значению символ:

```
>>> for i in range(33, 127): print("%s => %c" % (i, i))
```

* `d` и `i` — возвращают целую часть числа:

```
>>> print("%d %d %d" % (10, 25.6, -80))
10 25 -80
>>> print("%i %i %i" % (10, 25.6, -80))
10 25 -80
```

* `o` — восьмеричное значение:

```
>>> print("%o %o %o" % (0o77, 10, 10.5))
77 12 12
>>> print("%#o %#o %#o" % (0o77, 10, 10.5))
0o77 0o12 0o12
```

* `x` — шестнадцатеричное значение в нижнем регистре:

```
>>> print("%x %x %x" % (0xff, 10, 10.5))
ff a a
>>> print("%#x %#x %#x" % (0xff, 10, 10.5))
0xff 0xa 0xa
```

* `X` — шестнадцатеричное значение в верхнем регистре:

```
>>> print("%X %X %X" % (0xff, 10, 10.5))
FF A A
>>> print("%#X %#X %#X" % (0xff, 10, 10.5))
0XFF 0XA 0XA
```

* `f` и `F` — вещественное число в десятичном представлении:

```
>>> print("%f %f %f" % (300, 18.65781452, -12.5))
300.000000 18.657815 -12.500000
>>> print("%F %F %F" % (300, 18.65781452, -12.5))
300.000000 18.657815 -12.500000
>>> print("%#.0F %.0F" % (300, 300))
300. 300
```

- `e` — вещественное число в экспоненциальной форме (буква `e` в нижнем регистре):

```
>>> print("%e %e" % (3000, 18657.81452))
3.000000e+03 1.865781e+04
```

- `E` — вещественное число в экспоненциальной форме (буква `E` в верхнем регистре):

```
>>> print("%E %E" % (3000, 18657.81452))
3.000000E+03 1.865781E+04
```

- `g` — эквивалентно `f` или `e` (выбирается более короткая запись числа):

```
>>> print("%g %g %g" % (0.086578, 0.000086578, 1.865E-005))
0.086578 8.6578e-05 1.865e-05
```

- `G` — эквивалентно `f` или `E` (выбирается более короткая запись числа):

```
>>> print("%G %G %G" % (0.086578, 0.000086578, 1.865E-005))
0.086578 8.6578E-05 1.865E-05
```

Если внутри строки необходимо использовать символ процента, этот символ следует удвоить, иначе будет выведено сообщение об ошибке:

```
>>> print("% %s" % ("- это символ процента")) # Ошибка
Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    print("% %s" % ("- это символ процента")) # Ошибка
TypeError: not all arguments converted during string formatting
>>> print("%% %s" % ("- это символ процента")) # Нормально
% — это символ процента
```

Форматирование строк очень удобно использовать при передаче данных в шаблон веб-страницы. Для этого заполняем словарь данными и указываем его справа от символа `%`, а сам шаблон — слева. Продемонстрируем это на примере (листинг 6.1).

Листинг 6.1. Пример использования форматирования строк

```
# -*- coding: utf-8 -*-
html = """<html>
<head><title>%(title)s</title>
</head>
<body>
<h1>%(h1)s</h1>
<div>%(content)s</div>
</body>
</html>"""
arr = {"title":    "Это название документа",
       "h1":       "Это заголовок первого уровня",
       "content":  "Это основное содержание страницы"}
print(html % arr) # Подставляем значения и выводим шаблон
input()
```

Результат выполнения:

```
<html>
<head><title>Это название документа</title>
</head>
```

```
<body>
<h1>Это заголовок первого уровня</h1>
<div>Это основное содержание страницы</div>
</body>
</html>
```

Для форматирования строк также можно использовать следующие методы:

- `expandtabs([<Ширина поля>])` — заменяет символ табуляции пробелами таким образом, чтобы общая ширина фрагмента вместе с текстом, расположенным перед символом табуляции, была равна указанной величине. Если параметр не указан, то ширина поля предполагается равной 8 символам:

  ```
  >>> s = "1\t12\t123\t"
  >>> "'%s'" % s.expandtabs(4)
  "'1   12  123 '"
  ```

  В этом примере ширина задана равной четырем символам. Поэтому во фрагменте `1\t` табуляция будет заменена тремя пробелами, во фрагменте `12\t` — двумя пробелами, а во фрагменте `123\t` — одним пробелом. Во всех трех фрагментах ширина будет равна четырем символам.

  Если перед символом табуляции нет текста или количество символов перед табуляцией равно указанной в вызове метода ширине, то табуляция заменяется указанным количеством пробелов:

  ```
  >>> s = "\t"
  >>> "'%s' - '%s'" % (s.expandtabs(), s.expandtabs(4))
  "'        ' - '    '"
  >>> s = "1234\t"
  >>> "'%s'" % s.expandtabs(4)
  "'1234    '"
  ```

  Если количество символов перед табуляцией больше ширины, то табуляция заменяется пробелами таким образом, чтобы ширина фрагмента вместе с текстом делилась без остатка на указанную ширину:

  ```
  >>> s = "12345\t123456\t1234567\t1234567890\t"
  >>> "'%s'" % s.expandtabs(4)
  "'12345   123456  1234567 1234567890  '"
  ```

  Таким образом, если количество символов перед табуляцией больше 4, но менее 8, то фрагмент дополняется пробелами до 8 символов. Если количество символов больше 8, но менее 12, то фрагмент дополняется пробелами до 12 символов и т. д. Все это справедливо при указании в качестве параметра числа 4;

- `center(<Ширина>[, <Символ>])` — производит выравнивание строки по центру внутри поля указанной ширины. Если второй параметр не указан, справа и слева от исходной строки будут добавлены пробелы:

  ```
  >>> s = "str"
  >>> s.center(15), s.center(11, "-")
  ('      str      ', '----str----')
  ```

  Теперь произведем выравнивание трех фрагментов шириной 15 символов. Первый фрагмент будет выровнен по правому краю, второй — по левому, а третий — по центру:

```
>>> s = "str"
>>> "'%15s' '%-15s' '%s'" % (s, s, s.center(15))
"'            str' 'str            ' '      str      '"
```

Если количество символов в строке превышает ширину поля, то значение ширины игнорируется, и строка возвращается полностью:

```
>>> s = "string"
>>> s.center(6), s.center(5)
('string', 'string')
```

- `ljust(<Ширина>[, <Символ>])` — производит выравнивание строки по левому краю внутри поля указанной ширины. Если второй параметр не указан, справа от исходной строки будут добавлены пробелы. Если количество символов в строке превышает ширину поля, значение ширины игнорируется, и строка возвращается полностью:

```
>>> s = "string"
>>> s.ljust(15), s.ljust(15, "-")
('string         ', 'string---------')
>>> s.ljust(6), s.ljust(5)
('string', 'string')
```

- `rjust(<Ширина>[, <Символ>])` — производит выравнивание строки по правому краю внутри поля указанной ширины. Если второй параметр не указан, слева от исходной строки будут добавлены пробелы. Если количество символов в строке превышает ширину поля, значение ширины игнорируется, и строка возвращается полностью:

```
>>> s = "string"
>>> s.rjust(15), s.rjust(15, "-")
('         string', '---------string')
>>> s.rjust(6), s.rjust(5)
('string', 'string')
```

- `zfill(<Ширина>)` — производит выравнивание фрагмента по правому краю внутри поля указанной ширины. Слева от фрагмента будут добавлены нули. Если количество символов в строке превышает ширину поля, значение ширины игнорируется, и строка возвращается полностью:

```
>>> "5".zfill(20), "123456".zfill(5)
('00000000000000000005', '123456')
```

6.5. Метод *format()*

Помимо операции форматирования, мы можем использовать для этой же цели метод `format()`. Он имеет следующий синтаксис:

```
<Строка> = <Строка специального формата>.format(*args, **kwargs)
```

В параметре `<Строка специального формата>` внутри символов фигурных скобок `{` и `}` указываются спецификаторы, имеющие следующий синтаксис:

```
{[<Поле>][!<Функция>][:<Формат>]}
```

Все символы, расположенные вне фигурных скобок, выводятся без преобразований. Если внутри строки необходимо использовать символы `{` и `}`, то эти символы следует удвоить, иначе возбуждается исключение `ValueError`:

```
>>> print("Символы {{ и }} — {0}".format("специальные"))
Символы { и } — специальные
```

♦ В качестве параметра <Поле> можно указать порядковый номер (нумерация начинается с нуля) или ключ параметра, указанного в методе `format()`. Допустимо комбинировать позиционные и именованные параметры, при этом именованные параметры следует указать последними:

```
>>> "{0} - {1} - {2}".format(10, 12.3, "string")      # Индексы
'10 - 12.3 - string'
>>> arr = [10, 12.3, "string"]
>>> "{0} - {1} - {2}".format(*arr)                    # Индексы
'10 - 12.3 - string'
>>> "{model} - {color}".format(color="red", model="BMW") # Ключи
'BMW - red'
>>> d = {"color": "red", "model": "BMW"}
>>> "{model} - {color}".format(**d)                   # Ключи
'BMW - red'
>>> "{color} - {0}".format(2015, color="red")         # Комбинация
'red - 2015'
```

В вызове метода `format()` можно указать список, словарь или объект. Для доступа к элементам списка по индексу внутри строки формата применяются квадратные скобки, а для доступа к элементам словаря или атрибутам объекта используется точечная нотация:

```
>>> arr = [10, [12.3, "string"] ]
>>> "{0[0]} - {0[1][0]} - {0[1][1]}".format(arr)      # Индексы
'10 - 12.3 - string'
>>> "{arr[0]} - {arr[1][1]}".format(arr=arr)          # Индексы
'10 - string'
>>> class Car: color, model = "red", "BMW"

>>> car = Car()
>>> "{0.model} - {0.color}".format(car)               # Атрибуты
'BMW - red'
```

Существует также краткая форма записи, при которой <Поле> не указывается. В этом случае скобки без указанного индекса нумеруются слева направо, начиная с нуля:

```
>>> "{} - {} - {} - {n}".format(1, 2, 3, n=4) # "{0} - {1} - {2} - {n}"
'1 - 2 - 3 - 4'
>>> "{} - {} - {n} - {}".format(1, 2, 3, n=4) # "{0} - {1} - {n} - {2}"
'1 - 2 - 4 - 3'
```

♦ Параметр <Функция> задает функцию, с помощью которой обрабатываются данные перед вставкой в строку. Если указано значение `s`, то данные обрабатываются функцией `str()`, если значение `r`, то функцией `repr()`, а если значение `a`, то функцией `ascii()`. Если параметр не указан, для преобразования данных в строку используется функция `str()`:

```
>>> print("{0!s}".format("строка"))                   # str()
строка
>>> print("{0!r}".format("строка"))                   # repr()
'строка'
>>> print("{0!a}".format("строка"))                   # ascii()
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

- В параметре `<Формат>` указывается значение, имеющее следующий синтаксис:

  ```
  [[<Заполнитель>]<Выравнивание>][<Знак>][#][0][<Ширина>][,]
  [.<Точность>][<Преобразование>]
  ```

  - Параметр `<Ширина>` задает минимальную ширину поля. Если строка не помещается в указанную ширину, то значение игнорируется, и строка выводится полностью:

    ```
    >>> "'{0:10}' '{1:3}'".format(3, "string")
    "'         3' 'string'"
    ```

    Ширину поля можно передать в качестве параметра в методе `format()`. В этом случае вместо числа указывается индекс параметра внутри фигурных скобок:

    ```
    >>> "'{0:{1}}'".format(3, 10) # 10 — это ширина поля
    "'         3'"
    ```

  - Параметр `<Выравнивание>` управляет выравниванием значения внутри поля. Поддерживаются следующие значения:

    - `<` — по левому краю;
    - `>` — по правому краю (поведение по умолчанию);
    - `^` — по центру поля. Пример:

      ```
      >>> "'{0:<10}' '{1:>10}' '{2:^10}'".format(3, 3, 3)
      "'3         ' '         3' '    3     '"
      ```

    - `=` — знак числа выравнивается по левому краю, а число по правому краю:

      ```
      >>> "'{0:=10}' '{1:=10}'".format(-3, 3)
      "'-        3' '         3'"
      ```

      Как видно из приведенного примера, пространство между знаком и числом по умолчанию заполняется пробелами, а знак положительного числа не указывается. Чтобы вместо пробелов пространство заполнялось нулями, необходимо указать нуль перед шириной поля:

      ```
      >>> "'{0:=010}' '{1:=010}'".format(-3, 3)
      "'-000000003' '0000000003'"
      ```

  - Параметр `<Заполнитель>` задает символ, которым будет заполняться свободное пространство в поле (по умолчанию — пробел). Такого же эффекта можно достичь, указав нуль в параметре `<Заполнитель>`:

    ```
    >>> "'{0:0=10}' '{1:0=10}'".format(-3, 3)
    "'-000000003' '0000000003'"
    >>> "'{0:*<10}' '{1:+>10}' '{2:.^10}'".format(3, 3, 3)
    "'3*********' '+++++++++3' '....3.....'"
    ```

  - Параметр `<Знак>` управляет выводом знака числа. Допустимые значения:

    - `+` — задает обязательный вывод знака как для отрицательных, так и для положительных чисел;
    - `-` — вывод знака только для отрицательных чисел (значение по умолчанию);
    - `пробел` — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Пример:

```
>>> "'{0:+}' '{1:+}' '{0:-}' '{1:-}'".format(3, -3)
"'+3' '-3' '3' '-3'"
>>> "'{0: }' '{1: }'".format(3, -3)        # Пробел
"' 3' '-3'"
```

- Для целых чисел в параметре `<Преобразование>` могут быть указаны следующие опции:
  - `b` — преобразование в двоичную систему счисления:

    ```
    >>> "'{0:b}' '{0:#b}'".format(3)
    "'11' '0b11'"
    ```

  - `c` — преобразование числа в соответствующий символ:

    ```
    >>> "'{0:c}'".format(100)
    "'d'"
    ```

  - `d` — преобразование в десятичную систему счисления;
  - `n` — аналогично опции `d`, но учитывает настройки локали. Например, выведем большое число с разделением тысячных разрядов пробелом:

    ```
    >>> import locale
    >>> locale.setlocale(locale.LC_NUMERIC, 'Russian_Russia.1251')
    'Russian_Russia.1251'
    >>> print("{0:n}".format(100000000).replace("\uffa0", " "))
    100 000 000
    ```

    В Python 3 между разрядами вставляется символ с кодом `\uffa0`, который отображается квадратиком. Чтобы вывести символ пробела, мы производим замену в строке с помощью метода `replace()`. В Python версии 2 поведение было другим. Там вставлялся символ с кодом `\xa0` и не нужно было производить замену. Чтобы в Python 3 вставлялся символ с кодом `\xa0`, следует воспользоваться функцией `format()` из модуля `locale`:

    ```
    >>> import locale
    >>> locale.setlocale(locale.LC_NUMERIC, "Russian_Russia.1251")
    'Russian_Russia.1251'
    >>> print(locale.format("%d", 100000000, grouping = True))
    100 000 000
    >>> locale.localeconv()["thousands_sep"]
    '\xa0'
    ```

    Можно также разделить тысячные разряды запятой, указав ее в строке формата:

    ```
    >>> print("{0:,d}".format(100000000))
    100,000,000
    ```

  - `o` — преобразование в восьмеричную систему счисления:

    ```
    >>> "'{0:d}' '{0:o}' '{0:#o}'".format(511)
    "'511' '777' '0o777'"
    ```

  - `x` — преобразование в шестнадцатеричную систему счисления в нижнем регистре:

    ```
    >>> "'{0:x}' '{0:#x}'".format(255)
    "'ff' '0xff'"
    ```

- `X` — преобразование в шестнадцатеричную систему счисления в верхнем регистре:

```
>>> "'{0:X}' '{0:#X}'".format(255)
"'FF' '0XFF'"
```

- Для вещественных чисел в параметре `<Преобразование>` могут быть указаны следующие опции:

  - `f` и `F` — преобразование в десятичную систему счисления:

```
>>> "'{0:f}' '{1:f}' '{2:f}'".format(30, 18.6578145, -2.5)
"'30.000000' '18.657815' '-2.500000'"
```

По умолчанию выводимое число имеет шесть знаков после запятой. Задать другое количество знаков после запятой мы можем в параметре `<Точность>`:

```
>>> "'{0:.7f}' '{1:.2f}'".format(18.6578145, -2.5)
"'18.6578145' '-2.50'"
```

  - `e` — вывод в экспоненциальной форме (буква `e` в нижнем регистре):

```
>>> "'{0:e}' '{1:e}'".format(3000, 18657.81452)
"'3.000000e+03' '1.865781e+04'"
```

  - `E` — вывод в экспоненциальной форме (буква `E` в верхнем регистре):

```
>>> "'{0:E}' '{1:E}'".format(3000, 18657.81452)
"'3.000000E+03' '1.865781E+04'"
```

Здесь по умолчанию количество знаков после запятой также равно шести, но мы можем указать другую величину этого параметра:

```
>>> "'{0:.2e}' '{1:.2E}'".format(3000, 18657.81452)
"'3.00e+03' '1.87E+04'"
```

  - `g` — эквивалентно `f` или `e` (выбирается более короткая запись числа):

```
>>> "'{0:g}' '{1:g}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578e-05'"
```

  - `n` — аналогично опции `g`, но учитывает настройки локали;
  - `G` — эквивалентно `f` или `E` (выбирается более короткая запись числа):

```
>>> "'{0:G}' '{1:G}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578E-05'"
```

  - `%` — умножает число на 100 и добавляет символ процента в конец. Значение отображается в соответствии с опцией `f`:

```
>>> "'{0:%}' '{1:.4%}'".format(0.086578, 0.000086578)
"'8.657800%' '0.0087%'"
```

6.5.1. Форматируемые строки

В Python 3.6 появилась весьма удобная альтернатива методу `format()` — *форматируемые строки*.

Форматируемая строка обязательно должна предваряться буквой `f` или `F`. В нужных местах такой строки записываются команды на вставку в эти места значений, хранящихся в пере-

менных, — точно так же, как и в строках специального формата, описанных ранее. Такие команды имеют следующий синтаксис:

```
{[<Переменная>][!<Функция>][:<Формат>]}
```

Параметр `<Переменная>` задает имя переменной, из которой будет извлечено вставляемое в строку значение. Вместо имени переменной можно записать выражение, вычисляющее значение, которое нужно вывести. Параметры `<Функция>` и `<Формат>` имеют то же назначение и записываются так же, как и в случае метода `format()`:

```
>>> a = 10; b = 12.3; s = "string"
>>> f"{a} - {b} - {s}"                  # Простой вывод чисел и строк
'10 - 12.3 - string'
>>> f"{a} - {b:5.2f} - {s}"             # Вывод с форматированием
'10 - 12.30 - string'
>>> d = 3
>>> f"{a} - {b:5.{d}f} - {s}"           # В командах можно использовать
                                        # значения из переменных
'10 - 12.300 - string'
>>> arr = [3, 4]
>>> f"{arr[0]} - {arr[1]}"              # Вывод элементов массива
'3 - 4'
>>> f"{arr[0]} - {arr[1] * 2}"          # Использование выражений
'3 - 8'
```

6.6. Функции и методы для работы со строками

Рассмотрим основные функции для работы со строками:

- `str([<Объект>])` — преобразует любой объект в строку. Если параметр не указан, возвращается пустая строка. Используется функцией `print()` для вывода объектов:

  ```
  >>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
  ('', '[1, 2]', '(3, 4)', "{'x': 1}")
  >>> print("строка1\nстрока2")
  строка1
  строка2
  ```

- `repr(<Объект>)` — возвращает строковое представление объекта. Используется при выводе объектов в окне **Python Shell** редактора IDLE:

  ```
  >>> repr("Строка"), repr([1, 2, 3]), repr({"x": 5})
  ("'Строка'", '[1, 2, 3]', "{'x': 5}")
  >>> repr("строка1\nстрока2")
  "'строка1\\nстрока2'"
  ```

- `ascii(<Объект>)` — возвращает строковое представление объекта. В строке могут быть символы только из кодировки ASCII:

  ```
  >>> ascii([1, 2, 3]), ascii({"x": 5})
  ('[1, 2, 3]', "{'x': 5}")
  >>> ascii("строка")
  "'\\u0441\\u0442\\u0440\\u043e\\u043a\\u0430'"
  ```

♦ `len(<Строка>)` — возвращает длину строки — количество символов в ней:

```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
(6, 3, 6)
>>> len("строка")
6
```

Приведем перечень основных методов для работы со строками:

♦ `strip([<Символы>])` — удаляет указанные в параметре символы в начале и в конце строки. Если параметр не задан, удаляются пробельные символы: пробел, символ перевода строки (`\n`), символ возврата каретки (`\r`), символы горизонтальной (`\t`) и вертикальной (`\v`) табуляции:

```
>>> s1, s2 = "     str\n\r\v\t", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.strip(), s2.strip("tsr"))
"'str' — 'ok'"
```

♦ `lstrip([<Символы>])` — удаляет пробельные или указанные символы в начале строки:

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.lstrip(), s2.lstrip("tsr"))
"'str     ' — 'okstrstrstr'"
```

♦ `rstrip([<Символы>])` — удаляет пробельные или указанные символы в конце строки:

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.rstrip(), s2.rstrip("tsr"))
"'     str' — 'strstrstrok'"
```

♦ `split([<Разделитель>[, <Лимит>]])` — разделяет строку на подстроки по указанному разделителю и добавляет эти подстроки в список, который возвращается в качестве результата. Если первый параметр не указан или имеет значение `None`, то в качестве разделителя используется символ пробела. Во втором параметре можно задать количество подстрок в результирующем списке — если он не указан или равен `-1`, в список попадут все подстроки. Если подстрок больше указанного количества, то список будет содержать еще один элемент — с остатком строки:

```
>>> s = "word1 word2 word3"
>>> s.split(), s.split(None, 1)
(['word1', 'word2', 'word3'], ['word1', 'word2 word3'])
>>> s = "word1\nword2\nword3"
>>> s.split("\n")
['word1', 'word2', 'word3']
```

Если в строке содержатся несколько пробелов подряд и разделитель не указан, то пустые элементы не будут добавлены в список:

```
>>> s = "word1           word2 word3     "
>>> s.split()
['word1', 'word2', 'word3']
```

При использовании другого разделителя могут возникнуть пустые элементы:

```
>>> s = ",,word1,,word2,,word3,,"
>>> s.split(",")
['', '', 'word1', '', 'word2', '', 'word3', '', '']
>>> "1,,2,,3".split(",")
['1', '', '2', '', '3']
```

Если разделитель не найден в строке, то список будет состоять из одного элемента, представляющего исходную строку:

```
>>> "word1 word2 word3".split("\n")
['word1 word2 word3']
```

♦ `rsplit([<Разделитель>[, <Лимит>]])` — аналогичен методу `split()`, но поиск символа-разделителя производится не слева направо, а наоборот — справа налево:

```
>>> s = "word1 word2 word3"
>>> s.rsplit(), s.rsplit(None, 1)
(['word1', 'word2', 'word3'], ['word1 word2', 'word3'])
>>> "word1\nword2\nword3".rsplit("\n")
['word1', 'word2', 'word3']
```

♦ `splitlines([False])` — разделяет строку на подстроки по символу перевода строки (`\n`) и добавляет их в список. Символы новой строки включаются в результат, только если необязательный параметр имеет значение `True`. Если разделитель не найден в строке, список будет содержать только один элемент:

```
>>> "word1\nword2\nword3".splitlines()
['word1', 'word2', 'word3']
>>> "word1\nword2\nword3".splitlines(True)
['word1\n', 'word2\n', 'word3']
>>> "word1\nword2\nword3".splitlines(False)
['word1', 'word2', 'word3']
>>> "word1 word2 word3".splitlines()
['word1 word2 word3']
```

♦ `partition(<Разделитель>)` — находит первое вхождение символа-разделителя в строку и возвращает кортеж из трех элементов: первый элемент будет содержать фрагмент, расположенный перед разделителем, второй элемент — сам разделитель, а третий элемент — фрагмент, расположенный после разделителя. Поиск производится слева направо. Если символ-разделитель не найден, то первый элемент кортежа будет содержать всю строку, а остальные элементы останутся пустыми:

```
>>> "word1 word2 word3".partition(" ")
('word1', ' ', 'word2 word3')
>>> "word1 word2 word3".partition("\n")
('word1 word2 word3', '', '')
```

♦ `rpartition(<Разделитель>)` — метод аналогичен методу `partition()`, но поиск символа-разделителя производится не слева направо, а наоборот — справа налево. Если символ-разделитель не найден, то первые два элемента кортежа окажутся пустыми, а третий элемент будет содержать всю строку:

```
>>> "word1 word2 word3".rpartition(" ")
('word1 word2', ' ', 'word3')
>>> "word1 word2 word3".rpartition("\n")
('', '', 'word1 word2 word3')
```

♦ `join()` — преобразует последовательность в строку. Элементы добавляются через указанный разделитель. Формат метода:

```
<Строка> = <Разделитель>.join(<Последовательность>)
```

В качестве примера преобразуем список и кортеж в строку:

```
>>> " => ".join(["word1", "word2", "word3"])
'word1 => word2 => word3'
>>> " ".join(("word1", "word2", "word3"))
'word1 word2 word3'
```

Обратите внимание на то, что элементы последовательностей должны быть строками, иначе возбуждается исключение `TypeError`:

```
>>> " ".join(("word1", "word2", 5))
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    " ".join(("word1", "word2", 5))
TypeError: sequence item 2: expected str instance, int found
```

Как вы уже знаете, строки относятся к неизменяемым типам данных. Если попытаться изменить символ по индексу, возникнет ошибка. Чтобы изменить символ по индексу, можно преобразовать строку в список с помощью функции `list()`, произвести изменения, а затем с помощью метода `join()` преобразовать список обратно в строку:

```
>>> s = "Python"
>>> arr = list(s); arr        # Преобразуем строку в список
['P', 'y', 't', 'h', 'o', 'n']
>>> arr[0] = "J"; arr         # Изменяем элемент по индексу
['J', 'y', 't', 'h', 'o', 'n']
>>> s = "".join(arr); s       # Преобразуем список в строку
'Jython'
```

В Python 3 также можно преобразовать строку в тип `bytearray`, а затем изменить символ по индексу:

```
>>> s = "Python"
>>> b = bytearray(s, "cp1251"); b
bytearray(b'Python')
>>> b[0] = ord("J"); b
bytearray(b'Jython')
>>> s = b.decode("cp1251"); s
'Jython'
```

6.7. Настройка локали

Для установки *локали* (совокупности локальных настроек системы) служит функция `setlocale()` из модуля `locale`. Прежде чем использовать функцию, необходимо подключить модуль с помощью выражения:

```
import locale
```

Функция `setlocale()` имеет следующий формат:

```
setlocale(<Категория>[, <Локаль>]);
```

Параметр `<Категория>` может принимать следующие значения:

- `locale.LC_ALL` — устанавливает локаль для всех режимов;
- `locale.LC_COLLATE` — для сравнения строк;

- `locale.LC_CTYPE` — для перевода символов в нижний или верхний регистр;
- `locale.LC_MONETARY` — для отображения денежных единиц;
- `locale.LC_NUMERIC` — для форматирования чисел;
- `locale.LC_TIME` — для форматирования вывода даты и времени.

Получить текущее значение локали позволяет функция `getlocale([<Категория>])`. В качестве примера настроим локаль под Windows вначале на кодировку Windows-1251, потом на кодировку UTF-8, а затем на кодировку по умолчанию. Далее выведем текущее значение локали для всех категорий и только для `locale.LC_COLLATE`:

```
>>> import locale
>>> # Для кодировки windows-1251
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> # Устанавливаем локаль по умолчанию
>>> locale.setlocale(locale.LC_ALL, "")
'Russian_Russia.1251'
>>> # Получаем текущее значение локали для всех категорий
>>> locale.getlocale()
('Russian_Russia', '1251')
>>> # Получаем текущее значение категории locale.LC_COLLATE
>>> locale.getlocale(locale.LC_COLLATE)
('Russian_Russia', '1251')
```

Получить настройки локали позволяет функция `localeconv()`. Функция возвращает словарь с настройками. Результат выполнения функции для локали `Russian_Russia.1251` выглядит следующим образом:

```
>>> locale.localeconv()
{'decimal_point': ',', 'thousands_sep': '\xa0', 'grouping': [3, 0],
'int_curr_symbol': 'RUB', 'currency_symbol': '?', 'mon_decimal_point': ',',
'mon_thousands_sep': '\xa0', 'mon_grouping': [3, 0], 'positive_sign': '',
'negative_sign': '-', 'int_frac_digits': 2, 'frac_digits': 2, 'p_cs_precedes': 0,
'p_sep_by_space': 1, 'n_cs_precedes': 0, 'n_sep_by_space': 1, 'p_sign_posn': 1,
'n_sign_posn': 1}
```

6.8. Изменение регистра символов

Для изменения регистра символов предназначены следующие методы:

- `upper()` — заменяет все символы строки соответствующими прописными буквами:

  ```
  >>> print("строка".upper())
  СТРОКА
  ```

- `lower()` — заменяет все символы строки соответствующими строчными буквами:

  ```
  >>> print("СТРОКА".lower())
  строка
  ```

- `swapcase()` — заменяет все строчные символы соответствующими прописными буквами, а все прописные символы — строчными:

```
>>> print("СТРОКА строка".swapcase())
строка СТРОКА
```

- `capitalize()` — делает первую букву строки прописной:

```
>>> print("строка строка".capitalize())
Строка строка
```

- `title()` — делает первую букву каждого слова прописной:

```
>>> s = "первая буква каждого слова станет прописной"
>>> print(s.title())
Первая Буква Каждого Слова Станет Прописной
```

- `casefold()` — то же самое, что и `lower()`, но дополнительно преобразует все символы с диакритическими знаками и лигатуры в буквы стандартной латиницы. Обычно применяется для сравнения строк:

```
>>> "Python".casefold() == "python".casefold()
True
>>> "grosse".casefold() == "große".casefold()
True
```

6.9. Функции для работы с символами

Для работы с отдельными символами предназначены следующие функции:

- `chr(<Код символа>)` — возвращает символ по указанному коду:

```
>>> print(chr(1055))
П
```

- `ord(<Символ>)` — возвращает код указанного символа:

```
>>> print(ord("П"))
1055
```

6.10. Поиск и замена в строке

Для поиска и замены в строке используются следующие методы:

- `find()` — ищет подстроку в строке. Возвращает номер позиции, с которой начинается вхождение подстроки в строку. Если подстрока в строку не входит, то возвращается значение `-1`. Метод зависит от регистра символов. Формат метода:

```
<Строка>.find(<Подстрока>[, <Начало>[, <Конец>]])
```

Если начальная позиция не указана, то поиск будет осуществляться с начала строки. Если параметры `<Начало>` и `<Конец>` указаны, то производится операция извлечения среза:

```
<Строка>[<Начало>:<Конец>]
```

и поиск подстроки будет выполняться в этом фрагменте:

```
>>> s = "пример пример Пример"
>>> s.find("при"), s.find("При"), s.find("тест")
(0, 14, -1)
>>> s.find("при", 9), s.find("при", 0, 6), s.find("при", 7, 12)
(-1, 0, 7)
```

- `index()` — метод аналогичен методу `find()`, но если подстрока в строку не входит, возбуждается исключение `ValueError`. Формат метода:

  ```
  <Строка>.index(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Пример:

  ```
  >>> s = "пример пример Пример"
  >>> s.index("при"), s.index("при", 7, 12), s.index("При", 1)
  (0, 7, 14)
  >>> s.index("тест")
  Traceback (most recent call last):
    File "<pyshell#24>", line 1, in <module>
      s.index("тест")
  ValueError: substring not found
  ```

- `rfind()` — ищет подстроку в строке. Возвращает позицию последнего вхождения подстроки в строку. Если подстрока в строку не входит, возвращается значение `-1`. Метод зависит от регистра символов. Формат метода:

  ```
  <Строка>.rfind(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Если начальная позиция не указана, то поиск будет производиться с начала строки. Если параметры `<Начало>` и `<Конец>` указаны, то выполняется операция извлечения среза, и поиск подстроки будет производиться в этом фрагменте:

  ```
  >>> s = "пример пример Пример Пример"
  >>> s.rfind("при"), s.rfind("При"), s.rfind("тест")
  (7, 21, -1)
  >>> s.find("при", 0, 6), s.find("При", 10, 20)
  (0, 14)
  ```

- `rindex()` — метод аналогичен методу `rfind()`, но если подстрока в строку не входит, возбуждается исключение `ValueError`. Формат метода:

  ```
  <Строка>.rindex(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Пример:

  ```
  >>> s = "пример пример Пример Пример"
  >>> s.rindex("при"), s.rindex("При"), s.rindex("при", 0, 6)
  (7, 21, 0)
  >>> s.rindex("тест")
  Traceback (most recent call last):
    File "<pyshell#30>", line 1, in <module>
      s.rindex("тест")
  ValueError: substring not found
  ```

- `count()` — возвращает число вхождений подстроки в строку. Если подстрока в строку не входит, возвращается значение `0`. Метод зависит от регистра символов. Формат метода:

  ```
  <Строка>.count(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Пример:

  ```
  >>> s = "пример пример Пример Пример"
  >>> s.count("при"), s.count("при", 6), s.count("При")
  (2, 1, 2)
  ```

```
>>> s.count("тест")
0
```

♦ `startswith()` — проверяет, начинается ли строка с указанной подстроки. Если начинается, возвращается значение `True`, в противном случае — `False`. Метод зависит от регистра символов. Формат метода:

```
<Строка>.startswith(<Подстрока>[, <Начало>[, <Конец>]])
```

Если начальная позиция не указана, сравнение будет производиться с началом строки. Если параметры `<Начало>` и `<Конец>` указаны, выполняется операция извлечения среза, и сравнение будет производиться с началом фрагмента:

```
>>> s = "пример пример Пример Пример"
>>> s.startswith("при"), s.startswith("При")
(True, False)
>>> s.startswith("при", 6), s.startswith("При", 14)
(False, True)
```

Начиная с Python версии 2.5, параметр `<Подстрока>` может быть кортежем:

```
>>> s = "пример пример Пример Пример"
>>> s.startswith(("при", "При"))
True
```

♦ `endswith()` — проверяет, заканчивается ли строка указанной подстрокой. Если заканчивается, то возвращается значение `True`, в противном случае — `False`. Метод зависит от регистра символов. Формат метода:

```
<Строка>.endswith(<Подстрока>[, <Начало>[, <Конец>]])
```

Если начальная позиция не указана, то сравнение будет производиться с концом строки. Если параметры `<Начало>` и `<Конец>` указаны, то выполняется операция извлечения среза, и сравнение будет производиться с концом фрагмента:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith("ока"), s.endswith("ОКА")
(False, True)
>>> s.endswith("ока", 0, 9)
True
```

Начиная с Python версии 2.5, параметр `<Подстрока>` может быть кортежем:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith(("ока", "ОКА"))
True
```

♦ `replace()` — производит замену всех вхождений заданной подстроки в строке на другую подстроку и возвращает результат в виде новой строки. Метод зависит от регистра символов. Формат метода:

```
<Строка>.replace(<Подстрока для замены>, <Новая подстрока>[,
                 <Максимальное количество замен>])
```

Если количество замен не указано, будет выполнена замена всех найденных подстрок:

```
>>> s = "Привет, Петя"
>>> print(s.replace("Петя", "Вася"))
Привет, Вася
```

```
>>> print(s.replace("петя", "вася")) # Зависит от регистра
Привет, Петя
>>> s = "strstrstrstrstr"
>>> s.replace("str", ""), s.replace("str", "", 3)
('', 'strstr')
```

- `translate(<Таблица символов>)` — заменяет символы в соответствии с параметром `<Таблица символов>`. Параметр `<Таблица символов>` должен быть словарем, ключами которого являются Unicode-коды заменяемых символов, а значениями — Unicode-коды заменяющих символов. Если в качестве значения указать `None`, то символ будет удален. Для примера удалим букву `П`, а также изменим регистр всех букв `р`:

```
>>> s = "Пример"
>>> d = {ord("П"): None, ord("р"): ord("Р")}
>>> d
{1088: 1056, 1055: None}
>>> s.translate(d)
'РимеР'
```

Упростить создание параметра `<Таблица символов>` позволяет статический метод `maketrans()`. Формат метода:

```
str.maketrans(<X>[, <Y>[, <Z>]])
```

Если указан только первый параметр, то он должен быть словарем:

```
>>> t = str.maketrans({"а": "А", "о": "О", "с": None})
>>> t
{1072: 'А', 1089: None, 1086: 'О'}
>>> "строка".translate(t)
'трОкА'
```

Если указаны два первых параметра, то они должны быть строками одинаковой длины. В результате будет создан словарь с ключами из строки `<X>` и значениями из строки `<Y>`, расположенными в той же позиции. Изменим регистр некоторых символов:

```
>>> t = str.maketrans("абвгдежзи", "АБВГДЕЖЗИ")
>>> t
{1072: 1040, 1073: 1041, 1074: 1042, 1075: 1043, 1076: 1044,
1077: 1045, 1078: 1046, 1079: 1047, 1080: 1048}
>>> "абвгдежзи".translate(t)
'АБВГДЕЖЗИ'
```

В третьем параметре можно дополнительно указать строку из символов, которым будет сопоставлено значение `None`. После выполнения метода `translate()` эти символы будут удалены из строки. Заменим все цифры на `0`, а некоторые буквы удалим из строки:

```
>>> t = str.maketrans("123456789", "0" * 9, "str")
>>> t
{116: None, 115: None, 114: None, 49: 48, 50: 48, 51: 48,
52: 48, 53: 48, 54: 48, 55: 48, 56: 48, 57: 48}
>>> "str123456789str".translate(t)
'000000000'
```

6.11. Проверка типа содержимого строки

Для проверки типа содержимого предназначены следующие методы:

- `isalnum()` — возвращает `True`, если строка содержит только буквы и (или) цифры, в противном случае — `False`. Если строка пустая, возвращается значение `False`:

```
>>> "0123".isalnum(), "123abc".isalnum(), "abc123".isalnum()
(True, True, True)
>>> "строка".isalnum()
True
>>> "".isalnum(), "123 abc".isalnum(), "abc, 123.".isalnum()
(False, False, False)
```

- `isalpha()` — возвращает `True`, если строка содержит только буквы, в противном случае — `False`. Если строка пустая, возвращается значение `False`:

```
>>> "string".isalpha(), "строка".isalpha(), "".isalpha()
(True, True, False)
>>> "123abc".isalpha(), "str str".isalpha(), "st,st".isalpha()
(False, False, False)
```

- `isdigit()` — возвращает `True`, если строка содержит только цифры, в противном случае — `False`:

```
>>> "0123".isdigit(), "123abc".isdigit(), "abc123".isdigit()
(True, False, False)
```

- `isdecimal()` — возвращает `True`, если строка содержит только десятичные символы, в противном случае — `False`. К десятичным символам относятся десятичные цифры в кодировке ASCII, а также надстрочные и подстрочные десятичные цифры в других языках:

```
>>> "123".isdecimal(), "123стр".isdecimal()
(True, False)
```

- `isnumeric()` — возвращает `True`, если строка содержит только числовые символы, в противном случае — `False`. К числовым символам относятся десятичные цифры в кодировке ASCII, символы римских чисел, дробные числа и др.:

```
>>> "\u2155".isnumeric(), "\u2155".isdigit()
(True, False)
>>> print("\u2155") # Выведет символ "1/5"
```

- `isupper()` — возвращает `True`, если строка содержит буквы только верхнего регистра, в противном случае — `False`:

```
>>> "STRING".isupper(), "СТРОКА".isupper(), "".isupper()
(True, True, False)
>>> "STRING1".isupper(), "СТРОКА, 123".isupper(), "123".isupper()
(True, True, False)
>>> "string".isupper(), "STRing".isupper()
(False, False)
```

- `islower()` — возвращает `True`, если строка содержит буквы только нижнего регистра, в противном случае — `False`:

```
>>> "srting".islower(), "строка".islower(), "".islower()
(True, True, False)
>>> "string1".islower(), "str, 123".islower(), "123".islower()
(True, True, False)
>>> "STRING".islower(), "Строка".islower()
(False, False)
```

- `istitle()` — возвращает `True`, если все слова в строке начинаются с заглавной буквы, в противном случае — `False`. Если строка пустая, также возвращается `False`:

```
>>> "Str Str".istitle(), "Стр Стр".istitle()
(True, True)
>>> "Str Str 123".istitle(), "Стр Стр 123".istitle()
(True, True)
>>> "Str str".istitle(), "Стр стр".istitle()
(False, False)
>>> "".istitle(), "123".istitle()
(False, False)
```

- `isprintable()` — возвращает `True`, если строка содержит только печатаемые символы, в противном случае — `False`. Отметим, что пробел относится к печатаемым символам:

```
>>> "123".isprintable()
True
>>> "PHP Python".isprintable()
True
>>> "\n".isprintable()
False
```

- `isspace()` — возвращает `True`, если строка содержит только пробельные символы, в противном случае — `False`:

```
>>> "".isspace(), " \n\r\t".isspace(), "str str".isspace()
(False, True, False)
```

- `isidentifier()` — возвращает `True`, если строка представляет собой допустимое с точки зрения Python имя переменной, функции или класса, в противном случае — `False`:

```
>>> "s".isidentifier()
True
>>> "func".isidentifier()
True
>>> "123func".isidentifier()
False
```

Следует иметь в виду, что метод `isidentifier()` лишь проверяет, удовлетворяет ли заданное имя правилам языка. Он не проверяет, совпадает ли это имя с ключевым словом Python. Для этого надлежит применять функцию `iskeyword()`, объявленную в модуле `keyword`, которая возвращает `True`, если переданная ей строка совпадает с одним из ключевых слов:

```
>>> keyword.iskeyword("else")
True
>>> keyword.iskeyword("elsewhere")
False
```

Переделаем нашу программу суммирования произвольного количества целых чисел, введенных пользователем (см. листинг 4.12), таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой. Кроме того, предусмотрим возможность ввода отрицательных целых чисел (листинг 6.2).

Листинг 6.2. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    if x == "stop":
        break # Выход из цикла
    if x == "":
        print("Вы не ввели значение!")
        continue
    if x[0] == "-": # Если первым символом является минус
        if not x[1:].isdigit(): # Если фрагмент не состоит из цифр
            print("Необходимо ввести число, а не строку!")
            continue
    else: # Если минуса нет, то проверяем всю строку
        if not x.isdigit(): # Если строка не состоит из цифр
            print("Необходимо ввести число, а не строку!")
            continue
    x = int(x) # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода значений и получения результата выглядит так (значения, введенные пользователем, выделены здесь полужирным шрифтом):

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число:
Вы не ввели значение!
Введите число: str
Необходимо ввести число, а не строку!
Введите число: -5
Введите число: -str
Необходимо ввести число, а не строку!
Введите число: stop
Сумма чисел равна: 5
```

6.12. Тип данных *bytes*

Тип данных `str` отлично подходит для хранения текста, но что делать, если необходимо обрабатывать изображения? Ведь изображение не имеет кодировки, а значит, оно не может быть преобразовано в Unicode-строку. Для решения этой проблемы в Python 3 были введе-

ны типы `bytes` и `bytearray`, которые позволяют хранить последовательность целых чисел в диапазоне от `0` до `255`, т. е. байтов. Тип данных `bytes` относится к неизменяемым типам, как и строки, а тип данных `bytearray` — к изменяемым, как и списки.

Создать объект типа `bytes` можно следующими способами:

- с помощью функции `bytes([<Строка>, <Кодировка>[, <Обработка ошибок>]])`. Если параметры не указаны, то возвращается пустой объект. Чтобы преобразовать строку в объект типа `bytes`, необходимо передать минимум два первых параметра. Если строка указана только в первом параметре, то возбуждается исключение `TypeError`:

```
>>> bytes()
b''
>>> bytes("строка", "cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка")
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    bytes("строка")
TypeError: string argument without an encoding
```

  В третьем параметре могут быть указаны значения `"strict"` (при ошибке возбуждается исключение `UnicodeEncodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом вопроса) или `"ignore"` (неизвестные символы игнорируются):

```
>>> bytes("string\uFFFD", "cp1251", "strict")
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    bytes("string\uFFFD", "cp1251", "strict")
  File "C:\Python36\lib\encodings\cp1251.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character
'\ufffd' in position 6: character maps to <undefined>
>>> bytes("string\uFFFD", "cp1251", "replace")
b'string?'
>>> bytes("string\uFFFD", "cp1251", "ignore")
b'string'
```

- с помощью строкового метода `encode([encoding="utf-8"][, errors="strict"])`. Если кодировка не указана, строка преобразуется в последовательность байтов в кодировке UTF-8. В параметре `errors` могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"`, `"ignore"`, `"xmlcharrefreplace"` или `"backslashreplace"`:

```
>>> "строка".encode()
b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
>>> "строка".encode(encoding="cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> "строка\uFFFD".encode(encoding="cp1251",
                          errors="xmlcharrefreplace")
b'\xf1\xf2\xf0\xee\xea\xe0&#65533;'
>>> "строка\uFFFD".encode(encoding="cp1251",
                          errors="backslashreplace")
b'\xf1\xf2\xf0\xee\xea\xe0\\ufffd'
```

♦ указав букву b (регистр не имеет значения) перед строкой в апострофах, кавычках, тройных апострофах или тройных кавычках. Обратите внимание на то, что в строке могут быть только символы с кодами, входящими в кодировку ASCII. Все остальные символы должны быть представлены специальными последовательностями:

```
>>> b"string", b'string', b"""string""", b'''string'''
(b'string', b'string', b'string', b'string')
>>> b"строка"
SyntaxError: bytes can only contain ASCII literal characters.
>>> b"\xf1\xf2\xf0\xee\xea\xe0"
b'\xf1\xf2\xf0\xee\xea\xe0'
```

♦ с помощью функции `bytes(<Последовательность>)`, которая преобразует последовательность целых чисел от 0 до 255 в объект типа bytes. Если число не попадает в диапазон, возбуждается исключение ValueError:

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

♦ с помощью функции `bytes(<Число>)`, которая задает количество элементов в последовательности. Каждый элемент будет содержать нулевой символ:

```
>>> bytes(10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

♦ с помощью метода `bytes.fromhex(<Строка>)`. Строка в этом случае должна содержать шестнадцатеричные значения символов:

```
>>> b = bytes.fromhex(" e1 e2e0ae aaa0 ")
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

Объекты типа bytes относятся к последовательностям. Каждый элемент такой последовательности может хранить целое число от 0 до 255, которое обозначает код символа. Как и все последовательности, объекты поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение:

```
>>> b = bytes("string", "cp1251")
>>> b
b'string'
>>> b[0]                      # Обращение по индексу
115
>>> b[1:3]                    # Получение среза
b'tr'
>>> b + b"123"                # Конкатенация
b'string123'
>>> b * 3                     # Повторение
b'stringstringstring'
>>> 115 in b, b"tr" in b, b"as" in b
(True, True, False)
```

Как видно из примера, при выводе объекта целиком, а также при извлечении среза, производится попытка отображения символов. Однако доступ по индексу возвращает целое число, а не символ. Если преобразовать объект в список, то мы получим последовательность целых чисел:

```
>>> list(bytes("string", "cp1251"))
[115, 116, 114, 105, 110, 103]
```

Тип `bytes` относится к неизменяемым типам. Это означает, что можно получить значение по индексу, но изменить его нельзя:

```
>>> b = bytes("string", "cp1251")
>>> b[0] = 168
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    b[0] = 168
TypeError: 'bytes' object does not support item assignment
```

Объекты типа `bytes` поддерживают большинство строковых методов, которые мы рассматривали в предыдущих разделах. Однако некоторые из этих методов могут некорректно работать с русскими буквами — в этих случаях следует использовать тип `str`, а не тип `bytes`. Не поддерживаются объектами типа `bytes` строковые методы `encode()`, `isidentifier()`, `isprintable()`, `isnumeric()`, `isdecimal()`, `format_map()` и `format()`, а также операция форматирования.

При использовании методов следует учитывать, что в параметрах нужно указывать объекты типа `bytes`, а не строки:

```
>>> b = bytes("string", "cp1251")
>>> b.replace(b"s", b"S")
b'String'
```

Необходимо также помнить, что смешивать строки и объекты типа `bytes` в выражениях нельзя. Предварительно необходимо явно преобразовать объекты к одному типу, а лишь затем производить операцию:

```
>>> b"string" + "string"
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    b"string" + "string"
TypeError: can't concat bytes to str
>>> b"string" + "string".encode("ascii")
b'stringstring'
```

Объект типа `bytes` может содержать как однобайтовые, так и многобайтовые символы. При использовании многобайтовых символов некоторые функции могут работать не так, как предполагалось, — например, функция `len()` вернет количество байтов, а не символов:

```
>>> len("строка")
6
>>> len(bytes("строка", "cp1251"))
6
>>> len(bytes("строка", "utf-8"))
12
```

Преобразовать объект типа `bytes` в строку позволяет метод `decode()`. Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр `encoding` задает кодировку символов (по умолчанию UTF-8) в объекте `bytes`, а параметр `errors` — способ обработки ошибок при преобразовании. В параметре `errors` можно указать значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`. Пример преобразования:

```
>>> b = bytes("строка", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
('строка', 'строка')
```

Для преобразования можно также воспользоваться функцией `str()`:

```
>>> b = bytes("строка", "cp1251")
>>> str(b, "cp1251")
'строка'
```

Чтобы изменить кодировку данных, следует сначала преобразовать тип `bytes` в строку, а затем произвести обратное преобразование, указав нужную кодировку. Преобразуем данные из кодировки Windows-1251 в кодировку KOI8-R, а затем обратно:

```
>>> w = bytes("Строка", "cp1251") # Данные в кодировке windows-1251
>>> k = w.decode("cp1251").encode("koi8-r")
>>> k, str(k, "koi8-r")           # Данные в кодировке KOI8-R
(b'\xf3\xd4\xd2\xcf\xcb\xc1', 'Строка')
>>> w = k.decode("koi8-r").encode("cp1251")
>>> w, str(w, "cp1251")           # Данные в кодировке windows-1251
(b'\xd1\xf2\xf0\xee\xea\xe0', 'Строка')
```

В Python 3.5 появились два полезных инструмента для работы с типом данных `bytes`. Во-первых, теперь можно форматировать такие данные с применением описанного в *разд. 6.4* оператора `%`:

```
>>> b"%i - %i - %f" % (10, 20, 30)
b'10 - 20 - 30.000000'
```

Однако здесь нужно помнить, что тип преобразования `s` (т. е. вывод в виде Unicode-строки) в этом случае не поддерживается, и его использование приведет к возбуждению исключения `TypeError`:

```
>>> b"%s - %s - %s" % (10, 20, 30)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    b"%s - %s - %s" % (10, 20, 30)
TypeError: %b requires a bytes-like object, or an object that implements __bytes__,
not 'int'
```

Во-вторых, тип `bytes` получил поддержку метода `hex()`, который возвращает строку с шестнадцатеричным представлением значения:

```
>>> b"string".hex()
'737472696e67'
```

6.13. Тип данных *bytearray*

Тип данных `bytearray` является разновидностью типа `bytes` и поддерживает те же самые методы и операции (включая оператор форматирования `%` и метод `hex()`, описанные ранее). В отличие от типа `bytes`, тип `bytearray` допускает возможность непосредственного изменения объекта и содержит дополнительные методы, позволяющие выполнять эти изменения.

Создать объект типа `bytearray` можно следующими способами:

- с помощью функции `bytearray([<Строка>, <Кодировка>[, <Обработка ошибок>]])`. Если параметры не указаны, то возвращается пустой объект. Чтобы преобразовать строку в объект типа `bytearray`, необходимо передать минимум два первых параметра. Если строка указана только в первом параметре, то возбуждается исключение `TypeError`:

```
>>> bytearray()
bytearray(b'')
>>> bytearray("строка", "cp1251")
bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
>>> bytearray("строка")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    bytearray("строка")
TypeError: string argument without an encoding
```

  В третьем параметре могут быть указаны значения `"strict"` (при ошибке возбуждается исключение `UnicodeEncodeError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом вопроса) или `"ignore"` (неизвестные символы игнорируются):

```
>>> bytearray("string\uFFFD", "cp1251", "strict")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    bytearray("string\uFFFD", "cp1251", "strict")
  File "C:\Python36\lib\encodings\cp1251.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character
'\ufffd' in position 6: character maps to <undefined>
>>> bytearray("string\uFFFD", "cp1251", "replace")
bytearray(b'string?')
>>> bytearray("string\uFFFD", "cp1251", "ignore")
bytearray(b'string')
```

- с помощью функции `bytearray(<Последовательность>)`, которая преобразует последовательность целых чисел от `0` до `255` в объект типа `bytearray`. Если число не попадает в диапазон, возбуждается исключение `ValueError`:

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

- с помощью функции `bytearray(<Число>)`, которая задает количество элементов в последовательности. Каждый элемент будет содержать нулевой символ:

```
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
```

♦ с помощью метода `bytearray.fromhex(<Строка>)`. Строка в этом случае должна содержать шестнадцатеричные значения символов:

```
>>> b = bytearray.fromhex(" e1 e2e0ae aaa0 ")
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

Тип `bytearray` относится к изменяемым типам. Поэтому можно не только получить значение по индексу, но и изменить его (что не свойственно строкам):

```
>>> b = bytearray("Python", "ascii")
>>> b[0]                                # Можем получить значение
80
>>> b[0] = b"J"[0]                      # Можем изменить значение
>>> b
bytearray(b'Jython')
```

При изменении значения важно помнить, что присваиваемое значение должно быть целым числом в диапазоне от `0` до `255`. Чтобы получить число в предыдущем примере, мы создали объект типа `bytes`, а затем присвоили значение, расположенное по индексу `0` (`b[0] = b"J"[0]`). Можно, конечно, сразу указать код символа, но ведь держать все коды символов в памяти свойственно компьютеру, а не человеку.

Для изменения объекта можно также использовать следующие методы:

♦ `append(<Число>)` — добавляет один элемент в конец объекта. Метод изменяет текущий объект и ничего не возвращает:

```
>>> b = bytearray("string", "ascii")
>>> b.append(b"1"[0]); b
bytearray(b'string1')
```

♦ `extend(<Последовательность>)` — добавляет элементы последовательности в конец объекта. Метод изменяет текущий объект и ничего не возвращает:

```
>>> b = bytearray("string", "ascii")
>>> b.extend(b"123"); b
bytearray(b'string123')
```

Добавить несколько элементов можно с помощью операторов + и +=:

```
>>> b = bytearray("string", "ascii")
>>> b + b"123"                  # Возвращает новый объект
bytearray(b'string123')
>>> b += b"456"; b              # Изменяет текущий объект
bytearray(b'string456')
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> b = bytearray("string", "ascii")
>>> b[len(b):] = b"123"         # Добавляем элементы в последовательность
>>> b
bytearray(b'string123')
```

♦ `insert(<Индекс>, <Число>)` — добавляет один элемент в указанную позицию. Остальные элементы смещаются. Метод изменяет текущий объект и ничего не возвращает. Добавим элемент в начало объекта:

```
>>> b = bytearray("string", "ascii")
>>> b.insert(0, b"1"[0]); b
bytearray(b'1string')
```

Метод `insert()` позволяет добавить только один элемент. Чтобы добавить несколько элементов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало объекта:

```
>>> b = bytearray("string", "ascii")
>>> b[:0] = b"123"; b
bytearray(b'123string')
```

♦ `pop([<Индекс>])` — удаляет элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, удаляет и возвращает последний элемент:

```
>>> b = bytearray("string", "ascii")
>>> b.pop()                  # Удаляем последний элемент
103
>>> b
bytearray(b'strin')
>>> b.pop(0)                 # Удаляем первый элемент
115
>>> b
bytearray(b'trin')
```

Удалить элемент списка позволяет также оператор `del`:

```
>>> b = bytearray("string", "ascii")
>>> del b[5]                # Удаляем последний элемент
>>> b
bytearray(b'strin')
>>> del b[:2]               # Удаляем первый и второй элементы
>>> b
bytearray(b'rin')
```

♦ `remove(<Число>)` — удаляет первый элемент, содержащий указанное значение. Если элемент не найден, возбуждается исключение `ValueError`. Метод изменяет текущий объект и ничего не возвращает:

```
>>> b = bytearray("strstr", "ascii")
>>> b.remove(b"s"[0])       # Удаляет только первый элемент
>>> b
bytearray(b'trstr')
```

♦ `reverse()` — изменяет порядок следования элементов на противоположный. Метод изменяет текущий объект и ничего не возвращает:

```
>>> b = bytearray("string", "ascii")
>>> b.reverse(); b
bytearray(b'gnirts')
```

Преобразовать объект типа `bytearray` в строку позволяет метод `decode()`. Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр `encoding` задает кодировку символов (по умолчанию UTF-8) в объекте `bytearray`, а параметр `errors` — способ обработки ошибок при преобразовании. В параметре `errors` можно указать значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`. Пример преобразования:

```
>>> b = bytearray("строка", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
('строка', 'строка')
```

Для преобразования можно также воспользоваться функцией `str()`:

```
>>> b = bytearray("строка", "cp1251")
>>> str(b, "cp1251")
'строка'
```

6.14. Преобразование объекта в последовательность байтов

Преобразовать объект в последовательность байтов (выполнить его *сериализацию*), а затем восстановить (*десериализовать*) объект позволяет модуль `pickle`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import pickle
```

Для преобразования предназначены две функции:

♦ `dumps(<Объект>[, protocol=None][, fix_imports=True])` — производит сериализацию объекта и возвращает последовательность байтов специального формата. Формат зависит от указанного во втором параметре протокола, который задается в виде числа от `0` до значения константы `pickle.HIGHEST_PROTOCOL`. Если второй параметр не указан, будет использован протокол `3` (константа `pickle.DEFAULT_PROTOCOL`). Пример преобразования списка и кортежа:

```
>>> import pickle
>>> obj1 = [1, 2, 3, 4, 5]       # Список
>>> obj2 = (6, 7, 8, 9, 10)      # Кортеж
>>> pickle.dumps(obj1)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
>>> pickle.dumps(obj2)
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

♦ `loads(<Последовательность байтов>[, fix_imports=True][, encoding="ASCII"][, errors="strict"])` — преобразует последовательность байтов специального формата обратно в объект, выполняя его десериализацию. Пример восстановления списка и кортежа:

```
>>> pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
[1, 2, 3, 4, 5]
>>> pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
(6, 7, 8, 9, 10)
```

6.15. Шифрование строк

Для шифрования строк предназначен модуль `hashlib`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import hashlib
```

Модуль предоставляет следующие функции: `md5()`, `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, в Python 3.6 появилась поддержка функций `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()` и `shake_256()`. В качестве необязательного параметра функциям можно передать шифруемую последовательность байтов:

```
>>> import hashlib
>>> h = hashlib.sha1(b"password")
```

Передать последовательность байтов можно также с помощью метода `update()`. В этом случае объект присоединяется к предыдущему значению:

```
>>> h = hashlib.sha1()
>>> h.update(b"password")
```

Получить зашифрованную последовательность байтов и строку позволяют два метода: `digest()` и `hexdigest()`. Первый метод возвращает значение, относящееся к типу `bytes`, а второй — строку, содержащую шестнадцатеричные цифры:

```
>>> h = hashlib.sha1(b"password")
>>> h.digest()
b'[\xaaa\xe4\xc9\xb9??\x06\x82%\x0bl\xf83\x1b~\xe6\x8f\xd8'
>>> h.hexdigest()
'5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8'
```

Наиболее часто применяемой является функция `md5()`, которая шифрует строку с помощью алгоритма MD5. Эта функция используется для шифрования паролей, т. к. не существует алгоритма для дешифровки зашифрованных ей значений. Для сравнения введенного пользователем пароля с сохраненным в базе необходимо зашифровать введенный пароль, а затем произвести сравнение:

```
>>> import hashlib
>>> h = hashlib.md5(b"password")
>>> p = h.hexdigest()
>>> p                               # Пароль, сохраненный в базе
'5f4dcc3b5aa765d61d8327deb882cf99'
>>> h2 = hashlib.md5(b"password")    # Пароль, введенный пользователем
>>> if p == h2.hexdigest(): print("Пароль правильный")

Пароль правильный
```

Свойство `digest_size` хранит размер значения, генерируемого описанными ранее функциями шифрования, в байтах:

```
>>> h = hashlib.sha512(b"password")
>>> h.digest_size
64
```

Поддерживается еще несколько функций, выполняющих устойчивое к взлому шифрование паролей:

- pbkdf2_hmac(<Основной алгоритм шифрования>, <Шифруемый пароль>, <"Соль">, <Количество проходов шифрования>, dklen=None). В качестве основного алгоритма шифрования следует указать строку с наименованием этого алгоритма: "md5", "sha1", "sha224", "sha256", "sha384" и "sha512". Шифруемый пароль указывается в виде значения типа bytes. "Соль" — это особая величина типа bytes, выступающая в качестве ключа шифрования, — ее длина не должна быть менее 16 символов. Количество проходов шифрования следует указать достаточно большим (так, при использовании алгоритма SHA512 оно должно составлять 100 000). Последним параметром функции pbkdf2_hmac() можно указать длину результирующего закодированного значения в байтах — если она не задана или равна None, будет создано значение стандартной для выбранного алгоритма длины (64 байта для алгоритма SHA512). Закодированный пароль возвращается в виде величины типа bytes:

```
>>> import hashlib
>>> dk = hashlib.pbkdf2_hmac('sha512', b'1234567', b'saltsaltsaltsalt', 100000)
>>> dk
b"Sb\x85tc-\xcb@\xc5\x97\x19\x90\x94@\x9f\xde\x07\xa4p-\x83\x94\xf4\x94\x99\x07\
xec\xfa\xf3\xcd\xc3\x88jv\xd1\xe5\x9a\x119\x15/\xa4\xc2\xd3N\xaba\x02\xc0s\xc1\
xd1\x0b\x86xj(\x8c>Mr'@\xbb"
```

> ### Примечание
>
> Кодирование данных с применением функции pbkdf2_hmac() отнимает очень много системных ресурсов и может занять значительное время, особенно на маломощных компьютерах.

Поддержка двух следующих функций появилась в Python 3.6:

- blake2s([<Шифруемый пароль>][, digest_size=32], [salt=b””]) — шифрует пароль по алгоритму BLAKE2s, оптимизированному для 32-разрядных систем. Второй параметр задает размер значения в виде числа от 1 до 32, третий — «соль» в виде величины типа bytes, которая может иметь в длину не более 8 байтов. Возвращает объект, хранящий закодированный пароль:

```
>>> h = hashlib.blake2s(b"string", digest_size=16, salt=b"saltsalt")
>>> h.digest()
b'\x96l\xe0\xfa\xb4\xe7Bw\x11\xf7D\xc2\xa4\xcf\x06\xf7'
```

- blake2b([<Шифруемый пароль>][, digest_size=64], [salt=b””]) — шифрует пароль по алгоритму BLAKE2b, оптимизированному для 64-разрядных систем. Второй параметр задает размер значения в виде числа от 1 до 64, третий — «соль» в виде величины типа bytes, которая может иметь в длину не более 16 байтов. Возвращает объект, хранящий закодированный пароль:

```
>>> h = hashlib.blake2b(b"string", digest_size=48, salt=b"saltsaltsalt")
>>> h.digest()
b'\x0e\xcf\xb9\xc8G;q\xbaV\xbdV\x16\xd4@/J\x97W\x0c\xc4\xc5{\xd4\xb6\x12\x01z\
x9f\xdd\xf6\xf1\x03o\x97&v\xfd\xa6\x90\x81\xc4T\xb8z\xaf\xc3\x9a\xd9'
```

> ### Примечание
>
> Функции blake2b() и blake2s() поддерживают большое количество параметров, которые применяются только в специфических случаях. Полное описание этих функций можно найти в документации по Python.

ГЛАВА 7

Регулярные выражения

Регулярные выражения предназначены для выполнения в строке сложного поиска или замены. В языке Python использовать регулярные выражения позволяет модуль `re`. Прежде чем задействовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import re
```

7.1. Синтаксис регулярных выражений

Создать откомпилированный шаблон регулярного выражения позволяет функция `compile()`. Функция имеет следующий формат:

```
<Шаблон> = re.compile(<Регулярное выражение>[, <Модификатор>])
```

В параметре `<Модификатор>` могут быть указаны следующие флаги (или их комбинация через оператор `|`):

- `I` или `IGNORECASE` — поиск без учета регистра:

  ```
  >>> import re
  >>> p = re.compile(r"^[а-яе]+$", re.I | re.U)
  >>> print("Найдено" if p.search("АБВГДЕЕ") else "Нет")
  Найдено
  >>> p = re.compile(r"^[а-яе]+$", re.U)
  >>> print("Найдено" if p.search("АБВГДЕЕ") else "Нет")
  Нет
  ```

- `M` или `MULTILINE` — поиск в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`"\n"`). Символ `^` соответствует привязке к началу каждой подстроки, а символ `$` — позиции перед символом перевода строки;
- `S` или `DOTALL` — метасимвол «точка» по умолчанию соответствует любому символу, кроме символа перевода строки (`\n`). Символу перевода строки метасимвол «точка» будет соответствовать в присутствии дополнительного модификатора. Символ `^` соответствует привязке к началу всей строки, а символ `$` — привязке к концу всей строки:

  ```
  >>> p = re.compile(r"^.$")
  >>> print("Найдено" if p.search("\n") else "Нет")
  Нет
  >>> p = re.compile(r"^.$", re.M)
  ```

```
>>> print("Найдено" if p.search("\n") else "Нет")
Нет
>>> p = re.compile(r"^.$", re.S)
>>> print("Найдено" if p.search("\n") else "Нет")
Найдено
```

- `X` или `VERBOSE` — если флаг указан, то пробелы и символы перевода строки будут проигнорированы. Внутри регулярного выражения можно использовать и комментарии:

```
>>> p = re.compile(r"""^ # Привязка к началу строки
        [0-9]+ # Строка должна содержать одну цифру (или более)
        $      # Привязка к концу строки
        """, re.X | re.S)
>>> print("Найдено" if p.search("1234567890") else "Нет")
Найдено
>>> print("Найдено" if p.search("abcd123") else "Нет")
Нет
```

- `A` или `ASCII` — классы `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` будут соответствовать символам в кодировке ASCII (по умолчанию указанные классы соответствуют Unicode-символам);

> **Примечание**
>
> Флаги `U` и `UNICODE`, включающие режим соответствия Unicode-символам классов `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S`, сохранены в Python 3 лишь для совместимости с ранними версиями этого языка и никакого влияния на обработку регулярных выражений не оказывают.

- `L` или `LOCALE` — учитываются настройки текущей локали. Начиная с Python 3.6, могут быть использованы только в том случае, когда регулярное выражение задается в виде значения типов `bytes` или `bytearray`.

Как видно из примеров, перед всеми строками, содержащими регулярные выражения, указан модификатор `r`. Иными словами, мы используем неформатированные строки. Если модификатор не указать, то все слэши необходимо экранировать. Например, строку:

```
p = re.compile(r"^\w+$")
```

нужно было бы записать так:

```
p = re.compile("^\\w+$")
```

Внутри регулярного выражения символы `.`, `^`, `$`, `*`, `+`, `?`, `{`, `[`, `]`, `\`, `|`, `(` и `)` имеют специальное значение. Если эти символы должны трактоваться как есть, их следует экранировать с помощью слэша. Некоторые специальные символы теряют свое особое значение, если их разместить внутри квадратных скобок, — в этом случае экранировать их не нужно. Например, как уже было отмечено ранее, метасимвол «точка» по умолчанию соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед точкой нужно указать символ `\` или разместить точку внутри квадратных скобок: `[.]`. Продемонстрируем это на примере проверки правильности введенной даты (листинг 7.1).

Листинг 7.1. Проверка правильности ввода даты

```
# -*- coding: utf-8 -*-
import re          # Подключаем модуль
d = "29,12.2009"  # Вместо точки указана запятая
p = re.compile(r"^[0-3][0-9].[01][0-9].[12][09][0-9][0-9]$")
```

```
# Символ "\" не указан перед точкой
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Так как точка означает любой символ,
# выведет: Дата введена правильно

p = re.compile(r"^[0-3][0-9]\.[01][0-9]\.[12][09][0-9][0-9]$")
# Символ "\" указан перед точкой
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Так как перед точкой указан символ "\",
# выведет: Дата введена неправильно

p = re.compile(r"^[0-3][0-9][.][01][0-9][.][12][09][0-9][0-9]$")
# Точка внутри квадратных скобок
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
# Выведет: Дата введена неправильно
input()
```

В этом примере мы осуществляли привязку к началу и концу строки с помощью следующих метасимволов:

- `^` — привязка к началу строки или подстроки. Она зависит от флагов `M` (или `MULTILINE`) и `S` (или `DOTALL`);
- `$` — привязка к концу строки или подстроки. Она зависит от флагов `M` (или `MULTILINE`) и `S` (или `DOTALL`);
- `\A` — привязка к началу строки (не зависит от модификатора);
- `\Z` — привязка к концу строки (не зависит от модификатора).

Если в параметре `<Модификатор>` указан флаг `M` (или `MULTILINE`), то поиск производится в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`\n`). В этом случае символ `^` соответствует привязке к началу каждой подстроки, а символ `$` — позиции перед символом перевода строки:

```
>>> p = re.compile(r"^.+$")       # Точка не соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Ничего не найдено
[]
>>> p = re.compile(r"^.+$", re.S) # Теперь точка соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Строка полностью соответствует
['str1\nstr2\nstr3']
>>> p = re.compile(r"^.+$", re.M) # Многострочный режим
>>> p.findall("str1\nstr2\nstr3") # Получили каждую подстроку
['str1', 'str2', 'str3']
```

Привязку к началу и концу строки следует использовать, если строка должна полностью соответствовать регулярному выражению. Например, для проверки, содержит ли строка число (листинг 7.2).

Листинг 7.2. Проверка наличия целого числа в строке

```
# -*- coding: utf-8 -*-
import re                         # Подключаем модуль
p = re.compile(r"^[0-9]+$", re.S)
if p.search("245"):
    print("Число")                # Выведет: Число
else:
    print("Не число")
if p.search("Строка245"):
    print("Число")
else:
    print("Не число")             # Выведет: Не число
input()
```

Если убрать привязку к началу и концу строки, то любая строка, содержащая хотя бы одну цифру, будет распознана как `Число` (листинг 7.3).

Листинг 7.3. Отсутствие привязки к началу или концу строки

```
# -*- coding: utf-8 -*-
import re                         # Подключаем модуль
p = re.compile(r"[0-9]+", re.S)
if p.search("Строка245"):
    print("Число")                # Выведет: Число
else:
    print("Не число")
input()
```

Кроме того, можно указать привязку только к началу или только к концу строки (листинг 7.4).

Листинг 7.4. Привязка к началу и концу строки

```
# -*- coding: utf-8 -*-
import re                         # Подключаем модуль
p = re.compile(r"[0-9]+$", re.S)
if p.search("Строка245"):
    print("Есть число в конце строки")
else:
    print("Нет числа в конце строки")
# Выведет: Есть число в конце строки
p = re.compile(r"^[0-9]+", re.S)
if p.search("Строка245"):
    print("Есть число в начале строки")
else:
    print("Нет числа в начале строки")
# Выведет: Нет числа в начале строки
input()
```

Также поддерживаются два метасимвола, позволяющие указать привязку к началу или концу слова:

- `\b` — привязка к началу слова (началом слова считается пробел или любой символ, не являющийся буквой, цифрой или знаком подчеркивания);
- `\B` — привязка к позиции, не являющейся началом слова.

Рассмотрим несколько примеров:

```
>>> p = re.compile(r"\bpython\b")
>>> print("Найдено" if p.search("python") else "Нет")
Найдено
>>> print("Найдено" if p.search("pythonware") else "Нет")
Нет
>>> p = re.compile(r"\Bth\B")
>>> print("Найдено" if p.search("python") else "Нет")
Найдено
>>> print("Найдено" if p.search("this") else "Нет")
Нет
```

В квадратных скобках `[]` можно указать символы, которые могут встречаться на этом месте в строке. Разрешается записать символы подряд или указать диапазон через дефис:

- `[09]` — соответствует числу 0 или 9;
- `[0-9]` — соответствует любому числу от 0 до 9;
- `[абв]` — соответствует буквам «а», «б» и «в»;
- `[а-г]` — соответствует буквам «а», «б», «в» и «г»;
- `[а-яё]` — соответствует любой букве от «а» до «я»;
- `[АБВ]` — соответствует буквам «А», «Б» и «В»;
- `[А-ЯЁ]` — соответствует любой букве от «А» до «Я»;
- `[а-яА-ЯёЁ]` — соответствует любой русской букве в любом регистре;
- `[0-9а-яА-ЯёЁa-zA-Z]` — любая цифра и любая буква независимо от регистра и языка.

> **Внимание!**
>
> Буква «ё» не входит в диапазон `[а-я]`, а буква «Ё» — в диапазон `[А-Я]`.

Значение в скобках инвертируется, если после первой скобки вставить символ `^`. Таким образом можно указать символы, которых не должно быть на этом месте в строке:

- `[^09]` — не цифра 0 или 9;
- `[^0-9]` — не цифра от 0 до 9;
- `[^а-яА-ЯёЁa-zA-Z]` — не буква.

Как вы уже знаете, точка теряет свое специальное значение, если ее заключить в квадратные скобки. Кроме того, внутри квадратных скобок могут встретиться символы, которые имеют специальное значение (например, `^` и `-`). Символ `^` теряет свое специальное значение, если он не расположен сразу после открывающей квадратной скобки. Чтобы отменить специальное значение символа `-`, его необходимо указать после всех символов, перед закрывающей квадратной скобкой или сразу после открывающей квадратной скобки. Все специальные символы можно сделать обычными, если перед ними указать символ `\`.

Метасимвол `|` позволяет сделать выбор между альтернативными значениями. Выражение `n|m` соответствует одному из символов: `n` или `m`:

```
>>> p = re.compile(r"красн((ая)|(ое))")
>>> print("Найдено" if p.search("красная") else "Нет")
Найдено
>>> print("Найдено" if p.search("красное") else "Нет")
Найдено
>>> print("Найдено" if p.search("красный") else "Нет")
Нет
```

Вместо указания символов можно использовать стандартные классы:

- `\d` — соответствует любой цифре. При указании флага `A` (`ASCII`) эквивалентно `[0-9]`;
- `\w` — соответствует любой букве, цифре или символу подчеркивания. При указании флага `A` (`ASCII`) эквивалентно `[a-zA-Z0-9_]`;
- `\s` — любой пробельный символ. При указании флага `A` (`ASCII`) эквивалентно `[\t\n\r\f\v]`;
- `\D` — не цифра. При указании флага `A` (`ASCII`) эквивалентно `[^0-9]`;
- `\W` — не буква, не цифра и не символ подчеркивания. При указании флага `A` (`ASCII`) эквивалентно `[^a-zA-Z0-9_]`;
- `\S` — не пробельный символ. При указании флага `A` (`ASCII`) эквивалентно `[^\t\n\r\f\v]`.

> **Примечание**
>
> В Python 3 поддержка Unicode в регулярных выражениях установлена по умолчанию. При этом все классы трактуются гораздо шире. Так, класс `\d` соответствует не только десятичным цифрам, но и другим цифрам из кодировки Unicode, — например, дробям, класс `\w` включает не только латинские буквы, но и любые другие, а класс `\s` охватывает также неразрывные пробелы. Поэтому на практике лучше явно указывать символы внутри квадратных скобок, а не использовать классы.

Количество вхождений символа в строку задается с помощью *квантификаторов*:

- `{n}` — `n` вхождений символа в строку. Например, шаблон `r"^[0-9]{2}$"` соответствует двум вхождениям любой цифры;
- `{n,}` — `n` или более вхождений символа в строку. Например, шаблон `r"^[0-9]{2,}$"` соответствует двум и более вхождениям любой цифры;
- `{n,m}` — не менее `n` и не более `m` вхождений символа в строку. Числа указываются через запятую без пробела. Например, шаблон `r"^[0-9]{2,4}$"` соответствует от двух до четырех вхождений любой цифры;
- `*` — ноль или большее число вхождений символа в строку. Эквивалентно комбинации `{0,}`;
- `+` — одно или большее число вхождений символа в строку. Эквивалентно комбинации `{1,}`;
- `?` — ни одного или одно вхождение символа в строку. Эквивалентно комбинации `{0,1}`.

Все квантификаторы являются «жадными». При поиске соответствия ищется самая длинная подстрока, соответствующая шаблону, и не учитываются более короткие соответствия. Рассмотрим это на примере и получим содержимое всех тегов `<b>` вместе с тегами:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*</b>", re.S)
```

```
>>> p.findall(s)
['<b>Text1</b>Text2<b>Text3</b>']
```

Вместо желаемого результата мы получили полностью строку. Чтобы ограничить «жадность», необходимо после квантификатора указать символ `?`:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*?</b>", re.S)
>>> p.findall(s)
['<b>Text1</b>', '<b>Text3</b>']
```

Этот код вывел то, что мы искали. Если необходимо получить содержимое без тегов, то нужный фрагмент внутри шаблона следует разместить внутри круглых скобок:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>(.*?)</b>", re.S)
>>> p.findall(s)
['Text1', 'Text3']
```

Круглые скобки часто используются для группировки фрагментов внутри шаблона. В этом случае не требуется, чтобы фрагмент запоминался и был доступен в результатах поиска. Чтобы избежать захвата фрагмента, следует после открывающей круглой скобки разместить символы `?:` (вопросительный знак и двоеточие):

```
>>> s = "test text"
>>> p = re.compile(r"([a-z]+((st)|(xt)))", re.S)
>>> p.findall(s)
[('test', 'st', 'st', ''), ('text', 'xt', '', 'xt')]
>>> p = re.compile(r"([a-z]+(?:(?:st)|(?:xt)))", re.S)
>>> p.findall(s)
['test', 'text']
```

В первом примере мы получили список с двумя элементами. Каждый элемент списка является кортежем, содержащим четыре элемента. Все эти элементы соответствуют фрагментам, заключенным в шаблоне в круглые скобки. Первый элемент кортежа содержит фрагмент, расположенный в первых круглых скобках, второй — во вторых круглых скобках и т. д. Три последних элемента кортежа являются лишними. Чтобы они не выводились в результатах, мы добавили символы `?:` после каждой открывающей круглой скобки. В результате список состоит только из фрагментов, полностью соответствующих регулярному выражению.

К найденному фрагменту в круглых скобках внутри шаблона можно обратиться с помощью механизма *обратных ссылок*. Для этого порядковый номер круглых скобок в шаблоне указывается после слэша, например, так: `\1`. Нумерация скобок внутри шаблона начинается с `1`. Для примера получим текст между одинаковыми парными тегами:

```
>>> s = "<b>Text1</b>Text2<I>Text3</I><b>Text4</b>"
>>> p = re.compile(r"<([a-z]+)>(.*?)</\1>", re.S | re.I)
>>> p.findall(s)
[('b', 'Text1'), ('I', 'Text3'), ('b', 'Text4')]
```

Фрагментам внутри круглых скобок можно дать имена, создав тем самым именованные фрагменты. Для этого после открывающей круглой скобки следует указать комбинацию символов `?P<name>`. В качестве примера разберем e-mail на составные части:

```
>>> email = "test@mail.ru"
>>> p = re.compile(r"""(?P<name>[a-z0-9_.-]+) # Название ящика
        @                                     # Символ "@"
        (?P<host>(?:[a-z0-9-]+\.)+[a-z]{2,6}) # Домен
        """, re.I | re.VERBOSE)
>>> r = p.search(email)
>>> r.group("name")                           # Название ящика
'test'
>>> r.group("host")                           # Домен
'mail.ru'
```

Чтобы внутри шаблона обратиться к именованным фрагментам, используется следующий синтаксис: `(?P=name)`. Для примера получим текст между одинаковыми парными тегами:

```
>>> s = "<b>Text1</b>Text2<I>Text3</I>"
>>> p = re.compile(r"<(?P<tag>[a-z]+)>(.*?)</(?P=tag)>", re.S | re.I)
>>> p.findall(s)
[('b', 'Text1'), ('I', 'Text3')]
```

Кроме того, внутри круглых скобок могут быть расположены следующие конструкции:

- `(?#...)` — комментарий. Текст внутри круглых скобок игнорируется;
- `(?=...)` — положительный просмотр вперед. Выведем все слова, после которых расположена запятая:

  ```
  >>> s = "text1, text2, text3 text4"
  >>> p = re.compile(r"\w+(?=[,])", re.S | re.I)
  >>> p.findall(s)
  ['text1', 'text2']
  ```

- `(?!...)` — отрицательный просмотр вперед. Выведем все слова, после которых нет запятой:

  ```
  >>> s = "text1, text2, text3 text4"
  >>> p = re.compile(r"[a-z]+[0-9](?![,])", re.S | re.I)
  >>> p.findall(s)
  ['text3', 'text4']
  ```

- `(?<=...)` — положительный просмотр назад. Выведем все слова, перед которыми расположена запятая с пробелом:

  ```
  >>> s = "text1, text2, text3 text4"
  >>> p = re.compile(r"(?<=[,][ ])[a-z]+[0-9]", re.S | re.I)
  >>> p.findall(s)
  ['text2', 'text3']
  ```

- `(?<!...)` — отрицательный просмотр назад. Выведем все слова, перед которыми расположен пробел, но перед пробелом нет запятой:

  ```
  >>> s = "text1, text2, text3 text4"
  >>> p = re.compile(r"(?<![,]) ([a-z]+[0-9])", re.S | re.I)
  >>> p.findall(s)
  ['text4']
  ```

- `(?(id` или `name)шаблон1|шаблон2)` — если группа с номером или названием найдена, то должно выполняться условие из параметра `шаблон1`, в противном случае должно выпол-

няться условие из параметра `шаблон2`. Выведем все слова, которые расположены внутри апострофов. Если перед словом нет апострофа, то в конце слова должна быть запятая:

```
>>> s = "text1 'text2' 'text3 text4, text5"
>>> p = re.compile(r"(')?([a-z]+[0-9])(?(1)'|,)", re.S | re.I)
>>> p.findall(s)
[("'", 'text2'), ('', 'text4')]
```

- `(?aiLmsux)` — позволяет установить опции регулярного выражения. Буквы `a`, `i`, `L`, `m`, `s`, `u` и `x` имеют такое же назначение, что и одноименные модификаторы в функции `compile()`.

> ***Внимание!***
> Начиная с Python 3.6, опции, задаваемые внутри регулярного выражения в круглых скобках, объявлены устаревшими и не рекомендованными к использованию. В будущих версиях Python их поддержка будет удалена.

Рассмотрим небольшой пример. Предположим, необходимо получить все слова, расположенные после дефиса, причем перед дефисом и после слов должны следовать пробельные символы:

```
>>> s = "-word1 -word2 -word3 -word4 -word5"
>>> re.findall(r"\s\-([a-z0-9]+)\s", s, re.S | re.I)
['word2', 'word4']
```

Как видно из примера, мы получили только два слова вместо пяти. Первое и последнее слова не попали в результат, т. к. расположены в начале и в конце строки. Чтобы эти слова попали в результат, необходимо добавить альтернативный выбор `(^|\s)` — для начала строки и `(\s|$)` — для конца строки. Чтобы найденные выражения внутри круглых скобок не попали в результат, следует добавить символы `?:` после открывающей скобки:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?:\s|$)", s, re.S | re.I)
['word1', 'word3', 'word5']
```

Здесь первое и последнее слова успешно попали в результат. Почему же слова `word2` и `word4` не попали в список совпадений — ведь перед дефисом есть пробел и после слова есть пробел? Чтобы понять причину, рассмотрим поиск по шагам. Первое слово успешно попадает в результат, т. к. перед дефисом расположено начало строки и после слова есть пробел. После поиска указатель перемещается, и строка для дальнейшего поиска примет следующий вид:

```
"-word1 <Указатель>-word2 -word3 -word4 -word5"
```

Обратите внимание на то, что перед фрагментом `-word2` больше нет пробела, и дефис не расположен в начале строки. Поэтому следующим совпадением окажется слово `word3`, и указатель снова будет перемещен:

```
"-word1 -word2 -word3 <Указатель>-word4 -word5"
```

Опять перед фрагментом `-word4` нет пробела, и дефис не расположен в начале строки. Поэтому следующим совпадением окажется слово `word5`, и поиск будет завершен. Таким образом, слова `word2` и `word4` не попадают в результат, поскольку пробел до фрагмента уже был использован в предыдущем поиске. Чтобы этого избежать, следует воспользоваться положительным просмотром вперед `(?=...)`:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?=\s|$)", s, re.S | re.I)
['word1', 'word2', 'word3', 'word4', 'word5']
```

В этом примере мы заменили фрагмент `(?:\s|$)` на `(?=\s|$)`. Поэтому все слова успешно попали в список совпадений.

7.2. Поиск первого совпадения с шаблоном

Для поиска первого совпадения с шаблоном предназначены следующие функции и методы:

- `match()` — проверяет соответствие с началом строки. Формат метода:

  ```
  match(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствие найдено, возвращается объект `Match`, в противном случае — значение `None`:

  ```
  >>> import re
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if p.match("str123") else "Нет")
  Нет
  >>> print("Найдено" if p.match("str123", 3) else "Нет")
  Найдено
  >>> print("Найдено" if p.match("123str") else "Нет")
  Найдено
  ```

  Вместо метода `match()` можно воспользоваться функцией `match()`. Формат функции:

  ```
  re.match(<Шаблон>, <Строка>[, <Модификатор>])
  ```

  В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Если соответствие найдено, то возвращается объект `Match`, в противном случае — значение `None`:

  ```
  >>> p = r"[0-9]+"
  >>> print("Найдено" if re.match(p, "str123") else "Нет")
  Нет
  >>> print("Найдено" if re.match(p, "123str") else "Нет")
  Найдено
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if re.match(p, "123str") else "Нет")
  Найдено
  ```

- `search()` — проверяет соответствие с любой частью строки. Формат метода:

  ```
  search(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствие найдено, возвращается объект `Match`, в противном случае — значение `None`:

  ```
  >>> p = re.compile(r"[0-9]+")
  >>> print("Найдено" if p.search("str123") else "Нет")
  Найдено
  >>> print("Найдено" if p.search("123str") else "Нет")
  Найдено
  >>> print("Найдено" if p.search("123str", 3) else "Нет")
  Нет
  ```

  Вместо метода `search()` можно воспользоваться функцией `search()`. Формат функции:

  ```
  re.search(<Шаблон>, <Строка>[, <Модификатор>])
  ```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Если соответствие найдено, возвращается объект `Match`, в противном случае — значение `None`:

```
>>> p = r"[0-9]+"
>>> print("Найдено" if re.search(p, "str123") else "Нет")
Найдено
>>> p = re.compile(r"[0-9]+")
>>> print("Найдено" if re.search(p, "str123") else "Нет")
Найдено
```

♦ `fullmatch()` — выполняет проверку, соответствует ли переданная строка регулярному выражению целиком. Формат метода:

```
fullmatch(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Если соответствие найдено, то возвращается объект `Match`, в противном случае — значение `None`:

```
>>> p = re.compile("[Pp]ython")
>>> print("Найдено" if p.fullmatch("Python") else "Нет")
Найдено
>>> print("Найдено" if p.fullmatch("py") else "Нет")
Нет
>>> print("Найдено" if p.fullmatch("PythonWare") else "Нет")
Нет
>>> print("Найдено" if p.fullmatch("PythonWare", 0, 6) else "Нет")
Найдено
```

Вместо метода `fullmatch()` можно воспользоваться функцией `fullmatch()`. Формат функции:

```
re.fullmatch(<Шаблон>, <Строка>[, <Модификатор>])
```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Если строка полностью совпадает с шаблоном, возвращается объект `Match`, в противном случае — значение `None`:

```
>>> p = "[Pp]ython"
>>> print("Найдено" if re.fullmatch(p, "Python") else "Нет")
Найдено
>>> print("Найдено" if re.fullmatch(p, "py") else "Нет")
Нет
```

В качестве примера переделаем нашу программу суммирования произвольного количества целых чисел, введенных пользователем (см. листинг 4.12), таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой. Предусмотрим также возможность ввода отрицательных целых чисел (листинг 7.5).

Листинг 7.5. Суммирование произвольного количества чисел

```
# -*- coding: utf-8 -*-
import re
print("Введите слово 'stop' для получения результата")
summa = 0
```

```
p = re.compile(r"^[-]?[0-9]+$", re.S)
while True:
    x = input("Введите число: ")
    if x == "stop":
        break    # Выход из цикла
    if not p.search(x):
        print("Необходимо ввести число, а не строку!")
        continue # Переходим на следующую итерацию цикла
    x = int(x)   # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Объект `Match`, возвращаемый методами (функциями) `match()`, `search()` и `fullmatch()`, имеет следующие атрибуты и методы:

- `re` — ссылка на скомпилированный шаблон, указанный в методах (функциях) `match()`, `search()` и `fullmatch()`. Через эту ссылку доступны следующие атрибуты:
  - `groups` — количество групп в шаблоне;
  - `groupindex` — словарь с названиями групп и их номерами;
  - `pattern` — исходная строка с регулярным выражением;
  - `flags` — комбинация флагов, заданных при создании регулярного выражения в функции `compile()`, и флагов, указанных в самом регулярном выражении, в конструкции `(?aiLmsux)`;
- `string` — значение параметра `<Строка>` в методах (функциях) `match()`, `search()` и `fullmatch()`;
- `pos` — значение параметра `<Начальная позиция>` в методах `match()`, `search()` и `fullmatch()`;
- `endpos` — значение параметра `<Конечная позиция>` в методах `match()`, `search()` и `fullmatch()`;
- `lastindex` — возвращает номер последней группы или значение `None`, если поиск завершился неудачей;
- `lastgroup` — возвращает название последней группы или значение `None`, если эта группа не имеет имени или поиск завершился неудачей:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m
<_sre.SRE_Match object at 0x00FC9DE8>
>>> m.re.groups, m.re.groupindex
(2, {'num': 1, 'str': 2})
>>> p.groups, p.groupindex
(2, {'num': 1, 'str': 2})
>>> m.string
'123456string 67890text'
>>> m.lastindex, m.lastgroup
(2, 'str')
>>> m.pos, m.endpos
(0, 22)
```

♦ `group([<id1 или name1>[, ..., <idN или nameN>]])` — возвращает фрагменты, соответствующие шаблону. Если параметр не задан или указано значение `0`, возвращается фрагмент, полностью соответствующий шаблону. Если указан номер или название группы, возвращается фрагмент, совпадающий с этой группой. Через запятую можно указать несколько номеров или названий групп — в этом случае возвращается кортеж, содержащий фрагменты, что соответствует группам. Если нет группы с указанным номером или названием, то возбуждается исключение `IndexError`:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m.group(), m.group(0) # Полное соответствие шаблону
('123456string', '123456string')
>>> m.group(1), m.group(2)         # Обращение по индексу
('123456', 'string')
>>> m.group("num"), m.group("str") # Обращение по названию
('123456', 'string')
>>> m.group(1, 2), m.group("num", "str") # Несколько параметров
(('123456', 'string'), ('123456', 'string'))
```

♦ `groupdict([<Значение по умолчанию>])` — возвращает словарь, содержащий значения именованных групп. С помощью необязательного параметра можно указать значение, которое будет выводиться вместо значения `None` для групп, не имеющих совпадений:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groupdict()
{'num': '123456', 'str': None}
>>> m.groupdict("")
{'num': '123456', 'str': ''}
```

♦ `groups([<Значение по умолчанию>])` — возвращает кортеж, содержащий значения всех групп. С помощью необязательного параметра можно указать значение, которое будет выводиться вместо значения `None` для групп, не имеющих совпадений:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groups()
('123456', None)
>>> m.groups("")
('123456', '')
```

♦ `start([<Номер или название группы>])` — возвращает индекс начала фрагмента, соответствующего заданной группе. Если параметр не указан, то фрагментом является полное соответствие с шаблоном. Если соответствия нет, возвращается значение `-1`;

♦ `end([<Номер или название группы>])` — возвращает индекс конца фрагмента, соответствующего заданной группе. Если параметр не указан, то фрагментом является полное соответствие с шаблоном. Если соответствия нет, возвращается значение `-1`;

♦ `span([<Номер или название группы>])` — возвращает кортеж, содержащий начальный и конечный индексы фрагмента, соответствующего заданной группе. Если параметр не указан, то фрагментом является полное соответствие с шаблоном. Если соответствия нет, возвращается значение `(-1, -1)`:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> s = "str123456str"
>>> m = p.search(s)
>>> m.start(), m.end(), m.span()
(3, 12, (3, 12))
>>> m.start(1), m.end(1), m.start("num"), m.end("num")
(3, 9, 3, 9)
>>> m.start(2), m.end(2), m.start("str"), m.end("str")
(9, 12, 9, 12)
>>> m.span(1), m.span("num"), m.span(2), m.span("str")
((3, 9), (3, 9), (9, 12), (9, 12))
>>> s[m.start(1):m.end(1)], s[m.start(2):m.end(2)]
('123456', 'str')
```

- `expand(<Шаблон>)` — производит замену в строке. Внутри указанного шаблона можно использовать обратные ссылки: `\номер группы`, `\g<номер группы>` и `\g<название группы>`. Для примера поменяем два тега местами:

```
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> m = p.search("<br><hr>")
>>> m.expand(r"<\2><\1>")              # \номер
'<hr><br>'
>>> m.expand(r"<\g<2>><\g<1>>")        # \g<номер>
'<hr><br>'
>>> m.expand(r"<\g<tag2>><\g<tag1>>") # \g<название>
'<hr><br>'
```

В качестве примера использования метода `search()` проверим на соответствие шаблону введенный пользователем адрес электронной почты (листинг 7.6).

Листинг 7.6. Проверка e-mail на соответствие шаблону

```
# -*- coding: utf-8 -*-
import re
email = input("Введите e-mail: ")
pe = r"^([a-z0-9_.-]+)@(([a-z0-9-]+\.)+[a-z]{2,6})$"
p = re.compile(pe, re.I | re.S)
m = p.search(email)
if not m:
    print("E-mail не соответствует шаблону")
else:
    print("E-mail", m.group(0), "соответствует шаблону")
    print("ящик:", m.group(1), "домен:", m.group(2))
input()
```

Результат выполнения (введенное пользователем значение выделено полужирным шрифтом):

```
Введите e-mail: user@mail.ru
E-mail user@mail.ru соответствует шаблону
ящик: user домен: mail.ru
```

7.3. Поиск всех совпадений с шаблоном

Для поиска всех совпадений с шаблоном предназначено несколько функций и методов.

- Метод `findall()` ищет все совпадения с шаблоном. Формат метода:

  ```
  findall(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствия найдены, возвращается список с фрагментами, в противном случае возвращается пустой список. Если внутри шаблона есть более одной группы, то каждый элемент списка будет кортежем, а не строкой:

  ```
  >>> import re
  >>> p = re.compile(r"[0-9]+")
  >>> p.findall("2007, 2008, 2009, 2010, 2011")
  ['2007', '2008', '2009', '2010', '2011']
  >>> p = re.compile(r"[a-z]+")
  >>> p.findall("2007, 2008, 2009, 2010, 2011")
  []
  >>> t = r"(([0-9]{3})-([0-9]{2})-([0-9]{2}))"
  >>> p = re.compile(t)
  >>> p.findall("322-77-20, 528-22-98")
  [('322-77-20', '322', '77', '20'),
   ('528-22-98', '528', '22', '98')]
  ```

- Вместо метода `findall()` можно воспользоваться функцией `findall()`. Формат функции:

  ```
  re.findall(<Шаблон>, <Строка>[, <Модификатор>])
  ```

  В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`:

  ```
  >>> re.findall(r"[0-9]+", "1 2 3 4 5 6")
  ['1', '2', '3', '4', '5', '6']
  >>> p = re.compile(r"[0-9]+")
  >>> re.findall(p, "1 2 3 4 5 6")
  ['1', '2', '3', '4', '5', '6']
  ```

- Метод `finditer()` аналогичен методу `findall()`, но возвращает итератор, а не список. На каждой итерации цикла возвращается объект `Match`. Формат метода:

  ```
  finditer(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Пример:

  ```
  >>> p = re.compile(r"[0-9]+")
  >>> for m in p.finditer("2007, 2008, 2009, 2010, 2011"):
      print(m.group(0), "start:", m.start(), "end:", m.end())

  2007 start: 0 end: 4
  2008 start: 6 end: 10
  2009 start: 12 end: 16
  2010 start: 18 end: 22
  2011 start: 24 end: 28
  ```

♦ Вместо метода `finditer()` можно воспользоваться функцией `finditer()`. Ее формат:

```
re.finditer(<Шаблон>, <Строка>[, <Модификатор>])
```

В параметре `<Шаблон>` указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Получим содержимое между тегами:

```
>>> p = re.compile(r"<b>(.+?)</b>", re.I | re.S)
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> for m in re.finditer(p, s):
        print(m.group(1))

Text1
Text3
```

7.4. Замена в строке

Метод `sub()` ищет все совпадения с шаблоном и заменяет их указанным значением. Если совпадения не найдены, возвращается исходная строка. Метод имеет следующий формат:

```
sub(<Новый фрагмент или ссылка на функцию>, <Строка для замены>
    [, <Максимальное количество замен>])
```

Внутри нового фрагмента можно использовать обратные ссылки `\номер группы`, `\g<номер группы>` и `\g<название группы>`. Для примера поменяем два тега местами:

```
>>> import re
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> p.sub(r"<\2><\1>", "<br><hr>")              # \номер
'<hr><br>'
>>> p.sub(r"<\g<2>><\g<1>>", "<br><hr>")        # \g<номер>
'<hr><br>'
>>> p.sub(r"<\g<tag2>><\g<tag1>>", "<br><hr>") # \g<название>
'<hr><br>'
```

В качестве первого параметра можно указать ссылку на функцию. В эту функцию будет передаваться объект `Match`, соответствующий найденному фрагменту. Результат, возвращаемый этой функцией, служит фрагментом для замены. Для примера найдем все числа в строке и прибавим к ним число `10` (листинг 7.7).

Листинг 7.7. Поиск чисел в строке

```
# -*- coding: utf-8 -*-
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    x = int(m.group(0))
    x += 10
    return "{0}".format(x)

p = re.compile(r"[0-9]+")
# Заменяем все вхождения
print(p.sub(repl, "2008, 2009, 2010, 2011"))
```

```
# Заменяем только первые два вхождения
print(p.sub(repl, "2008, 2009, 2010, 2011", 2))
input()
```

Результат выполнения:

```
2018, 2019, 2020, 2021
2018, 2019, 2010, 2011
```

> ***Внимание!***
> Название функции указывается без круглых скобок.

Вместо метода `sub()` можно воспользоваться функцией `sub()`. Формат функции:

```
re.sub(<Шаблон>, <Новый фрагмент или ссылка на функцию>,
       <Строка для замены>[, <Максимальное количество замен>
       [, flags=0]])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Для примера поменяем два тега местами, а также изменим регистр букв (листинг 7.8).

Листинг 7.8. Перестановка тегов с изменением регистра букв

```
# -*- coding: utf-8 -*-
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    tag1 = m.group("tag1").upper()
    tag2 = m.group("tag2").upper()
    return "<{0}><{1}>".format(tag2, tag1)
p = r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>"
print(re.sub(p, repl, "<br><hr>"))
input()
```

Результат выполнения:

```
<HR><BR>
```

Метод `subn()` аналогичен методу `sub()`, но возвращает не строку, а кортеж из двух элементов: измененной строки и количества произведенных замен. Метод имеет следующий формат:

```
subn(<Новый фрагмент или ссылка на функцию>, <Строка для замены>
    [, <Максимальное количество замен>])
```

Заменим все числа в строке на `0`:

```
>>> p = re.compile(r"[0-9]+")
>>> p.subn("0", "2008, 2009, 2010, 2011")
('0, 0, 0, 0', 4)
```

Вместо метода `subn()` можно воспользоваться функцией `subn()`. Формат функции:

```
re.subn(<Шаблон>, <Новый фрагмент или ссылка на функцию>,
        <Строка для замены>[, <Максимальное количество замен>
       [, flags=0]])
```

В качестве параметра <Шаблон> можно указать строку с регулярным выражением или скомпилированное регулярное выражение:

```
>>> p = r"200[79]"
>>> re.subn(p, "2001", "2007, 2008, 2009, 2010")
('2001, 2008, 2001, 2010', 2)
```

Для выполнения замен также можно использовать метод `expand()`, поддерживаемый объектом `Match`. Формат метода:

```
expand(<Шаблон>)
```

Внутри указанного шаблона можно использовать обратные ссылки: `\номер группы`, `\g<номер группы>` и `\g<название группы>`:

```
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> m = p.search("<br><hr>")
>>> m.expand(r"<\2><\1>")            # \номер
'<hr><br>'
>>> m.expand(r"<\g<2>><\g<1>>")      # \g<номер>
'<hr><br>'
>>> m.expand(r"<\g<tag2>><\g<tag1>>") # \g<название>
'<hr><br>'
```

7.5. Прочие функции и методы

Метод `split()` разбивает строку по шаблону и возвращает список подстрок. Его формат:

```
split(<Исходная строка>[, <Лимит>])
```

Если во втором параметре задано число, то в списке окажется указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент — с остатком строки:

```
>>> import re
>>> p = re.compile(r"[\s,.]+")
>>> p.split("word1, word2\nword3\r\nword4.word5")
['word1', 'word2', 'word3', 'word4', 'word5']
>>> p.split("word1, word2\nword3\r\nword4.word5", 2)
['word1', 'word2', 'word3\r\nword4.word5']
```

Если разделитель в строке не найден, список будет состоять только из одного элемента, содержащего исходную строку:

```
>>> p = re.compile(r"[0-9]+")
>>> p.split("word, word\nword")
['word, word\nword']
```

Вместо метода `split()` можно воспользоваться функцией `split()`. Формат функции:

```
re.split(<Шаблон>, <Исходная строка>[, <Лимит>[, flags=0]])
```

В качестве параметра <Шаблон> можно указать строку с регулярным выражением или скомпилированное регулярное выражение:

```
>>> p = re.compile(r"[\s,.]+")
>>> re.split(p, "word1, word2\nword3")
['word1', 'word2', 'word3']
```

```
>>> re.split(r"[\s,.]+", "word1, word2\nword3")
['word1', 'word2', 'word3']
```

Функция `escape(<Строка>)` экранирует все специальные символы в строке, после чего ее можно безопасно использовать внутри регулярного выражения:

```
>>> print(re.escape(r"[]().*"))
\[\]\(\)\.\*
```

Функция `purge()` выполняет очистку кэша, в котором хранятся промежуточные данные, используемые в процессе выполнения регулярных выражений. Ее рекомендуется вызывать после обработки большого количества регулярных выражений. Результата эта функция не возвращает:

```
>>> re.purge()
```

ГЛАВА 8

Списки, кортежи, множества и диапазоны

Списки, *кортежи*, *множества* и *диапазоны* — это нумерованные наборы объектов. Каждый элемент набора содержит лишь ссылку на объект — по этой причине они могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Позиция элемента в наборе задается *индексом*. Обратите внимание на то, что нумерация элементов начинается с 0, а не с 1.

Списки и кортежи являются просто упорядоченными последовательностями элементов. Как и все последовательности, они поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор +), повторение (оператор *), проверку на вхождение (оператор in) и невхождение (оператор not in).

♦ *Списки* относятся к изменяемым типам данных. Это означает, что мы можем не только получить элемент по индексу, но и изменить его:

```
>>> arr = [1, 2, 3]          # Создаем список
>>> arr[0]                   # Получаем элемент по индексу
1
>>> arr[0] = 50              # Изменяем элемент по индексу
>>> arr
[50, 2, 3]
```

♦ *Кортежи* относятся к неизменяемым типам данных. Иными словами, можно получить элемент по индексу, но изменить его нельзя:

```
>>> t = (1, 2, 3)            # Создаем кортеж
>>> t[0]                     # Получаем элемент по индексу
1
>>> t[0] = 50                # Изменить элемент по индексу нельзя!
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    t[0] = 50                # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

♦ *Множества* могут быть как изменяемыми, так и неизменяемыми. Их основное отличие от только что рассмотренных типов данных — хранение лишь уникальных значений (неуникальные значения автоматически отбрасываются):

```
>>> set([0, 1, 1, 2, 3, 3, 4])
{0, 1, 2, 3, 4}
```

- Что касается *диапазонов*, то они представляют собой наборы чисел, сформированные на основе заданных начального, конечного значений и величины шага между числами. Их важнейшее преимущество перед всеми остальными наборами объектов — небольшой объем занимаемой оперативной памяти:

```
>>> r = range(0, 101, 10)
>>> for i in r: print(i, end = " ")

0 10 20 30 40 50 60 70 80 90 100
```

Рассмотрим все упомянутые типы данных более подробно.

8.1. Создание списка

Создать список можно следующими способами:

- с помощью функции `list([<Последовательность>])`. Функция позволяет преобразовать любую последовательность в список. Если параметр не указан, создается пустой список:

```
>>> list()                   # Создаем пустой список
[]
>>> list("String")           # Преобразуем строку в список
['S', 't', 'r', 'i', 'n', 'g']
>>> list((1, 2, 3, 4, 5))    # Преобразуем кортеж в список
[1, 2, 3, 4, 5]
```

- указав все элементы списка внутри квадратных скобок:

```
>>> arr = [1, "str", 3, "4"]
>>> arr
[1, 'str', 3, '4']
```

- заполнив список поэлементно с помощью метода `append()`:

```
>>> arr = []              # Создаем пустой список
>>> arr.append(1)         # Добавляем элемент1 (индекс 0)
>>> arr.append("str")     # Добавляем элемент2 (индекс 1)
>>> arr
[1, 'str']
```

В некоторых языках программирования (например, в PHP) можно добавить элемент, указав пустые квадратные скобки или индекс больше последнего индекса. В языке Python все эти способы приведут к ошибке:

```
>>> arr = []
>>> arr[] = 10
SyntaxError: invalid syntax
>>> arr[0] = 10
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    arr[0] = 10
IndexError: list assignment index out of range
```

При создании списка в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел и строк, но для списков этого делать нельзя. Рассмотрим пример:

```
>>> x = y = [1, 2]        # Якобы создали два объекта
>>> x, y
([1, 2], [1, 2])
```

В этом примере мы создали список из двух элементов и присвоили значение переменным x и y. Теперь попробуем изменить значение в переменной y:

```
>>> y[1] = 100            # Изменяем второй элемент
>>> x, y                  # Изменилось значение сразу в двух переменных
([1, 100], [1, 100])
```

Как видно из примера, изменение значения в переменной y привело также к изменению значения в переменной x. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> x, y = [1, 2], [1, 2]
>>> y[1] = 100            # Изменяем второй элемент
>>> x, y
([1, 2], [1, 100])
```

Точно такая же ситуация возникает при использовании оператора повторения *. Например, в следующей инструкции производится попытка создания двух вложенных списков с помощью оператора *:

```
>>> arr = [ [] ] * 2      # Якобы создали два вложенных списка
>>> arr
[[], []]
>>> arr[0].append(5)      # Добавляем элемент
>>> arr                   # Изменились два элемента
[[5], [5]]
```

Создавать вложенные списки следует с помощью метода append() внутри цикла:

```
>>> arr = []
>>> for i in range(2): arr.append([])


>>> arr
[[], []]
>>> arr[0].append(5); arr
[[5], []]
```

Можно также воспользоваться генераторами списков:

```
>>> arr = [ [] for i in range(2) ]
>>> arr
[[], []]
>>> arr[0].append(5); arr
[[5], []]
```

Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор is. Если переменные ссылаются на один и тот же объект, оператор is возвращает значение True:

```
>>> x = y = [1, 2]         # Неправильно
>>> x is y # Переменные содержат ссылку на один и тот же список
True
```

```
>>> x, y = [1, 2], [1, 2] # Правильно
>>> x is y                # Это разные объекты
False
```

Но что же делать, если необходимо создать копию списка? Первый способ заключается в применении операции извлечения среза, второй — в использовании функции `list()`, а третий — в вызове метода `copy()`:

```
>>> x = [1, 2, 3, 4, 5] # Создали список
>>> # Создаем копию списка
>>> y = list(x) # или с помощью среза: y = x[:]
>>>             # или вызовом метода copy(): y = x.copy()
>>> y
[1, 2, 3, 4, 5]
>>> x is y # Оператор показывает, что это разные объекты
False
>>> y[1] = 100 # Изменяем второй элемент
>>> x, y       # Изменился только список в переменной y
([1, 2, 3, 4, 5], [1, 100, 3, 4, 5])
```

На первый взгляд может показаться, что мы получили копию — оператор `is` показывает, что это разные объекты, а изменение элемента затронуло лишь значение переменной `y`. В данном случае вроде все нормально. Но проблема заключается в том, что списки в языке Python могут иметь неограниченную степень вложенности. Рассмотрим это на примере:

```
>>> x = [1, [2, 3, 4, 5]] # Создали вложенный список
>>> y = list(x)           # Якобы сделали копию списка
>>> x is y                # Разные объекты
False
>>> y[1][1] = 100         # Изменяем элемент
>>> x, y                  # Изменение затронуло переменную x!!!
([1, [2, 100, 4, 5]], [1, [2, 100, 4, 5]])
```

Здесь мы создали список, в котором второй элемент является вложенным списком, после чего с помощью функции `list()` попытались создать копию списка. Как и в предыдущем примере, оператор `is` показывает, что это разные объекты, но посмотрите на результат — изменение переменной `y` затронуло и значение переменной `x`. Таким образом, функция `list()` и операция извлечения среза создают лишь *поверхностную копию* списка.

Чтобы получить полную копию списка, следует воспользоваться функцией `deepcopy()` из модуля `copy`:

```
>>> import copy            # Подключаем модуль copy
>>> x = [1, [2, 3, 4, 5]]
>>> y = copy.deepcopy(x)   # Делаем полную копию списка
>>> y[1][1] = 100          # Изменяем второй элемент
>>> x, y                   # Изменился только список в переменной y
([1, [2, 3, 4, 5]], [1, [2, 100, 4, 5]])
```

Функция `deepcopy()` создает копию каждого объекта, при этом сохраняя внутреннюю структуру списка. Иными словами, если в списке существуют два элемента, ссылающиеся на один объект, то будет создана копия объекта, и элементы будут ссылаться на этот новый объект, а не на разные объекты:

```
>>> import copy          # Подключаем модуль copy
>>> x = [1, 2]
```

```
>>> y = [x, x]            # Два элемента ссылаются на один объект
>>> y
[[1, 2], [1, 2]]
>>> z = copy.deepcopy(y) # Сделали копию списка
>>> z[0] is x, z[1] is x, z[0] is z[1]
(False, False, True)
>>> z[0][0] = 300         # Изменили один элемент
>>> z                     # Значение изменилось сразу в двух элементах!
[[300, 2], [300, 2]]
>>> x                     # Начальный список не изменился
[1, 2]
```

8.2. Операции над списками

Обращение к элементам списка осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Нумерация элементов списка начинается с нуля. Выведем все элементы списка:

```
>>> arr = [1, "str", 3.2, "4"]
>>> arr[0], arr[1], arr[2], arr[3]
(1, 'str', 3.2, '4')
```

С помощью позиционного присваивания можно присвоить значения элементов списка каким-либо переменным. Количество элементов справа и слева от оператора = должно совпадать, иначе будет выведено сообщение об ошибке:

```
>>> x, y, z = [1, 2, 3] # Позиционное присваивание
>>> x, y, z
(1, 2, 3)
>>> x, y = [1, 2, 3]    # Количество элементов должно совпадать
Traceback (most recent call last):
  File "<pyshell#86>", line 1, in <module>
    x, y = [1, 2, 3]    # Количество элементов должно совпадать
ValueError: too many values to unpack (expected 2)
```

В Python 3 при позиционном присваивании перед одной из переменных слева от оператора = можно указать звездочку. В этой переменной будет сохраняться список, состоящий из «лишних» элементов. Если таких элементов нет, список будет пустым:

```
>>> x, y, *z = [1, 2, 3]; x, y, z
(1, 2, [3])
>>> x, y, *z = [1, 2, 3, 4, 5]; x, y, z
(1, 2, [3, 4, 5])
>>> x, y, *z = [1, 2]; x, y, z
(1, 2, [])
>>> *x, y, z = [1, 2]; x, y, z
([], 1, 2)
>>> x, *y, z = [1, 2, 3, 4, 5]; x, y, z
(1, [2, 3, 4], 5)
>>> *z, = [1, 2, 3, 4, 5]; z
[1, 2, 3, 4, 5]
```

Так как нумерация элементов списка начинается с 0, индекс последнего элемента будет на единицу меньше количества элементов. Получить количество элементов списка позволяет функция len():

```
>>> arr = [1, 2, 3, 4, 5]
>>> len(arr)              # Получаем количество элементов
5
>>> arr[len(arr)-1]       # Получаем последний элемент
5
```

Если элемент, соответствующий указанному индексу, отсутствует в списке, возбуждается исключение IndexError:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[5]                # Обращение к несуществующему элементу
Traceback (most recent call last):
  File "<pyshell#99>", line 1, in <module>
    arr[5]                # Обращение к несуществующему элементу
IndexError: list index out of range
```

В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца списка, а точнее — чтобы получить положительный индекс, значение вычитается из общего количества элементов списка:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[-1], arr[len(arr)-1] # Обращение к последнему элементу
(5, 5)
```

Так как списки относятся к изменяемым типам данных, мы можем изменить элемент по индексу:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[0] = 600             # Изменение элемента по индексу
>>> arr
[600, 2, 3, 4, 5]
```

Кроме того, списки поддерживают операцию извлечения среза, которая возвращает указанный фрагмент списка. Формат операции:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры не являются обязательными. Если параметр <Начало> не указан, используется значение 0. Если параметр <Конец> не указан, возвращается фрагмент до конца списка. Следует также заметить, что элемент с индексом, указанным в этом параметре, не входит в возвращаемый фрагмент. Если параметр <Шаг> не указан, используется значение 1. В качестве значения параметров можно указать отрицательные значения.

Теперь рассмотрим несколько примеров:

- сначала получим поверхностную копию списка:

  ```
  >>> arr = [1, 2, 3, 4, 5]
  >>> m = arr[:]; m # Создаем поверхностную копию и выводим значения
  [1, 2, 3, 4, 5]
  >>> m is arr      # Оператор is показывает, что это разные объекты
  False
  ```

♦ затем выведем символы в обратном порядке:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[::-1]              # Шаг -1
[5, 4, 3, 2, 1]
```

♦ выведем список без первого и последнего элементов:

```
>>> arr[1:]                # Без первого элемента
[2, 3, 4, 5]
>>> arr[:-1]               # Без последнего элемента
[1, 2, 3, 4]
```

♦ получим первые два элемента списка:

```
>>> arr[0:2]               # Символ с индексом 2 не входит в диапазон
[1, 2]
```

♦ а так получим последний элемент:

```
>>> arr[-1:]               # Последний элемент списка
[5]
```

♦ и, наконец, выведем фрагмент от второго элемента до четвертого включительно:

```
>>> arr[1:4] # Возвращаются элементы с индексами 1, 2 и 3
[2, 3, 4]
```

С помощью среза можно изменить фрагмент списка. Если срезу присвоить пустой список, то элементы, попавшие в срез, будут удалены:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[1:3] = [6, 7] # Изменяем значения элементов с индексами 1 и 2
>>> arr
[1, 6, 7, 4, 5]
>>> arr[1:3] = []      # Удаляем элементы с индексами 1 и 2
>>> arr
[1, 4, 5]
```

Объединить два списка в один список позволяет оператор `+`. Результатом объединения будет новый список:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = [6, 7, 8, 9]
>>> arr3 = arr1 + arr2
>>> arr3
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Вместо оператора `+` можно использовать оператор `+=`. Следует учитывать, что в этом случае элементы добавляются в текущий список:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr += [6, 7, 8, 9]
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Кроме рассмотренных операций, списки поддерживают операцию повторения и проверку на вхождение. Повторить список указанное количество раз можно с помощью оператора `*`, а выполнить проверку на вхождение элемента в список позволяет оператор `in`:

```
>>> [1, 2, 3] * 3                                # Операция повторения
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5] # Проверка на вхождение
(True, False)
```

8.3. Многомерные списки

Любой элемент списка может содержать объект произвольного типа. Например, элемент списка может быть числом, строкой, списком, кортежем, словарем и т. д. Создать вложенный список можно, например, так:

```
>>> arr = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Как вы уже знаете, выражение внутри скобок может располагаться на нескольких строках. Следовательно, предыдущий пример можно записать иначе:

```
>>> arr = [
               [1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]
          ]
```

Чтобы получить значение элемента во вложенном списке, следует указать два индекса:

```
>>> arr[1][1]
5
```

Элементы вложенного списка также могут иметь элементы произвольного типа. Количество вложений не ограничено, и мы можем создать объект любой степени сложности. В этом случае для доступа к элементам указывается несколько индексов подряд:

```
>>> arr = [ [1, ["a", "b"], 3], [4, 5, 6], [7, 8, 9] ]
>>> arr[0][1][0]
'a'
>>> arr = [ [1, { "a": 10, "b": ["s", 5] } ] ]
>>> arr[0][1]["b"][0]
's'
```

8.4. Перебор элементов списка

Перебрать все элементы списка можно с помощью цикла `for`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> for i in arr: print(i, end=" ")

1 2 3 4 5
```

Следует заметить, что переменную `i` внутри цикла можно изменить, но если она ссылается на неизменяемый тип данных (например, на число или строку), это не отразится на исходном списке:

```
>>> arr = [1, 2, 3, 4]        # Элементы имеют неизменяемый тип (число)
>>> for i in arr: i += 10

>>> arr                       # Список не изменился
[1, 2, 3, 4]
```

```
>>> arr = [ [1, 2], [3, 4] ] # Элементы имеют изменяемый тип (список)
>>> for i in arr: i[0] += 10
>>> arr                      # Список изменился
[[11, 2], [13, 4]]
```

Для генерации индексов элементов можно воспользоваться функцией `range()`, которая возвращает объект-диапазон, поддерживающий итерации. С помощью такого диапазона внутри цикла `for` можно получить текущий индекс. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если он не указан, используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон значений. Если параметр `<Шаг>` не указан, используется значение `1`. Для примера умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)                   # Результат выполнения: [2, 4, 6, 8]
```

Можно также воспользоваться функцией `enumerate(<Объект>[, start=0])`, которая на каждой итерации цикла `for` возвращает кортеж из индекса и значения текущего элемента списка. Умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i, elem in enumerate(arr):
    arr[i] *= 2
print(arr)                   # Результат выполнения: [2, 4, 6, 8]
```

Кроме того, перебрать элементы можно с помощью цикла `while`, но нужно помнить, что он выполняется медленнее цикла `for`. Для примера умножим каждый элемент списка на 2, используя цикл `while`:

```
arr = [1, 2, 3, 4]
i, c = 0, len(arr)
while i < c:
    arr[i] *= 2
    i += 1
print(arr)                   # Результат выполнения: [2, 4, 6, 8]
```

8.5. Генераторы списков и выражения-генераторы

В предыдущем разделе мы изменяли элементы списка следующим образом:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)                   # Результат выполнения: [2, 4, 6, 8]
```

С помощью генераторов списков тот же самый код можно записать более компактно, к тому же генераторы списков работают быстрее цикла `for`. Однако вместо изменения исходного списка возвращается новый список:

```
arr = [1, 2, 3, 4]
arr = [ i * 2 for i in arr ]
print(arr)                   # Результат выполнения: [2, 4, 6, 8]
```

Как видно из примера, мы поместили цикл `for` внутри квадратных скобок, а также изменили порядок следования параметров, — инструкция, выполняемая внутри цикла, находится перед циклом. Обратите внимание и на то, что выражение внутри цикла не содержит оператора присваивания, — на каждой итерации цикла будет генерироваться новый элемент, которому неявным образом присваивается результат выполнения выражения внутри цикла. В итоге будет создан новый список, содержащий измененные значения элементов исходного списка.

Генераторы списков могут иметь сложную структуру — например, состоять из нескольких вложенных циклов `for` и (или) содержать оператор ветвления `if` после цикла. Для примера получим четные элементы списка и умножим их на 10:

```
arr = [1, 2, 3, 4]
arr = [ i * 10 for i in arr if i % 2 == 0 ]
print(arr)                    # Результат выполнения: [20, 40]
```

Этот код эквивалентен коду:

```
arr = []
for i in [1, 2, 3, 4]:
    if i % 2 == 0:            # Если число четное
        arr.append(i * 10)    # Добавляем элемент
print(arr)                    # Результат выполнения: [20, 40]
```

Усложним наш пример — получим четные элементы вложенного списка и умножим их на 10:

```
arr = [[1, 2], [3, 4], [5, 6]]
arr = [ j * 10 for i in arr for j in i if j % 2 == 0 ]
print(arr)                    # Результат выполнения: [20, 40, 60]
```

Этот код эквивалентен коду:

```
arr = []
for i in [[1, 2], [3, 4], [5, 6]]:
    for j in i:
        if j % 2 == 0:            # Если число четное
            arr.append(j * 10)    # Добавляем элемент
print(arr)                        # Результат выполнения: [20, 40, 60]
```

Если выражение разместить внутри не квадратных, а круглых скобок, то будет возвращаться не список, а итератор. Такие конструкции называются *выражениями-генераторами*. В качестве примера просуммируем четные числа в списке:

```
>>> arr = [1, 4, 12, 45, 10]
>>> sum((i for i in arr if i % 2 == 0))
26
```

8.6. Функции *map()*, *zip()*, *filter()* и *reduce()*

Встроенная функция `map()` позволяет применить заданную в параметре функцию к каждому элементу последовательности. Она имеет такой формат:

```
map(<Функция>, <Последовательность1>[, ..., <ПоследовательностьN>])
```

Функция `map()` возвращает объект, поддерживающий итерации (а не список, как это было ранее в Python 2). Чтобы получить список в версии Python 3, возвращенный результат необходимо передать в функцию `list()`.

В качестве параметра `<Функция>` указывается ссылка на функцию (название функции без круглых скобок), которой будет передаваться текущий элемент последовательности. Внутри этой функции необходимо вернуть новое значение. Для примера прибавим к каждому элементу списка число 10 (листинг 8.1).

Листинг 8.1. Функция `map()`

```
def func(elem):
    """ Увеличение значения каждого элемента списка """
    return elem + 10 # Возвращаем новое значение

arr = [1, 2, 3, 4, 5]
print( list( map(func, arr) ) )
# Результат выполнения: [11, 12, 13, 14, 15]
```

Функции `map()` можно передать несколько последовательностей. В этом случае в функцию обратного вызова будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковом смещении. Просуммируем элементы трех списков (листинг 8.2).

Листинг 8.2. Суммирование элементов трех списков

```
def func(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3 # Возвращаем новое значение

arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222, 333, 444, 555]
```

Если количество элементов в последовательностях различается, за основу выбирается последовательность с минимальным количеством элементов (листинг 8.3).

Листинг 8.3. Суммирование элементов трех списков разной длины

```
def func(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3

arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20]
arr3 = [100, 200, 300, 400, 500]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222]
```

Встроенная функция `zip()` на каждой итерации возвращает кортеж, содержащий элементы последовательностей, которые расположены на одинаковом смещении. Функция возвраща-

ет объект, поддерживающий итерации (а не список, как это было ранее в Python 2). Чтобы получить список в версии Python 3, необходимо результат передать в функцию `list()`. Формат функции:

```
zip(<Последовательность1>[, ..., <ПоследовательностьN>])
```

Пример:

```
>>> zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
<zip object at 0x00FCAC88>
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Если количество элементов в последовательностях будет разным, то в результат попадут только элементы, которые существуют во всех последовательностях на одинаковом смещении:

```
>>> list(zip([1, 2, 3], [4, 6], [7, 8, 9, 10]))
[(1, 4, 7), (2, 6, 8)]
```

В качестве еще одного примера переделаем нашу программу суммирования элементов трех списков (см. листинг 8.3) и используем функцию `zip()` вместо функции `map()` (листинг 8.4).

Листинг 8.4. Суммирование элементов трех списков с помощью функции `zip()`

```
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
arr = [x + y + z for (x, y, z) in zip(arr1, arr2, arr3)]
print(arr)
# Результат выполнения: [111, 222, 333, 444, 555]
```

Функция `filter()` позволяет выполнить проверку элементов последовательности. Формат функции:

```
filter(<Функция>, <Последовательность>)
```

Если в первом параметре вместо названия функции указать значение `None`, то каждый элемент последовательности будет проверен на соответствие значению `True`. Если элемент в логическом контексте возвращает значение `False`, то он не будет добавлен в возвращаемый результат. Функция возвращает объект, поддерживающий итерации (а не список или кортеж, как это было ранее в Python 2). Чтобы получить список в версии Python 3, необходимо результат передать в функцию `list()`:

```
>>> filter(None, [1, 0, None, [], 2])
<filter object at 0x00FD58B0>
>>> list(filter(None, [1, 0, None, [], 2]))
[1, 2]
```

Аналогичная операция с использованием генераторов списков выглядит так:

```
>>> [ i for i in [1, 0, None, [], 2] if i ]
[1, 2]
```

В первом параметре можно указать ссылку на функцию, в которую в качестве параметра будет передаваться текущий элемент последовательности. Если элемент нужно добавить

в возвращаемое функцией filter() значение, то внутри функции, указанной в качестве первого параметра, следует вернуть значение True, в противном случае — значение False. Для примера удалим все отрицательные значения из списка (листинг 8.5).

Листинг 8.5. Пример использования функции filter()

```
def func(elem):
    return elem >= 0

arr = [-1, 2, -3, 4, 0, -20, 10]
arr = list(filter(func, arr))
print(arr)                              # Результат: [2, 4, 0, 10]
# Использование генераторов списков
arr = [-1, 2, -3, 4, 0, -20, 10]
arr = [ i for i in arr if func(i) ]
print(arr)                              # Результат: [2, 4, 0, 10]
```

Функция reduce() из модуля functools применяет указанную функцию к парам элементов и накапливает результат. Функция имеет следующий формат:

```
reduce(<Функция>, <Последовательность>[, <Начальное значение>])
```

В параметр <Функция> в качестве параметров передаются два элемента: первый элемент будет содержать результат предыдущих вычислений, а второй — значение текущего элемента. Получим сумму всех элементов списка (листинг 8.6).

Листинг 8.6. Пример использования функции reduce()

```
from functools import reduce # Подключаем модуль

def func(x, y):
    print("({0}, {1})".format(x, y), end=" ")
    return x + y

arr = [1, 2, 3, 4, 5]
summa = reduce(func, arr)
# Последовательность: (1, 2) (3, 3) (6, 4) (10, 5)
print(summa) # Результат выполнения: 15
summa = reduce(func, arr, 10)
# Последовательность: (10, 1) (11, 2) (13, 3) (16, 4) (20, 5)
print(summa) # Результат выполнения: 25
summa = reduce(func, [], 10)
print(summa) # Результат выполнения: 10
```

8.7. Добавление и удаление элементов списка

Для добавления и удаления элементов списка используются следующие методы:

- append(<Объект>) — добавляет один объект в конец списка. Метод изменяет текущий список и ничего не возвращает:

```
>>> arr = [1, 2, 3]
>>> arr.append(4); arr          # Добавляем число
[1, 2, 3, 4]
>>> arr.append([5, 6]); arr     # Добавляем список
[1, 2, 3, 4, [5, 6]]
>>> arr.append((7, 8)); arr     # Добавляем кортеж
[1, 2, 3, 4, [5, 6], (7, 8)]
```

- `extend(<Последовательность>)` — добавляет элементы последовательности в конец списка. Метод изменяет текущий список и ничего не возвращает:

```
>>> arr = [1, 2, 3]
>>> arr.extend([4, 5, 6])       # Добавляем список
>>> arr.extend((7, 8, 9))       # Добавляем кортеж
>>> arr.extend("abc")           # Добавляем буквы из строки
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c']
```

Добавить несколько элементов можно с помощью операции конкатенации или оператора +=:

```
>>> arr = [1, 2, 3]
>>> arr + [4, 5, 6]             # Возвращает новый список
[1, 2, 3, 4, 5, 6]
>>> arr += [4, 5, 6]            # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6]
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> arr = [1, 2, 3]
>>> arr[len(arr):] = [4, 5, 6]  # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6]
```

- `insert(<Индекс>, <Объект>)` — добавляет один объект в указанную позицию. Остальные элементы смещаются. Метод изменяет текущий список и ничего не возвращает:

```
>>> arr = [1, 2, 3]
>>> arr.insert(0, 0); arr   # Вставляем 0 в начало списка
[0, 1, 2, 3]
>>> arr.insert(-1, 20); arr # Можно указать отрицательные числа
[0, 1, 2, 20, 3]
>>> arr.insert(2, 100); arr # Вставляем 100 в позицию 2
[0, 1, 100, 2, 20, 3]
>>> arr.insert(10, [4, 5]); arr # Добавляем список
[0, 1, 100, 2, 20, 3, [4, 5]]
```

Метод `insert()` позволяет добавить только один объект. Чтобы добавить несколько объектов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало списка:

```
>>> arr = [1, 2, 3]
>>> arr[:0] = [-2, -1, 0]
>>> arr
[-2, -1, 0, 1, 2, 3]
```

- `pop([<Индекс>])` — удаляет элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, то удаляет и возвращает последний элемент списка. Если элемента с указанным индексом нет, или список пустой, возбуждается исключение `IndexError`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.pop()          # Удаляем последний элемент списка
5
>>> arr                # Список изменился
[1, 2, 3, 4]
>>> arr.pop(0)         # Удаляем первый элемент списка
1
>>> arr                # Список изменился
[2, 3, 4]
```

  Удалить элемент списка позволяет также оператор `del`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> del arr[4]; arr  # Удаляем последний элемент списка
[1, 2, 3, 4]
>>> del arr[:2]; arr # Удаляем первый и второй элементы
[3, 4]
```

- `remove(<Значение>)` — удаляет первый элемент, содержащий указанное значение. Если элемент не найден, возбуждается исключение `ValueError`. Метод изменяет текущий список и ничего не возвращает:

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.remove(1)    # Удаляет только первый элемент
>>> arr
[2, 3, 1, 1]
>>> arr.remove(5)    # Такого элемента нет
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    arr.remove(5)    # Такого элемента нет
ValueError: list.remove(x): x not in list
```

- `clear()` — удаляет все элементы списка, очищая его. Никакого результата при этом не возвращается:

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.clear()
>>> arr
[]
```

Если необходимо удалить все повторяющиеся элементы списка, то можно преобразовать список во множество, а затем множество обратно преобразовать в список. Обратите внимание на то, что список должен содержать только неизменяемые объекты (например, числа, строки или кортежи). В противном случае возбуждается исключение `TypeError`:

```
>>> arr = [1, 2, 3, 1, 1, 2, 2, 3, 3]
>>> s = set(arr)          # Преобразуем список во множество
>>> s
{1, 2, 3}
>>> arr = list(s)         # Преобразуем множество в список
>>> arr                   # Все повторы были удалены
[1, 2, 3]
```

8.8. Поиск элемента в списке и получение сведений о значениях, входящих в список

Как вы уже знаете, выполнить проверку на *вхождение* элемента в список позволяет оператор `in`: если элемент входит в список, то возвращается значение `True`, в противном случае — `False`. Аналогичный оператор `not in` выполняет проверку на *невхождение* элемента в список: если элемент отсутствует в списке, возвращается `True`, в противном случае — `False`:

```
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5] # Проверка на вхождение
(True, False)
>>> 2 not in [1, 2, 3, 4, 5], 6 not in [1, 2, 3, 4, 5] # Проверка на невхождение
(False, True)
```

Тем не менее оба этих оператора не дают никакой информации о местонахождении элемента внутри списка. Чтобы узнать индекс элемента внутри списка, следует воспользоваться методом `index()`. Формат метода:

```
index(<Значение>[, <Начало>[, <Конец>]])
```

Метод `index()` возвращает индекс элемента, имеющего указанное значение. Если значение не входит в список, то возбуждается исключение `ValueError`. Если второй и третий параметры не указаны, поиск будет производиться с начала и до конца списка:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.index(1), arr.index(2)
(0, 1)
>>> arr.index(1, 1), arr.index(1, 3, 5)
(2, 4)
>>> arr.index(3)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    arr.index(3)
ValueError: 3 is not in list
```

Узнать общее количество элементов с указанным значением позволяет метод `count(<Значение>)`. Если элемент не входит в список, возвращается значение `0`:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.count(1), arr.count(2)
(3, 2)
>>> arr.count(3)                    # Элемент не входит в список
0
```

С помощью функций `max()` и `min()` можно узнать максимальное и минимальное значение из всех, что входят в список, соответственно:

```
>>> arr = [1, 2, 3, 4, 5]
>>> max(arr), min(arr)
(5, 1)
```

Функция `any(<Последовательность>)` возвращает значение `True`, если в последовательности существует хотя бы один элемент, который в логическом контексте возвращает значение `True`. Если последовательность не содержит элементов, возвращается значение `False`:

```
>>> any([0, None]), any([0, None, 1]), any([])
(False, True, False)
```

Функция `all(<Последовательность>)` возвращает значение `True`, если все элементы последовательности в логическом контексте возвращают значение `True` или последовательность не содержит элементов:

```
>>> all([0, None]), all([0, None, 1]), all([]), all(["str", 10])
(False, False, True, True)
```

8.9. Переворачивание и перемешивание списка

Метод `reverse()` изменяет порядок следования элементов списка на противоположный. Метод изменяет текущий список и ничего не возвращает:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.reverse()              # Изменяется текущий список
>>> arr
[5, 4, 3, 2, 1]
```

Если необходимо изменить порядок следования и получить новый список, следует воспользоваться функцией `reversed(<Последовательность>)`. Она возвращает итератор, который можно преобразовать в список с помощью функции `list()` или генератора списков:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> reversed(arr)
<list_reverseiterator object at 0x00FD5150>
>>> list(reversed(arr))             # Использование функции list()
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> for i in reversed(arr): print(i, end=" ") # Вывод с помощью цикла

10 9 8 7 6 5 4 3 2 1
>>> [i for i in reversed(arr)] # Использование генератора списков
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Функция `shuffle(<Список>[, <Число от 0.0 до 1.0>])` из модуля `random` перемешивает список случайным образом. Функция перемешивает сам список и ничего не возвращает. Если второй параметр не указан, используется значение, возвращаемое функцией `random()`:

```
>>> import random          # Подключаем модуль random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)    # Перемешиваем список случайным образом
>>> arr
[2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
```

8.10. Выбор элементов случайным образом

Получить элементы из списка случайным образом позволяют следующие функции из модуля `random`:

- `choice(<Последовательность>)` — возвращает случайный элемент из любой последовательности (строки, списка, кортежа):

  ```
  >>> import random # Подключаем модуль random
  >>> random.choice(["s", "t", "r"]) # Список
  's'
  ```

♦ `sample(<Последовательность>, <Количество элементов>)` — возвращает список из указанного количества элементов. В этот список попадут элементы из последовательности, выбранные случайным образом. В качестве последовательности можно указать любые объекты, поддерживающие итерации:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
[7, 10]
>>> arr                          # Сам список не изменяется
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

8.11. Сортировка списка

Отсортировать список позволяет метод `sort()`. Он имеет следующий формат:

```
sort([key=None][, reverse=False])
```

Все параметры не являются обязательными. Метод изменяет текущий список и ничего не возвращает. Отсортируем список по возрастанию с параметрами по умолчанию:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort()                               # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Чтобы отсортировать список по убыванию, следует в параметре `reverse` указать значение `True`:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort(reverse=True)                   # Сортировка по убыванию
>>> arr
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Надо заметить, что стандартная сортировка зависит от регистра символов (листинг 8.7).

Листинг 8.7. Стандартная сортировка

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort()
for i in arr:
    print(i, end=" ")
# Результат выполнения: Единица2 Единый единица1
```

В параметре `key` метода `sort()` можно указать функцию, выполняющую какое-либо действие над каждым элементом списка. В качестве единственного параметра она должна принимать значение очередного элемента списка, а в качестве результата — возвращать результат действий над ним. Этот результат будет участвовать в процессе сортировки, но значения самих элементов списка не изменятся.

Выполнив пример из листинга 8.7, мы получили неправильный результат сортировки, ведь `Единый` и `Единица2` больше `единица1`. Чтобы регистр символов не учитывался, в параметре `key` мы укажем функцию для изменения регистра символов (листинг 8.8).

Листинг 8.8. Пользовательская сортировка

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=str.lower)                    # Указываем метод lower()
for i in arr:
    print(i, end=" ")
# Результат выполнения: единица1 Единица2 Единый
```

Метод `sort()` сортирует сам список и не возвращает никакого значения. В некоторых случаях необходимо получить отсортированный список, а текущий список оставить без изменений. Для этого следует воспользоваться функцией `sorted()`. Функция имеет следующий формат:

```
sorted(<Последовательность>[, key=None][, reverse=False])
```

В первом параметре указывается список, который необходимо отсортировать. Остальные параметры эквивалентны параметрам метода `sort()`. Вот пример использования функции `sorted()`:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> sorted(arr)               # Возвращает новый список!
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sorted(arr, reverse=True) # Возвращает новый список!
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> arr = ["единица1", "Единый", "Единица2"]
>>> sorted(arr, key=str.lower)
['единица1', 'Единица2', 'Единый']
```

8.12. Заполнение списка числами

Создать список, содержащий диапазон чисел, можно с помощью функции `range()`. Она возвращает диапазон, который преобразуется в список вызовом функции `list()`. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение, а если он не указан, используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон. Если параметр `<Шаг>` не указан, используется значение `1`. В качестве примера заполним список числами от `0` до `10`:

```
>>> list(range(11))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Создадим список, состоящий из диапазона чисел от `1` до `15`:

```
>>> list(range(1, 16))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Теперь изменим порядок следования чисел на противоположный:

```
>>> list(range(15, 0, -1))
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Если необходимо получить список со случайными числами (или случайными элементами из другого списка), то следует воспользоваться функцией `sample(<Последовательность>, <Количество элементов>)` из модуля `random`:

```
>>> import random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 3)
[1, 9, 5]
>>> random.sample(range(300), 5)
[259, 294, 142, 292, 245]
```

8.13. Преобразование списка в строку

Преобразовать список в строку позволяет метод `join()`. Элементы добавляются в формируемую строку через указанный разделитель. Формат метода:

```
<Строка> = <Разделитель>.join(<Последовательность>)
```

Пример:

```
>>> arr = ["word1", "word2", "word3"]
>>> " — ".join(arr)
'word1 — word2 — word3'
```

Обратите внимание на то, что элементы списка должны быть строками, иначе возвращается исключение `TypeError`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " — ".join(arr)
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    " — ".join(arr)
TypeError: sequence item 3: expected str instance, int found
```

Избежать этого исключения можно с помощью выражения-генератора, внутри которого текущий элемент списка преобразуется в строку с помощью функции `str()`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " — ".join( ( str(i) for i in arr ) )
'word1 — word2 — word3 — 2'
```

Кроме того, с помощью функции `str()` можно сразу получить строковое представление списка:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> str(arr)
"['word1', 'word2', 'word3', 2]"
```

8.14. Кортежи

Кортежи, как и списки, являются упорядоченными последовательностями элементов. Они во многом аналогичны спискам, но имеют одно очень важное отличие — изменить кортеж нельзя. Можно сказать, что кортеж — это список, доступный только для чтения.

Создать кортеж можно следующими способами:

- с помощью функции `tuple([<Последовательность>])`. Функция позволяет преобразовать любую последовательность в кортеж. Если параметр не указан, создается пустой кортеж:

```
>>> tuple()                  # Создаем пустой кортеж
()
>>> tuple("String")          # Преобразуем строку в кортеж
('S', 't', 'r', 'i', 'n', 'g')
>>> tuple([1, 2, 3, 4, 5]) # Преобразуем список в кортеж
(1, 2, 3, 4, 5)
```

♦ указав все элементы через запятую внутри круглых скобок (или без скобок):

```
>>> t1 = ()          # Создаем пустой кортеж
>>> t2 = (5,)        # Создаем кортеж из одного элемента
>>> t3 = (1, "str", (3, 4)) # Кортеж из трех элементов
>>> t4 = 1, "str", (3, 4)   # Кортеж из трех элементов
>>> t1, t2, t3, t4
((), (5,), (1, 'str', (3, 4)), (1, 'str', (3, 4)))
```

Обратите особое внимание на вторую строку примера. Чтобы создать кортеж из одного элемента, необходимо в конце указать запятую. Именно запятые формируют кортеж, а не круглые скобки. Если внутри круглых скобок нет запятых, будет создан объект другого типа:

```
>>> t = (5); type(t)       # Получили число, а не кортеж!
<class 'int'>
>>> t = ("str"); type(t)  # Получили строку, а не кортеж!
<class 'str'>
```

Четвертая строка в исходном примере также доказывает, что не скобки формируют кортеж, а запятые. Помните, что любое выражение в языке Python можно заключить в круглые скобки, а чтобы получить кортеж, необходимо указать запятые.

Позиция элемента в кортеже задается *индексом*. Обратите внимание на то, что нумерация элементов кортежа (как и списка) начинается с `0`, а не с `1`. Как и все последовательности, кортежи поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`) и невхождение (оператор `not in`):

```
>>> t = (1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t[0]                    # Получаем значение первого элемента кортежа
1
>>> t[::-1]                 # Изменяем порядок следования элементов
(9, 8, 7, 6, 5, 4, 3, 2, 1)
>>> t[2:5]                  # Получаем срез
(3, 4, 5)
>>> 8 in t, 0 in t          # Проверка на вхождение
(True, False)
>>> (1, 2, 3) * 3           # Повторение
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> (1, 2, 3) + (4, 5, 6) # Конкатенация
(1, 2, 3, 4, 5, 6)
```

Кортежи, как уже неоднократно отмечалось, относятся к неизменяемым типам данных. Иными словами, можно получить элемент по индексу, но изменить его нельзя:

```
>>> t = (1, 2, 3)           # Создаем кортеж
>>> t[0]                    # Получаем элемент по индексу
1
```

```
>>> t[0] = 50              # Изменить элемент по индексу нельзя!
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in <module>
    t[0] = 50              # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

Кортежи поддерживают уже знакомые нам по спискам функции `len()`, `min()`, `max()`, методы `index()` и `count()`:

```
>>> t = (1, 2, 3)           # Создаем кортеж
>>> len(t)                  # Получаем количество элементов
3
>>> t = (1, 2, 1, 2, 1)
>>> t.index(1), t.index(2) # Ищем элементы в кортеже
(0, 1)
```

8.15. Множества

Множество — это неупорядоченная последовательность уникальных элементов. Объявить множество можно с помощью функции `set()`:

```
>>> s = set()
>>> s
set([])
```

Функция `set()` также позволяет преобразовать элементы последовательности во множество:

```
>>> set("string")                    # Преобразуем строку
set(['g', 'i', 'n', 's', 'r', 't'])
>>> set([1, 2, 3, 4, 5])             # Преобразуем список
set([1, 2, 3, 4, 5])
>>> set((1, 2, 3, 4, 5))             # Преобразуем кортеж
set([1, 2, 3, 4, 5])
>>> set([1, 2, 3, 1, 2, 3]) # Остаются только уникальные элементы
set([1, 2, 3])
```

Перебрать элементы множества позволяет цикл `for`:

```
>>> for i in set([1, 2, 3]): print i
1 2 3
```

Получить количество элементов множества позволяет функция `len()`:

```
>>> len(set([1, 2, 3]))
3
```

Для работы с множествами предназначены следующие операторы и соответствующие им методы:

- `|` и `union()` — объединяют два множества:

  ```
  >>> s = set([1, 2, 3])
  >>> s.union(set([4, 5, 6])), s | set([4, 5, 6])
  (set([1, 2, 3, 4, 5, 6]), set([1, 2, 3, 4, 5, 6]))
  ```

  Если элемент уже содержится во множестве, то он повторно добавлен не будет:

  ```
  >>> set([1, 2, 3]) | set([1, 2, 3])
  set([1, 2, 3])
  ```

♦ `a |= b` и `a.update(b)` — добавляют элементы множества `b` во множество `a`:

```
>>> s = set([1, 2, 3])
>>> s.update(set([4, 5, 6]))
>>> s
set([1, 2, 3, 4, 5, 6])
>>> s |= set([7, 8, 9])
>>> s
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

♦ `-` и `difference()` — вычисляют разницу множеств:

```
>>> set([1, 2, 3]) - set([1, 2, 4])
set([3])
>>> s = set([1, 2, 3])
>>> s.difference(set([1, 2, 4]))
set([3])
```

♦ `a -= b` и `a.difference_update(b)` — удаляют элементы из множества `a`, которые существуют и во множестве `a`, и во множестве `b`:

```
>>> s = set([1, 2, 3])
>>> s.difference_update(set([1, 2, 4]))
>>> s
set([3])
>>> s -= set([3, 4, 5])
>>> s
set([])
```

♦ `&` и `intersection()` — пересечение множеств. Позволяют получить элементы, которые существуют в обоих множествах:

```
>>> set([1, 2, 3]) & set([1, 2, 4])
set([1, 2])
>>> s = set([1, 2, 3])
>>> s.intersection(set([1, 2, 4]))
set([1, 2])
```

♦ `a &= b` и `a.intersection_update(b)` — во множестве `a` останутся элементы, которые существуют и во множестве `a`, и во множестве `b`:

```
>>> s = set([1, 2, 3])
>>> s.intersection_update(set([1, 2, 4]))
>>> s
set([1, 2])
>>> s &= set([1, 6, 7])
>>> s
set([1])
```

♦ `^` и `symmetric_difference()` — возвращают все элементы обоих множеств, исключая элементы, которые присутствуют в обоих этих множествах:

```
>>> s = set([1, 2, 3])
>>> s ^ set([1, 2, 4]), s.symmetric_difference(set([1, 2, 4]))
(set([3, 4]), set([3, 4]))
>>> s ^ set([1, 2, 3]), s.symmetric_difference(set([1, 2, 3]))
(set([]), set([]))
```

```
>>> s ^ set([4, 5, 6]), s.symmetric_difference(set([4, 5, 6]))
(set([1, 2, 3, 4, 5, 6]), set([1, 2, 3, 4, 5, 6]))
```

- `a ^= b` и `a.symmetric_difference_update(b)` — во множестве `a` будут все элементы обоих множеств, исключая те, что присутствуют в обоих этих множествах:

```
>>> s = set([1, 2, 3])
>>> s.symmetric_difference_update(set([1, 2, 4]))
>>> s
set([3, 4])
>>> s ^= set([3, 5, 6])
>>> s
set([4, 5, 6])
```

Операторы сравнения множеств:

- `in` — проверка наличия элемента во множестве:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 in s, 12 in s
(True, False)
```

- `not in` — проверка отсутствия элемента во множестве:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 in s, 12 in s
(False, True)
```

- `==` — проверка на равенство:

```
>>> set([1, 2, 3]) == set([1, 2, 3])
True
>>> set([1, 2, 3]) == set([3, 2, 1])
True
>>> set([1, 2, 3]) == set([1, 2, 3, 4])
False
```

- `a <= b` и `a.issubset(b)` — проверяют, входят ли все элементы множества `a` во множество `b`:

```
>>> s = set([1, 2, 3])
>>> s <= set([1, 2]), s <= set([1, 2, 3, 4])
(False, True)
>>> s.issubset(set([1, 2])), s.issubset(set([1, 2, 3, 4]))
(False, True)
```

- `a < b` — проверяет, входят ли все элементы множества `a` во множество `b`, причем множество `a` не должно быть равно множеству `b`:

```
>>> s = set([1, 2, 3])
>>> s < set([1, 2, 3]), s < set([1, 2, 3, 4])
(False, True)
```

- `a >= b` и `a.issuperset(b)` — проверяют, входят ли все элементы множества `b` во множество `a`:

```
>>> s = set([1, 2, 3])
>>> s >= set([1, 2]), s >= set([1, 2, 3, 4])
(True, False)
```

```
>>> s.issuperset(set([1, 2])), s.issuperset(set([1, 2, 3, 4]))
(True, False)
```

- `a > b` — проверяет, входят ли все элементы множества `b` во множество `a`, причем множество `a` не должно быть равно множеству `b`:

```
>>> s = set([1, 2, 3])
>>> s > set([1, 2]), s > set([1, 2, 3])
(True, False)
```

- `a.isdisjoint(b)` — проверяет, являются ли множества `a` и `b` полностью разными, т. е. не содержащими ни одного совпадающего элемента:

```
>>> s = set([1, 2, 3])
>>> s.isdisjoint(set([4, 5, 6]))
True
>>> s.isdisjoint(set([1, 3, 5]))
False
```

Для работы с множествами предназначены следующие методы:

- `copy()` — создает копию множества. Обратите внимание на то, что оператор `=` присваивает лишь ссылку на тот же объект, а не копирует его:

```
>>> s = set([1, 2, 3])
>>> c = s; s is c # С помощью = копию создать нельзя!
True
>>> c = s.copy()  # Создаем копию объекта
>>> c
set([1, 2, 3])
>>> s is c        # Теперь это разные объекты
False
```

- `add(<Элемент>)` — добавляет `<Элемент>` во множество:

```
>>> s = set([1, 2, 3])
>>> s.add(4); s
set([1, 2, 3, 4])
```

- `remove(<Элемент>)` — удаляет `<Элемент>` из множества. Если элемент не найден, то возбуждается исключение `KeyError`:

```
>>> s = set([1, 2, 3])
>>> s.remove(3); s          # Элемент существует
set([1, 2])
>>> s.remove(5)             # Элемент НЕ существует
Traceback (most recent call last):
  File "<pyshell#78>", line 1, in <module>
    s.remove(5)             # Элемент НЕ существует
KeyError: 5
```

- `discard(<Элемент>)` — удаляет `<Элемент>` из множества, если он присутствует. Если указанный элемент не существует, никакого исключения не возбуждается:

```
>>> s = set([1, 2, 3])
>>> s.discard(3); s         # Элемент существует
set([1, 2])
```

```
>>> s.discard(5); s       # Элемент НЕ существует
set([1, 2])
```

♦ `pop()` — удаляет произвольный элемент из множества и возвращает его. Если элементов нет, возбуждается исключение `KeyError`:

```
>>> s = set([1, 2])
>>> s.pop(), s
(1, set([2]))
>>> s.pop(), s
(2, set([]))
>>> s.pop() # Если нет элементов, то будет ошибка
Traceback (most recent call last):
  File "<pyshell#89>", line 1, in <module>
    s.pop() # Если нет элементов, то будет ошибка
KeyError: 'pop from an empty set'
```

♦ `clear()` — удаляет все элементы из множества:

```
>>> s = set([1, 2, 3])
>>> s.clear(); s
set([])
```

Помимо генераторов списков и генераторов словарей, язык Python 3 поддерживает *генераторы множеств*. Их синтаксис похож на синтаксис генераторов списков, но выражение заключается в фигурные скобки, а не в квадратные. Так как результатом является множество, все повторяющиеся элементы будут удалены:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3]}
{1, 2, 3}
```

Генераторы множеств могут иметь сложную структуру. Например, состоять из нескольких вложенных циклов `for` и (или) содержать оператор ветвления `if` после цикла. Создадим из элементов исходного списка множество, содержащее только уникальные элементы с четными значениями:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3] if x % 2 == 0}
{2}
```

Язык Python поддерживает еще один тип множеств — `frozenset`. В отличие от типа `set`, множество типа `frozenset` нельзя изменить. Объявить такое множество можно с помощью функции `frozenset()`:

```
>>> f = frozenset()
>>> f
frozenset([])
```

Функция `frozenset()` позволяет также преобразовать элементы последовательности во множество:

```
>>> frozenset("string")                # Преобразуем строку
frozenset(['g', 'i', 'n', 's', 'r', 't'])
>>> frozenset([1, 2, 3, 4, 4])         # Преобразуем список
frozenset([1, 2, 3, 4])
>>> frozenset((1, 2, 3, 4, 4))         # Преобразуем кортеж
frozenset([1, 2, 3, 4])
```

Множества `frozenset` поддерживают операторы, которые не изменяют само множество, а также следующие методы: `copy()`, `difference()`, `intersection()`, `issubset()`, `issuperset()`, `symmetric_difference()` и `union()`.

8.16. Диапазоны

Диапазоны, как следует из самого их названия, — это последовательности целых чисел с заданными начальным и конечным значениями и шагом (промежутком между соседними числами). Как и списки, кортежи и множества, диапазоны представляют собой последовательности и, подобно кортежам, являются неизменяемыми.

Важнейшим преимуществом диапазонов перед другими видами последовательностей является их компактность — вне зависимости от количества входящих в него элементов-чисел, диапазон всегда занимает один и тот же объем оперативной памяти. Однако в диапазон могут входить лишь числа, последовательно стоящие друг за другом, — сформировать диапазон на основе произвольного набора чисел или данных другого типа, даже чисел с плавающей точкой, невозможно.

Диапазоны чаще всего используются для проверки вхождения значения в какой-либо интервал и для организации циклов.

Для создания диапазона применяется функция `range()`:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение — если он не указан, используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон. Если параметр `<Шаг>` не указан, используется значение `1`:

```
>>> r = range(1, 10)
>>> for i in r: print(i, end = " ")
1 2 3 4 5 6 7 8 9
>>> r = range(10, 110, 10)
>>> for i in r: print(i, end = " ")
10 20 30 40 50 60 70 80 90 100
>>> r = range(10, 1, -1)
>>> for i in r: print(i, end = " ")
10 9 8 7 6 5 4 3 2
```

Преобразовать диапазон в список, кортеж, обычное или неизменяемое множество можно с помощью функций `list()`, `tuple()`, `set()` или `frozenset()` соответственно:

```
>>> list(range(1, 10))          # Преобразуем в список
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(1, 10))         # Преобразуем в кортеж
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> set(range(1, 10))           # Преобразуем в множество
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Множества поддерживают доступ к элементу по индексу, получение среза (в результате возвращается также диапазон), проверку на вхождение и невхождение, функции `len()`, `min()`, `max()`, методы `index()` и `count()`:

```
>>> r = range(1, 10)
>>> r[2], r[-1]
(3, 9)
```

```
>>> r[2:4]
range(3, 5)
>>> 2 in r, 12 in r
(True, False)
>>> 3 not in r, 13 not in r
(False, True)
>>> len(r), min(r), max(r)
(9, 1, 9)
>>> r.index(4), r.count(4)
(3, 1)
```

Поддерживается ряд операторов, позволяющих сравнить два диапазона:

- `==` — возвращает `True`, если диапазоны равны, и `False` — в противном случае. Диапазоны считаются равными, если они содержат одинаковые последовательности чисел:

  ```
  >>> range(1, 10) == range(1, 10, 1)
  True
  >>> range(1, 10, 2) == range(1, 11, 2)
  True
  >>> range(1, 10, 2) == range(1, 12, 2)
  False
  ```

- `!=` — возвращает `True`, если диапазоны не равны, и `False` — в противном случае:

  ```
  >>> range(1, 10, 2) != range(1, 12, 2)
  True
  >>> range(1, 10) != range(1, 10, 1)
  False
  ```

Также диапазоны поддерживают атрибуты `start`, `stop` и `step`, возвращающие, соответственно, начальную, конечную границы диапазона и его шаг:

```
>>> r = range(1, 10)
>>> r.start, r.stop, r.step
(1, 10, 1)
```

8.17. Модуль *itertools*

Модуль `itertools` содержит функции, позволяющие генерировать различные последовательности на основе других последовательностей, производить фильтрацию элементов и др. Все функции возвращают объекты, поддерживающие итерации. Прежде чем использовать функции, необходимо подключить модуль с помощью инструкции:

```
import itertools
```

8.17.1. Генерирование неопределенного количества значений

Для генерации неопределенного количества значений предназначены следующие функции:

- `count([start=0][, step=1])` — создает бесконечно нарастающую последовательность значений. Начальное значение задается параметром `start`, а шаг — параметром `step`:

  ```
  >>> import itertools
  >>> for i in itertools.count():
  ```

```
        if i > 10: break
        print(i, end=" ")

0 1 2 3 4 5 6 7 8 9 10
>>> list(zip(itertools.count(), "абвгд"))
[(0, 'а'), (1, 'б'), (2, 'в'), (3, 'г'), (4, 'д')]
>>> list(zip(itertools.count(start=2, step=2), "абвгд"))
[(2, 'а'), (4, 'б'), (6, 'в'), (8, 'г'), (10, 'д')]
```

- `cycle(<Последовательность>)` — на каждой итерации возвращает очередной элемент последовательности. Когда будет достигнут конец последовательности, перебор начнется сначала, и так до бесконечности:

```
>>> n = 1
>>> for i in itertools.cycle("абв"):
        if n > 10: break
        print(i, end=" ")
        n += 1

а б в а б в а б в а
>>> list(zip(itertools.cycle([0, 1]), "абвгд"))
[(0, 'а'), (1, 'б'), (0, 'в'), (1, 'г'), (0, 'д')]
```

- `repeat(<Объект>[, <Количество повторов>])` — возвращает объект указанное количество раз. Если количество повторов не указано, объект возвращается бесконечно:

```
>>> list(itertools.repeat(1, 10))
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> list(zip(itertools.repeat(5), "абвгд"))
[(5, 'а'), (5, 'б'), (5, 'в'), (5, 'г'), (5, 'д')]
```

8.17.2. Генерирование комбинаций значений

Получить различные комбинации значений позволяют следующие функции:

- `combinations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. При этом элементы в кортеже гарантированно будут разными. Формат функции:

```
combinations(<Последовательность>, <Количество элементов>)
```

Примеры:

```
>>> import itertools
>>> list(itertools.combinations('абвг', 2))
[('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'в'), ('б', 'г'),
 ('в', 'г')]
>>> ["".join(i) for i in itertools.combinations('абвг', 2)]
['аб', 'ав', 'аг', 'бв', 'бг', 'вг']
>>> list(itertools.combinations('вгаб', 2))
[('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'а'), ('г', 'б'),
 ('а', 'б')]
>>> list(itertools.combinations('абвг', 3))
[('а', 'б', 'в'), ('а', 'б', 'г'), ('а', 'в', 'г'),
 ('б', 'в', 'г')]
```

- `combinations_with_replacement()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. При этом кортеж может содержать одинаковые элементы. Формат функции:

```
combinations_with_replacement(<Последовательность>,
                              <Количество элементов>)
```

Примеры:

```
>>> list(itertools.combinations_with_replacement('абвг', 2))
[('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'б'),
 ('б', 'в'), ('б', 'г'), ('в', 'в'), ('в', 'г'), ('г', 'г')]
>>> list(itertools.combinations_with_replacement('вгаб', 2))
[('в', 'в'), ('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'г'),
 ('г', 'а'), ('г', 'б'), ('а', 'а'), ('а', 'б'), ('б', 'б')]
```

- `permutations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. Если количество элементов не указано, то используется длина последовательности. Формат функции:

```
permutations(<Последовательность>[, <Количество элементов>])
```

Примеры:

```
>>> list(itertools.permutations('абвг', 2))
[('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'), ('б', 'в'),
 ('б', 'г'), ('в', 'а'), ('в', 'б'), ('в', 'г'), ('г', 'а'),
 ('г', 'б'), ('г', 'в')]
>>> ["".join(i) for i in itertools.permutations('абвг')]
['абвг', 'абгв', 'авбг', 'авгб', 'агбв', 'агвб', 'бавг',
 'багв', 'бваг', 'бвга', 'бгав', 'бгва', 'вабг', 'вагб',
 'вбаг', 'вбга', 'вгаб', 'вгба', 'габв', 'гавб', 'гбав',
 'гбва', 'гваб', 'гвба']
```

- `product()` — на каждой итерации возвращает кортеж, содержащий комбинацию из элементов одной или нескольких последовательностей. Формат функции:

```
product(<Последовательность1>[,..., <ПоследовательностьN>][,
        repeat=1])
```

Примеры:

```
>>> from itertools import product
>>> list(product('абвг', repeat=2))
[('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'),
 ('б', 'б'), ('б', 'в'), ('б', 'г'), ('в', 'а'), ('в', 'б'),
 ('в', 'в'), ('в', 'г'), ('г', 'а'), ('г', 'б'), ('г', 'в'),
 ('г', 'г')]
>>> ["".join(i) for i in product('аб', 'вг', repeat=1)]
['ав', 'аг', 'бв', 'бг']
>>> ["".join(i) for i in product('аб', 'вг', repeat=2)]
['авав', 'аваг', 'авбв', 'авбг', 'агав', 'агаг', 'агбв',
 'агбг', 'бвав', 'бваг', 'бвбв', 'бвбг', 'бгав', 'бгаг',
 'бгбв', 'бгбг']
```

8.17.3. Фильтрация элементов последовательности

Для фильтрации элементов последовательности предназначены следующие функции:

- `filterfalse(<Функция>, <Последовательность>)` — возвращает элементы последовательности (по одному на каждой итерации), для которых функция, указанная в первом параметре, вернет значение `False`:

```
>>> import itertools
>>> def func(x): return x > 3

>>> list(itertools.filterfalse(func, [4, 5, 6, 0, 7, 2, 3]))
[0, 2, 3]
>>> list(filter(func, [4, 5, 6, 0, 7, 2, 3]))
[4, 5, 6, 7]
```

  Если в первом параметре вместо названия функции указать значение `None`, то каждый элемент последовательности будет проверен на соответствие значению `False`. Если элемент в логическом контексте возвращает значение `True`, то он не войдет в возвращаемый результат:

```
>>> list(itertools.filterfalse(None, [0, 5, 6, 0, 7, 0, 3]))
[0, 0, 0]
>>> list(filter(None, [0, 5, 6, 0, 7, 0, 3]))
[5, 6, 7, 3]
```

- `dropwhile(<Функция>, <Последовательность>)` — возвращает элементы последовательности (по одному на каждой итерации), начиная с элемента, для которого функция, указанная в первом параметре, вернет значение `False`:

```
>>> def func(x): return x > 3

>>> list(itertools.dropwhile(func, [4, 5, 6, 0, 7, 2, 3]))
[0, 7, 2, 3]
>>> list(itertools.dropwhile(func, [4, 5, 6, 7, 8]))
[]
>>> list(itertools.dropwhile(func, [1, 2, 4, 5, 6, 7, 8]))
[1, 2, 4, 5, 6, 7, 8]
```

- `takewhile(<Функция>, <Последовательность>)` — возвращает элементы последовательности (по одному на каждой итерации), пока не встретится элемент, для которого функция, указанная в первом параметре, вернет значение `False`:

```
>>> def func(x): return x > 3

>>> list(itertools.takewhile(func, [4, 5, 6, 0, 7, 2, 3]))
[4, 5, 6]
>>> list(itertools.takewhile(func, [4, 5, 6, 7, 8]))
[4, 5, 6, 7, 8]
>>> list(itertools.takewhile(func, [1, 2, 4, 5, 6, 7, 8]))
[]
```

- `compress()` — производит фильтрацию последовательности, указанной в первом параметре. Элемент возвращается, только если соответствующий элемент (с таким же индексом) из второй последовательности трактуется как истина. Сравнение заканчивается, когда достигнут конец одной из последовательностей. Формат функции:

```
compress(<Фильтруемая последовательность>,
         <Последовательность логических значений>)
```

Примеры:

```
>>> list(itertools.compress('абвгде', [1, 0, 0, 0, 1, 1]))
['а', 'д', 'е']
>>> list(itertools.compress('абвгде', [True, False, True]))
['а', 'в']
```

8.17.4. Прочие функции

Помимо функций, которые мы рассмотрели в предыдущих подразделах, модуль `itertools` содержит несколько дополнительных функций:

- `islice()` — на каждой итерации возвращает очередной элемент последовательности. Поддерживаются форматы:

  ```
  islice(<Последовательность>, <Конечная граница>)
  ```

  и

  ```
  islice(<Последовательность>, <Начальная граница>, <Конечная граница>[, <Шаг>])
  ```

  Если `<Шаг>` не указан, будет использовано значение `1`.

  Примеры:

  ```
  >>> list(itertools.islice("абвгдезж", 3))
  ['а', 'б', 'в']
  >>> list(itertools.islice("абвгдезж", 3, 6))
  ['г', 'д', 'е']
  >>> list(itertools.islice("абвгдезж", 3, 6, 2))
  ['г', 'е']
  ```

- `starmap(<Функция>, <Последовательность>)` — формирует последовательность на основании значений, возвращенных указанной функцией. Исходная последовательность должна содержать в качестве элементов кортежи — именно над элементами этих кортежей функция и станет вычислять значения, которые войдут в генерируемую последовательность. Примеры суммирования значений:

  ```
  >>> import itertools
  >>> def func1(x, y): return x + y

  >>> list(itertools.starmap(func1, [(1, 2), (4, 5), (6, 7)]))
  [3, 9, 13]
  >>> def func2(x, y, z): return x + y + z

  >>> list(itertools.starmap(func2, [(1, 2, 3), (4, 5, 6)]))
  [6, 15]
  ```

- `zip_longest()` — на каждой итерации возвращает кортеж, содержащий элементы последовательностей, которые расположены на одинаковом смещении. Если последовательности имеют разное количество элементов, вместо отсутствующего элемента вставляется объект, указанный в параметре `fillvalue`. Формат функции:

  ```
  zip_longest(<Последовательность1>[, ..., <ПоследовательностьN>]
              [, fillvalue=None])
  ```

Примеры:

```
>>> list(itertools.zip_longest([1, 2, 3], [4, 5, 6]))
[(1, 4), (2, 5), (3, 6)]
>>> list(itertools.zip_longest([1, 2, 3], [4]))
[(1, 4), (2, None), (3, None)]
>>> list(itertools.zip_longest([1, 2, 3], [4], fillvalue=0))
[(1, 4), (2, 0), (3, 0)]
>>> list(zip([1, 2, 3], [4]))
[(1, 4)]
```

- `accumulate(<Последовательность>[, <Функция>])` — на каждой итерации возвращает результат, полученный выполнением определенного действия над текущим элементом и результатом, полученным на предыдущей итерации. Выполняемая операция задается параметром `<Функция>`, а если он не указан, выполняется операция сложения. Функция, выполняющая операцию, должна принимать два параметра и возвращать результат. На первой итерации всегда возвращается первый элемент переданной последовательности:

```
>>> # Выполняем сложение
>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6]))
[1, 3, 6, 10, 15, 21]
>>> # [1, 1+2, 3+3, 6+4, 10+5, 15+6]
>>> # Выполняем умножение
>>> def func(x, y): return x * y

>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6], func))
[1, 2, 6, 24, 120, 720]
>>> # [1, 1*2, 2*3, 6*4, 24*5, 120*6]
```

- `chain()` — на каждой итерации возвращает элементы сначала из первой последовательности, затем из второй и т. д. Формат функции:

```
chain(<Последовательность1>[, ..., <ПоследовательностьN>])
```

Примеры:

```
>>> arr1, arr2, arr3 = [1, 2, 3], [4, 5], [6, 7, 8, 9]
>>> list(itertools.chain(arr1, arr2, arr3))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(itertools.chain("abc", "defg", "hij"))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> list(itertools.chain("abc", ["defg", "hij"]))
['a', 'b', 'c', 'defg', 'hij']
```

- `chain.from_iterable(<Последовательность>)` — аналогична функции `chain()`, но принимает одну последовательность, каждый элемент которой считается отдельной последовательностью:

```
>>> list(itertools.chain.from_iterable(["abc", "defg", "hij"]))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

- `tee(<Последовательность>[, <Количество>])` — возвращает кортеж, содержащий несколько одинаковых итераторов для последовательности. Если второй параметр не указан, то возвращается кортеж из двух итераторов:

```
>>> arr = [1, 2, 3]
>>> itertools.tee(arr)
(<itertools.tee object at 0x00FD8760>,
<itertools.tee object at 0x00FD8738>)
>>> itertools.tee(arr, 3)
(<itertools.tee object at 0x00FD8710>,
<itertools.tee object at 0x00FD87D8>,
<itertools.tee object at 0x00FD87B0>)
>>> list(itertools.tee(arr)[0])
[1, 2, 3]
>>> list(itertools.tee(arr)[1])
[1, 2, 3]
```

ГЛАВА 9

Словари

Словари — это наборы объектов, доступ к которым осуществляется не по индексу, а по ключу. В качестве ключа можно указать неизменяемый объект, например: число, строку или кортеж. Элементы словаря могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Следует также заметить, что элементы в словарях располагаются в произвольном порядке. Чтобы получить элемент, необходимо указать ключ, который использовался при сохранении значения.

Словари относятся к отображениям, а не к последовательностям. По этой причине функции, предназначенные для работы с последовательностями, а также операции извлечения среза, конкатенации, повторения и др., к словарям не применимы. Равно как и списки, словари относятся к изменяемым типам данных. Иными словами, мы можем не только получить значение по ключу, но и изменить его.

9.1. Создание словаря

Создать словарь можно следующими способами:

- с помощью функции `dict()`. Форматы функции:

```
dict(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
dict(<Словарь>)
dict(<Список кортежей с двумя элементами (Ключ, Значение)>)
dict(<Список списков с двумя элементами [Ключ, Значение]>)
```

  Если параметры не указаны, создается пустой словарь:

```
>>> d = dict(); d                    # Создаем пустой словарь
{}
>>> d = dict(a=1, b=2); d
{'a': 1, 'b': 2}
>>> d = dict({"a": 1, "b": 2}); d     # Словарь
{'a': 1, 'b': 2}
>>> d = dict([("a", 1), ("b", 2)]); d # Список кортежей
{'a': 1, 'b': 2}
>>> d = dict([["a", 1], ["b", 2]]); d # Список списков
{'a': 1, 'b': 2}
```

Объединить два списка в список кортежей позволяет функция `zip()`:

```
>>> k = ["a", "b"]          # Список с ключами
>>> v = [1, 2]              # Список со значениями
>>> list(zip(k, v))         # Создание списка кортежей
[('a', 1), ('b', 2)]
>>> d = dict(zip(k, v)); d # Создание словаря
{'a': 1, 'b': 2}
```

- указав все элементы словаря внутри фигурных скобок — это наиболее часто используемый способ создания словаря. Между ключом и значением ставится двоеточие, а пары «ключ/значение» записываются через запятую:

```
>>> d = {}; d               # Создание пустого словаря
{}
>>> d = { "a": 1, "b": 2 }; d
{'a': 1, 'b': 2}
```

- заполнив словарь поэлементно. В этом случае ключ указывается внутри квадратных скобок:

```
>>> d = {}             # Создаем пустой словарь
>>> d["a"] = 1         # Добавляем элемент1 (ключ "a")
>>> d["b"] = 2         # Добавляем элемент2 (ключ "b")
>>> d
{'a': 1, 'b': 2}
```

- с помощью метода `dict.fromkeys(<Последовательность>[, <Значение>])`. Метод создает новый словарь, ключами которого станут элементы последовательности, переданной первым параметром, а их значениями — величина, переданная вторым параметром. Если второй параметр не указан, элементы словаря получат в качестве значения `None`:

```
>>> d = dict.fromkeys(["a", "b", "c"])
>>> d
{'a': None, 'c': None, 'b': None}
>>> d = dict.fromkeys(["a", "b", "c"], 0) # Указан список
>>> d
{'a': 0, 'c': 0, 'b': 0}
>>> d = dict.fromkeys(("a", "b", "c"), 0) # Указан кортеж
>>> d
{'a': 0, 'c': 0, 'b': 0}
```

При создании словаря в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел и строк, но для списков и словарей этого делать нельзя. Рассмотрим пример:

```
>>> d1 = d2 = { "a": 1, "b": 2 } # Якобы создали два объекта
>>> d2["b"] = 10
>>> d1, d2                  # Изменилось значение в двух переменных !!!
({'a': 1, 'b': 10}, {'a': 1, 'b': 10})
```

Сразу видно, что изменение значения в переменной `d2` привело также к изменению значения в переменной `d1`, т. е. обе переменные ссылаются на один и тот же объект. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> d1, d2 = { "a": 1, "b": 2 }, { "a": 1, "b": 2 }
>>> d2["b"] = 10
>>> d1, d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Создать поверхностную копию словаря позволяет функция `dict()`:

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = dict(d1)           # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
False
>>> d2["b"] = 10
>>> d1, d2   # Изменилось только значение в переменной d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Также можно воспользоваться методом `copy()`:

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = d1.copy()          # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
False
>>> d2["b"] = 10
>>> d1, d2   # Изменилось только значение в переменной d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Чтобы создать полную копию словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy`:

```
>>> d1 = { "a": 1, "b": [20, 30, 40] }
>>> d2 = dict(d1)          # Создаем поверхностную копию
>>> d2["b"][0] = "test"
>>> d1, d2                 # Изменились значения в двух переменных!!!
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 30, 40]})
>>> import copy
>>> d3 = copy.deepcopy(d1) # Создаем полную копию
>>> d3["b"][1] = 800
>>> d1, d3                 # Изменилось значение только в переменной d3
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 800, 40]})
```

9.2. Операции над словарями

Обращение к элементам словаря осуществляется с помощью квадратных скобок, в которых указывается ключ. В качестве ключа можно указать неизменяемый объект, например: число, строку или кортеж.

Выведем все элементы словаря:

```
>>> d = { 1: "int", "a": "str", (1, 2): "tuple" }
>>> d[1], d["a"], d[(1, 2)]
('int', 'str', 'tuple')
```

Если элемент, соответствующий указанному ключу, отсутствует в словаре, возбуждается исключение `KeyError`:

```
>>> d = { "a": 1, "b": 2 }
>>> d["c"]              # Обращение к несуществующему элементу
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    d["c"]              # Обращение к несуществующему элементу
KeyError: 'c'
```

Проверить существование ключа в словаре можно с помощью оператора `in`. Если ключ найден, возвращается значение `True`, в противном случае — `False`:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d            # Ключ существует
True
>>> "c" in d            # Ключ не существует
False
```

Проверить, отсутствует ли какой-либо ключ в словаре, позволит оператор `not in`. Если ключ отсутствует, возвращается `True`, иначе — `False`:

```
>>> d = { "a": 1, "b": 2 }
>>> "c" not in d            # Ключ не существует
True
>>> "a" not in d            # Ключ существует
False
```

Метод `get(<Ключ>[, <Значение по умолчанию>])` позволяет избежать возбуждения исключения `KeyError` при отсутствии в словаре указанного ключа. Если ключ присутствует в словаре, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, возвращается `None` или значение, указанное во втором параметре:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```

Кроме того, можно воспользоваться методом `setdefault(<Ключ>[, <Значение по умолчанию>])`. Если ключ присутствует в словаре, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, в словаре создается новый элемент со значением, указанным во втором параметре. Если второй параметр не указан, значением нового элемента будет `None`:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"), d.setdefault("c"), d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'c': None, 'b': 2, 'd': 0}
```

Так как словари относятся к изменяемым типам данных, мы можем изменить элемент по ключу. Если элемент с указанным ключом отсутствует в словаре, он будет создан:

```
>>> d = { "a": 1, "b": 2 }
>>> d["a"] = 800                # Изменение элемента по ключу
>>> d["c"] = "string"           # Будет добавлен новый элемент
>>> d
{'a': 800, 'c': 'string', 'b': 2}
```

Получить количество ключей в словаре позволяет функция `len()`:

```
>>> d = { "a": 1, "b": 2 }
>>> len(d)          # Получаем количество ключей в словаре
2
```

Удалить элемент из словаря можно с помощью оператора `del`:

```
>>> d = { "a": 1, "b": 2 }
>>> del d["b"]; d   # Удаляем элемент с ключом "b" и выводим словарь
{'a': 1}
```

9.3. Перебор элементов словаря

Перебрать все элементы словаря можно с помощью цикла `for`, хотя словари и не являются последовательностями. Для примера выведем элементы словаря двумя способами. Первый способ использует метод `keys()`, возвращающий объект с ключами словаря. Во втором случае мы просто указываем словарь в качестве параметра. На каждой итерации цикла будет возвращаться ключ, с помощью которого внутри цикла можно получить значение, соответствующее этому ключу (листинг 9.1).

Листинг 9.1. Перебор элементов словаря

```
d = {"x": 1, "y": 2, "z": 3}
for key in d.keys():          # Использование метода keys()
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (y => 2) (x => 1) (z => 3)
print()                       # Вставляем символ перевода строки
for key in d:                 # Словари также поддерживают итерации
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (y => 2) (x => 1) (z => 3)
```

Поскольку словари являются неупорядоченными структурами, элементы словаря выводятся в произвольном порядке. Чтобы вывести элементы с сортировкой по ключам, следует получить список ключей, а затем воспользоваться методом `sort()` (листинг 9.2).

Листинг 9.2. Упорядоченный вывод элементов словаря с помощью метода `sort()`

```
d = {"x": 1, "y": 2, "z": 3}
k = list(d.keys())            # Получаем список ключей
k.sort()                      # Сортируем список ключей
for key in k:
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

Для сортировки ключей вместо метода `sort()` можно воспользоваться функцией `sorted()` (листинг 9.3).

Листинг 9.3. Упорядоченный вывод элементов словаря с помощью функции sorted()

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d.keys()):
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

Так как на каждой итерации возвращается ключ словаря, функции `sorted()` можно сразу передать объект словаря, а не результат выполнения метода `keys()` (листинг 9.4).

Листинг 9.4. Упорядоченный вывод элементов словаря с помощью функции sorted()

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d):
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

9.4. Методы для работы со словарями

Для работы со словарями предназначены следующие методы:

- `keys()` — возвращает объект `dict_keys`, содержащий все ключи словаря. Этот объект поддерживает итерации, а также операции над множествами:

```
>>> d1, d2 = { "a": 1, "b": 2 }, { "a": 3, "c": 4, "d": 5 }
>>> d1.keys(), d2.keys() # Получаем объект dict_keys
(dict_keys(['a', 'b']), dict_keys(['a', 'c', 'd']))
>>> list(d1.keys()), list(d2.keys()) # Получаем список ключей
(['a', 'b'], ['a', 'c', 'd'])
>>> for k in d1.keys(): print(k, end=" ")

a b
>>> d1.keys() | d2.keys()    # Объединение
{'a', 'c', 'b', 'd'}
>>> d1.keys() — d2.keys()    # Разница
{'b'}
>>> d2.keys() — d1.keys()    # Разница
{'c', 'd'}
>>> d1.keys() & d2.keys()    # Одинаковые ключи
{'a'}
>>> d1.keys() ^ d2.keys()    # Уникальные ключи
{'c', 'b', 'd'}
```

- `values()` — возвращает объект `dict_values`, содержащий все значения словаря. Этот объект поддерживает итерации:

```
>>> d = { "a": 1, "b": 2 }
>>> d.values()                  # Получаем объект dict_values
dict_values([1, 2])
>>> list(d.values())            # Получаем список значений
[1, 2]
```

```
>>> [ v for v in d.values() ]
[1, 2]
```

♦ `items()` — возвращает объект `dict_items`, содержащий все ключи и значения в виде кортежей. Этот объект поддерживает итерации:

```
>>> d = { "a": 1, "b": 2 }
>>> d.items()                    # Получаем объект dict_items
dict_items([('a', 1), ('b', 2)])
>>> list(d.items())              # Получаем список кортежей
[('a', 1), ('b', 2)]
```

♦ `<Ключ> in <Словарь>` — проверяет существование указанного ключа в словаре. Если ключ найден, возвращается значение `True`, в противном случае — `False`:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d                     # Ключ существует
True
>>> "c" in d                     # Ключ не существует
False
```

♦ `<Ключ> not in <Словарь>` — проверяет отсутствие указанного ключа в словаре. Если такового ключа нет, возвращается значение `True`, в противном случае — `False`:

```
>>> d = { "a": 1, "b": 2 }
>>> "c" not in d                     # Ключ не существует
True
>>> "a" not in d                     # Ключ существует
False
```

♦ `get(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует в словаре, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, возвращается `None` или значение, указанное во втором параметре:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```

♦ `setdefault(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует в словаре, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, в словаре создается новый элемент со значением, указанным во втором параметре. Если второй параметр не указан, значением нового элемента будет `None`:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"),d.setdefault("c"),d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'c': None, 'b': 2, 'd': 0}
```

♦ `pop(<Ключ>[, <Значение по умолчанию>])` — удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, возвращается значение из второго параметра. Если ключ отсутствует и второй параметр не указан, возбуждается исключение `KeyError`:

```
>>> d = { "a": 1, "b": 2, "c": 3 }
>>> d.pop("a"), d.pop("n", 0)
(1, 0)
```

```
>>> d.pop("n") # Ключ отсутствует и нет второго параметра
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    d.pop("n") # Ключ отсутствует и нет второго параметра
KeyError: 'n'
>>> d
{'c': 3, 'b': 2}
```

♦ `popitem()` — удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если словарь пустой, возбуждается исключение `KeyError`:

```
>>> d = { "a": 1, "b": 2 }
>>> d.popitem() # Удаляем произвольный элемент
('a', 1)
>>> d.popitem() # Удаляем произвольный элемент
('b', 2)
>>> d.popitem() # Словарь пустой. Возбуждается исключение
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    d.popitem() # Словарь пустой. Возбуждается исключение
KeyError: 'popitem(): dictionary is empty'
```

♦ `clear()` — удаляет все элементы словаря. Метод ничего не возвращает в качестве значения:

```
>>> d = { "a": 1, "b": 2 }
>>> d.clear()                  # Удаляем все элементы
>>> d                          # Словарь теперь пустой
{}
```

♦ `update()` — добавляет элементы в словарь. Метод изменяет текущий словарь и ничего не возвращает. Форматы метода:

```
update(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
update(<Словарь>)
update(<Список кортежей с двумя элементами>)
update(<Список списков с двумя элементами>)
```

Если элемент с указанным ключом уже присутствует в словаре, то его значение будет перезаписано.

Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> d.update(c=3, d=4)
>>> d
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
>>> d.update({"c": 10, "d": 20})        # Словарь
>>> d # Значения элементов перезаписаны
{'a': 1, 'c': 10, 'b': 2, 'd': 20}
>>> d.update([("d", 80), ("e", 6)])     # Список кортежей
>>> d
{'a': 1, 'c': 10, 'b': 2, 'e': 6, 'd': 80}
```

```
>>> d.update([["a", "str"], ["i", "t"]])  # Список списков
>>> d
{'a': 'str', 'c': 10, 'b': 2, 'e': 6, 'd': 80, 'i': 't'}
```

- `copy()` — создает поверхностную копию словаря:

```
>>> d1 = { "a": 1, "b": [10, 20] }
>>> d2 = d1.copy() # Создаем поверхностную копию
>>> d1 is d2       # Это разные объекты
False
>>> d2["a"] = 800  # Изменяем значение
>>> d1, d2         # Изменилось значение только в d2
({'a': 1, 'b': [10, 20]}, {'a': 800, 'b': [10, 20]})
>>> d2["b"][0] = "new" # Изменяем значение вложенного списка
>>> d1, d2         # Изменились значения и в d1, и в d2!!!
({'a': 1, 'b': ['new', 20]}, {'a': 800, 'b': ['new', 20]})
```

  Чтобы создать полную копию словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy`.

9.5. Генераторы словарей

Помимо генераторов списков, язык Python 3 поддерживает генераторы словарей. Синтаксис генераторов словарей похож на синтаксис генераторов списков, но имеет два различия:

- выражение заключается в фигурные скобки, а не в квадратные;
- внутри выражения перед циклом `for` указываются два значения через двоеточие, а не одно. Значение, расположенное слева от двоеточия, становится ключом, а значение, расположенное справа от двоеточия, — значением элемента.

Пример:

```
>>> keys = ["a", "b"]           # Список с ключами
>>> values = [1, 2]             # Список со значениями
>>> {k: v for (k, v) in zip(keys, values)}
{'a': 1, 'b': 2}
>>> {k: 0 for k in keys}
{'a': 0, 'b': 0}
```

Генераторы словарей могут иметь сложную структуру — например, состоять из нескольких вложенных циклов `for` и (или) содержать оператор ветвления `if` после цикла. Создадим из исходного словаря новый словарь, содержащий только элементы с четными значениями:

```
>>> d = { "a": 1, "b": 2, "c": 3, "d": 4 }
>>> {k: v for (k, v) in d.items() if v % 2 == 0}
{'b': 2, 'd': 4}
```

ГЛАВА 10

Работа с датой и временем

Для работы с датой и временем в языке Python предназначены следующие модули:

- `time` — позволяет получить текущие дату и время, а также произвести их форматированный вывод;
- `datetime` — позволяет манипулировать датой и временем. Например, производить арифметические операции, сравнивать даты, выводить дату и время в различных форматах и др.;
- `calendar` — позволяет вывести календарь в виде простого текста или в HTML-формате;
- `timeit` — позволяет измерить время выполнения небольших фрагментов кода с целью оптимизации программы.

10.1. Получение текущих даты и времени

Получить текущие дату и время позволяют следующие функции из модуля `time`:

- `time()` — возвращает вещественное число, представляющее количество секунд, прошедшее с начала эпохи (обычно с 1 января 1970 г.):

```
>>> import time              # Подключаем модуль time
>>> time.time()              # Получаем количество секунд
1511273856.8787858
```

- `gmtime([<Количество секунд>])` — возвращает объект `struct_time`, представляющий универсальное время (UTC). Если параметр не указан, возвращается текущее время. Если параметр указан, время будет не текущим, а соответствующим количеству секунд, прошедших с начала эпохи:

```
>>> time.gmtime(0)                    # Начало эпохи
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
>>> time.gmtime()                     # Текущая дата и время
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=14, tm_min=17,
tm_sec=55, tm_wday=1, tm_yday=325, tm_isdst=0)
>>> time.gmtime(1511273856.0)         # Дата 21-11-2017
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=14, tm_min=17,
tm_sec=36, tm_wday=1, tm_yday=325, tm_isdst=0)
```

Получить значение конкретного атрибута можно, указав его название или индекс внутри объекта:

```
>>> d = time.gmtime()
>>> d.tm_year, d[0]
(2017, 2017)
>>> tuple(d)              # Преобразование в кортеж
(2017, 11, 21, 14, 19, 34, 1, 325, 0)
```

- `localtime([<Количество секунд>])` — возвращает объект `struct_time`, представляющий локальное время. Если параметр не указан, возвращается текущее время. Если параметр указан, время будет не текущим, а соответствующим количеству секунд, прошедших с начала эпохи:

```
>>> time.localtime()               # Текущая дата и время
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=17, tm_min=20,
tm_sec=4, tm_wday=1, tm_yday=325, tm_isdst=0)
>>> time.localtime(1511273856.0)  # Дата 21-11-2017
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=17, tm_min=17,
tm_sec=36, tm_wday=1, tm_yday=325, tm_isdst=0)
```

- `mktime(<Объект struct_time>)` — возвращает вещественное число, представляющее количество секунд, прошедших с начала эпохи. В качестве параметра указывается объект `struct_time` или кортеж из девяти элементов. Если указанная дата некорректна, возбуждается исключение `OverflowError`:

```
>>> d = time.localtime(1511273856.0)
>>> time.mktime(d)
1511273856.0
>>> tuple(time.localtime(1511273856.0))
(2017, 11, 21, 17, 17, 36, 1, 325, 0)
>>> time.mktime((2017, 11, 21, 17, 17, 36, 1, 325, 0))
1511273856.0
>>> time.mktime((1940, 0, 31, 5, 23, 43, 5, 31, 0))
... Фрагмент опущен ...
OverflowError: mktime argument out of range
```

Объект `struct_time`, возвращаемый функциями `gmtime()` и `localtime()`, содержит следующие атрибуты (указаны тройки вида «имя атрибута — индекс — описание»):

- `tm_year` — `0` — год;
- `tm_mon` — `1` — месяц (число от `1` до `12`);
- `tm_mday` — `2` — день месяца (число от `1` до `31`);
- `tm_hour` — `3` — час (число от `0` до `23`);
- `tm_min` — `4` — минуты (число от `0` до `59`);
- `tm_sec` — `5` — секунды (число от `0` до `59`, изредка до `61`);
- `tm_wday` — `6` — день недели (число от `0` для понедельника до `6` для воскресенья);
- `tm_yday` — `7` — количество дней, прошедшее с начала года (число от `1` до `366`);
- `tm_isdst` — `8` — флаг коррекции летнего времени (значения `0`, `1` или `-1`).

Выведем текущие дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 10.1).

Листинг 10.1. Вывод текущих даты и времени

```
# -*- coding: utf-8 -*-
import time # Подключаем модуль time
d = [ "понедельник", "вторник", "среда", "четверг",
      "пятница", "суббота", "воскресенье" ]
m = [ "", "января", "февраля", "марта", "апреля", "мая",
      "июня", "июля", "августа", "сентября", "октября",
      "ноября", "декабря" ]
t = time.localtime() # Получаем текущее время
print( "Сегодня:\n%s %s %s %s %02d:%02d:%02d\n%02d.%02d.%02d" %
     ( d[t[6]], t[2], m[t[1]], t[0], t[3], t[4], t[5],
       t[2], t[1], t[0] ) )
input()
```

Примерный результат выполнения:

```
Сегодня:
вторник 21 ноября 2017 17:20:04
21.11.2017
```

10.2. Форматирование даты и времени

Форматирование даты и времени выполняют следующие функции из модуля `time`:

- `strftime(<Строка формата>[, <Объект struct_time>])` — возвращает строковое представление даты в соответствии со строкой формата. Если второй параметр не указан, будут выведены текущие дата и время. Если во втором параметре указан объект `struct_time` или кортеж из девяти элементов, дата будет соответствовать указанному значению. Функция зависит от настройки локали:

```
>>> import time
>>> time.strftime("%d.%m.%Y") # Форматирование даты
'21.11.2017'
>>> time.strftime("%H:%M:%S") # Форматирование времени
'17:23:27'
>>> time.strftime("%d.%m.%Y", time.localtime(1321954972.0))
'22.11.2011'
```

- `strptime(<Строка с датой>[, <Строка формата>])` — разбирает строку, указанную в первом параметре, в соответствии со строкой формата. Возвращает объект `struct_time`. Если строка не соответствует формату, возбуждается исключение `ValueError`. Если строка формата не указана, используется строка `"%a %b %d %H:%M:%S %Y"`. Функция учитывает текущую локаль:

```
>>> time.strptime("Tue Nov 21 17:34:22 2017")
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=17, tm_min=34,
tm_sec=22, tm_wday=1, tm_yday=325, tm_isdst=-1)
>>> time.strptime("21.11.2017", "%d.%m.%Y")
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=1, tm_yday=325, tm_isdst=-1)
>>> time.strptime("21-11-2017", "%d.%m.%Y")
... Фрагмент опущен ...
ValueError: time data '21-11-2017' does not match format '%d.%m.%Y'
```

- asctime([<Объект struct_time>]) — возвращает строку формата "%a %b %d %H:%M:%S %Y". Если параметр не указан, будут выведены текущие дата и время. Если в параметре указан объект struct_time или кортеж из девяти элементов, то дата будет соответствовать указанному значению:

```
>>> time.asctime()                             # Текущая дата
'Tue Nov 21 17:34:45 2017'
>>> time.asctime(time.localtime(1321954972.0))  # Дата в прошлом
'Tue Nov 22 12:42:52 2011'
```

- ctime([<Количество секунд>]) — функция аналогична asctime(), но в качестве параметра принимает не объект struct_time, а количество секунд, прошедших с начала эпохи:

```
>>> time.ctime()                   # Текущая дата
'Tue Nov 21 17:35:37 2017'
>>> time.ctime(1321954972.0)       # Дата в прошлом
'Tue Nov 22 12:42:52 2011'
```

В параметре <Строка формата> в функциях strftime() и strptime() могут быть использованы следующие комбинации специальных символов:

- %y — год из двух цифр (от "00" до "99");
- %Y — год из четырех цифр (например, "2011");
- %m — номер месяца с предваряющим нулем (от "01" до "12");
- %b — аббревиатура месяца в зависимости от настроек локали (например, "янв" для января);
- %B — название месяца в зависимости от настроек локали (например, "Январь");
- %d — номер дня в месяце с предваряющим нулем (от "01" до "31");
- %j — день с начала года (от "001" до "366");
- %U — номер недели в году (от "00" до "53"). Неделя начинается с воскресенья. Все дни с начала года до первого воскресенья относятся к неделе с номером 0;
- %W — номер недели в году (от "00" до "53"). Неделя начинается с понедельника. Все дни с начала года до первого понедельника относятся к неделе с номером 0;
- %w — номер дня недели ("0" — для воскресенья, "6" — для субботы);
- %a — аббревиатура дня недели в зависимости от настроек локали (например, "Пн" для понедельника);
- %A — название дня недели в зависимости от настроек локали (например, "понедельник");
- %H — часы в 24-часовом формате (от "00" до "23");
- %I — часы в 12-часовом формате (от "01" до "12");
- %M — минуты (от "00" до "59");
- %S — секунды (от "00" до "59", изредка до "61");
- %p — эквивалент значений AM и PM в текущей локали;
- %c — представление даты и времени в текущей локали;
- %x — представление даты в текущей локали;
- %X — представление времени в текущей локали:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> print(time.strftime("%x"))   # Представление даты
21.11.2017
>>> print(time.strftime("%X"))   # Представление времени
17:37:00
>>> print(time.strftime("%c"))   # Дата и время
21.11.2017 17:37:14
```

- `%Z` — название часового пояса или пустая строка (например, "`Московское время`", "`UTC`");
- `%%` — символ "`%`".

В качестве примера выведем текущие дату и время с помощью функции `strftime()` (листинг 10.2).

Листинг 10.2. Форматирование даты и времени

```
# -*- coding: utf-8 -*-
import time
import locale
locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
s = "Сегодня:\n%A %d %b %Y %H:%M:%S\n%d.%m.%Y"
print(time.strftime(s))
input()
```

Примерный результат выполнения:

```
Сегодня:
вторник 21 ноя 2017 17:38:31
21.11.2017
```

10.3. «Засыпание» скрипта

Функция `sleep(<Время в секундах>)` из модуля `time` прерывает выполнение скрипта на указанное время, по истечении которого скрипт продолжит работу. В качестве параметра можно указать целое или вещественное число:

```
>>> import time              # Подключаем модуль time
>>> time.sleep(5)            # "Засыпаем" на 5 секунд
```

10.4. Модуль *datetime*: манипуляции датой и временем

Модуль `datetime` позволяет манипулировать датой и временем: выполнять арифметические операции, сравнивать даты, выводить дату и время в различных форматах и др. Прежде чем использовать классы из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import datetime
```

Модуль содержит пять классов:

- `timedelta` — дата в виде количества дней, секунд и микросекунд. Экземпляр этого класса можно складывать с экземплярами классов `date` и `datetime`. Кроме того, результат вычитания двух дат будет экземпляром класса `timedelta`;
- `date` — представление даты в виде объекта;
- `time` — представление времени в виде объекта;
- `datetime` — представление комбинации даты и времени в виде объекта;
- `tzinfo` — абстрактный класс, отвечающий за зону времени. За подробной информацией по этому классу обращайтесь к документации по модулю `datetime`.

10.4.1. Класс *timedelta*

Класс `timedelta` из модуля `datetime` позволяет выполнять операции над датами: складывать, вычитать, сравнивать и др. Конструктор класса имеет следующий формат:

```
timedelta([days][, seconds][, microseconds][, milliseconds][, minutes]
          [, hours][, weeks])
```

Все параметры не являются обязательными и по умолчанию имеют значение `0`. Первые три параметра считаются основными:

- `days` — дни (диапазон `-999999999 <= days <= 999999999`);
- `seconds` — секунды (диапазон `0 <= seconds < 3600*24`);
- `microseconds` — микросекунды (диапазон `0 <= microseconds < 1000000`).

Все остальные параметры автоматически преобразуются в следующие значения:

- `milliseconds` — миллисекунды (одна миллисекунда преобразуется в 1000 микросекунд):

  ```
  >>> import datetime
  >>> datetime.timedelta(milliseconds=1)
  datetime.timedelta(0, 0, 1000)
  ```

- `minutes` — минуты (одна минута преобразуется в 60 секунд):

  ```
  >>> datetime.timedelta(minutes=1)
  datetime.timedelta(0, 60)
  ```

- `hours` — часы (один час преобразуется в 3600 секунд):

  ```
  >>> datetime.timedelta(hours=1)
  datetime.timedelta(0, 3600)
  ```

- `weeks` — недели (одна неделя преобразуется в 7 дней):

  ```
  >>> datetime.timedelta(weeks=1)
  datetime.timedelta(7)
  ```

Значения можно указать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера укажем один час:

```
>>> datetime.timedelta(0, 0, 0, 0, 0, 1)
datetime.timedelta(0, 3600)
>>> datetime.timedelta(hours=1)
datetime.timedelta(0, 3600)
```

Получить результат можно с помощью следующих атрибутов:

- `days` — дни;
- `seconds` — секунды;
- `microseconds` — микросекунды.

Пример:

```
>>> d = datetime.timedelta(hours=1, days=2, milliseconds=1)
>>> d
datetime.timedelta(2, 3600, 1000)
>>> d.days, d.seconds, d.microseconds
(2, 3600, 1000)
>>> repr(d), str(d)
('datetime.timedelta(2, 3600, 1000)', '2 days, 1:00:00.001000')
```

Получить результат в секундах позволяет метод `total_seconds()`:

```
>>> d = datetime.timedelta(minutes=1)
>>> d.total_seconds()
60.0
```

Над экземплярами класса `timedelta` можно производить арифметические операции `+`, `-`, `/`, `//`, `%` и `*`, использовать унарные операторы `+` и `-`, а также получать абсолютное значение с помощью функции `abs()`:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d1 + d2, d2 — d1                              # Сложение и вычитание
(datetime.timedelta(9), datetime.timedelta(5))
>>> d2 / d1                                       # Деление
3.5
>>> d1 / 2, d2 / 2.5                              # Деление
(datetime.timedelta(1), datetime.timedelta(2, 69120))
>>> d2 // d1                                      # Деление
3
>>> d1 // 2, d2 // 2                              # Деление
(datetime.timedelta(1), datetime.timedelta(3, 43200))
>>> d2 % d1                                       # Остаток
datetime.timedelta(1)
>>> d1 * 2, d2 * 2                                # Умножение
(datetime.timedelta(4), datetime.timedelta(14))
>>> 2 * d1, 2 * d2                                # Умножение
(datetime.timedelta(4), datetime.timedelta(14))
>>> d3 = -d1
>>> d3, abs(d3)
(datetime.timedelta(-2), datetime.timedelta(2))
```

Кроме того, можно использовать операторы сравнения `==`, `!=`, `<`, `<=`, `>` и `>=`:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d3 = datetime.timedelta(weeks=1)
>>> d1 == d2, d2 == d3                    # Проверка на равенство
(False, True)
```

```
>>> d1 != d2, d2 != d3                  # Проверка на неравенство
(True, False)
>>> d1 < d2, d2 <= d3                   # Меньше, меньше или равно
(True, True)
>>> d1 > d2, d2 >= d3                   # Больше, больше или равно
(False, True)
```

Также можно получать строковое представление экземпляра класса `timedelta` с помощью функций `str()` и `repr()`:

```
>>> d = datetime.timedelta(hours = 25, minutes = 5, seconds = 27)
>>> str(d)
'1 day, 1:05:27'
>>> repr(d)
'datetime.timedelta(1, 3927)'
```

Еще поддерживаются следующие атрибуты класса:

- `min` — минимальное значение, которое может иметь экземпляр класса `timedelta`;
- `max` — максимальное значение, которое может иметь экземпляр класса `timedelta`;
- `resolution` — минимальное возможное различие между значениями `timedelta`.

Выведем значения этих атрибутов:

```
>>> datetime.timedelta.min
datetime.timedelta(-999999999)
>>> datetime.timedelta.max
datetime.timedelta(999999999, 86399, 999999)
>>> datetime.timedelta.resolution
datetime.timedelta(0, 0, 1)
```

10.4.2. Класс *date*

Класс `date` из модуля `datetime` позволяет выполнять операции над датами. Конструктор класса имеет следующий формат:

```
date(<Год>, <Месяц>, <День>)
```

Все параметры являются обязательными. В параметрах можно указать следующий диапазон значений:

- `<Год>` — в виде числа, расположенного в диапазоне между значениями, хранящимися в константах `MINYEAR` и `MAXYEAR` класса `datetime` (о нем речь пойдет позже). Выведем значения этих констант:

  ```
  >>> import datetime
  >>> datetime.MINYEAR, datetime.MAXYEAR
  (1, 9999)
  ```

- `<Месяц>` — от `1` до `12` включительно;
- `<День>` — от `1` до количества дней в месяце.

Если значения выходят за диапазон, возбуждается исключение `ValueError`:

```
>>> datetime.date(2017, 11, 21)
datetime.date(2017, 11, 21)
```

```
>>> datetime.date(2017, 13, 3)   # Неправильное значение для месяца
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.date(2017, 11, 21)
>>> repr(d), str(d)
('datetime.date(2017, 11, 21)', '2017-11-21')
```

Для создания экземпляра класса `date` также можно воспользоваться следующими методами этого класса:

- `today()` — возвращает текущую дату:

  ```
  >>> datetime.date.today() # Получаем текущую дату
  datetime.date(2017, 11, 21)
  ```

- `fromtimestamp(<Количество секунд>)` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи:

  ```
  >>> import datetime, time
  >>> datetime.date.fromtimestamp(time.time())  # Текущая дата
  datetime.date(2017, 11, 21)
  >>> datetime.date.fromtimestamp(1321954972.0) # Дата 22-11-2011
  datetime.date(2011, 11, 22)
  ```

- `fromordinal(<Количество дней с 1-го года>)` — возвращает дату, соответствующую количеству дней, прошедших с первого года. В качестве параметра указывается число от `1` до `datetime.date.max.toordinal()`:

  ```
  >>> datetime.date.max.toordinal()
  3652059
  >>> datetime.date.fromordinal(3652059)
  datetime.date(9999, 12, 31)
  >>> datetime.date.fromordinal(1)
  datetime.date(1, 1, 1)
  ```

Получить результат можно с помощью следующих атрибутов:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от `1` до `12`);
- `day` — день (число от `1` до количества дней в месяце).

Пример:

```
>>> d = datetime.date.today() # Текущая дата (21-11-2017)
>>> d.year, d.month, d.day
(2017, 11, 21)
```

Над экземплярами класса `date` можно производить следующие операции:

- `date2 = date1 + timedelta` — прибавляет к дате указанный период в днях. Значения атрибутов `timedelta.seconds` и `timedelta.microseconds` игнорируются;
- `date2 = date1 - timedelta` — вычитает из даты указанный период в днях. Значения атрибутов `timedelta.seconds` и `timedelta.microseconds` игнорируются;
- `timedelta = date1 - date2` — возвращает разницу между датами (период в днях). Атрибуты `timedelta.seconds` и `timedelta.microseconds` будут иметь значение `0`.

Можно также сравнивать две даты с помощью операторов сравнения:

```
>>> d1 = datetime.date(2017, 11, 21)
>>> d2 = datetime.date(2017, 1, 1)
>>> t = datetime.timedelta(days=10)
>>> d1 + t, d1 - t            # Прибавляем и вычитаем 10 дней
(datetime.date(2017, 12, 1), datetime.date(2017, 11, 11))
>>> d1 - d2                   # Разница между датами
datetime.timedelta(324)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
(False, True, False, True)
>>> d1 == d2, d1 != d2
(False, True)
```

Класс `date` поддерживает следующие методы:

- `replace([year][, month][, day])` — возвращает дату с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра:

```
>>> d = datetime.date(2017, 11, 21)
>>> d.replace(2016, 12) # Заменяем год и месяц
datetime.date(2016, 12, 21)
>>> d.replace(year=2017, month=1, day=31)
datetime.date(2017, 1, 31)
>>> d.replace(day=30)   # Заменяем только день
datetime.date(2017, 1, 30)
```

- `strftime(<Строка формата>)` — возвращает отформатированную строку. В строке формата можно задавать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`:

```
>>> d = datetime.date(2017, 11, 21)
>>> d.strftime("%d.%m.%Y")
'21.11.2017'
```

- `isoformat()` — возвращает дату в формате `ГГГГ-ММ-ДД`:

```
>>> d = datetime.date(2017, 11, 21)
>>> d.isoformat()
'2017-11-21'
```

- `ctime()` — возвращает строку формата `"%a %b %d %H:%M:%S %Y"`:

```
>>> d = datetime.date(2017, 11, 21)
>>> d.ctime()
'Tue Nov 21 00:00:00 2017'
```

- `timetuple()` — возвращает объект `struct_time` с датой и временем:

```
>>> d = datetime.date(2017, 11, 21)
>>> d.timetuple()
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=1, tm_yday=325, tm_isdst=-1)
```

- `toordinal()` — возвращает количество дней, прошедших с 1-го года:

```
>>> d = datetime.date(2017, 11, 21)
>>> d.toordinal()
736654
```

```
>>> datetime.date.fromordinal(736654)
datetime.date(2017, 11, 21)
```

- `weekday()` — возвращает порядковый номер дня в неделе (`0` — для понедельника, `6` — для воскресенья):

```
>>> d = datetime.date(2017, 11, 21)
>>> d.weekday() # 1 — это вторник
1
```

- `isoweekday()` — возвращает порядковый номер дня в неделе (`1` — для понедельника, `7` — для воскресенья):

```
>>> d = datetime.date(2017, 11, 21)
>>> d.isoweekday() # 2 — это вторник
2
```

- `isocalendar()` — возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе):

```
>>> d = datetime.date(2017, 11, 21)
>>> d.isocalendar()
(2017, 47, 2)
```

Наконец, имеется поддержка следующих атрибутов класса:

- `min` — минимально возможное значение даты;
- `max` — максимально возможное значение даты;
- `resolution` — минимальное возможное различие между значениями даты.

Выведем значения этих атрибутов:

```
>>> datetime.date.min
datetime.date(1, 1, 1)
>>> datetime.date.max
datetime.date(9999, 12, 31)
>>> datetime.date.resolution
datetime.timedelta(1)
```

10.4.3. Класс *time*

Класс `time` из модуля `datetime` позволяет выполнять операции над значениями времени. Конструктор класса имеет следующий формат:

```
time([hour][, minute][, second][, microsecond][, tzinfo], [fold])
```

Все параметры не являются обязательными. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В параметрах можно указать следующий диапазон значений:

- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`);

♦ `fold` — порядковый номер отметки времени. Значение `0` (используется по умолчанию) обозначает первую отметку, значение `1` — вторую. Введено в Python 3.6 для тех случаев, когда в данной временной зоне практикуется перевод часов с зимнего на летнее время и обратно, в результате чего часы могут дважды в сутки показывать одинаковое время.

Если значения выходят за диапазон, возбуждается исключение `ValueError`:

```
>>> import datetime
>>> datetime.time(23, 12, 38, 375000)
datetime.time(23, 12, 38, 375000)
>>> t = datetime.time(hour=23, second=38, minute=12)
>>> repr(t), str(t)
('datetime.time(23, 12, 38)', '23:12:38')
>>> datetime.time(25, 12, 38, 375000)
... Фрагмент опущен ...
ValueError: hour must be in 0..23
```

Получить результат можно с помощью следующих атрибутов:

♦ `hour` — часы (число от `0` до `23`);

♦ `minute` — минуты (число от `0` до `59`);

♦ `second` — секунды (число от `0` до `59`);

♦ `microsecond` — микросекунды (число от `0` до `999999`);

♦ `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`);

♦ `fold` — порядковый номер отметки времени (число `0` или `1`). Поддержка этого атрибута появилась в Python 3.6.

Пример:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.hour, t.minute, t.second, t.microsecond
(23, 12, 38, 375000)
```

Над экземплярами класса `time` нельзя выполнять арифметические операции. Можно только производить сравнения:

```
>>> t1 = datetime.time(23, 12, 38, 375000)
>>> t2 = datetime.time(12, 28, 17)
>>> t1 < t2, t1 > t2, t1 <= t2, t1 >= t2
(False, True, False, True)
>>> t1 == t2, t1 != t2
(False, True)
```

Класс `time` поддерживает следующие методы:

♦ `replace([hour][, minute][, second][, microsecond][, tzinfo])` — возвращает время с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.replace(10, 52)      # Заменяем часы и минуты
datetime.time(10, 52, 38, 375000)
>>> t.replace(second=21)   # Заменяем только секунды
datetime.time(23, 12, 21, 375000)
```

- isoformat() — возвращает время в формате ISO 8601:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.isoformat()
'23:12:38.375000'
```

- strftime(<Строка формата>) — возвращает отформатированную строку. В строке формата можно указывать комбинации специальных символов, которые используются в функции strftime() из модуля time:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.strftime("%H:%M:%S")
'23:12:38'
```

Дополнительно класс time поддерживает такие атрибуты:

- min — минимально возможное значение времени;
- max — максимально возможное значение времени;
- resolution — минимальное возможное различие между значениями времени.

Вот значения этих атрибутов:

```
>>> datetime.time.min
datetime.time(0, 0)
>>> datetime.time.max
datetime.time(23, 59, 59, 999999)
>>> datetime.time.resolution
datetime.timedelta(0, 0, 1)
```

> **Примечание**
>
> Класс time поддерживает также методы dst(), utcoffset() и tzname(). За подробной информацией по этим методам, а также по абстрактному классу tzinfo обращайтесь к документации по модулю datetime.

10.4.4. Класс *datetime*

Класс datetime из модуля datetime позволяет выполнять операции над комбинацией даты и времени. Конструктор класса имеет следующий формат:

```
datetime(<Год>, <Месяц>, <День>[, hour][, minute][, second]
        [, microsecond][, tzinfo][, fold])
```

Первые три параметра являются обязательными. Остальные значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В параметрах можно указать следующий диапазон значений:

- <Год> — в виде числа, расположенного в диапазоне между значениями, хранящимися в константах MINYEAR (1) и MAXYEAR (9999);
- <Месяц> — число от 1 до 12 включительно;
- <День> — число от 1 до количества дней в месяце;
- hour — часы (число от 0 до 23);
- minute — минуты (число от 0 до 59);
- second — секунды (число от 0 до 59);

- ♦ `microsecond` — микросекунды (число от `0` до `999999`);
- ♦ `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`);
- ♦ `fold` — порядковый номер отметки времени. Значение `0` (используется по умолчанию) обозначает первую отметку, значение `1` — вторую. Введено в Python 3.6 для тех случаев, когда в данной временной зоне практикуется перевод часов с зимнего на летнее время и обратно, в результате чего часы могут дважды в сутки показывать одинаковое время.

Если значения выходят за диапазон, возбуждается исключение `ValueError`:

```
>>> import datetime
>>> datetime.datetime(2017, 11, 21)
datetime.datetime(2017, 11, 21, 0, 0)
>>> datetime.datetime(2017, 11, 21, hour=17, minute=47)
datetime.datetime(2017, 11, 21, 17, 47)
>>> datetime.datetime(2017, 32, 20)
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.datetime(2017, 11, 21, 17, 47, 43)
>>> repr(d), str(d)
('datetime.datetime(2017, 11, 21, 17, 47, 43)', '2017-11-21 17:47:43')
```

Для создания экземпляра класса `datetime` также можно воспользоваться следующими методами:

- ♦ `today()` — возвращает текущие дату и время:

```
>>> datetime.datetime.today()
datetime.datetime(2017, 11, 21, 17, 48, 27, 932332)
```

- ♦ `now([<Зона>])` — возвращает текущие дату и время. Если параметр не задан, то метод аналогичен методу `today()`:

```
>>> datetime.datetime.now()
datetime.datetime(2017, 11, 21, 17, 48, 51, 703618)
```

- ♦ `utcnow()` — возвращает текущее универсальное время (UTC):

```
>>> datetime.datetime.utcnow()
datetime.datetime(2017, 11, 21, 14, 49, 4, 497928)
```

- ♦ `fromtimestamp(<Количество секунд>[, <Зона>])` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи:

```
>>> import datetime, time
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2017, 11, 21, 17, 49, 27, 394796)
>>> datetime.datetime.fromtimestamp(1511273856.0)
datetime.datetime(2017, 11, 21, 17, 17, 36)
```

- ♦ `utcfromtimestamp(<Количество секунд>)` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи, в универсальном времени (UTC):

```
>>> datetime.datetime.utcfromtimestamp(time.time())
datetime.datetime(2017, 11, 21, 14, 50, 10, 596706)
>>> datetime.datetime.utcfromtimestamp(1511273856.0)
datetime.datetime(2017, 11, 21, 14, 17, 36)
```

- `fromordinal(<Количество дней с 1-го года>)` — возвращает дату, соответствующую количеству дней, прошедших с 1-го года. В качестве параметра указывается число от `1` до `datetime.datetime.max.toordinal()`:

```
>>> datetime.datetime.max.toordinal()
3652059
>>> datetime.datetime.fromordinal(3652059)
datetime.datetime(9999, 12, 31, 0, 0)
>>> datetime.datetime.fromordinal(1)
datetime.datetime(1, 1, 1, 0, 0)
```

- `combine(<Экземпляр класса date>, <Экземпляр класса time>)` — возвращает экземпляр класса `datetime`, созданный на основе переданных ему экземпляров классов `date` и `time`:

```
>>> d = datetime.date(2017, 11, 21)  # Экземпляр класса date
>>> t = datetime.time(17, 51, 22)   # Экземпляр класса time
>>> datetime.datetime.combine(d, t)
datetime.datetime(2017, 11, 21, 17, 51, 22)
```

- `strptime(<Строка с датой>, <Строка формата>)` — разбирает строку, указанную в первом параметре, в соответствии со строкой формата, создает на основе полученных из разобранной строки данных экземпляр класса `datetime` и возвращает его. Если строка не соответствует формату, возбуждается исключение `ValueError`. Метод учитывает текущую локаль:

```
>>> datetime.datetime.strptime("21.11.2017", "%d.%m.%Y")
datetime.datetime(2017, 11, 21, 0, 0)
>>> datetime.datetime.strptime("21.11.2017", "%d-%m-%Y")
... Фрагмент опущен ...
ValueError: time data '21.11.2017'
does not match format '%d-%m-%Y'
```

Получить результат можно с помощью следующих атрибутов:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от `1` до `12`);
- `day` — день (число от `1` до количества дней в месяце);
- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).
- `fold` — порядковый номер отметки времени (число `0` или `1`). Поддержка этого атрибута появилась в Python 3.6.

Примеры:

```
>>> d = datetime.datetime(2017, 11, 21, 17, 53, 58)
>>> d.year, d.month, d.day
(2017, 11, 21)
>>> d.hour, d.minute, d.second, d.microsecond
(17, 53, 58, 0)
```

Над экземплярами класса `datetime` можно производить следующие операции:

- `datetime2 = datetime1 + timedelta` — прибавляет к дате указанный период;
- `datetime2 = datetime1 — timedelta` — вычитает из даты указанный период;
- `timedelta = datetime1 — datetime2` — возвращает разницу между датами.

Можно также сравнивать две даты с помощью операторов сравнения.

Примеры:

```
>>> d1 = datetime.datetime(2017, 11, 21, 17, 54, 8)
>>> d2 = datetime.datetime(2017, 11, 1, 12, 31, 4)
>>> t = datetime.timedelta(days=10, minutes=10)
>>> d1 + t                    # Прибавляем 10 дней и 10 минут
datetime.datetime(2017, 12, 1, 18, 4, 8)
>>> d1 - t                    # Вычитаем 10 дней и 10 минут
datetime.datetime(2017, 11, 11, 17, 44, 8)
>>> d1 - d2                   # Разница между датами
datetime.timedelta(20, 19384)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
(False, True, False, True)
>>> d1 == d2, d1 != d2
(False, True)
```

Класс `datetime` поддерживает следующие методы:

- `date()` — возвращает экземпляр класса `date`, хранящий дату:

  ```
  >>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
  >>> d.date()
  datetime.date(2017, 11, 21)
  ```

- `time()` — возвращает экземпляр класса `time`, хранящий время:

  ```
  >>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
  >>> d.time()
  datetime.time(17, 56, 41)
  ```

- `timetz()` — возвращает экземпляр класса `time`, хранящий время. Метод учитывает параметр `tzinfo`;
- `timestamp()` — возвращает вещественное число, представляющее количество секунд, прошедшее с начала эпохи (обычно с 1 января 1970 г.):

  ```
  >>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
  >>> d.timestamp()
  1511276201.0
  ```

- `replace([year][, month][, day][, hour][, minute][, second][, microsecond][, tzinfo])` — возвращает дату с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра:

  ```
  >>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
  >>> d.replace(2016, 12)
  datetime.datetime(2016, 12, 21, 17, 56, 41)
  >>> d.replace(hour=12, month=10)
  datetime.datetime(2016, 10, 21, 12, 56, 41)
  ```

♦ `timetuple()` — возвращает объект `struct_time` с датой и временем:

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.timetuple()
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=17, tm_min=56,
tm_sec=41, tm_wday=1, tm_yday=325, tm_isdst=-1)
```

♦ `utctimetuple()` — возвращает объект `struct_time` с датой в универсальном времени (UTC):

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.utctimetuple()
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=21, tm_hour=17, tm_min=56,
tm_sec=41, tm_wday=1, tm_yday=325, tm_isdst=0)
```

♦ `toordinal()` — возвращает количество дней, прошедшее с 1-го года:

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.toordinal()
736654
```

♦ `weekday()` — возвращает порядковый номер дня в неделе (`0` — для понедельника, `6` — для воскресенья):

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.weekday() # 1 — это вторник
1
```

♦ `isoweekday()` — возвращает порядковый номер дня в неделе (`1` — для понедельника, `7` — для воскресенья):

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.isoweekday() # 2 — это вторник
2
```

♦ `isocalendar()` — возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе):

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.isocalendar()
(2017, 47, 2)
```

♦ `isoformat([<Разделитель даты и времени>])` — возвращает дату в формате ISO 8601. Если разделитель не указан, используется буква `T`:

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.isoformat()          # Разделитель не указан
'2017-11-21T17:56:41'
>>> d.isoformat(" ")       # Пробел в качестве разделителя
'2017-11-21 17:56:41'
```

♦ `ctime()` — возвращает строку формата `"%a %b %d %H:%M:%S %Y"`:

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.ctime()
'Tue Nov 21 17:56:41 2017'
```

♦ `strftime(<Строка формата>)` — возвращает отформатированную строку. В строке формата можно указывать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`:

```
>>> d = datetime.datetime(2017, 11, 21, 17, 56, 41)
>>> d.strftime("%d.%m.%Y %H:%M:%S")
'21.11.2017 17:56:41'
```

Поддерживаются также следующие атрибуты класса:

- `min` — минимально возможные значения даты и времени;
- `max` — максимально возможные значения даты и времени;
- `resolution` — минимальное возможное различие между значениями даты и времени.

Значения этих атрибутов:

```
>>> datetime.datetime.min
datetime.datetime(1, 1, 1, 0, 0)
>>> datetime.datetime.max
datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)
>>> datetime.datetime.resolution
datetime.timedelta(0, 0, 1)
```

> ***Примечание***
>
> Класс `datetime` также поддерживает методы `astimezone()`, `dst()`, `utcoffset()` и `tzname()`. За подробной информацией по этим методам, а также по абстрактному классу `tzinfo`, обращайтесь к документации по модулю `datetime`.

10.5. Модуль *calendar*: вывод календаря

Модуль `calendar` формирует календарь в виде простого текста или HTML-кода. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import calendar
```

Модуль предоставляет следующие классы:

- `Calendar` — базовый класс, который наследуют все остальные классы. Формат конструктора:

  ```
  Calendar([<Первый день недели>])
  ```

  В качестве примера получим двумерный список всех дней в ноябре 2017 года, распределенных по дням недели:

  ```
  >>> import calendar
  >>> c = calendar.Calendar(0)
  >>> c.monthdayscalendar(2017, 11) # 11 — это ноябрь
  [[0, 0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11, 12], [13, 14, 15, 16, 17, 18, 19],
  [20, 21, 22, 23, 24, 25, 26], [27, 28, 29, 30, 0, 0, 0]]
  ```

- `TextCalendar` — позволяет вывести календарь в виде простого текста. Формат конструктора:

  ```
  TextCalendar([<Первый день недели>])
  ```

  Выведем календарь на весь 2017 год:

  ```
  >>> c = calendar.TextCalendar(0)
  >>> print(c.formatyear(2017)) # Текстовый календарь на 2017 год
  ```

В качестве результата мы получим большую строку, содержащую календарь в виде отформатированного текста;

- `LocaleTextCalendar` — позволяет вывести календарь в виде простого текста. Названия месяцев и дней недели выводятся в соответствии с указанной локалью. Формат конструктора:

  ```
  LocaleTextCalendar([<Первый день недели>[, <Название локали>]])
  ```

  Выведем календарь на весь 2017 год на русском языке:

  ```
  >>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
  >>> print(c.formatyear(2017))
  ```

- `HTMLCalendar` — позволяет вывести календарь в формате HTML. Формат конструктора:

  ```
  HTMLCalendar([<Первый день недели>])
  ```

  Выведем календарь на весь 2017 год:

  ```
  >>> c = calendar.HTMLCalendar(0)
  >>> print(c.formatyear(2017))
  ```

  Результатом будет большая строка с HTML-кодом календаря, отформатированного в виде таблицы;

- `LocaleHTMLCalendar` — позволяет вывести календарь в формате HTML. Названия месяцев и дней недели выводятся в соответствии с указанной локалью. Формат конструктора:

  ```
  LocaleHTMLCalendar([<Первый день недели>[, <Название локали>]])
  ```

  Выведем календарь на весь 2017 год на русском языке в виде отдельной веб-страницы:

  ```
  >>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
  >>> xhtml = c.formatyearpage(2017, encoding="windows-1251")
  >>> print(xhtml.decode("cp1251"))
  ```

В первом параметре всех конструкторов указывается число от `0` (для понедельника) до `6` (для воскресенья). Если параметр не указан, его значение принимается равным `0`. Вместо чисел можно использовать встроенные константы: `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`, поддерживаемые классом `calendar`. Изменить значение параметра позволяет метод `setfirstweekday(<Первый день недели>)`.

В качестве примера выведем текстовый календарь на январь 2017 года, где первым днем недели является воскресенье:

```
>>> c = calendar.TextCalendar()          # Первый день понедельник
>>> c.setfirstweekday(calendar.SUNDAY) # Первый день теперь воскресенье
>>> print(c.formatmonth(2017, 1))      # Текстовый календарь на январь 2017 г.
```

10.5.1. Методы классов *TextCalendar* и *LocaleTextCalendar*

Классы `TextCalendar` и `LocaleTextCalendar` поддерживают следующие методы:

- `formatmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — возвращает текстовый календарь на указанный месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов перевода строки между строками.

Выведем календарь на декабрь 2017 года:

```
>>> import calendar
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatmonth(2017, 12))
    Декабрь 2017
Пн Вт Ср Чт Пт Сб Вс
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

♦ `prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — аналогичен методу `formatmonth()`, но не возвращает календарь в виде строки, а сразу выводит его на экран.

  Распечатаем календарь на декабрь 2017 года, указав ширину поля с днем, равной 4 символам:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.prmonth(2017, 12, 4)
           Декабрь 2017
 Пн   Вт   Ср   Чт   Пт   Сб   Вс
                      1    2    3
  4    5    6    7    8    9   10
 11   12   13   14   15   16   17
 18   19   20   21   22   23   24
 25   26   27   28   29   30   31
```

♦ `formatyear(<Год>[, w=2][, l=1][, c=6][, m=3])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:

  - `w` — ширина поля с днем (по умолчанию `2`);
  - `l` — количество символов перевода строки между строками (по умолчанию `1`);
  - `c` — количество пробелов между месяцами (по умолчанию `6`);
  - `m` — количество месяцев на строке (по умолчанию `3`).

  Значения можно записывать через запятую в порядке следования параметров или присвоить значение названию параметра.

  В качестве примера сформируем календарь на 2018 год, при этом на одной строке выведем сразу четыре месяца и установим количество пробелов между месяцами:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2018, m=4, c=2))
```

♦ `pryear(<Год>[, w=2][, l=1][, c=6][, m=3])` — аналогичен методу `formatyear()`, но не возвращает календарь в виде строки, а сразу выводит его.

  В качестве примера распечатаем календарь на 2018 год по два месяца на строке, расстояние между месяцами установим равным 4-м символам, ширину поля с датой — равной 2-м символам, а строки разделим одним символом перевода строки:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.pryear(2018, 2, 1, 4, 2)
```

10.5.2. Методы классов *HTMLCalendar* и *LocaleHTMLCalendar*

Классы `HTMLCalendar` и `LocaleHTMLCalendar` поддерживают следующие методы:

- `formatmonth(<Год>, <Месяц>[, <True | False>])` — возвращает календарь на указанный месяц в году в виде HTML-кода. Если в третьем параметре указано значение `True` (значение по умолчанию), то в заголовке таблицы после названия месяца будет указан год. Календарь будет отформатирован в виде HTML-таблицы. Для каждой ячейки таблицы задается стилевой класс, с помощью которого можно управлять внешним видом календаря. Названия стилевых классов доступны через атрибут `cssclasses`, который содержит список названий для каждого дня недели:

```
>>> import calendar
>>> c = calendar.HTMLCalendar(0)
>>> print(c.cssclasses)
['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun']
```

  Выведем календарь на ноябрь 2017 года, для будних дней укажем класс `"workday"`, а для выходных дней — класс `"week-end"`:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> c.cssclasses = ["workday", "workday", "workday", "workday",
                    "workday", "week-end", "week-end"]
>>> print(c.formatmonth(2017, 11, False))
```

- `formatyear(<Год>[, <Количество месяцев на строке>])` — возвращает календарь на указанный год в виде HTML-кода. Календарь будет отформатирован с помощью нескольких HTML-таблиц.

  Для примера выведем календарь на 2017 год так, чтобы на одной строке выводились сразу четыре месяца:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2017, 4))
```

- `formatyearpage(<Год>[, width][, css][, encoding])` — возвращает календарь на указанный год в виде отдельной веб-страницы. Параметры имеют следующее предназначение:
  - `width` — количество месяцев на строке (по умолчанию `3`);
  - `css` — название файла с таблицей стилей (по умолчанию `"calendar.css"`);
  - `encoding` — кодировка файла. Название кодировки будет указано в параметре `encoding` XML-пролога, а также в теге `<meta>`.

  Значения можно указать через запятую в порядке следования параметров или присвоить значение названию параметра.

  Для примера выведем календарь на 2017 год так, чтобы на одной строке выводилось четыре месяца, дополнительно указав кодировку:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> xhtml = c.formatyearpage(2017, 4, encoding="windows-1251")
>>> type(xhtml) # Возвращаемая строка имеет тип данных bytes
<class 'bytes'>
>>> print(xhtml.decode("cp1251"))
```

10.5.3. Другие полезные функции

Модуль `calendar` предоставляет еще несколько функций, которые позволяют вывести текстовый календарь без создания экземпляра соответствующего класса и получить дополнительную информацию о дате:

- `setfirstweekday(<Первый день недели>)` — устанавливает первый день недели для календаря. В качестве параметра указывается число от `0` (для понедельника) до `6` (для воскресенья). Вместо чисел можно использовать встроенные константы: `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`. Получить текущее значение параметра можно с помощью функции `firstweekday()`.\

  Установим воскресенье первым днем недели:

```
>>> import calendar
>>> calendar.firstweekday()     # По умолчанию 0
0
>>> calendar.setfirstweekday(6) # Изменяем значение
>>> calendar.firstweekday()     # Проверяем установку
6
```

- `month(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — возвращает текстовый календарь на указанный месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов перевода строки между строками.

  Выведем календарь на ноябрь 2017 года:

```
>>> calendar.setfirstweekday(0)
>>> print(calendar.month(2017, 11)) # Ноябрь 2017 года
   November 2017
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

- `prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]])` — аналогична функции `month()`, но не возвращает календарь в виде строки, а сразу выводит его.

  Выведем календарь на ноябрь 2017 года:

```
>>> calendar.prmonth(2017, 11) # Ноябрь 2017 года
```

- `monthcalendar(<Год>, <Месяц>)` — возвращает двумерный список всех дней в указанном месяце, распределенных по дням недели. Дни, выходящие за пределы месяца, будут представлены нулями.

  Выведем массив для ноября 2017 года:

```
>>> calendar.monthcalendar(2017, 11)
[[0, 0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11, 12], [13, 14, 15, 16, 17, 18, 19],
[20, 21, 22, 23, 24, 25, 26], [27, 28, 29, 30, 0, 0, 0]]
```

- `monthrange(<Год>, <Месяц>)` — возвращает кортеж из двух элементов: номера дня недели, приходящегося на первое число указанного месяца, и количества дней в месяце:

```
>>> print(calendar.monthrange(2017, 11))
(2, 30)
>>> # Ноябрь 2017 года начинается со среды (2) и включает 30 дней
```

♦ `calendar(<Год>[, w][, l][, c][, m])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
   - `w` — ширина поля с днем (по умолчанию `2`);
   - `l` — количество символов перевода строки между строками (по умолчанию `1`);
   - `c` — количество пробелов между месяцами (по умолчанию `6`);
   - `m` — количество месяцев на строке (по умолчанию `3`).

   Значения можно указать через запятую в порядке следования параметров или присвоить значение названию параметра.

   Для примера выведем календарь на 2017 год так, чтобы на одной строке выводилось сразу четыре месяца, установив при этом количество пробелов между месяцами:

```
>>> print(calendar.calendar(2017, m=4, c=2))
```

♦ `prcal(<Год>[, w][, l][, c][, m])` — аналогична функции `calendar()`, но не возвращает календарь в виде строки, а сразу выводит его.

   Для примера выведем календарь на 2017 год по два месяца на строке, расстояние между месяцами установим равным 4-м символам, ширину поля с датой — равной 2-м символам, а строки разделим одним символом перевода строки:

```
>>> calendar.prcal(2017, 2, 1, 4, 2)
```

♦ `weekheader(<n>)` — возвращает строку, которая содержит аббревиатуры дней недели с учетом текущей локали, разделенные пробелами. Единственный параметр задает длину каждой аббревиатуры в символах:

```
>>> calendar.weekheader(4)
'Mon  Tue  Wed  Thu  Fri  Sat  Sun '
>>> calendar.weekheader(2)
'Mo Tu We Th Fr Sa Su'
>>> import locale                    # Задаем другую локаль
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> calendar.weekheader(2)
'Пн Вт Ср Чт Пт Сб Вс'
```

♦ `isleap(<Год>)` — возвращает значение `True`, если указанный год является високосным, в противном случае — `False`:

```
>>> calendar.isleap(2017), calendar.isleap(2016)
(False, True)
```

♦ `leapdays(<Год1>, <Год2>)` — возвращает количество високосных лет в диапазоне от `<Год1>` до `<Год2>` (`<Год2>` не учитывается):

```
>>> calendar.leapdays(2013, 2016) # 2016 не учитывается
0
>>> calendar.leapdays(2010, 2016) # 2012 — високосный год
1
>>> calendar.leapdays(2010, 2017) # 2012 и 2016 — високосные года
2
```

- `weekday(<Год>, <Месяц>, <День>)` — возвращает номер дня недели (`0` — для понедельника, `6` — для воскресенья):

```
>>> calendar.weekday(2017, 11, 22)
2
```

- `timegm(<Объект struct_time>)` — возвращает число, представляющее количество секунд, прошедших с начала эпохи. В качестве параметра указывается объект `struct_time` с датой и временем, возвращаемый функцией `gmtime()` из модуля `time`:

```
>>> import calendar, time
>>> d = time.gmtime(1511348777.0) # Дата 22-11-2017
>>> d
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=22, tm_hour=11, tm_min=6,
tm_sec=17, tm_wday=2, tm_yday=326, tm_isdst=0)
>>> tuple(d)
(2017, 11, 22, 11, 6, 17, 2, 326, 0)
>>> calendar.timegm(d)
1511348777
>>> calendar.timegm((2017, 11, 22, 11, 6, 17, 2, 326, 0))
1511348777
```

Модуль `calendar` также предоставляет несколько атрибутов:

- `day_name` — список полных названий дней недели в текущей локали:

```
>>> [i for i in calendar.day_name]
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> [i for i in calendar.day_name]
['понедельник', 'вторник', 'среда', 'четверг', 'пятница', 'суббота',
'воскресенье']
```

- `day_abbr` — список аббревиатур названий дней недели в текущей локали:

```
>>> [i for i in calendar.day_abbr]
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> [i for i in calendar.day_abbr]
['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Вс']
```

- `month_name` — список полных названий месяцев в текущей локали:

```
>>> [i for i in calendar.month_name]
['', 'January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> [i for i in calendar.month_name]
['', 'Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', 'Июль',
'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь']
```

♦ `month_abbr` — список аббревиатур названий месяцев в текущей локали:

```
>>> [i for i in calendar.month_abbr]
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> [i for i in calendar.month_abbr]
['', 'янв', 'фев', 'мар', 'апр', 'май', 'июн', 'июл', 'авг', 'сен',
'окт', 'ноя', 'дек']
```

10.6. Измерение времени выполнения фрагментов кода

Модуль `timeit` позволяет измерить время выполнения небольших фрагментов кода с целью оптимизации программы. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
from timeit import Timer
```

Измерения производятся с помощью класса `Timer`. Конструктор класса имеет следующий формат:

```
Timer([stmt='pass'][, setup='pass'][, timer=<timer function>])
```

В параметре `stmt` указывается код (в виде строки), время выполнения которого предполагается измерить. Параметр `setup` позволяет указать код, который будет выполнен перед измерением времени выполнения кода в параметре `stmt`. Например, в параметре `setup` можно подключить модуль.

Получить время выполнения можно с помощью метода `timeit([number=1000000])`. В параметре `number` указывается количество повторений.

Для примера просуммируем числа от `1` до `10000` тремя способами и выведем время выполнения каждого способа (листинг 10.3).

Листинг 10.3. Измерение времени выполнения

```
# -*- coding: utf-8 -*-
from timeit import Timer
code1 = """\
i, j = 1, 0
while i < 10001:
    j += i
    i += 1
"""
t1 = Timer(stmt=code1)
print("while:", t1.timeit(number=10000))
code2 = """\
j = 0
for i in range(1, 10001):
    j += i
```

```
"""
t2 = Timer(stmt=code2)
print("for:", t2.timeit(number=10000))
code3 = """\
j = sum(range(1, 10001))
"""
t3 = Timer(stmt=code3)
print("sum:", t3.timeit(number=10000))
input()
```

Примерный результат выполнения (зависит от мощности компьютера):

```
while: 10.487761735853875
for: 6.378136742560729
sum: 2.2042291718107094
```

Сразу видно, что цикл `for` работает почти в два раза быстрее цикла `while`, а функция `sum()` в данном случае вне конкуренции.

Метод `repeat([repeat=3][, number=1000000])` вызывает метод `timeit()` указанное в параметре `repeat` количество раз и возвращает список значений. Аргумент `number` передается в качестве параметра методу `timeit()`.

Для примера создадим список со строковыми представлениями чисел от `1` до `10000`: в первом случае для создания списка используем цикл `for` и метод `append()`, а во втором — генератор списков (листинг 10.4).

Листинг 10.4. Использование метода `repeat()`

```
# -*- coding: utf-8 -*-
from timeit import Timer
code1 = """\
arr1 = []
for i in range(1, 10001):
    arr1.append(str(i))
"""
t1 = Timer(stmt=code1)
print("append:", t1.repeat(repeat=3, number=2000))
code2 = """\
arr2 = [str(i) for i in range(1, 10001)]
"""
t2 = Timer(stmt=code2)
print("генератор:", t2.repeat(repeat=3, number=2000))
input()
```

Примерный результат выполнения:

```
append: [6.27173358307843, 6.222750011887982, 6.239843531272257]
генератор: [4.6601598507632325, 4.648098189899006, 4.618446638727157]
```

Как видно из результата, генераторы списков выполняются быстрее.

ГЛАВА 11

Пользовательские функции

Функция — это фрагмент кода, который можно вызвать из любого места программы. В предыдущих главах мы уже не один раз использовали встроенные функции языка Python — например, с помощью функции `len()` получали количество элементов последовательности. В этой главе мы рассмотрим создание пользовательских функций, которые позволят уменьшить избыточность программного кода и повысить его структурированность.

11.1. Определение функции и ее вызов

Функция создается (или, как говорят программисты, определяется) с помощью ключевого слова `def` в следующем формате:

```
def <Имя функции>([<Параметры>]):
    [""" Строка документирования """]
    <Тело функции>
    [return <Результат>]
```

Имя функции должно быть уникальным идентификатором, состоящим из латинских букв, цифр и знаков подчеркивания, причем имя функции не может начинаться с цифры. В качестве имени нельзя использовать ключевые слова, кроме того, следует избегать совпадений с названиями встроенных идентификаторов. Регистр символов в названии функции также имеет значение.

После имени функции в круглых скобках можно указать один или несколько параметров через запятую, а если функция не принимает параметры, указываются только круглые скобки. После круглых скобок ставится двоеточие.

Тело функции представляет собой составную конструкцию. Как и в любой составной конструкции, инструкции внутри функции выделяются одинаковым количеством пробелов слева. Концом функции считается инструкция, перед которой находится меньшее количество пробелов. Если тело функции не содержит инструкций, то внутри ее необходимо разместить оператор `pass`, который не выполняет никаких действий. Этот оператор удобно использовать на этапе отладки программы, когда мы определили функцию, а тело решили дописать позже. Вот пример функции, которая ничего не делает:

```
def func():
    pass
```

Необязательная инструкция `return` позволяет вернуть из функции какое-либо значение в качестве результата. После исполнения этой инструкции выполнение функции будет остановлено, и последующие инструкции никогда не будут выполнены:

```
>>> def func():
        print("Текст до инструкции return")
        return "Возвращаемое значение"
        print("Эта инструкция никогда не будет выполнена")

>>> print(func()) # Вызываем функцию
```

Результат выполнения:

```
Текст до инструкции return
Возвращаемое значение
```

Инструкции `return` может не быть вообще. В этом случае выполняются все инструкции внутри функции, и в качестве результата возвращается значение `None`.

Для примера создадим три функции (листинг 11.1).

Листинг 11.1. Определение функций

```
def print_ok():
    """ Пример функции без параметров """
    print("Сообщение при удачно выполненной операции")

def echo(m):
    """ Пример функции с параметром """
    print(m)

def summa(x, y):
    """ Пример функции с параметрами,
        возвращающей сумму двух переменных """
    return x + y
```

При вызове функции значения ее параметров указываются внутри круглых скобок через запятую. Если функция не принимает параметров, оставляются только круглые скобки. Необходимо также заметить, что количество параметров в определении функции должно совпадать с количеством параметров при вызове, иначе будет выведено сообщение об ошибке. Вызвать функции из листинга 11.1 можно способами, указанными в листинге 11.2.

Листинг 11.2. Вызов функций

```
print_ok()              # Вызываем функцию без параметров
echo("Сообщение")       # Функция выведет сообщение
x = summa(5, 2)         # Переменной x будет присвоено значение 7
a, b = 10, 50
y = summa(a, b)         # Переменной y будет присвоено значение 60
```

Как видно из последнего примера, имя переменной в вызове функции может не совпадать с именем соответствующего параметра в определении функции. Кроме того, *глобальные* переменные `x` и `y` не конфликтуют с одноименными переменными, созданными в определении функции, т. к. они расположены в разных областях видимости. Переменные, указанные в определении функции, являются локальными и доступны только внутри функции. Более подробно области видимости мы рассмотрим в *разд. 11.9*.

Оператор +, используемый в функции `summa()`, служит не только для сложения чисел, но и позволяет объединить последовательности. То есть функция `summa()` может использоваться не только для сложения чисел. В качестве примера выполним конкатенацию строк и объединение списков (листинг 11.3).

Листинг 11.3. Многоцелевая функция

```
def summa(x, y):
    return x + y

print(summa("str", "ing"))   # Выведет: string
print(summa([1, 2], [3, 4])) # Выведет: [1, 2, 3, 4]
```

Как вы уже знаете, все в языке Python представляет собой объекты: строки, списки и даже сами типы данных. Не являются исключением и функции. Инструкция `def` создает объект, имеющий тип `function`, и сохраняет ссылку на него в идентификаторе, указанном после инструкции `def`. Таким образом, мы можем сохранить ссылку на функцию в другой переменной — для этого название функции указывается без круглых скобок. Сохраним ссылку в переменной и вызовем функцию через нее (листинг 11.4).

Листинг 11.4. Сохранение ссылки на функцию в переменной

```
def summa(x, y):
    return x + y

f = summa                         # Сохраняем ссылку в переменной f
v = f(10, 20)                     # Вызываем функцию через переменную f
```

Можно также передать ссылку на функцию другой функции в качестве параметра (листинг 11.5). Функции, передаваемые по ссылке, обычно называются *функциями обратного вызова*.

Листинг 11.5. Функции обратного вызова

```
def summa(x, y):
    return x + y

def func(f, a, b):
    """ Через переменную f будет доступна ссылка на
        функцию summa() """
    return f(a, b) # Вызываем функцию summa()

# Передаем ссылку на функцию в качестве параметра
v = func(summa, 10, 20)
```

Объекты функций поддерживают множество атрибутов, обратиться к которым можно, указав атрибут после названия функции через точку. Например, через атрибут `__name__` можно получить имя функции в виде строки, через атрибут `__doc__` — строку документирования и т. д. Для примера выведем названия всех атрибутов функции с помощью встроенной функции `dir()`:

```
>>> def summa(x, y):
        """ Суммирование двух чисел """
        return x + y

>>> dir(summa)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
>>> summa.__name__
'summa'
>>> summa.__code__.co_varnames
('x', 'y')
>>> summa.__doc__
' Суммирование двух чисел '
```

11.2. Расположение определений функций

Все инструкции в программе выполняются последовательно. Это означает, что прежде чем использовать в программе идентификатор, его необходимо предварительно определить, присвоив ему значение. Поэтому определение функции должно быть расположено перед вызовом функции.

Правильно:

```
def summa(x, y):
    return x + y
v = summa(10, 20) # Вызываем после определения. Все нормально
```

Неправильно:

```
v = summa(10, 20) # Идентификатор еще не определен. Это ошибка!!!
def summa(x, y):
    return x + y
```

В последнем случае будет выведено сообщение об ошибке: `NameError: name 'summa' is not defined`. Чтобы избежать ошибки, определение функции размещают в самом начале программы после подключения модулей или в отдельном модуле (о них речь пойдет в *главе 12*).

С помощью оператора ветвления `if` можно изменить порядок выполнения программы — например, разместить внутри условия несколько определений функций с одинаковым названием, но разной реализацией (листинг 11.6).

Листинг 11.6. Определение функции в зависимости от условия

```
# -*- coding: utf-8 -*-
n = input("Введите 1 для вызова первой функции: ")
if n == "1":
    def echo():
        print("Вы ввели число 1")
```

```
else:
    def echo():
        print("Альтернативная функция")

echo() # Вызываем функцию
input()
```

При вводе числа 1 мы получим сообщение "Вы ввели число 1", в противном случае — "Альтернативная функция".

Помните, что инструкция def всего лишь присваивает ссылку на объект функции идентификатору, расположенному после ключевого слова def. Если определение одной функции встречается в программе несколько раз, будет использоваться функция, которая была определена последней:

```
>>> def echo():
        print("Вы ввели число 1")
>>> def echo():
        print("Альтернативная функция")
>>> echo() # Всегда выводит "Альтернативная функция"
```

11.3. Необязательные параметры и сопоставление по ключам

Чтобы сделать некоторые параметры функции необязательными, следует в определении функции присвоить этому параметру начальное значение. Переделаем функцию суммирования двух чисел и сделаем второй параметр необязательным (листинг 11.7).

Листинг 11.7. Необязательные параметры

```
def summa(x, y=2):          # y — необязательный параметр
    return x + y
a = summa(5)               # Переменной a будет присвоено значение 7
b = summa(10, 50)          # Переменной b будет присвоено значение 60
```

Таким образом, если второй параметр не задан, он получит значение 2. Обратите внимание на то, что необязательные параметры должны следовать после обязательных, иначе будет выведено сообщение об ошибке.

До сих пор мы использовали позиционную передачу параметров в функцию:

```
def summa(x, y):
    return x + y
print(summa(10, 20))        # Выведет: 30
```

Переменной x при сопоставлении будет присвоено значение 10, а переменной y — значение 20. Но язык Python позволяет также передать значения в функцию, используя сопоставление по ключам. Для этого при вызове функции параметрам присваиваются значения, причем последовательность указания параметров в этом случае может быть произвольной (листинг 11.8).

Листинг 11.8. Сопоставление по ключам

```
def summa(x, y):
    return x + y
print(summa(y=20, x=10))   # Сопоставление по ключам
```

Сопоставление по ключам очень удобно использовать, если функция имеет несколько необязательных параметров. В этом случае не нужно указывать все значения, а достаточно присвоить значение нужному параметру:

```
>>> def summa(a=2, b=3, c=4): # Все параметры являются необязательными
        return a + b + c
>>> print(summa(2, 3, 20))    # Позиционное присваивание
>>> print(summa(c=20))        # Сопоставление по ключам
```

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед этим кортежем или списком следует указать символ `*`. Пример передачи значений из кортежа и списка приведен в листинге 11.9.

Листинг 11.9. Пример передачи значений из кортежа и списка

```
def summa(a, b, c):
    return a + b + c
t1, arr = (1, 2, 3), [1, 2, 3]
print(summa(*t1))                  # Распаковываем кортеж
print(summa(*arr))                 # Распаковываем список
t2 = (2, 3)
print(summa(1, *t2))               # Можно комбинировать значения
```

Если значения параметров содержатся в словаре, то перед ним следует поставить две звездочки: `**` (листинг 11.10).

Листинг 11.10. Пример передачи значений из словаря

```
def summa(a, b, c):
    return a + b + c
d1 = {"a": 1, "b": 2, "c": 3}
print(summa(**d1))                 # Распаковываем словарь
t, d2 = (1, 2), {"c": 3}
print(summa(*t, **d2))             # Можно комбинировать значения
```

Объекты в функцию передаются по ссылке. Если объект относится к неизменяемому типу, то изменение значения внутри функции не затронет значение переменной вне функции:

```
>>> def func(a, b):
        a, b = 20, "str"
>>> x, s = 80, "test"
>>> func(x, s)              # Значения переменных x и s не изменяются
>>> print(x, s)             # Выведет: 80 test
```

В этом примере значения в переменных `x` и `s` не изменились. Однако, если объект относится к изменяемому типу, ситуация будет другой:

```
>>> def func(a, b):
        a[0], b["a"] = "str", 800
>>> x = [1, 2, 3]             # Список
>>> y = {"a": 1, "b": 2}      # Словарь
>>> func(x, y)                # Значения будут изменены!!!
>>> print(x, y)               # Выведет: ['str', 2, 3] {'a': 800, 'b': 2}
```

Как видно из примера, значения в переменных x и y изменились, поскольку список и словарь относятся к изменяемым типам. Чтобы избежать изменения значений, внутри функции следует создать копию объекта (листинг 11.11).

Листинг 11.11. Передача изменяемого объекта в функцию

```
def func(a, b):
    a = a[:]                    # Создаем поверхностную копию списка
    b = b.copy()                # Создаем поверхностную копию словаря
    a[0], b["a"] = "str", 800
x = [1, 2, 3]                   # Список
y = {"a": 1, "b": 2}            # Словарь
func(x, y)                      # Значения останутся прежними
print(x, y)                     # Выведет: [1, 2, 3] {'a': 1, 'b': 2}
```

Можно также передать копию объекта непосредственно в вызове функции:

```
func(x[:], y.copy())
```

Если указать объект, имеющий изменяемый тип, в качестве значения параметра по умолчанию, этот объект будет сохраняться между вызовами функции:

```
>>> def func(a=[]):
        a.append(2)
        return a
>>> print(func())               # Выведет: [2]
>>> print(func())               # Выведет: [2, 2]
>>> print(func())               # Выведет: [2, 2, 2]
```

Как видно из примера, значения накапливаются внутри списка. Обойти эту проблему можно, например, следующим образом:

```
>>> def func(a=None):
        # Создаем новый список, если значение равно None
        if a is None:
            a = []
        a.append(2)
        return a
>>> print(func())               # Выведет: [2]
>>> print(func([1]))            # Выведет: [1, 2]
>>> print(func())               # Выведет: [2]
```

11.4. Переменное число параметров в функции

Если перед параметром в определении функции указать символ *, то функции можно будет передать произвольное количество параметров. Все переданные параметры сохраняются

в кортеже. Для примера напишем функцию суммирования произвольного количества чисел (листинг 11.12).

Листинг 11.12. Передача функции произвольного количества параметров

```
def summa(*t):
    """ Функция принимает произвольное количество параметров """
    res = 0
    for i in t:      # Перебираем кортеж с переданными параметрами
        res += i
    return res
print(summa(10, 20))                     # Выведет: 30
print(summa(10, 20, 30, 40, 50, 60))   # Выведет: 210
```

Можно также вначале указать несколько обязательных параметров и параметров, имеющих значения по умолчанию (листинг 11.13).

Листинг 11.13. Функция с параметрами разных типов

```
def summa(x, y=5, *t): # Комбинация параметров
    res = x + y
    for i in t:        # Перебираем кортеж с переданными параметрами
        res += i
    return res
print(summa(10))                         # Выведет: 15
print(summa(10, 20, 30, 40, 50, 60))   # Выведет: 210
```

Если перед параметром в определении функции указать две звездочки: `**`, то все именованные параметры будут сохранены в словаре (листинг 11.14).

Листинг 11.14. Сохранение переданных данных в словаре

```
def func(**d):
    for i in d:     # Перебираем словарь с переданными параметрами
        print("{0} => {1}".format(i, d[i]), end=" ")
func(a=1, b=2, c=3) # Выведет: a => 1 c => 3 b => 2
```

При комбинировании параметров параметр с двумя звездочками записывается самым последним. Если в определении функции указывается комбинация параметров с одной звездочкой и двумя звездочками, то функция примет любые переданные ей параметры (листинг 11.15).

Листинг 11.15. Комбинирование параметров

```
def func(*t, **d):
    """ Функция примет любые параметры """
    for i in t:
        print(i, end=" ")
    for i in d:    # Перебираем словарь с переданными параметрами
        print("{0} => {1}".format(i, d[i]), end=" ")
```

```
func(35, 10, a=1, b=2, c=3) # Выведет: 35 10 a => 1 c => 3 b => 2
func(10)                    # Выведет: 10
func(a=1, b=2)              # Выведет: a => 1 b => 2
```

В определении функции можно указать, что некоторые параметры передаются только по именам. Такие параметры должны указываться после параметра с одной звездочкой, но перед параметром с двумя звездочками. Именованные параметры могут иметь значения по умолчанию:

```
>>> def func(*t, a, b=10, **d):
        print(t, a, b, d)
>>> func(35, 10, a=1, c=3) # Выведет: (35, 10) 1 10 {'c': 3}
>>> func(10, a=5)          # Выведет: (10,) 5 10 {}
>>> func(a=1, b=2)         # Выведет: () 1 2 {}
>>> func(1, 2, 3)          # Ошибка. Параметр a обязателен!
```

В этом примере переменная `t` примет любое количество значений, которые будут объединены в кортеж. Переменные `a` и `b` должны передаваться только по именам, причем переменной `a` при вызове функции обязательно нужно передать значение. Переменная `b` имеет значение по умолчанию, поэтому при вызове допускается не передавать ей значение, но если значение передается, оно должно быть указано после названия параметра и оператора `=`. Переменная `d` примет любое количество именованных параметров и сохранит их в словаре. Обратите внимание на то, что хотя переменные `a` и `b` являются именованными, они не попадут в этот словарь.

Параметра с двумя звездочками в определении функции может не быть, а вот параметр с одной звездочкой при указании параметров, передаваемых только по именам, должен присутствовать обязательно. Если функция не должна принимать переменного количества параметров, но необходимо использовать переменные, передаваемые только по именам, то можно указать только звездочку без переменной:

```
>>> def func(x=1, y=2, *, a, b=10):
        print(x, y, a, b)
>>> func(35, 10, a=1)      # Выведет: 35 10 1 10
>>> func(10, a=5)          # Выведет: 10 2 5 10
>>> func(a=1, b=2)         # Выведет: 1 2 1 2
>>> func(a=1, y=8, x=7)    # Выведет: 7 8 1 10
>>> func(1, 2, 3)          # Ошибка. Параметр a обязателен!
```

В этом примере значения переменным `x` и `y` можно передавать как по позициям, так и по именам. Поскольку переменные имеют значения по умолчанию, допускается вообще не передавать им значений. Переменные `a` и `b` расположены после параметра с одной звездочкой, поэтому передать значения при вызове можно только по именам. Так как переменная `b` имеет значение по умолчанию, допускается не передавать ей значение при вызове, а вот переменная `a` обязательно должна получить значение, причем только по имени.

11.5. Анонимные функции

Помимо обычных, язык Python позволяет использовать *анонимные функции*, которые также называются *лямбда-функциями*. Анонимная функция не имеет имени и описывается с помощью ключевого слова `lambda` в следующем формате:

```
lambda [<Параметр1>[, ..., <ПараметрN>]]: <Возвращаемое значение>
```

После ключевого слова `lambda` можно указать передаваемые параметры. В качестве параметра `<Возвращаемое значение>` указывается выражение, результат выполнения которого будет возвращен функцией.

В качестве значения анонимная функция возвращает ссылку на объект-функцию, которую можно сохранить в переменной или передать в качестве параметра другой функции. Вызвать анонимную функцию можно, как и обычную, с помощью круглых скобок, внутри которых расположены передаваемые параметры. Пример использования анонимных функций приведен в листинге 11.16.

Листинг 11.16. Пример использования анонимных функций

```
f1 = lambda: 10 + 20                    # Функция без параметров
f2 = lambda x, y: x + y                 # Функция с двумя параметрами
f3 = lambda x, y, z: x + y + z          # Функция с тремя параметрами
print(f1())                             # Выведет: 30
print(f2(5, 10))                        # Выведет: 15
print(f3(5, 10, 30))                    # Выведет: 45
```

Как и у обычных функций, некоторые параметры анонимных функций могут быть необязательными. Для этого параметрам в определении функции присваивается значение по умолчанию (листинг 11.17).

Листинг 11.17. Необязательные параметры в анонимных функциях

```
f = lambda x, y=2: x + y
print(f(5))                             # Выведет: 7
print(f(5, 6))                          # Выведет: 11
```

Чаще всего анонимную функцию не сохраняют в переменной, а сразу передают в качестве параметра в другую функцию. Например, метод списков `sort()` позволяет указать пользовательскую функцию в параметре `key`. Отсортируем список без учета регистра символов, указав в качестве параметра анонимную функцию (листинг 11.18).

Листинг 11.18. Сортировка без учета регистра символов

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=lambda s: s.lower())
for i in arr:
    print(i, end=" ")
# Результат выполнения: единица1 Единица2 Единый
```

11.6. Функции-генераторы

Функцией-генератором называется функция, которая при последовательных вызовах возвращает очередной элемент какой-либо последовательности. Приостановить выполнение функции и превратить функцию в генератор позволяет ключевое слово `yield`. Для примера напишем функцию, которая возводит элементы последовательности в указанную степень (листинг 11.19).

Листинг 11.19. Пример использования функций-генераторов

```
def func(x, y):
    for i in range(1, x+1):
        yield i ** y
for n in func(10, 2):
    print(n, end=" ")     # Выведет: 1 4 9 16 25 36 49 64 81 100
print()                   # Вставляем пустую строку
for n in func(10, 3):
    print(n, end=" ")     # Выведет: 1 8 27 64 125 216 343 512 729 1000
```

Функции-генераторы поддерживают метод `__next__()`, который позволяет получить следующее значение. Когда значения заканчиваются, метод возбуждает исключение `StopIteration`. Вызов метода `__next__()` в цикле `for` производится незаметно для нас. Для примера перепишем предыдущую программу, использовав метод `__next__()` вместо цикла `for` (листинг 11.20).

Листинг 11.20. Использование метода `__next__()`

```
def func(x, y):
    for i in range(1, x+1):
       yield i ** y

i = func(3, 3)
print(i.__next__())       # Выведет: 1 (1 ** 3)
print(i.__next__())       # Выведет: 8 (2 ** 3)
print(i.__next__())       # Выведет: 27 (3 ** 3)
print(i.__next__())       # Исключение StopIteration
```

Получается, что с помощью обычных функций мы можем вернуть все значения сразу в виде списка, а с помощью функций-генераторов — только одно значение за раз. Эта особенность очень полезна при обработке большого количества значений, т. к. не понадобится загружать весь список со значениями в память.

Существует возможность вызвать одну функцию-генератор из другой. Для этого применяется расширенный синтаксис ключевого слова `yield`:

```
yield from <Вызываемая функция-генератор>
```

Рассмотрим следующий пример. Пусть у нас есть список чисел, и нам требуется получить другой список, включающий числа в диапазоне от 1 до каждого из чисел в первом списке. Чтобы создать такой список, напишем код, показанный в листинге 11.21.

Листинг 11.21. Вызов одной функции-генератора из другой (простой случай)

```
def gen(l):
    for e in l:
        yield from range(1, e + 1)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Здесь мы в функции-генераторе `gen` перебираем переданный ей в качестве параметра список и для каждого его элемента вызываем другую функцию-генератор. В качестве последней выступает выражение, создающее диапазон от 1 до значения очередного элемента, увеличенного на единицу (чтобы это значение вошло в диапазон). В результате на выходе мы получим вполне ожидаемый результат:

```
1 2 3 4 5 1 2 3 4 5 6 7 8 9 10
```

Усложним задачу, включив в результирующий список числа, умноженные на 2. Код, выполняющий эту задачу, показан в листинге 11.22.

Листинг 11.22. Вызов одной функции-генератора из другой (сложный случай)

```
def gen2(n):
    for e in range(1, n + 1):
        yield e * 2

def gen(l):
    for e in l:
        yield from gen2(e)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Здесь мы вызываем из функции-генератора `gen` написанную нами самими функцию-генератор `gen2`. Последняя создает диапазон, перебирает все входящие в него числа и возвращает их умноженными на 2. Результат работы приведенного в листинге кода таков:

```
2 4 6 8 10 2 4 6 8 10 12 14 16 18 20
```

Что нам и требовалось получить.

11.7. Декораторы функций

Декораторы позволяют изменить поведение обычных функций — например, выполнить какие-либо действия перед выполнением функции. Рассмотрим это на примере (листинг 11.23).

Листинг 11.23. Декораторы функций

```
def deco(f):                          # Функция-декоратор
    print("Вызвана функция func()")
    return f                          # Возвращаем ссылку на функцию
@deco
def func(x):
    return "x = {0}".format(x)

print(func(10))
```

Результат выполнения этого примера:

```
Вызвана функция func()
x = 10
```

Здесь перед определением функции `func()` было указано имя функции `deco()` с предваряющим символом `@`:

```
@deco
```

Таким образом, функция `deco()` стала декоратором функции `func()`. В качестве параметра функция-декоратор принимает ссылку на функцию, поведение которой необходимо изменить, и должна возвращать ссылку на ту же функцию или какую-либо другую. Наш предыдущий пример эквивалентен коду, показанному в листинге 11.24.

Листинг 11.24. Эквивалент использования декоратора

```
def deco(f):
    print("Вызвана функция func()")
    return f
def func(x):
    return "x = {0}".format(x)
# Вызываем функцию func() через функцию deco()
print(deco(func)(10))
```

Перед определением функции можно указать сразу несколько функций-декораторов. Для примера обернем функцию `func()` в два декоратора: `deco1()` и `deco2()` (листинг 11.25).

Листинг 11.25. Указание нескольких декораторов

```
def deco1(f):
    print("Вызвана функция deco1()")
    return f
def deco2(f):
    print("Вызвана функция deco2()")
    return f
@deco1
@deco2
def func(x):
    return "x = {0}".format(x)
print(func(10))
```

Вот что мы увидим после выполнения примера:

```
Вызвана функция deco2()
Вызвана функция deco1()
x = 10
```

Использование двух декораторов эквивалентно следующему коду:

```
func = deco1(deco2(func))
```

Здесь сначала будет вызвана функция `deco2()`, а затем функция `deco1()`. Результат выполнения будет присвоен идентификатору `func`.

В качестве еще одного примера использования декораторов рассмотрим выполнение функции только при правильно введенном пароле (листинг 11.26).

Листинг 11.26. Ограничение доступа с помощью декоратора

```
passw = input("Введите пароль: ")

def test_passw(p):
    def deco(f):
        if p == "10":         # Сравниваем пароли
            return f
        else:
            return lambda: "Доступ закрыт"
    return deco               # Возвращаем функцию-декоратор

@test_passw(passw)
def func():
    return "Доступ открыт"
print(func())                 # Вызываем функцию
```

Здесь после символа `@` указана не ссылка на функцию, а выражение, которое возвращает декоратор. Иными словами, декоратором является не функция `test_passw()`, а результат ее выполнения (функция `deco()`). Если введенный пароль является правильным, то выполнится функция `func()`, в противном случае будет выведена надпись `"Доступ закрыт"`, которую вернет анонимная функция.

11.8. Рекурсия. Вычисление факториала

Рекурсия — это возможность функции вызывать саму себя. Рекурсию удобно использовать для перебора объекта, имеющего заранее неизвестную структуру, или для выполнения неопределенного количества операций. В качестве примера рассмотрим вычисление факториала (листинг 11.27).

Листинг 11.27. Вычисление факториала

```
def factorial(n):
    if n == 0 or n == 1: return 1
    else:
        return n * factorial(n - 1)

while True:
    x = input("Введите число: ")
    if x.isdigit():           # Если строка содержит только цифры
        x = int(x)            # Преобразуем строку в число
        break                 # Выходим из цикла
    else:
        print("Вы ввели не число!")
print("Факториал числа {0} = {1}".format(x, factorial(x)))
```

Впрочем, проще всего для вычисления факториала воспользоваться функцией `factorial()` из модуля `math`:

```
>>> import math
>>> math.factorial(5), math.factorial(6)
(120, 720)
```

Количество вызовов функции самой себя (*проходов рекурсии*) ограничено. Узнать его можно, вызвав функцию `getrecursionlimit()` из модуля `sys`:

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

При превышении допустимого количества проходов рекурсии будет возбуждено исключение:

- `RuntimeError` — в версиях Python, предшествующих 3.5;
- `RecursionError` — в Python 3.5 и последующих версиях.

11.9. Глобальные и локальные переменные

Глобальные переменные — это переменные, объявленные в программе вне функции. В Python глобальные переменные видны в любой части модуля, включая функции (листинг 11.28).

Листинг 11.28. Глобальные переменные

```
def func(glob2):
    print("Значение глобальной переменной glob1 =", glob1)
    glob2 += 10
    print("Значение локальной переменной glob2 =", glob2)

glob1, glob2 = 10, 5
func(77) # Вызываем функцию
print("Значение глобальной переменной glob2 =", glob2)
```

Результат выполнения:

```
Значение глобальной переменной glob1 = 10
Значение локальной переменной glob2 = 87
Значение глобальной переменной glob2 = 5
```

Переменной `glob2` внутри функции присваивается значение, переданное при вызове функции. В результате создается новая переменная с тем же именем, но являющаяся локальной. Все изменения этой переменной внутри функции не затронут значение одноименной глобальной переменной.

Локальные переменные — это переменные, объявляемые внутри функций. Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной переменной не изменяется. Локальные переменные видны только внутри тела функции (листинг 11.29).

Листинг 11.29. Локальные переменные

```
def func():
    local1 = 77                       # Локальная переменная
    glob1 = 25                        # Локальная переменная
    print("Значение glob1 внутри функции =", glob1)
glob1 = 10                            # Глобальная переменная
func()                                # Вызываем функцию
print("Значение glob1 вне функции =", glob1)
try:
    print(local1)                     # Вызовет исключение NameError
except NameError:                     # Обрабатываем исключение
    print("Переменная local1 не видна вне функции")
```

Результат выполнения:

```
Значение glob1 внутри функции = 25
Значение glob1 вне функции = 10
Переменная local1 не видна вне функции
```

Как видно из примера, переменная `local1`, объявленная внутри функции `func()`, недоступна вне функции. Объявление внутри функции локальной переменной `glob1` не изменило значения одноименной глобальной переменной.

Если обращение к переменной производится до присваивания ей значения (даже если существует одноименная глобальная переменная), будет возбуждено исключение `UnboundLocalError` (листинг 11.30).

Листинг 11.30. Ошибка при обращении к переменной до присваивания значения

```
def func():
    # Локальная переменная еще не определена
    print(glob1)                      # Эта строка вызовет ошибку!!!
    glob1 = 25                        # Локальная переменная
glob1 = 10                            # Глобальная переменная
func()                                # Вызываем функцию
# Результат выполнения:
# UnboundLocalError: local variable 'glob1' referenced before assignment
```

Для того чтобы значение глобальной переменной можно было изменить внутри функции, необходимо объявить переменную глобальной с помощью ключевого слова `global`. Продемонстрируем это на примере (листинг 11.31).

Листинг 11.31. Использование ключевого слова `global`

```
def func():
    # Объявляем переменную glob1 глобальной
    global glob1
    glob1 = 25          # Изменяем значение глобальной переменной
    print("Значение glob1 внутри функции =", glob1)
```

```
glob1 = 10              # Глобальная переменная
print("Значение glob1 вне функции =", glob1)
func()                  # Вызываем функцию
print("Значение glob1 после функции =", glob1)
```

Результат выполнения:

```
Значение glob1 вне функции = 10
Значение glob1 внутри функции = 25
Значение glob1 после функции = 25
```

Таким образом, поиск идентификатора, используемого внутри функции, будет производиться в следующем порядке:

1. Поиск объявления идентификатора внутри функции (в локальной области видимости).
2. Поиск объявления идентификатора в глобальной области.
3. Поиск во встроенной области видимости (встроенные функции, классы и т. д.).

При использовании анонимных функций следует учитывать, что при указании внутри функции глобальной переменной будет сохранена ссылка на эту переменную, а не ее значение в момент определения функции:

```
>>> x = 5
>>> func = lambda: x          # Сохраняется ссылка, а не значение переменной x!!!
>>> x = 80                    # Изменили значение
>>> print(func())             # Выведет: 80, а не 5
```

Если необходимо сохранить именно текущее значение переменной, можно воспользоваться способом, приведенным в листинге 11.32.

Листинг 11.32. Сохранение значения переменной

```
x = 5
func = (lambda y: lambda: y)(x)       # Сохраняется значение переменной x
x = 80                                # Изменили значение
print(func())                         # Выведет: 5
```

Обратите внимание на вторую строку примера. В ней мы определили анонимную функцию с одним параметром, возвращающую ссылку на вложенную анонимную функцию. Далее мы вызываем первую функцию с помощью круглых скобок и передаем ей значение переменной `x`. В результате сохраняется текущее значение переменной, а не ссылка на нее.

Сохранить текущее значение переменной также можно, указав глобальную переменную в качестве значения параметра по умолчанию в определении функции (листинг 11.33).

Листинг 11.33. Сохранение значения с помощью параметра по умолчанию

```
x = 5
func = lambda x=x: x                   # Сохраняется значение переменной x
x = 80                                 # Изменили значение
print(func())                          # Выведет: 5
```

Получить все идентификаторы и их значения позволяют следующие функции:

- `globals()` — возвращает словарь с глобальными идентификаторами;
- `locals()` — возвращает словарь с локальными идентификаторами.

  Пример использования обеих этих функций показан в листинге 11.34.

Листинг 11.34. Использование функций `globals()` и `locals()`

```
def func():
    local1 = 54
    glob2 = 25
    print("Глобальные идентификаторы внутри функции")
    print(sorted(globals().keys()))
    print("Локальные идентификаторы внутри функции")
    print(sorted(locals().keys()))
glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(globals().keys()))
```

Результат выполнения:

```
Глобальные идентификаторы внутри функции
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'func', 'glob1', 'glob2']
Локальные идентификаторы внутри функции
['glob2', 'local1']
Глобальные идентификаторы вне функции
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'func', 'glob1', 'glob2']
```

- `vars([<Объект>])` — если вызывается без параметра внутри функции, возвращает словарь с локальными идентификаторами. Если вызывается без параметра вне функции, возвращает словарь с глобальными идентификаторами. При указании объекта в качестве параметра возвращает идентификаторы этого объекта (эквивалентно вызову `<Объект>.__dict__`). Пример использования этой функции можно увидеть в листинге 11.35.

Листинг 11.35. Использование функции `vars()`

```
def func():
    local1 = 54
    glob2 = 25
    print("Локальные идентификаторы внутри функции")
    print(sorted(vars().keys()))
glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(vars().keys()))
print("Указание объекта в качестве параметра")
print(sorted(vars(dict).keys()))
print("Альтернативный вызов")
print(sorted(dict.__dict__.keys()))
```

11.10. Вложенные функции

Одну функцию можно вложить в другую функцию, причем уровень вложенности не ограничен. При этом вложенная функция получает свою собственную локальную область видимости и имеет доступ к переменным, объявленным внутри функции, в которую она вложена (*функции-родителя*). Рассмотрим вложение функций на примере (листинг 11.36).

Листинг 11.36. Вложенные функции

```
def func1(x):
    def func2():
        print(x)
    return func2
f1 = func1(10)
f2 = func1(99)
f1()                    # Выведет: 10
f2()                    # Выведет: 99
```

Здесь мы определили функцию `func1()`, принимающую один параметр, а внутри нее — вложенную функцию `func2()`. Результатом выполнения функции `func1()` будет ссылка на эту вложенную функцию. Внутри функции `func2()` мы производим вывод значения переменной `x`, которая является локальной в функции `func1()`. Таким образом, помимо локальной, глобальной и встроенной областей видимости, добавляется *вложенная область видимости*. При этом поиск идентификаторов вначале производится внутри вложенной функции, затем внутри функции-родителя, далее в функциях более высокого уровня и лишь потом в глобальной и встроенной областях видимости. В нашем примере переменная `x` будет найдена в области видимости функции `func1()`.

Следует учитывать, что в момент определения функции сохраняются ссылки на переменные, а не их значения. Например, если после определения функции `func2()` произвести изменение переменной `x`, то будет использоваться это новое значение (листинг 11.37).

Листинг 11.37. При объявлении вложенной функции сохраняется ссылка на переменную

```
def func1(x):
    def func2():
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                    # Выведет: 30
f2()                    # Выведет: 30
```

Обратите внимание на результат выполнения. В обоих случаях мы получили значение `30`. Если необходимо сохранить именно значение переменной при определении вложенной функции, следует передать значение как значение по умолчанию (листинг 11.38).

Листинг 11.38. Принудительное сохранение значения переменной

```
def func1(x):
    def func2(x=x): # Сохраняем текущее значение, а не ссылку
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                     # Выведет: 10
f2()                     # Выведет: 99
```

Теперь попробуем из вложенной функции `func2()` изменить значение переменной `x`, объявленной внутри функции `func1()`. Если внутри функции `func2()` присвоить значение переменной `x`, будет создана новая локальная переменная с таким же именем. Если внутри функции `func2()` объявить переменную как глобальную и присвоить ей значение, то изменится значение глобальной переменной, а не значение переменной `x` внутри функции `func1()`. Таким образом, ни один из изученных ранее способов не позволяет из вложенной функции изменить значение переменной, объявленной внутри функции-родителя. Чтобы решить эту проблему, следует объявить необходимые переменные с помощью ключевого слова `nonlocal` (листинг 11.39).

Листинг 11.39. Ключевое слово `nonlocal`

```
def func1(a):
    x = a
    def func2(b):
        nonlocal x    # Объявляем переменную как nonlocal
        print(x)
        x = b         # Можем изменить значение x в func1()
    return func2

f = func1(10)
f(5)                      # Выведет: 10
f(12)                     # Выведет: 5
f(3)                      # Выведет: 12
```

При использовании ключевого слова `nonlocal` следует помнить, что переменная обязательно должна существовать внутри функции-родителя. В противном случае будет выведено сообщение об ошибке.

11.11. Аннотации функций

В Python функции могут содержать *аннотации*, которые вводят новый способ документирования. Теперь в заголовке функции допускается указывать предназначение каждого параметра, его тип данных, а также тип возвращаемого функцией значения. Аннотации имеют следующий формат:

```
def <Имя функции>(
    [<Параметр1>[: <Выражение>] [= <Значение по умолчанию>][, ...,
     <ПараметрN>[: <Выражение>] [= <Значение по умолчанию>]]]
    ) -> <Возвращаемое значение>:
    <Тело функции>
```

В параметрах `<Выражение>` и `<Возвращаемое значение>` можно указать любое допустимое выражение языка Python. Это выражение будет выполнено при создании функции.

Пример указания аннотаций:

```
>>> def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
        print(a, b)
```

В этом примере для переменной `a` создано описание `"Параметр1"`. Для переменной `b` выражение `10 + 5` является описанием, а число `3` — значением параметра по умолчанию. Кроме того, после закрывающей круглой скобки указан тип возвращаемого функцией значения: `None`. После создания функции все выражения будут выполнены, и результаты сохранятся в виде словаря в атрибуте `__annotations__` объекта функции.

Для примера выведем значение этого атрибута:

```
>>> def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
        pass

>>> func.__annotations__
{'a': 'Параметр1', 'b': 15, 'return': None}
```

ГЛАВА 12

Модули и пакеты

Модулем в языке Python называется любой файл с программным кодом. Каждый модуль может импортировать другой модуль, получая таким образом доступ к атрибутам (переменным, функциям и классам), объявленным внутри импортированного модуля. Следует заметить, что импортируемый модуль может содержать программу не только на Python — можно импортировать скомпилированный модуль, написанный на языке С.

Все программы, которые мы запускали ранее, были расположены в модуле с названием `"__main__"`. Получить имя модуля позволяет предопределенный атрибут `__name__`. Для запускаемого модуля он содержит значение `"__main__"`, а для импортируемого модуля — его имя. Выведем название модуля:

```
print(__name__)                    # Выведет: __main__
```

Проверить, является модуль главной программой или импортированным модулем, позволяет код, приведенный в листинге 12.1.

Листинг 12.1. Проверка способа запуска модуля

```
if __name__ == "__main__":
    print("Это главная программа")
else:
    print("Импортированный модуль")
```

12.1. Инструкция *import*

Импортировать модуль позволяет инструкция `import`. Мы уже не раз обращались к этой инструкции для подключения встроенных модулей. Например, подключали модуль `time` для получения текущей даты с помощью функции `strftime()`:

```
import time                          # Импортируем модуль time
print(time.strftime("%d.%m.%Y"))     # Выводим текущую дату
```

Инструкция `import` имеет следующий формат:

```
import <Название модуля 1> [as <Псевдоним 1>][, ...,
       <Название модуля N> [as <Псевдоним N>]]
```

После ключевого слова `import` указывается название модуля. Обратите внимание на то, что название не должно содержать расширения и пути к файлу. При именовании модулей необ-

ходимо учитывать, что операция импорта создает одноименный идентификатор, — это означает, что название модуля должно полностью соответствовать правилам именования переменных. Можно создать модуль с именем, начинающимся с цифры, но импортировать такой модуль нельзя. Кроме того, следует избегать совпадения имен модулей с ключевыми словами, встроенными идентификаторами и названиями модулей, входящих в стандартную библиотеку.

За один раз можно импортировать сразу несколько модулей, записав их через запятую. Для примера подключим модули `time` и `math` (листинг 12.2).

Листинг 12.2. Подключение нескольких модулей сразу

```
import time, math                    # Импортируем несколько модулей сразу
print(time.strftime("%d.%m.%Y")) # Текущая дата
print(math.pi)                       # Число pi
```

После импортирования модуля его название становится идентификатором, через который можно получить доступ к атрибутам, определенным внутри модуля. Доступ к атрибутам модуля осуществляется с помощью точечной нотации. Например, обратиться к константе `pi`, расположенной внутри модуля `math`, можно так:

```
math.pi
```

Функция `getattr()` позволяет получить значение атрибута модуля по его названию, заданному в виде строки. С помощью этой функции можно сформировать название атрибута динамически во время выполнения программы. Формат функции:

```
getattr(<Объект модуля>, <Атрибут>[, <Значение по умолчанию>])
```

Если указанный атрибут не найден, возбуждается исключение `AttributeError`. Чтобы избежать вывода сообщения об ошибке, в третьем параметре можно указать значение, которое будет возвращаться, если атрибут не существует. Пример использования функции приведен в листинге 12.3.

Листинг 12.3. Пример использования функции `getattr()`

```
import math
print(getattr(math, "pi"))       # Число pi
print(getattr(math, "x", 50))    # Число 50, т. к. x не существует
```

Проверить существование атрибута позволяет функция `hasattr(<Объект>, <Название атрибута>)`. Если атрибут существует, функция возвращает значение `True`. Напишем функцию проверки существования атрибута в модуле `math` (листинг 12.4).

Листинг 12.4. Проверка существования атрибута

```
import math
def hasattr_math(attr):
    if hasattr(math, attr):
        return "Атрибут существует"
    else:
        return "Атрибут не существует"
print(hasattr_math("pi"))       # Атрибут существует
print(hasattr_math("x"))        # Атрибут не существует
```

Если название модуля слишком длинное и его неудобно указывать каждый раз для доступа к атрибутам модуля, то можно создать *псевдоним*. Псевдоним задается после ключевого слова `as`. Создадим псевдоним для модуля `math` (листинг 12.5).

Листинг 12.5. Использование псевдонимов

```
import math as m                # Создание псевдонима
print(m.pi)                     # Число pi
```

Теперь доступ к атрибутам модуля `math` может осуществляться только с помощью идентификатора `m`. Идентификатор `math` в этом случае использовать уже нельзя.

Все содержимое импортированного модуля доступно только через название или псевдоним, указанный в инструкции `import`. Это означает, что любая глобальная переменная на самом деле является глобальной переменной модуля. По этой причине модули часто используются как пространства имен. Для примера создадим модуль под названием tests.py, в котором определим переменную `x` (листинг 12.6).

Листинг 12.6. Содержимое модуля tests.py

```
# -*- coding: utf-8 -*-
x = 50
```

В основной программе также определим переменную `x`, но с другим значением. Затем подключим файл tests.py и выведем значения переменных (листинг 12.7).

Листинг 12.7. Содержимое основной программы

```
# -*- coding: utf-8 -*-
import tests           # Подключаем файл tests.py
x = 22
print(tests.x)         # Значение переменной x внутри модуля
print(x)               # Значение переменной x в основной программе
input()
```

Оба файла размещаем в одной папке, а затем запускаем файл с основной программой с помощью двойного щелчка на значке файла. Как видно из результата, никакого конфликта имен нет, поскольку одноименные переменные расположены в разных пространствах имен.

Как говорилось еще в *главе 1*, перед собственно выполнением каждый модуль Python компилируется, преобразуясь в особое внутреннее представление (байт-код), — это делается для ускорения выполнения кода. Файлы с откомпилированным кодом хранятся в папке __pycache__, автоматически создающейся в папке, где находится сам файл с исходным, неоткомпилированным кодом модуля, и имеют имена вида <имя файла с исходным кодом>.cpython-<первые две цифры номера версии Python>.pyc. Так, при запуске на исполнение нашего файла tests.py откомпилированный код будет сохранен в файле tests.cpython-36.pyc.

Следует заметить, что для импортирования модуля достаточно иметь только файл с откомпилированным кодом, файл с исходным кодом в этом случае не нужен. Для примера переименуйте файл tests.py (например, в tests1.py), скопируйте файл tests.cpython-36.pyc из папки __pycache__ в папку с основной программой и переименуйте его в tests.pyc, а затем за-

пустите основную программу. Программа будет нормально выполняться. Таким образом, чтобы скрыть исходный код модулей, можно поставлять клиентам программу только с файлами, имеющими расширение pyc.

Существует еще одно обстоятельство, на которое следует обратить внимание. Импортирование модуля выполняется только при первом вызове инструкции `import` (или `from`, речь о которой пойдет позже). При каждом вызове инструкции `import` проверяется наличие объекта модуля в словаре `modules` из модуля `sys`. Если ссылка на модуль находится в этом словаре, то модуль повторно импортироваться не будет. Для примера выведем ключи словаря `modules`, предварительно отсортировав их (листинг 12.8).

Листинг 12.8. Вывод ключей словаря `modules`

```
# -*- coding: utf-8 -*-
import tests, sys                  # Подключаем модули tests и sys
print(sorted(sys.modules.keys()))
input()
```

Инструкция `import` требует явного указания объекта модуля. Так, нельзя передать название модуля в виде строки. Чтобы подключить модуль, название которого формируется программно, следует воспользоваться функцией `__import__()`. Для примера подключим модуль tests.py с помощью функции `__import__()` (листинг 12.9).

Листинг 12.9. Использование функции `__import__()`

```
# -*- coding: utf-8 -*-
s = "test" + "s"             # Динамическое создание названия модуля
m = __import__(s)            # Подключение модуля tests
print(m.x)                   # Вывод значения атрибута x
input()
```

Получить список всех идентификаторов внутри модуля позволяет функция `dir()`. Кроме того, можно воспользоваться словарем `__dict__`, который содержит все идентификаторы и их значения (листинг 12.10).

Листинг 12.10. Вывод списка всех идентификаторов

```
# -*- coding: utf-8 -*-
import tests
print(dir(tests))
print(sorted(tests.__dict__.keys()))
input()
```

12.2. Инструкция *from*

Для импортирования только определенных идентификаторов из модуля можно воспользоваться инструкцией `from`. Ее формат таков:

```
from <Название модуля> import <Идентификатор 1> [as <Псевдоним 1>]
                       [, ..., <Идентификатор N> [as <Псевдоним N>]]
```

```
from <Название модуля> import (<Идентификатор 1> [as <Псевдоним 1>],
                              [..., <Идентификатор N> [as <Псевдоним N>]])
from <Название модуля> import *
```

Первые два варианта позволяют импортировать модуль и сделать доступными только указанные идентификаторы. Для длинных имен можно назначить псевдонимы, указав их после ключевого слова `as`. В качестве примера сделаем доступными константу `pi` и функцию `floor()` из модуля `math`, а для названия функции создадим псевдоним (листинг 12.11).

Листинг 12.11. Инструкция `from`

```
# -*- coding: utf-8 -*-
from math import pi, floor as f
print(pi)                                    # Вывод числа pi
# Вызываем функцию floor() через идентификатор f
print(f(5.49))                               # Выведет: 5
input()
```

Идентификаторы можно разместить на нескольких строках, указав их названия через запятую внутри круглых скобок:

```
from math import (pi, floor,
                  sin, cos)
```

Третий вариант формата инструкции `from` позволяет импортировать из модуля все идентификаторы. Для примера импортируем все идентификаторы из модуля `math` (листинг 12.12).

Листинг 12.12. Импорт всех идентификаторов из модуля

```
# -*- coding: utf-8 -*-
from math import *    # Импортируем все идентификаторы из модуля math
print(pi)             # Вывод числа pi
print(floor(5.49))    # Вызываем функцию floor()
input()
```

Следует заметить, что идентификаторы, названия которых начинаются с символа подчеркивания, импортированы не будут. Кроме того, необходимо учитывать, что импортирование всех идентификаторов из модуля может нарушить пространство имен главной программы, т. к. идентификаторы, имеющие одинаковые имена, будут перезаписаны.

Создадим два модуля и подключим их с помощью инструкций `from` и `import`. Содержимое файла module1.py приведено в листинге 12.13.

Листинг 12.13. Содержимое файла module1.py

```
# -*- coding: utf-8 -*-
s = "Значение из модуля module1"
```

Содержимое файла module2.py приведено в листинге 12.14.

Листинг 12.14. Содержимое файла module2.py

```
# -*- coding: utf-8 -*-
s = "Значение из модуля module2"
```

Исходный код основной программы приведен в листинге 12.15.

Листинг 12.15. Код основной программы

```
# -*- coding: utf-8 -*-
from module1 import *
from module2 import *
import module1, module2
print(s)                 # Выведет: "Значение из модуля module2"
print(module1.s)         # Выведет: "Значение из модуля module1"
print(module2.s)         # Выведет: "Значение из модуля module2"
input()
```

Итак, в обоих модулях определена переменная с именем `s`. Размещаем все файлы в одной папке, а затем запускаем основную программу с помощью двойного щелчка на значке файла. При импортировании всех идентификаторов значением переменной `s` станет значение из модуля, который был импортирован последним, — в нашем случае это значение из модуля module2.py. Получить доступ к обеим переменным можно только при использовании инструкции `import`. Благодаря точечной нотации пространство имен не нарушается.

В атрибуте `__all__` можно указать список идентификаторов, которые будут импортироваться с помощью выражения `from module import *`. Идентификаторы внутри списка указываются в виде строки. Создадим файл module3.py (листинг 12.16).

Листинг 12.16. Использование атрибута `__all__`

```
# -*- coding: utf-8 -*-
x, y, z, _s = 10, 80, 22, "Строка"
__all__ = ["x", "_s"]
```

Затем напишем программу, которая будет его импортировать (листинг 12.17).

Листинг 12.17. Код основной программы

```
# -*- coding: utf-8 -*-
from module3 import *
print(sorted(vars().keys())) # Получаем список всех идентификаторов
input()
```

После запуска основной программы (с помощью двойного щелчка на значке файла) получим следующий результат:

```
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '_s', 'x']
```

Как видно из примера, были импортированы только переменные `_s` и `x`. Если бы мы не указали идентификаторы внутри списка `__all__`, результат был бы другим:

```
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'x', 'y', 'z']
```

Обратите внимание на то, что переменная `_s` в этом случае не импортируется, т. к. ее имя начинается с символа подчеркивания.

12.3. Пути поиска модулей

До сих пор мы размещали модули в одной папке с файлом основной программы. В этом случае нет необходимости настраивать пути поиска модулей, т. к. папка с исполняемым файлом автоматически добавляется в начало списка путей. Получить полный список путей поиска позволяет следующий код:

```
>>> import sys              # Подключаем модуль sys
>>> sys.path                # path содержит список путей поиска модулей
```

Список из переменной `sys.path` содержит пути поиска, получаемые из следующих источников:

- путь к папке с файлом основной программы;
- значение переменной окружения `PYTHONPATH`. Для добавления переменной в меню **Пуск** выбираем пункт **Панель управления** (или **Настройка | Панель управления**). В открывшемся окне выбираем пункт **Система** и щелкаем на ссылке **Дополнительные параметры системы**. Переходим на вкладку **Дополнительно** и нажимаем кнопку **Переменные среды**. В разделе **Переменные среды пользователя** нажимаем кнопку **Создать**. В поле **Имя переменной** вводим `PYTHONPATH`, а в поле **Значение переменной** задаем пути к папкам с модулями через точку с запятой — например, `C:\folder1;C:\folder2`. Закончив, не забудем нажать кнопки **OK** обоих открытых окон. После этого изменения перезагружать компьютер не нужно, достаточно заново запустить программу;
- пути поиска стандартных модулей;
- содержимое файлов с расширением pth, расположенных в каталогах поиска стандартных модулей, — например, в каталоге C:\Python36\Lib\site-packages. Названия таких файлов могут быть произвольными, главное, чтобы они имели расширение pth. Каждый путь (абсолютный или относительный) должен быть расположен на отдельной строке.

  Для примера создайте файл mypath.pth в каталоге C:\Python36\Lib\site-packages со следующим содержимым:

  ```
  # Это комментарий
  C:\folder1
  C:\folder2
  ```

> ***Примечание***
>
> Обратите внимание на то, что каталоги должны существовать, в противном случае они не будут добавлены в список `sys.path`.

При поиске модуля список `sys.path` просматривается от начала к концу. Поиск прекращается после первого найденного модуля. Таким образом, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, то будет использоваться модуль из папки C:\folder1, т. к. он расположен первым в списке путей поиска.

Список `sys.path` можно изменять программно с помощью соответствующих методов. Например, добавить каталог в конец списка можно с помощью метода `append()`, а в его начало — с помощью метода `insert()` (листинг 12.18).

Листинг 12.18. Изменение списка путей поиска модулей

```
# -*- coding: utf-8 -*-
import sys
sys.path.append(r"C:\folder1")          # Добавляем в конец списка
```

```
sys.path.insert(0, r"C:\folder2")       # Добавляем в начало списка
print(sys.path)
input()
```

В этом примере мы добавили папку C:\folder2 в начало списка. Теперь, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, будет использоваться модуль из папки C:\folder2, а не из папки C:\folder1, как в предыдущем примере.

Обратите внимание на символ `r` перед открывающей кавычкой. В этом режиме специальные последовательности символов не интерпретируются. Если используются обычные строки, то необходимо удвоить каждый слэш в пути:

```
sys.path.append("C:\\folder1\\folder2\\folder3")
```

В Python 3.6 появилась возможность указать полностью свои пути для поиска модулей, при этом список, хранящийся в переменной `sys.path`, будет проигнорирован. Для этого достаточно поместить в папку, где установлен Python, файл с именем python<первые две цифры номера версии Python>._pth (так, для Python 3.6 этот файл должен иметь имя python36._pth) или python._pth, в котором записать все нужные пути в том же формате, который используется при создании файлов pth. Первый файл будет использоваться программами, вызывающими библиотеку времени выполнения Python, в частности, **Python Shell**. А второй файл будет считан при запуске Python-программы щелчком мыши на ее файле.

> ***Внимание!***
>
> В файл *python<первые две цифры номера версии Python>._pth* обязательно следует включить пути для поиска модулей, составляющих стандартную библиотеку Python (их можно получить из списка, хранящегося в переменной `sys.path`). Если этого не сделать, утилита **Python Shell** вообще не запустится.

12.4. Повторная загрузка модулей

Как вы уже знаете, модуль загружается только один раз при первой операции импорта. Все последующие операции импортирования этого модуля будут возвращать уже загруженный объект модуля, даже если сам модуль был изменен. Чтобы повторно загрузить модуль, следует воспользоваться функцией `reload()` из модуля `imp`. Формат функции:

```
from imp import reload
reload(<Объект модуля>)
```

В качестве примера создадим модуль tests2.py, поместив его в папку C:\book (листинг 12.19).

Листинг 12.19. Содержимое файла tests2.py

```
# -*- coding: utf-8 -*-
x = 150
```

Подключим этот модуль в окне **Python Shell** редактора IDLE и выведем текущее значение переменной `x`:

```
>>> import sys
>>> sys.path.append(r"C:\book") # Добавляем путь к папке с модулем
>>> import tests2               # Подключаем модуль tests2.py
>>> print(tests2.x)             # Выводим текущее значение
150
```

Не закрывая окно **Python Shell**, изменим значение переменной `x` на `800`, а затем попробуем заново импортировать модуль и вывести текущее значение переменной:

```
>>> # Изменяем значение в модуле на 800
>>> import tests2
>>> print(tests2.x)            # Значение не изменилось
150
```

Как видно из примера, значение переменной `x` не изменилось. Теперь перезагрузим модуль с помощью функции `reload()`:

```
>>> from imp import reload
>>> reload(tests2)             # Перезагружаем модуль
<module 'tests2' from 'C:\book\tests2.py'>
>>> print(tests2.x)            # Значение изменилось
800
```

При использовании функции `reload()` следует учитывать, что идентификаторы, импортированные с помощью инструкции `from`, перезагружены не будут. Кроме того, повторно не загружаются скомпилированные модули, написанные на других языках программирования, — например, на языке C.

12.5. Пакеты

Пакетом называется папка с модулями, в которой расположен файл инициализации __init__.py. Файл инициализации может быть пустым или содержать код, который будет выполнен при первой операции импортирования любого модуля, входящего в состав пакета. В любом случае он обязательно должен присутствовать внутри папки с модулями.

В качестве примера создадим следующую структуру файлов и папок:

```
main.py                  # Основной файл с программой
folder1\                 # Папка на одном уровне вложенности с main.py
    __init__.py          # Файл инициализации
    module1.py           # Модуль folder1\module1.py
    folder2\             # Вложенная папка
        __init__.py      # Файл инициализации
        module2.py       # Модуль folder1\folder2\module2.py
        module3.py       # Модуль folder1\folder2\module3.py
```

Содержимое файлов __init__.py приведено в листинге 12.20.

Листинг 12.20. Содержимое файлов __init__.py

```
# -*- coding: utf-8 -*-
print("__init__ из", __name__)
```

Содержимое модулей module1.py, module2.py и module3.py приведено в листинге 12.21.

Листинг 12.21. Содержимое модулей module1.py, module2.py и module3.py

```
# -*- coding: utf-8 -*-
msg = "Модуль {0}".format(__name__)
```

Теперь импортируем эти модули в основном файле main.py и получим значение переменной `msg` разными способами. Файл main.py будем запускать с помощью двойного щелчка на значке файла. Содержимое файла main.py приведено в листинге 12.22.

Листинг 12.22. Содержимое файла main.py

```
# -*- coding: utf-8 -*-

# Доступ к модулю folder1\module1.py
import folder1.module1 as m1
                          # Выведет: __init__ из folder1
print(m1.msg)             # Выведет: Модуль folder1.module1
from folder1 import module1 as m2
print(m2.msg)             # Выведет: Модуль folder1.module1
from folder1.module1 import msg
print(msg)                # Выведет: Модуль folder1.module1

# Доступ к модулю folder1\folder2\module2.py
import folder1.folder2.module2 as m3
                          # Выведет: __init__ из folder1.folder2
print(m3.msg)             # Выведет: Модуль folder1.folder2.module2
from folder1.folder2 import module2 as m4
print(m4.msg)             # Выведет: Модуль folder1.folder2.module2
from folder1.folder2.module2 import msg
print(msg)                # Выведет: Модуль folder1.folder2.module2

input()
```

Как видно из примера, пакеты позволяют распределить модули по папкам. Чтобы импортировать модуль, расположенный во вложенной папке, необходимо указать путь к нему, перечислив имена папок через точку. Если модуль расположен в папке C:\folder1\folder2\, то путь к нему из C:\ должен быть записан так: `folder1.folder2`. При использовании инструкции `import` путь к модулю должен включать не только имена папок, но и название модуля без расширения:

```
import folder1.folder2.module2
```

Получить доступ к идентификаторам внутри импортированного модуля можно следующим образом:

```
print(folder1.folder2.module2.msg)
```

Так как постоянно указывать столь длинный идентификатор очень неудобно, можно создать псевдоним, указав его после ключевого слова `as`, и обращаться к идентификаторам модуля через него:

```
import folder1.folder2.module2 as m
print(m.msg)
```

При использовании инструкции `from` можно импортировать как объект модуля, так и определенные идентификаторы из модуля. Чтобы импортировать объект модуля, его название следует указать после ключевого слова `import`:

```
from folder1.folder2 import module2
print(module2.msg)
```

Для импортирования только определенных идентификаторов название модуля указывается в составе пути, а после ключевого слова `import` через запятую указываются идентификаторы:

```
from folder1.folder2.module2 import msg
print(msg)
```

Если необходимо импортировать все идентификаторы из модуля, то после ключевого слова `import` указывается символ `*`:

```
from folder1.folder2.module2 import *
print(msg)
```

Инструкция `from` позволяет также импортировать сразу несколько модулей из пакета. Для этого внутри файла инициализации **__init__.py** в атрибуте `__all__` необходимо указать список модулей, которые будут импортироваться с помощью выражения `from` *<Пакет>* `import *`. В качестве примера изменим содержимое файла **__init__.py** из каталога **folder1\folder2**:

```
# -*- coding: utf-8 -*-
__all__ = ["module2", "module3"]
```

Теперь создадим файл **main2.py** (листинг 12.23) и запустим его.

Листинг 12.23. Содержимое файла main2.py

```
# -*- coding: utf-8 -*-
from folder1.folder2 import *
print(module2.msg)          # Выведет: Модуль folder1.folder2.module2
print(module3.msg)          # Выведет: Модуль folder1.folder2.module3
input()
```

Как видно из примера, после ключевого слова `from` указывается лишь путь к папке без имени модуля. В результате выполнения инструкции `from` все модули, указанные в списке `__all__`, будут импортированы в пространство имен модуля **main.py**.

До сих пор мы рассматривали импортирование модулей из основной программы. Теперь рассмотрим импорт модулей внутри пакета. Для такого случая инструкция `from` поддерживает относительный импорт модулей. Чтобы импортировать модуль, расположенный в той же папке, перед названием модуля указывается точка:

```
from .module import *
```

Чтобы импортировать модуль, расположенный в родительской папке, перед названием модуля указываются две точки:

```
from ..module import *
```

Если необходимо обратиться уровнем еще выше, то указываются три точки:

```
from ...module import *
```

Чем выше уровень, тем больше точек необходимо указать. После ключевого слова `from` можно указывать одни только точки — в этом случае имя модуля вводится после ключевого слова `import`:

```
from .. import module
```

Рассмотрим относительный импорт на примере. Для этого создадим в папке C:\folder1\folder2\ модуль module4.py, чей код показан в листинге 12.24.

Листинг 12.24. Содержимое модуля module4.py

```
# -*- coding: utf-8 -*-

# Импорт модуля module2.py из текущего каталога
from . import module2 as m1
var1 = "Значение из: {0}".format(m1.msg)
from .module2 import msg as m2
var2 = "Значение из: {0}".format(m2)

# Импорт модуля module1.py из родительского каталога
from .. import module1 as m3
var3 = "Значение из: {0}".format(m3.msg)
from ..module1 import msg as m4
var4 = "Значение из: {0}".format(m4)
```

Теперь создадим файл main3.py (листинг 12.25) и запустим его с помощью двойного щелчка мышью.

Листинг 12.25. Содержимое файла main3.py

```
# -*- coding: utf-8 -*-
from folder1.folder2 import module4 as m
print(m.var1)          # Значение из: Модуль folder1.folder2.module2
print(m.var2)          # Значение из: Модуль folder1.folder2.module2
print(m.var3)          # Значение из: Модуль folder1.module1
print(m.var4)          # Значение из: Модуль folder1.module1
input()
```

При импортировании модуля внутри пакета с помощью инструкции `import` важно помнить, что в Python производится *абсолютный импорт*. Если при запуске Python-программы двойным щелчком на ее файле в список `sys.path` автоматически добавляется путь к каталогу с исполняемым файлом, то при импорте внутри пакета этого не происходит. Поэтому если изменить содержимое модуля module4.py показанным далее способом, то мы получим сообщение об ошибке или загрузим совсем другой модуль:

```
# -*- coding: utf-8 -*-
import module2        # Ошибка! Поиск модуля по абсолютному пути
var1 = "Значение из: {0}".format(module2.msg)
var2 = var3 = var4 = 0
```

В этом примере мы попытались импортировать модуль module2.py из модуля module4.py. При этом файл main3.py (см. листинг 12.25) мы запускаем с помощью двойного щелчка. Поскольку импорт внутри пакета выполняется по абсолютному пути, поиск модуля module2.py не будет производиться в папке folder1\folder2\. В результате модуль не будет найден. Если в путях поиска модулей находится модуль с таким же именем, то будет импортирован модуль, который мы и не предполагали подключать.

Чтобы подключить модуль, расположенный в той же папке внутри пакета, необходимо воспользоваться относительным импортом с помощью инструкции `from`:

```
from . import module2
```

Или указать полный путь относительно корневого каталога пакета:

```
import folder1.folder2.module2 as module2
```

ГЛАВА 13

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — это способ организации программы, позволяющий использовать один и тот же код многократно. В отличие от функций и модулей, ООП позволяет не только разделить программу на фрагменты, но и описать предметы реального мира в виде удобных сущностей — объектов, а также организовать связи между этими объектами.

Основным «кирпичиком» ООП является *класс* — сложный тип данных, включающий набор переменных и функций для управления значениями, хранящимися в этих переменных. Переменные называют *атрибутами* или *свойствами*, а функции — *методами*. Класс является фабрикой объектов, т. е. позволяет создать неограниченное количество экземпляров, основанных на этом классе.

13.1. Определение класса и создание экземпляра класса

Класс описывается с помощью ключевого слова `class` по следующей схеме:

```
class <Название класса>[(<Класс1>[, ..., <КлассN>])]:
    [""" Строка документирования """]
    <Описание атрибутов и методов>
```

Инструкция создает новый объект и присваивает ссылку на него идентификатору, указанному после ключевого слова `class`. Это означает, что название класса должно полностью соответствовать правилам именования переменных. После названия класса в круглых скобках можно указать один или несколько базовых классов через запятую. Если же класс не наследует базовые классы, то круглые скобки можно не указывать. Следует заметить, что все выражения внутри инструкции `class` выполняются при создании класса, а не его экземпляра. Для примера создадим класс, внутри которого просто выводится сообщение (листинг 13.1).

Листинг 13.1. Создание определения класса

```
# -*- coding: utf-8 -*-
class MyClass:
    """ Это строка документирования """
    print("Инструкции выполняются сразу")
input()
```

Этот пример содержит лишь определение класса `MyClass` и не создает экземпляр класса. Как только поток выполнения достигнет инструкции `class`, сообщение, указанное в функции `print()`, будет сразу выведено.

Создание атрибута класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, — с помощью инструкции `def`. Методам класса в первом параметре, который обязательно следует указать явно, автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть именем `self`, хотя это и не обязательно. Доступ к атрибутам и методам класса внутри определяемого метода производится через переменную `self` с помощью точечной нотации — к атрибуту `x` из метода класса можно обратиться так: `self.x`.

Чтобы использовать атрибуты и методы класса, необходимо создать экземпляр класса согласно следующему синтаксису:

```
<Экземпляр класса> = <Название класса>([<Параметры>])
```

При обращении к методам класса используется такой формат:

```
<Экземпляр класса>.<Имя метода>([<Параметры>])
```

Обратите внимание на то, что при вызове метода не нужно передавать ссылку на экземпляр класса в качестве параметра, как это делается в определении метода внутри класса. Ссылку на экземпляр класса интерпретатор передает автоматически.

Обращение к атрибутам класса осуществляется аналогично:

```
<Экземпляр класса>.<Имя атрибута>
```

Определим класс `MyClass` с атрибутом `x` и методом `print_x()`, выводящим значение этого атрибута, а затем создадим экземпляр класса и вызовем метод (листинг 13.2).

Листинг 13.2. Создание атрибута и метода

```
class MyClass:
    def __init__(self):     # Конструктор
        self.x = 10         # Атрибут экземпляра класса
    def print_x(self):      # self — это ссылка на экземпляр класса
        print(self.x)       # Выводим значение атрибута
c = MyClass()               # Создание экземпляра класса
                            # Вызываем метод print_x()
c.print_x()                 # self не указывается при вызове метода
print(c.x)                  # К атрибуту можно обратиться непосредственно
```

Для доступа к атрибутам и методам можно использовать и следующие функции:

- `getattr()` — возвращает значение атрибута по его названию, заданному в виде строки. С помощью этой функции можно сформировать имя атрибута динамически во время выполнения программы. Формат функции:

  ```
  getattr(<Объект>, <Атрибут>[, <Значение по умолчанию>])
  ```

  Если указанный атрибут не найден, возбуждается исключение `AttributeError`. Чтобы избежать вывода сообщения об ошибке, в третьем параметре можно указать значение, которое будет возвращаться, если атрибут не существует;

- `setattr()` — задает значение атрибута. Название атрибута указывается в виде строки. Формат функции:

  ```
  setattr(<Объект>, <Атрибут>, <Значение>)
  ```

Вторым параметром функции `setattr()` можно передать имя несуществующего атрибута — в этом случае атрибут с указанным именем будет создан;

- `delattr(<Объект>, <Атрибут>)` — удаляет атрибут, чье название указано в виде строки;
- `hasattr(<Объект>, <Атрибут>)` — проверяет наличие указанного атрибута. Если атрибут существует, функция возвращает значение `True`.

Продемонстрируем работу функций на примере (листинг 13.3).

Листинг 13.3. Функции `getattr()`, `setattr()` и `hasattr()`

```
class MyClass:
    def __init__(self):
        self.x = 10
    def get_x(self):
        return self.x
c = MyClass()                    # Создаем экземпляр класса
print(getattr(c, "x"))           # Выведет: 10
print(getattr(c, "get_x")())     # Выведет: 10
print(getattr(c, "y", 0))        # Выведет: 0, т. к. атрибут не найден
setattr(c, "y", 20)              # Создаем атрибут y
print(getattr(c, "y", 0))        # Выведет: 20
delattr(c, "y")                  # Удаляем атрибут y
print(getattr(c, "y", 0))        # Выведет: 0, т. к. атрибут не найден
print(hasattr(c, "x"))           # Выведет: True
print(hasattr(c, "y"))           # Выведет: False
```

Все атрибуты класса в языке Python являются открытыми (public), т. е. доступными для непосредственного изменения как из самого класса, так и из других классов и из основного кода программы.

Кроме того, атрибуты допускается создавать динамически после создания класса — можно создать как атрибут объекта класса, так и атрибут экземпляра класса. Рассмотрим это на примере (листинг 13.4).

Листинг 13.4. Атрибуты объекта класса и экземпляра класса

```
class MyClass:                    # Определяем пустой класс
    pass
MyClass.x = 50                    # Создаем атрибут объекта класса
c1, c2 = MyClass(), MyClass()     # Создаем два экземпляра класса
c1.y = 10                         # Создаем атрибут экземпляра класса
c2.y = 20                         # Создаем атрибут экземпляра класса
print(c1.x, c1.y)                 # Выведет: 50 10
print(c2.x, c2.y)                 # Выведет: 50 20
```

В этом примере мы определяем пустой класс, разместив в нем оператор `pass`. Далее создаем атрибут объекта класса: `x`. Этот атрибут будет доступен всем создаваемым экземплярам класса. Затем создаем два экземпляра класса и добавляем одноименные атрибуты: `y`. Значения этих атрибутов будут разными в каждом экземпляре класса. Но если создать новый экземпляр (например, `c3`), то атрибут `y` в нем определен не будет. Таким образом, с помощью

классов можно имитировать типы данных, поддерживаемые другими языками программирования (например, тип `struct`, доступный в языке C).

Очень важно понимать разницу между атрибутами объекта класса и атрибутами экземпляра класса. *Атрибут объекта класса* доступен всем экземплярам класса, и после изменения атрибута значение изменится во всех экземплярах класса. *Атрибут экземпляра класса* может хранить уникальное значение для каждого экземпляра, и изменение его в одном экземпляре класса не затронет значения одноименного атрибута в других экземплярах того же класса. Рассмотрим это на примере, создав класс с атрибутом объекта класса (`x`) и атрибутом экземпляра класса (`y`):

```
>>> class MyClass:
        x = 10                    # Атрибут объекта класса
        def __init__(self):
            self.y = 20           # Атрибут экземпляра класса
```

Теперь создадим два экземпляра этого класса:

```
>>> c1 = MyClass()                # Создаем экземпляр класса
>>> c2 = MyClass()                # Создаем экземпляр класса
```

Выведем значения атрибута `x`, а затем изменим значение и опять произведем вывод:

```
>>> print(c1.x, c2.x)             # 10 10
>>> MyClass.x = 88                # Изменяем атрибут объекта класса
>>> print(c1.x, c2.x)             # 88 88
```

Как видно из примера, изменение атрибута объекта класса затронуло значение в двух экземплярах класса сразу. Теперь произведем аналогичную операцию с атрибутом `y`:

```
>>> print(c1.y, c2.y)             # 20 20
>>> c1.y = 88                     # Изменяем атрибут экземпляра класса
>>> print(c1.y, c2.y)             # 88 20
```

В этом случае изменилось значение атрибута только в экземпляре `c1`.

Следует также учитывать, что в одном классе могут одновременно существовать атрибут объекта и атрибут экземпляра с одним именем. Изменение атрибута объекта класса мы производили следующим образом:

```
>>> MyClass.x = 88                # Изменяем атрибут объекта класса
```

Если после этой инструкции вставить инструкцию

```
>>> c1.x = 200                    # Создаем атрибут экземпляра
```

то будет создан атрибут экземпляра класса, а не изменено значение атрибута объекта класса. Чтобы увидеть разницу, выведем значения атрибутов:

```
>>> print(c1.x, MyClass.x)        # 200 88
```

13.2. Методы *__init__()* и *__del__()*

При создании экземпляра класса интерпретатор автоматически вызывает метод инициализации `__init__()`. В других языках программирования такой метод принято называть *конструктором класса*. Формат метода:

```
def __init__(self[, <Значение1>[, ..., <ЗначениеN>]]):
    <Инструкции>
```

С помощью метода `__init__()` атрибутам класса можно присвоить начальные значения. При создании экземпляра класса параметры этого метода указываются после имени класса в круглых скобках:

```
<Экземпляр класса> = <Имя класса>([<Значение1>[, ..., <ЗначениеN>]])
```

Пример использования метода `__init__()` приведен в листинге 13.5.

Листинг 13.5. Метод `__init__()`

```
class MyClass:
    def __init__(self, value1, value2): # Конструктор
        self.x = value1
        self.y = value2
c = MyClass(100, 300)                   # Создаем экземпляр класса
print(c.x, c.y)                         # Выведет: 100 300
```

Если конструктор вызывается при создании экземпляра, то перед уничтожением экземпляра автоматически вызывается метод, называемый *деструктором*. В языке Python деструктор реализуется в виде предопределенного метода `__del__()` (листинг 13.6). Следует заметить, что метод не будет вызван, если на экземпляр класса существует хотя бы одна ссылка. Впрочем, поскольку интерпретатор самостоятельно заботится об удалении объектов, использование деструктора в языке Python не имеет особого смысла.

Листинг 13.6. Метод `__del__()`

```
class MyClass:
    def __init__(self): # Конструктор класса
        print("Вызван метод __init__()")
    def __del__(self):  # Деструктор класса
        print("Вызван метод __del__()")
c1 = MyClass()          # Выведет: Вызван метод __init__()
del c1                  # Выведет: Вызван метод __del__()
c2 = MyClass()          # Выведет: Вызван метод __init__()
c3 = c2                 # Создаем ссылку на экземпляр класса
del c2                  # Ничего не выведет, т. к. существует ссылка
del c3                  # Выведет: Вызван метод __del__()
```

13.3. Наследование

Наследование является, пожалуй, самым главным понятием ООП. Предположим, у нас есть класс (например, `Class1`). При помощи наследования мы можем создать новый класс (например, `Class2`), в котором будет реализован доступ ко всем атрибутам и методам класса `Class1` (листинг 13.7).

Листинг 13.7. Наследование

```
class Class1:          # Базовый класс
    def func1(self):
        print("Метод func1() класса Class1")
```

```
    def func2(self):
        print("Метод func2() класса Class1")

class Class2(Class1): # Класс Class2 наследует класс Class1
    def func3(self):
        print("Метод func3() класса Class2")
c = Class2()          # Создаем экземпляр класса Class2
c.func1()             # Выведет: Метод func1() класса Class1
c.func2()             # Выведет: Метод func2() класса Class1
c.func3()             # Выведет: Метод func3() класса Class2
```

Как видно из примера, класс `Class1` указывается внутри круглых скобок в определении класса `Class2`. Таким образом, класс `Class2` наследует все атрибуты и методы класса `Class1`. Класс `Class1` называется *базовым* или *суперклассом*, а класс `Class2` — *производным* или *подклассом*.

Если имя метода в классе `Class2` совпадает с именем метода класса `Class1`, то будет использоваться метод из класса `Class2`. Чтобы вызвать одноименный метод из базового класса, перед методом следует через точку написать название базового класса, а в первом параметре метода — явно указать ссылку на экземпляр класса. Рассмотрим это на примере (листинг 13.8).

Листинг 13.8. Переопределение методов

```
class Class1:                   # Базовый класс
    def __init__(self):
        print("Конструктор базового класса")
    def func1(self):
        print("Метод func1() класса Class1")

class Class2(Class1):           # Класс Class2 наследует класс Class1
    def __init__(self):
        print("Конструктор производного класса")
        Class1.__init__(self) # Вызываем конструктор базового класса
    def func1(self):
        print("Метод func1() класса Class2")
        Class1.func1(self)      # Вызываем метод базового класса

c = Class2()                    # Создаем экземпляр класса Class2
c.func1()                       # Вызываем метод func1()
```

Выведет:

```
Конструктор производного класса
Конструктор базового класса
Метод func1() класса Class2
Метод func1() класса Class1
```

> ***Внимание!***
>
> Конструктор базового класса автоматически не вызывается, если он переопределен в производном классе.

Чтобы вызвать одноименный метод из базового класса, также можно воспользоваться функцией `super()`. Формат функции:

```
super([<Класс>, <Указатель self>])
```

С помощью функции `super()` инструкцию

```
Class1.__init__(self)          # Вызываем конструктор базового класса
```

можно записать так:

```
super().__init__()             # Вызываем конструктор базового класса
```

или так:

```
super(Class2, self).__init__() # Вызываем конструктор базового класса
```

Обратите внимание на то, что при использовании функции `super()` не нужно явно передавать указатель `self` в вызываемый метод. Кроме того, в первом параметре функции `super()` указывается производный класс, а не базовый. Поиск идентификатора будет производиться во всех базовых классах. Результатом поиска станет первый найденный идентификатор в цепочке наследования.

> ***Примечание***
>
> В последних версиях Python 2 существовало два типа классов: «классические» классы и классы нового стиля. Классы нового стиля должны были явно наследовать класс `object`. В Python 3 все классы являются классами нового стиля, но наследуют класс `object` неявно. Таким образом, все классы верхнего уровня являются наследниками этого класса, даже если он не указан в списке наследования. «Классические» классы (в понимании Python 2) в Python 3 больше не поддерживаются.

13.4. Множественное наследование

В определении класса в круглых скобках можно указать сразу несколько базовых классов через запятую. Рассмотрим множественное наследование на примере (листинг 13.9).

Листинг 13.9. Множественное наследование

```
class Class1:          # Базовый класс для класса Class2
    def func1(self):
        print("Метод func1() класса Class1")

class Class2(Class1): # Класс Class2 наследует класс Class1
    def func2(self):
        print("Метод func2() класса Class2")

class Class3(Class1): # Класс Class3 наследует класс Class1
    def func1(self):
        print("Метод func1() класса Class3")
    def func2(self):
        print("Метод func2() класса Class3")
    def func3(self):
        print("Метод func3() класса Class3")
    def func4(self):
        print("Метод func4() класса Class3")
```

```
class Class4(Class2, Class3): # Множественное наследование
    def func4(self):
        print("Метод func4() класса Class4")
c = Class4()              # Создаем экземпляр класса Class4
c.func1()                 # Выведет: Метод func1() класса Class3
c.func2()                 # Выведет: Метод func2() класса Class2
c.func3()                 # Выведет: Метод func3() класса Class3
c.func4()                 # Выведет: Метод func4() класса Class4
```

Метод `func1()` определен в двух классах: `Class1` и `Class3`. Так как вначале просматриваются все базовые классы, непосредственно указанные в определении текущего класса, метод `func1()` будет найден в классе `Class3` (поскольку он указан в числе базовых классов в определении `Class4`), а не в классе `Class1`.

Метод `func2()` также определен в двух классах: `Class2` и `Class3`. Так как класс `Class2` стоит первым в списке базовых классов, то метод будет найден именно в нем. Чтобы наследовать метод из класса `Class3`, следует указать это явным образом. Переделаем определение класса `Class4` из предыдущего примера и наследуем метод `func2()` из класса `Class3` (листинг 13.10).

Листинг 13.10. Указание класса при наследовании метода

```
class Class4(Class2, Class3): # Множественное наследование
    # Наследуем func2() из класса Class3, а не из класса Class2
    func2 = Class3.func2
    def func4(self):
        print("Метод func4() класса Class4")
```

Вернемся к листингу 13.9. Метод `func3()` определен только в классе `Class3`, поэтому метод наследуется от этого класса. Метод `func4()`, определенный в классе `Class3`, переопределяется в производном классе.

Если искомый метод найден в производном классе, то вся иерархия наследования просматриваться не будет.

Для получения перечня базовых классов можно воспользоваться атрибутом `__bases__`. В качестве значения атрибут возвращает кортеж. В качестве примера выведем базовые классы для всех классов из предыдущего примера:

```
>>> print(Class1.__bases__)
>>> print(Class2.__bases__)
>>> print(Class3.__bases__)
>>> print(Class4.__bases__)
```

Выведет:

```
(<class 'object'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class2'>, <class '__main__.Class3'>)
```

Рассмотрим порядок поиска идентификаторов при сложной иерархии множественного наследования (листинг 13.11).

Листинг 13.11. Поиск идентификаторов при множественном наследовании

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c = Class7()
print(c.x)
```

Последовательность поиска атрибута `x` будет такой:

```
Class7 -> Class4 -> Class3 — > Class6 -> Class5 -> Class2 -> Class1
```

Получить всю цепочку наследования позволяет атрибут `__mro__`:

```
>>> print(Class7.__mro__)
```

Результат выполнения:

```
(<class '__main__.Class7'>, <class '__main__.Class4'>,
 <class '__main__.Class3'>, <class '__main__.Class6'>,
 <class '__main__.Class5'>, <class '__main__.Class2'>,
 <class '__main__.Class1'>, <class 'object'>)
```

13.4.1. Примеси и их использование

Множественное наследование, поддерживаемое Python, позволяет реализовать интересный способ расширения функциональности классов с помощью так называемых *примесей* (mixins). Примесь — это класс, включающий какие-либо атрибуты и методы, которые необходимо добавить к другим классам. Объявляются они точно так же, как и обычные классы.

В качестве примера объявим класс-примесь `Mixin`, после чего объявим еще два класса, добавим к их функциональности ту, что определена в примеси `Mixin`, и проверим ее в действии (листинг 13.12).

Листинг 13.12. Расширение функциональности классов посредством примеси

```
class Mixin:                          # Определяем сам класс-примесь
    attr = 0                          # Определяем атрибут примеси
    def mixin_method(self):           # Определяем метод примеси
        print("Метод примеси")

class Class1 (Mixin):
    def method1(self):
        print("Метод класса Class1")

class Class2 (Class1, Mixin):
    def method2(self):
        print("Метод класса Class2")
```

```
c1 = Class1()
c1.method1()
c1.mixin_method()                    # Class1 поддерживает метод примеси

c2 = Class2()
c2.method1()
c2.method2()
c2.mixin_method()                    # Class2 также поддерживает метод примеси
```

Вот результат выполнения кода, приведенного в листинге 13.12:

```
Метод класса Class1
Метод примеси
Метод класса Class1
Метод класса Class2
Метод примеси
```

Примеси активно применяются в различных дополнительных библиотеках — в частности, в популярном веб-фреймворке Django.

13.5. Специальные методы

Классы поддерживают следующие специальные методы:

♦ `__call__()` — позволяет обработать вызов экземпляра класса как вызов функции. Формат метода:

  ```
  __call__(self[, <Параметр1>[, ..., <ПараметрN>]])
  ```

  Пример:

  ```
  class MyClass:
      def __init__(self, m):
          self.msg = m
      def __call__(self):
          print(self.msg)
  c1 = MyClass("Значение1") # Создание экземпляра класса
  c2 = MyClass("Значение2") # Создание экземпляра класса
  c1()                      # Выведет: Значение1
  c2()                      # Выведет: Значение2
  ```

♦ `__getattr__(self, <Атрибут>)` — вызывается при обращении к несуществующему атрибуту класса:

  ```
  class MyClass:
      def __init__(self):
          self.i = 20
      def __getattr__(self, attr):
          print("Вызван метод __getattr__()")
          return 0
  c = MyClass()
  # Атрибут i существует
  print(c.i)     # Выведет: 20. Метод __getattr__() не вызывается
  # Атрибут s не существует
  print(c.s)     # Выведет: Вызван метод __getattr__() 0
  ```

- `__getattribute__(self, <Атрибут>)` — вызывается при обращении к любому атрибуту класса. Необходимо учитывать, что использование точечной нотации (для обращения к атрибуту класса) внутри этого метода приведет к зацикливанию. Чтобы избежать зацикливания, следует вызвать метод `__getattribute__()` объекта `object` и внутри этого метода вернуть значение атрибута или возбудить исключение `AttributeError`:

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattribute__(self, attr):
        print("Вызван метод __getattribute__()")
        return object.__getattribute__(self, attr) # Только так!!!
c = MyClass()
print(c.i)     # Выведет: Вызван метод __getattribute__() 20
```

- `__setattr__(self, <Атрибут>, <Значение>)` — вызывается при попытке присваивания значения атрибуту экземпляра класса. Если внутри метода необходимо присвоить значение атрибуту, следует использовать словарь `__dict__`, поскольку при применении точечной нотации метод `__setattr__()` будет вызван повторно, что приведет к зацикливанию:

```
class MyClass:
    def __setattr__(self, attr, value):
        print("Вызван метод __setattr__()")
        self.__dict__[attr] = value                # Только так!!!
c = MyClass()
c.i = 10       # Выведет: Вызван метод __setattr__()
print(c.i)     # Выведет: 10
```

- `__delattr__(self, <Атрибут>)` — вызывается при удалении атрибута с помощью инструкции `del <Экземпляр класса>.<Атрибут>`;
- `__len__(self)` — вызывается при использовании функции `len()`, а также для проверки объекта на логическое значение при отсутствии метода `__bool__()`. Метод должен возвращать положительное целое число:

```
class MyClass:
    def __len__(self):
        return 50
c = MyClass()
print(len(c))              # Выведет: 50
```

- `__bool__(self)` — вызывается при использовании функции `bool()`;
- `__int__(self)` — вызывается при преобразовании объекта в целое число с помощью функции `int()`;
- `__float__(self)` — вызывается при преобразовании объекта в вещественное число с помощью функции `float()`;
- `__complex__(self)` — вызывается при преобразовании объекта в комплексное число с помощью функции `complex()`;
- `__round__(self, n)` — вызывается при использовании функции `round()`;
- `__index__(self)` — вызывается при использовании функций `bin()`, `hex()` и `oct()`;

- `__repr__(self)` и `__str__(self)` — служат для преобразования объекта в строку. Метод `__repr__()` вызывается при выводе в интерактивной оболочке, а также при использовании функции `repr()`. Метод `__str__()` вызывается при выводе с помощью функции `print()`, а также при использовании функции `str()`. Если метод `__str__()` отсутствует, будет вызван метод `__repr__()`. В качестве значения методы `__repr__()` и `__str__()` должны возвращать строку:

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __repr__(self):
        return "Вызван метод __repr__() {0}".format(self.msg)
    def __str__(self):
        return "Вызван метод __str__() {0}".format(self.msg)
c = MyClass("Значение")
print(repr(c)) # Выведет: Вызван метод __repr__() Значение
print(str(c))  # Выведет: Вызван метод __str__() Значение
print(c)       # Выведет: Вызван метод __str__() Значение
```

- `__hash__(self)` — этот метод следует переопределить, если экземпляр класса планируется использовать в качестве ключа словаря или внутри множества:

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __hash__(self):
        return hash(self.x)
c = MyClass(10)
d = {}
d[c] = "Значение"
print(d[c]) # Выведет: Значение
```

Классы поддерживают еще несколько специальных методов, которые применяются лишь в особых случаях и будут рассмотрены в *главе 15*.

13.6. Перегрузка операторов

Перегрузка операторов позволяет экземплярам классов участвовать в обычных операциях. Чтобы перегрузить оператор, необходимо в классе определить метод со специальным названием. Для перегрузки математических операторов используются следующие методы:

- `x + y` — сложение: `x.__add__(y)`;
- `y + x` — сложение (экземпляр класса справа): `x.__radd__(y)`;
- `x += y` — сложение и присваивание: `x.__iadd__(y)`;
- `x - y` — вычитание: `x.__sub__(y)`;
- `y - x` — вычитание (экземпляр класса справа): `x.__rsub__(y)`;
- `x -= y` — вычитание и присваивание: `x.__isub__(y)`;
- `x * y` — умножение: `x.__mul__(y)`;
- `y * x` — умножение (экземпляр класса справа): `x.__rmul__(y)`;

- `x *= y` — умножение и присваивание: `x.__imul__(y)`;
- `x / y` — деление: `x.__truediv__(y)`;
- `y / x` — деление (экземпляр класса справа): `x.__rtruediv__(y)`;
- `x /= y` — деление и присваивание: `x.__itruediv__(y)`;
- `x // y` — деление с округлением вниз: `x.__floordiv__(y)`;
- `y // x` — деление с округлением вниз (экземпляр класса справа): `x.__rfloordiv__(y)`;
- `x //= y` — деление с округлением вниз и присваивание: `x.__ifloordiv__(y)`;
- `x % y` — остаток от деления: `x.__mod__(y)`;
- `y % x` — остаток от деления (экземпляр класса справа): `x.__rmod__(y)`;
- `x %= y` — остаток от деления и присваивание: `x.__imod__(y)`;
- `x ** y` — возведение в степень: `x.__pow__(y)`;
- `y ** x` — возведение в степень (экземпляр класса справа): `x.__rpow__(y)`;
- `x **= y` — возведение в степень и присваивание: `x.__ipow__(y)`;
- `-x` — унарный минус: `x.__neg__()`;
- `+x` — унарный плюс: `x.__pos__()`;
- `abs(x)` — абсолютное значение: `x.__abs__()`.

Пример перегрузки математических операторов приведен в листинге 13.13.

Листинг 13.13. Пример перегрузки математических операторов

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __add__(self, y):          # Перегрузка оператора +
        print("Экземпляр слева")
        return self.x + y
    def __radd__(self, y):         # Перегрузка оператора +
        print("Экземпляр справа")
        return self.x + y
    def __iadd__(self, y):         # Перегрузка оператора +=
        print("Сложение с присваиванием")
        self.x += y
        return self
c = MyClass(50)
print(c + 10)                # Выведет: Экземпляр слева 60
print(20 + c)                # Выведет: Экземпляр справа 70
c += 30                      # Выведет: Сложение с присваиванием
print(c.x)                   # Выведет: 80
```

Методы перегрузки двоичных операторов:

- `~x` — двоичная инверсия: `x.__invert__()`;
- `x & y` — двоичное И: `x.__and__(y)`;

- `y & x` — двоичное И (экземпляр класса справа): `x.__rand__(y)`;
- `x &= y` — двоичное И и присваивание: `x.__iand__(y)`;
- `x | y` — двоичное ИЛИ: `x.__or__(y)`;
- `y | x` — двоичное ИЛИ (экземпляр класса справа): `x.__ror__(y)`;
- `x |= y` — двоичное ИЛИ и присваивание: `x.__ior__(y)`;
- `x ^ y` — двоичное исключающее ИЛИ: `x.__xor__(y)`;
- `y ^ x` — двоичное исключающее ИЛИ (экземпляр класса справа): `x.__rxor__(y)`;
- `x ^= y` — двоичное исключающее ИЛИ и присваивание: `x.__ixor__(y)`;
- `x << y` — сдвиг влево: `x.__lshift__(y)`;
- `y << x` — сдвиг влево (экземпляр класса справа): `x.__rlshift__(y)`;
- `x <<= y` — сдвиг влево и присваивание: `x.__ilshift__(y)`;
- `x >> y` — сдвиг вправо: `x.__rshift__(y)`;
- `y >> x` — сдвиг вправо (экземпляр класса справа): `x.__rrshift__(y)`;
- `x >>= y` — сдвиг вправо и присваивание: `x.__irshift__(y)`.

Перегрузка операторов сравнения производится с помощью следующих методов:

- `x == y` — равно: `x.__eq__(y)`;
- `x != y` — не равно: `x.__ne__(y)`;
- `x < y` — меньше: `x.__lt__(y)`;
- `x > y` — больше: `x.__gt__(y)`;
- `x <= y` — меньше или равно: `x.__le__(y)`;
- `x >= y` — больше или равно: `x.__ge__(y)`;
- `y in x` — проверка на вхождение: `x.__contains__(y)`.

Пример перегрузки операторов сравнения приведен в листинге 13.14.

Листинг 13.14. Пример перегрузки операторов сравнения

```
class MyClass:
    def __init__(self):
        self.x = 50
        self.arr = [1, 2, 3, 4, 5]
    def __eq__(self, y):           # Перегрузка оператора ==
        return self.x == y
    def __contains__(self, y):     # Перегрузка оператора in
        return y in self.arr
c = MyClass()
print("Равно" if c == 50 else "Не равно") # Выведет: Равно
print("Равно" if c == 51 else "Не равно") # Выведет: Не равно
print("Есть" if 5 in c else "Нет")        # Выведет: Есть
```

13.7. Статические методы и методы класса

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса (статический метод). Для этого перед определением метода внутри класса следует указать декоратор `@staticmethod`. Вызов статического метода без создания экземпляра класса осуществляется следующим образом:

```
<Название класса>.<Название метода>(<Параметры>)
```

Кроме того, можно вызвать статический метод через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Пример использования статических методов приведен в листинге 13.15.

Листинг 13.15. Статические методы

```
class MyClass:
    @staticmethod
    def func1(x, y):              # Статический метод
        return x + y
    def func2(self, x, y):        # Обычный метод в классе
        return x + y
    def func3(self, x, y):
        return MyClass.func1(x, y) # Вызов из метода класса

print(MyClass.func1(10, 20))      # Вызываем статический метод
c = MyClass()
print(c.func2(15, 6))             # Вызываем метод класса
print(c.func1(50, 12))            # Вызываем статический метод
                                  # через экземпляр класса
print(c.func3(23, 5))             # Вызываем статический метод
                                  # внутри класса
```

Обратите внимание на то, что в определении статического метода нет параметра `self`. Это означает, что внутри статического метода нет доступа к атрибутам и методам экземпляра класса.

Методы класса создаются с помощью декоратора `@classmethod`. В качестве первого параметра в метод класса передается ссылка на класс. Вызов метода класса осуществляется следующим образом:

```
<Название класса>.<Название метода>(<Параметры>)
```

Кроме того, можно вызвать метод класса через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Пример использования методов класса приведен в листинге 13.16.

Листинг 13.16. Методы класса

```
class MyClass:
    @classmethod
    def func(cls, x): # Метод класса
        print(cls, x)
```

```
MyClass.func(10)        # Вызываем метод через название класса
c = MyClass()
c.func(50)              # Вызываем метод класса через экземпляр
```

13.8. Абстрактные методы

Абстрактные методы содержат только определение метода без реализации. Предполагается, что производный класс должен переопределить метод и реализовать его функциональность. Чтобы такое предположение сделать более очевидным, часто внутри абстрактного метода возбуждают исключение (листинг 13.17).

Листинг 13.17. Абстрактные методы

```
class Class1:
    def func(self, x):       # Абстрактный метод
        # Возбуждаем исключение с помощью raise
        raise NotImplementedError("Необходимо переопределить метод")

class Class2(Class1):        # Наследуем абстрактный метод
    def func(self, x):       # Переопределяем метод
        print(x)

class Class3(Class1):        # Класс не переопределяет метод
    pass

c2 = Class2()
c2.func(50)                  # Выведет: 50
c3 = Class3()
try:                         # Перехватываем исключения
    c3.func(50)              # Ошибка. Метод func() не переопределен
except NotImplementedError as msg:
    print(msg)               # Выведет: Необходимо переопределить метод
```

В состав стандартной библиотеки входит модуль `abc`. В этом модуле определен декоратор `@abstractmethod`, который позволяет указать, что метод, перед которым расположен декоратор, является абстрактным. При попытке создать экземпляр производного класса, в котором не переопределен абстрактный метод, возбуждается исключение `TypeError`. Рассмотрим использование декоратора `@abstractmethod` на примере (листинг 13.18).

Листинг 13.18. Использование декоратора `@abstractmethod`

```
from abc import ABCMeta, abstractmethod
class Class1(metaclass=ABCMeta):
    @abstractmethod
    def func(self, x):       # Абстрактный метод
        pass

class Class2(Class1):        # Наследуем абстрактный метод
    def func(self, x):       # Переопределяем метод
        print(x)
```

```
class Class3(Class1):       # Класс не переопределяет метод
    pass

c2 = Class2()
c2.func(50)                 # Выведет: 50
try:
    c3 = Class3()           # Ошибка. Метод func() не переопределен
    c3.func(50)
except TypeError as msg:
    print(msg)              # Can't instantiate abstract class Class3
                            # with abstract methods func
```

Имеется возможность создания абстрактных статических методов и абстрактных методов класса, для чего необходимые декораторы указываются одновременно, друг за другом. Для примера объявим класс с абстрактными статическим методом и методом класса (листинг 13.19).

Листинг 13.19. Абстрактный статический метод и абстрактный метод класса

```
from abc import ABCMeta, abstractmethod
class MyClass(metaclass=ABCMeta):
    @staticmethod
    @abstractmethod
    def static_func(self, x):      # Абстрактный статический метод
        pass

    @classmethod
    @abstractmethod
    def class_func(self, x):       # Абстрактный метод класса
        pass
```

13.9. Ограничение доступа к идентификаторам внутри класса

Все идентификаторы внутри класса в языке Python являются открытыми, т. е. доступны для непосредственного изменения. Для имитации частных идентификаторов можно воспользоваться методами `__getattr__()`, `__getattribute__()` и `__setattr__()`, которые перехватывают обращения к атрибутам класса. Кроме того, можно воспользоваться идентификаторами, названия которых начинаются с двух символов подчеркивания. Такие идентификаторы называются *псевдочастными*. Псевдочастные идентификаторы доступны лишь внутри класса, но никак не вне его. Тем не менее изменить идентификатор через экземпляр класса все равно можно, зная, каким образом искажается название идентификатора. Например, идентификатор `__privateVar` внутри класса `Class1` будет доступен по имени `_Class1__privateVar`. Как можно видеть, здесь перед идентификатором добавляется название класса с предваряющим символом подчеркивания. Приведем пример использования псевдочастных идентификаторов (листинг 13.20).

Листинг 13.20. Псевдочастные идентификаторы

```
class MyClass:
    def __init__(self, x):
        self.__privateVar = x
    def set_var(self, x):           # Изменение значения
        self.__privateVar = x
    def get_var(self):              # Получение значения
        return self.__privateVar
c = MyClass(10)                     # Создаем экземпляр класса
print(c.get_var())                  # Выведет: 10
c.set_var(20)                       # Изменяем значение
print(c.get_var())                  # Выведет: 20
try:                                # Перехватываем ошибки
    print(c.__privateVar)           # Ошибка!!!
except AttributeError as msg:
    print(msg)                      # Выведет: 'MyClass' object has
                                    # no attribute '__privateVar'
c._MyClass__privateVar = 50         # Значение псевдочастных атрибутов
                                    # все равно можно изменить
print(c.get_var())                  # Выведет: 50
```

Можно также ограничить перечень атрибутов, разрешенных для экземпляров класса. Для этого разрешенные атрибуты указываются внутри класса в атрибуте `__slots__`. В качестве значения атрибуту можно присвоить строку или список строк с названиями идентификаторов. Если производится попытка обращения к атрибуту, не указанному в `__slots__`, возбуждается исключение `AttributeError` (листинг 13.21).

Листинг 13.21. Использование атрибута `__slots__`

```
class MyClass:
    __slots__ = ["x", "y"]
    def __init__(self, a, b):
        self.x, self.y = a, b
c = MyClass(1, 2)
print(c.x, c.y)                   # Выведет: 1 2
c.x, c.y = 10, 20                 # Изменяем значения атрибутов
print(c.x, c.y)                   # Выведет: 10 20
try:                              # Перехватываем исключения
    c.z = 50                      # Атрибут z не указан в __slots__,
                                  # поэтому возбуждается исключение
except AttributeError as msg:
    print(msg)                    # 'MyClass' object has no attribute 'z'
```

13.10. Свойства класса

Внутри класса можно создать идентификатор, через который в дальнейшем будут производиться операции получения и изменения значения какого-либо атрибута, а также его удаления. Создается такой идентификатор с помощью функции `property()`. Формат функции:

```
<Свойство> = property(<Чтение>[, <Запись>[, <Удаление>
                      [, <Строка документирования>]]])
```

В первых трех параметрах указывается ссылка на соответствующий метод класса. При попытке получить значение будет вызван метод, указанный в первом параметре. При операции присваивания значения будет вызван метод, указанный во втором параметре, — этот метод должен принимать один параметр. В случае удаления атрибута вызывается метод, указанный в третьем параметре. Если в качестве какого-либо параметра задано значение `None`, то это означает, что соответствующий метод не поддерживается. Рассмотрим свойства класса на примере (листинг 13.22).

Листинг 13.22. Свойства класса

```
class MyClass:
    def __init__(self, value):
        self.__var = value
    def get_var(self):        # Чтение
        return self.__var
    def set_var(self, value): # Запись
        self.__var = value
    def del_var(self):        # Удаление
        del self.__var
    v = property(get_var, set_var, del_var, "Строка документирования")
c = MyClass(5)
c.v = 35                     # Вызывается метод set_var()
print(c.v)                   # Вызывается метод get_var()
del c.v                      # Вызывается метод del_var()
```

Python поддерживает альтернативный метод определения свойств — с помощью методов `getter()`, `setter()` и `deleter()`, которые используются в декораторах. Соответствующий пример приведен в листинге 13.23.

Листинг 13.23. Методы `getter()`, `setter()` и `deleter()`

```
class MyClass:
    def __init__(self, value):
        self.__var = value
    @property
    def v(self):                          # Чтение
        return self.__var
    @v.setter
    def v(self, value):                   # Запись
        self.__var = value
    @v.deleter
    def v(self):                          # Удаление
        del self.__var
c = MyClass(5)
c.v = 35                                  # Запись
print(c.v)                                # Чтение
del c.v                                   # Удаление
```

Имеется возможность определить абстрактное свойство — в этом случае все реализующие его методы должны быть переопределены в подклассе. Выполняется это с помощью знакомого нам декоратора `@abstractmethod` из модуля `abc`. Пример определения абстрактного свойства показан в листинге 13.24.

Листинг 13.24. Определение абстрактного свойства

```
from abc import ABCMeta, abstractmethod
class MyClass1(metaclass=ABCMeta):
    def __init__(self, value):
        self.__var = value
    @property
    @abstractmethod
    def v(self):                        # Чтение
        return self.__var
    @v.setter
    @abstractmethod
    def v(self, value):                 # Запись
        self.__var = value
    @v.deleter
    @abstractmethod
    def v(self):                        # Удаление
        del self.__var
```

13.11. Декораторы классов

В языке Python, помимо декораторов функций, поддерживаются *декораторы классов*, которые позволяют изменить поведение самих классов. В качестве параметра декоратор принимает ссылку на объект класса, поведение которого необходимо изменить, и должен возвращать ссылку на тот же класс или какой-либо другой. Пример декорирования класса показан в листинге 13.25.

Листинг 13.25. Декоратор класса

```
def deco(C):                      # Принимает объект класса
    print("Внутри декоратора") # Производит какие-то действия
    return C                      # Возвращает объект класса

@deco
class MyClass:
    def __init__(self, value):
        self.v = value
c = MyClass(5)
print(c.v)
```

ГЛАВА 14

Обработка исключений

Исключения — это извещения интерпретатора, возбуждаемые в случае возникновения ошибки в программном коде или при наступлении какого-либо события. Если в коде не предусмотрена обработка исключения, выполнение программы прерывается, и выводится сообщение об ошибке.

Существуют три типа ошибок в программе:

- *синтаксические* — это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д. — т. е. ошибки в синтаксисе языка. Как правило, интерпретатор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки:

  ```
  >>> print("Нет завершающей кавычки!)
  SyntaxError: EOL while scanning string literal
  ```

- *логические* — это ошибки в логике программы, которые можно выявить только по результатам ее работы. Как правило, интерпретатор не предупреждает о наличии такой ошибки, и программа будет успешно выполняться, но результат ее выполнения окажется не тем, на который мы рассчитывали. Выявить и исправить такие ошибки весьма трудно;
- *ошибки времени выполнения* — это ошибки, которые возникают во время работы программы. Причиной являются события, не предусмотренные программистом. Классическим примером служит деление на ноль:

  ```
  >>> def test(x, y): return x / y

  >>> test(4, 2)                    # Нормально
  2.0
  >>> test(4, 0)                    # Ошибка
  Traceback (most recent call last):
    File "<pyshell#4>", line 1, in <module>
      test(4, 0)                    # Ошибка
    File "<pyshell#2>", line 1, in test
      def test(x, y): return x / y
  ZeroDivisionError: division by zero
  ```

Необходимо заметить, что в Python исключения возбуждаются не только при возникновении ошибки, но и как уведомление о наступлении каких-либо событий. Например, метод `index()` возбуждает исключение `ValueError`, если искомый фрагмент не входит в строку:

```
>>> "Строка".index("текст")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    "Строка".index("текст")
ValueError: substring not found
```

14.1. Инструкция *try...except...else...finally*

Для обработки исключений предназначена инструкция `try`. Формат инструкции:

```
try:
    <Блок, в котором перехватываются исключения>
[except [<Исключение1>[ as <Объект исключения>]]:
    <Блок, выполняемый при возникновении исключения>
[...
except [<ИсключениеN>[ as <Объект исключения>]]:
    <Блок, выполняемый при возникновении исключения>]]
[else:
    <Блок, выполняемый, если исключение не возникло>]
[finally:
    <Блок, выполняемый в любом случае>]
```

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока `try`. В блоке `except` в параметре `<Исключение1>` указывается класс обрабатываемого исключения. Например, обработать исключение, возникающее при делении на ноль, можно так, как показано в листинге 14.1.

Листинг 14.1. Обработка деления на ноль

```
try:                                # Перехватываем исключения
    x = 1 / 0                       # Ошибка: деление на 0
except ZeroDivisionError:           # Указываем класс исключения
    print("Обработали деление на 0")
    x = 0
print(x)                            # Выведет: 0
```

Если в блоке `try` возникло исключение, управление передается блоку `except`. В случае если исключение не соответствует указанному классу, управление передается следующему блоку `except`. Если ни один блок `except` не соответствует исключению, то исключение «всплывает» к обработчику более высокого уровня. Если исключение в программе вообще нигде не обрабатывается, оно передается обработчику по умолчанию, который останавливает выполнение программы и выводит стандартную информацию об ошибке. Таким образом, в обработчике может быть несколько блоков `except` с разными классами исключений. Кроме того, один обработчик можно вложить в другой (листинг 14.2).

Листинг 14.2. Вложенные обработчики

```
try:                                   # Обрабатываем исключения
    try:                               # Вложенный обработчик
        x = 1 / 0                      # Ошибка: деление на 0
```

```
    except NameError:
        print("Неопределенный идентификатор")
    except IndexError:
        print("Несуществующий индекс")
    print("Выражение после вложенного обработчика")
except ZeroDivisionError:
    print("Обработка деления на 0")
    x = 0
print(x)                                     # Выведет: 0
```

В этом примере во вложенном обработчике не указано исключение `ZeroDivisionError`, поэтому исключение «всплывает» к обработчику более высокого уровня.

После обработки исключения управление передается инструкции, расположенной сразу после обработчика. В нашем примере управление будет передано инструкции, выводящей значение переменной `x`, — `print(x)`. Обратите внимание на то, что инструкция `print("Выражение после вложенного обработчика")` выполнена не будет.

В инструкции `except` можно указать сразу несколько исключений, записав их через запятую внутри круглых скобок (листинг 14.3).

Листинг 14.3. Обработка нескольких исключений

```
try:
    x = 1 / 0
except (NameError, IndexError, ZeroDivisionError):
    # Обработка сразу нескольких исключений
    x = 0
print(x) # Выведет: 0
```

Получить информацию об обрабатываемом исключении можно через второй параметр в инструкции `except` (листинг 14.4).

Листинг 14.4. Получение информации об исключении

```
try:
    x = 1 / 0                     # Ошибка деления на 0
except (NameError, IndexError, ZeroDivisionError) as err:
    print(err.__class__.__name__) # Название класса исключения
    print(err)                    # Текст сообщения об ошибке
```

Результат выполнения:

```
ZeroDivisionError
division by zero
```

Для получения информации об исключении можно воспользоваться функцией `exc_info()` из модуля `sys`, которая возвращает кортеж из трех элементов: типа исключения, значения и объекта с трассировочной информацией. Преобразовать эти значения в удобочитаемый вид позволяет модуль `traceback`. Пример использования функции `exc_info()` и модуля `traceback` приведен в листинге 14.5.

Листинг 14.5. Пример использования функции `exc_info()`

```
import sys, traceback
try:
    x = 1 / 0
except ZeroDivisionError:
    Type, Value, Trace = sys.exc_info()
    print("Type: ", Type)
    print("Value:", Value)
    print("Trace:", Trace)
    print("\n", "print_exception()".center(40, "-"))
    traceback.print_exception(Type, Value, Trace, limit=5,
                              file=sys.stdout)
    print("\n", "print_tb()".center(40, "-"))
    traceback.print_tb(Trace, limit=1, file=sys.stdout)
    print("\n", "format_exception()".center(40, "-"))
    print(traceback.format_exception(Type, Value, Trace, limit=5))
    print("\n", "format_exception_only()".center(40, "-"))
    print(traceback.format_exception_only(Type, Value))
```

Результат выполнения примера:

```
Type:  <class 'ZeroDivisionError'>
Value: division by zero
Trace: <traceback object at 0x00000179D4142508>

 -----------print_exception()------------
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.5.py", line 3, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

 ---------------print_tb()---------------
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.5.py", line 3, in <module>
    x = 1 / 0

 -----------format_exception()-----------
['Traceback (most recent call last):\n', '  File
"D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.5.py", line 3, in <module>\n    x = 1 / 0\n', 'ZeroDivisionError:
division by zero\n']

 --------format_exception_only()---------
['ZeroDivisionError: division by zero\n']
```

Если в инструкции `except` не указан класс исключения, то такой блок будет перехватывать все исключения. На практике следует избегать пустых инструкций `except`, т. к. можно перехватить исключение, которое является лишь сигналом системе, а не ошибкой. Пример пустой инструкции `except` приведен в листинге 14.6.

Листинг 14.6. Пример перехвата всех исключений

```
try:
    x = 1 / 0                        # Ошибка деления на 0
except:                              # Обработка всех исключений
    x = 0
print(x)                             # Выведет: 0
```

Если в обработчике присутствует блок `else`, то инструкции внутри этого блока будут выполнены только при отсутствии ошибок. При необходимости выполнить какие-либо завершающие действия вне зависимости от того, возникло исключение или нет, следует воспользоваться блоком `finally`. Для примера выведем последовательность выполнения блоков (листинг 14.7).

Листинг 14.7. Блоки `else` и `finally`

```
try:
    x = 10 / 2                       # Нет ошибки
    #x = 10 / 0                      # Ошибка деления на 0
except ZeroDivisionError:
    print("Деление на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

Результат выполнения при отсутствии исключения:

```
Блок else
Блок finally
```

Последовательность выполнения блоков при наличии исключения будет другой:

```
Деление на 0
Блок finally
```

Необходимо заметить, что при наличии исключения и отсутствии блока `except` инструкции внутри блока `finally` будут выполнены, но исключение не будет обработано. Оно продолжит «всплывание» к обработчику более высокого уровня. Если пользовательский обработчик отсутствует, управление передается обработчику по умолчанию, который прерывает выполнение программы и выводит сообщение об ошибке:

```
>>> try:
        x = 10 / 0
finally: print("Блок finally")

Блок finally
Traceback (most recent call last):
  File "<pyshell#17>", line 2, in <module>
    x = 10 / 0
ZeroDivisionError: division by zero
```

В качестве примера переделаем нашу программу суммирования произвольного количества целых чисел, введенных пользователем (см. листинг 4.12), таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 14.8).

Листинг 14.8. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    if x == "stop":
        break      # Выход из цикла
    try:
        x = int(x) # Преобразуем строку в число
    except ValueError:
        print("Необходимо ввести целое число!")
    else:
        summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода значений и получения результата выглядит так (значения, введенные пользователем, выделены полужирным шрифтом):

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: str
Необходимо ввести целое число!
Введите число: -5
Введите число:
Необходимо ввести целое число!
Введите число: stop
Сумма чисел равна: 5
```

14.2. Инструкция *with...as*

Язык Python поддерживает *протокол менеджеров контекста*. Этот протокол гарантирует выполнение завершающих действий (например, закрытие файла) вне зависимости от того, произошло исключение внутри блока кода или нет.

Для работы с протоколом предназначена инструкция `with...as`. Инструкция имеет следующий формат:

```
with <Выражение1>[ as <Переменная>][, ...,
     <ВыражениеN>[ as <Переменная>]]:
    <Блок, в котором перехватываем исключения>
```

Вначале вычисляется `<Выражение1>`, которое должно возвращать объект, поддерживающий протокол. Этот объект должен поддерживать два метода: `__enter__()` и `__exit__()`. Метод `__enter__()` вызывается после создания объекта. Значение, возвращаемое этим методом, присваивается переменной, указанной после ключевого слова `as`. Если переменная не указана, возвращаемое значение игнорируется. Формат метода `__enter__()`:

```
__enter__(self)
```

Далее выполняются инструкции внутри тела инструкции `with`. Если при выполнении возникло исключение, то управление передается методу `__exit__()`. Метод имеет следующий формат:

```
__exit__(self, <Тип исключения>, <Значение>, <Объект traceback>)
```

Значения, доступные через последние три параметра, полностью эквивалентны значениям, возвращаемым функцией `exc_info()` из модуля `sys`. Если исключение обработано, метод должен вернуть значение `True`, в противном случае — `False`. Если метод возвращает `False`, исключение передается вышестоящему обработчику.

Если при выполнении инструкций, расположенных внутри тела инструкции `with`, исключение не возникло, управление все равно передается методу `__exit__()`. В этом случае последние три параметра будут содержать значение `None`.

Рассмотрим последовательность выполнения протокола на примере (листинг 14.9).

Листинг 14.9. Протокол менеджеров контекста

```
class MyClass:
    def __enter__(self):
        print("Вызван метод __enter__()")
        return self
    def __exit__(self, Type, Value, Trace):
        print("Вызван метод __exit__()")
        if Type is None:  # Если исключение не возникло
            print("Исключение не возникло")
        else:             # Если возникло исключение
            print("Value =", Value)
            return False  # False — исключение не обработано
                          # True  — исключение обработано
print("Последовательность при отсутствии исключения:")
with MyClass():
    print("Блок внутри with")
print("\nПоследовательность при наличии исключения:")
with MyClass() as obj:
    print("Блок внутри with")
    raise TypeError("Исключение TypeError")
```

Результат выполнения:

```
Последовательность при отсутствии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
Исключение не возникло
Последовательность при наличии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
Value = Исключение TypeError
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
```

```
II/Примеры/14/14.9.py", line 19, in <module>
    raise TypeError("Исключение TypeError")
TypeError: Исключение TypeError
```

Некоторые встроенные объекты, например файлы, поддерживают протокол по умолчанию. Если в инструкции `with` указана функция `open()`, то после выполнения инструкций внутри блока файл автоматически будет закрыт. Вот пример использования инструкции `with`:

```
with open("test.txt", "a", encoding="utf-8") as f:
    f.write("Строка\n") # Записываем строку в конец файла
```

Здесь файл test.txt открывается на дозапись данных. После выполнения функции `open()` переменной `f` будет присвоена ссылка на объект файла. С помощью этой переменной мы можем работать с файлом внутри тела инструкции `with`. После выхода из блока вне зависимости от наличия исключения файл будет закрыт.

14.3. Классы встроенных исключений

Все встроенные исключения в языке Python представляют собой классы. Иерархия встроенных классов исключений схематично выглядит так:

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError, OverflowError, ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
        LookupError
            IndexError, KeyError
        MemoryError
        NameError
            UnboundLocalError
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError, ConnectionAbortedError,
                ConnectionRefusedError, ConnectionResetError
            FileExistsError
            FileNotFoundError
            InterruptedError
            IsADirectoryError
            NotADirectoryError
            PermissionError
```

```
            ProcessLookupError
            TimeoutError
        RecursionError
        ReferenceError
        RuntimeError
            NotImplementedError
        SyntaxError
            IndentationError
                TabError
        SystemError
        TypeError
        ValueError
            UnicodeError
                UnicodeDecodeError, UnicodeEncodeError
                UnicodeTranslateError
        Warning
            BytesWarning, DeprecationWarning, FutureWarning, ImportWarning,
            PendingDeprecationWarning, ResourceWarning, RuntimeWarning,
            SyntaxWarning, UnicodeWarning, UserWarning
```

Основное преимущество использования классов для обработки исключений заключается в возможности указания базового класса для перехвата всех исключений соответствующих производных классов. Например, для перехвата деления на ноль мы использовали класс `ZeroDivisionError`, но если вместо него указать базовый класс `ArithmeticError`, будут перехватываться исключения классов `FloatingPointError`, `OverflowError` и `ZeroDivisionError`:

```
try:
    x = 1 / 0                         # Ошибка: деление на 0
except ArithmeticError:               # Указываем базовый класс
    print("Обработали деление на 0")
```

Рассмотрим основные классы встроенных исключений:

- `BaseException` — является классом самого верхнего уровня и базовым для всех прочих классов исключений;
- `Exception` — базовый класс для большинства встроенных в Python исключений. Именно его, а не `BaseException`, необходимо наследовать при создании пользовательского класса исключения;
- `AssertionError` — возбуждается инструкцией `assert`;
- `AttributeError` — попытка обращения к несуществующему атрибуту объекта;
- `EOFError` — возбуждается функцией `input()` при достижении конца файла;
- `ImportError` — невозможно импортировать модуль или пакет;
- `IndentationError` — неправильно расставлены отступы в программе;
- `IndexError` — указанный индекс не существует в последовательности;
- `KeyError` — указанный ключ не существует в словаре;
- `KeyboardInterrupt` — нажата комбинация клавиш <Ctrl>+<C>;
- `MemoryError` — интерпретатору существенно не хватает оперативной памяти;

- ♦ `NameError` — попытка обращения к идентификатору до его определения;
- ♦ `NotImplementedError` — должно возбуждаться в абстрактных методах;
- ♦ `OSError` — базовый класс для всех исключений, возбуждаемых в ответ на возникновение ошибок в операционной системе (отсутствие запрошенного файла, недостаток места на диске и пр.);
- ♦ `OverflowError` — число, получившееся в результате выполнения арифметической операции, слишком велико, чтобы Python смог его обработать;
- ♦ `RecursionError` — превышено максимальное количество проходов рекурсии;
- ♦ `RuntimeError` — неклассифицированная ошибка времени выполнения;
- ♦ `StopIteration` — возбуждается методом `__next__()` как сигнал об окончании итераций;
- ♦ `SyntaxError` — синтаксическая ошибка;
- ♦ `SystemError` — ошибка в самой программе интерпретатора Python;
- ♦ `TabError` — в исходном коде программы встретился символ табуляции, использование которого для создания отступов недопустимо;
- ♦ `TypeError` — тип объекта не соответствует ожидаемому;
- ♦ `UnboundLocalError` — внутри функции переменной присваивается значение после обращения к одноименной глобальной переменной;
- ♦ `UnicodeDecodeError` — ошибка преобразования последовательности байтов в строку;
- ♦ `UnicodeEncodeError` — ошибка преобразования строки в последовательность байтов;
- ♦ `UnicodeTranslationError` — ошибка преобразования строки в другую кодировку;
- ♦ `ValueError` — переданный параметр не соответствует ожидаемому значению;
- ♦ `ZeroDivisionError` — попытка деления на ноль.

14.4. Пользовательские исключения

Для возбуждения пользовательских исключений предназначены две инструкции: `raise` и `assert`.

Инструкция `raise` возбуждает заданное исключение. Она имеет несколько вариантов формата:

```
raise <Экземпляр класса>
raise <Название класса>
raise <Экземпляр или название класса> from <Объект исключения>
raise
```

В *первом варианте формата* инструкции `raise` указывается экземпляр класса возбуждаемого исключения. При создании экземпляра можно передать конструктору класса данные, которые станут доступны через второй параметр в инструкции `except`. Приведем пример возбуждения встроенного исключения `ValueError`:

```
>>> raise ValueError("Описание исключения")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise ValueError("Описание исключения")
ValueError: Описание исключения
```

Пример обработки этого исключения показан в листинге 14.10.

Листинг 14.10. Программное возбуждение исключения

```
try:
    raise ValueError("Описание исключения")
except ValueError as msg:
    print(msg) # Выведет: Описание исключения
```

В качестве исключения можно указать экземпляр пользовательского класса (листинг 14.11).

Листинг 14.11. Создание собственного исключения

```
class MyError(Exception):
    def __init__(self, value):
        self.msg = value
    def __str__(self):
        return self.msg
# Обработка пользовательского исключения
try:
    raise MyError("Описание исключения")
except MyError as err:
    print(err)        # Вызывается метод __str__()
    print(err.msg)    # Обращение к атрибуту класса
# Повторно возбуждаем исключение
raise MyError("Описание исключения")
```

Результат выполнения:

```
Описание исключения
Описание исключения
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.11.py", line 13, in <module>
    raise MyError("Описание исключения")
MyError: Описание исключения
```

Класс `Exception` поддерживает все необходимые методы для вывода сообщения об ошибке. Поэтому в большинстве случаев достаточно создать пустой класс, который наследует класс `Exception` (листинг 14.12).

Листинг 14.12. Упрощенный способ создания собственного исключения

```
class MyError(Exception): pass
try:
    raise MyError("Описание исключения")
except MyError as err:
    print(err)        # Выведет: Описание исключения
```

Во *втором варианте формата* инструкции `raise` в первом параметре задается объект класса, а не экземпляр:

```
try:
    raise ValueError # Эквивалентно: raise ValueError()
except ValueError:
    print("Сообщение об ошибке")
```

В *третьем варианте формата* инструкции `raise` в первом параметре задается экземпляр класса или просто название класса, а во втором параметре указывается объект исключения. В этом случае объект исключения сохраняется в атрибуте `__cause__`. При обработке вложенных исключений эти данные используются для вывода информации не только о последнем исключении, но и о первоначальном исключении. Пример этого варианта формата инструкции `raise` можно увидеть в листинге 14.13.

Листинг 14.13. Применение третьего варианта формата инструкции `raise`

```
try:
    x = 1 / 0
except Exception as err:
    raise ValueError() from err
```

Результат выполнения:

```
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.13.py", line 2, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.13.py", line 4, in <module>
    raise ValueError() from err
ValueError
```

Как видно из результата, мы получили информацию не только по исключению `ValueError`, но и по исключению `ZeroDivisionError`. Следует заметить, что при отсутствии инструкции `from` информация сохраняется неявным образом. Если убрать инструкцию `from` в предыдущем примере, мы получим следующий результат:

```
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.13.py", line 2, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.13.py", line 4, in <module>
    raise ValueError()
ValueError
```

Четвертый вариант формата инструкции `raise` позволяет повторно возбудить последнее исключение и обычно применяется в коде, следующем за инструкцией `except`. Пример этого варианта показан в листинге 14.14.

Листинг 14.14. Применение четвертого варианта формата инструкции `raise`

```
class MyError(Exception): pass
try:
    raise MyError("Сообщение об ошибке")
except MyError as err:
    print(err)
    raise              # Повторно возбуждаем исключение
```

Результат выполнения:

```
Сообщение об ошибке
Traceback (most recent call last):
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/14/14.14.py", line 3, in <module>
    raise MyError("Сообщение об ошибке")
MyError: Сообщение об ошибке
```

Инструкция `assert` возбуждает исключение `AssertionError`, если логическое выражение возвращает значение `False`. Инструкция имеет следующий формат:

```
assert <Логическое выражение>[, <Данные>]
```

Инструкция `assert` эквивалентна следующему коду:

```
if __debug__:
    if not <Логическое выражение>:
        raise AssertionError(<Данные>)
```

Если при запуске программы используется флаг `-O`, то переменная `__debug__` будет иметь ложное значение. Таким образом можно удалить все инструкции `assert` из байт-кода.

Пример использования инструкции `assert` представлен в листинге 14.15.

Листинг 14.15. Использование инструкции `assert`

```
try:
    x = -3
    assert x >= 0, "Сообщение об ошибке"
except AssertionError as err:
    print(err) # Выведет: Сообщение об ошибке
```

ГЛАВА 15

Итераторы, контейнеры и перечисления

Язык Python поддерживает средства для создания классов особого назначения: итераторов, контейнеров и перечислений.

Итераторы — это классы, генерирующие последовательности каких-либо значений. Такие классы мы можем задействовать, например, в циклах `for`:

```
class MyIterator:                # Определяем класс-итератор
    . . .
it = MyIterator()                # Создаем его экземпляр
for v in it:                     # и используем в цикле for
    . . .
```

Контейнеры — классы, которые могут выступать как последовательности (списки или кортежи) или отображения (словари). Мы можем обратиться к любому элементу экземпляра такого класса через его индекс или ключ:

```
class MyList:                    # Определяем класс-список
    . . .
class MyDict:                    # Определяем класс-словарь
    . . .
lst, dct = MyList(), MyDict()    # Используем их
lst[0] = 1
dct["first"] = 578
print(lst[1]), print(dct["second"])
```

Перечисления — особые классы, представляющие наборы каких-либо именованных величин. В этом смысле они аналогичны подобным типам данных, доступным в других языках программирования, — например, в C:

```
from enum import Enum            # Импортируем базовый класс Enum
class Versions(Enum):            # Определяем класс-перечисление
    Python2.7 = "2.7"
    Python3.6 = "3.6"
                                 # Используем его
if python_version == Versions.Python3.6:
    . . .
```

15.1. Итераторы

Для того чтобы превратить класс в итератор, нам следует переопределить в нем два специальных метода:

- `__iter__(self)` — говорит о том, что этот класс является итератором (поддерживает итерационный протокол, как говорят Python-программисты). Он должен возвращать сам экземпляр этого класса, а также при необходимости может выполнять всевозможные предустановки.

  Если в классе одновременно определены методы `__iter__()` и `__getitem__()` (о нем будет рассказано позже), предпочтение отдается первому методу;

- `__next__(self)` — вызывается при выполнении каждой итерации и должен возвращать очередное значение из последовательности. Если последовательность закончилась, в этом методе следует возбудить исключение `StopIteration`, которое сообщит вызывающему коду об окончании итераций.

Для примера рассмотрим класс, хранящий строку и на каждой итерации возвращающий очередной ее символ, начиная с конца (листинг 15.1).

Листинг 15.1. Класс-итератор

```
class ReverseString:
    def __init__(self, s):
        self.__s = s
    def __iter__(self):
        self.__i = 0
        return self
    def __next__(self):
        if self.__i > len(self.__s) - 1:
            raise StopIteration
        else:
            a = self.__s[-self.__i - 1]
            self.__i = self.__i + 1
            return a
```

Проверим его в действии:

```
>>> s = ReverseString("Python")
>>> for a in s: print(a, end="")
nohtyP
```

Результат вполне ожидаем — строка, выведенная задом наперед.

Также мы можем переопределить специальный метод `__len()__`, который вернет количество элементов в последовательности, и, разумеется, специальные методы `__str()__` и `__repr()__`, возвращающие строковое представление итератора (все эти методы были рассмотрены в *главе 13*).

Перепишем код нашего класса-итератора, добавив в него определение методов `__len()__` и `__str()__` (листинг 15.2 — часть кода опущена).

Листинг 15.2. Расширенный класс-итератор

```
class ReverseString:
    . . .
    def __len__(self):
        return len(self.__s)
    def __str__(self):
        return self.__s[::-1]
```

Теперь мы можем получить длину последовательности, хранящейся в экземпляре класса `ReverseString`, и его строковое представление:

```
>>> s = ReverseString("Python")
>>> print(len(s))
6
>>> print(str(s))
nohtyP
```

15.2. Контейнеры

Python позволяет создать как контейнеры-последовательности, аналогичные спискам и кортежам, так и контейнеры-отображения, т. е. словари. Сейчас мы узнаем, как это делается.

15.2.1. Контейнеры-последовательности

Чтобы класс смог реализовать функциональность последовательности, нам следует переопределить в нем следующие специальные методы:

- `__getitem__(self, <Индекс>)` — вызывается при извлечении элемента последовательности по его индексу с помощью операции `<Экземпляр класса>[<Индекс>]`. Метод должен возвращать значение, расположенное по этому индексу. Если индекс не является целым числом или срезом, должно возбуждаться исключение `TypeError`, а если индекса как такового не существует, следует возбудить исключение `IndexError`;
- `__setitem__(self, <Индекс>, <Значение>)` — вызывается в случае присваивания нового значения элементу последовательности с заданным индексом (операция `<Экземпляр класса>[<Индекс>] = <Новое значение>`). Метод не должен возвращать результата. В случае задания индекса недопустимого типа и отсутствия такого индекса в последовательности следует возбуждать те же исключения, что и в случае метода `__getitem__()`;
- `__delitem__(self, <Ключ>)` — вызывается в случае удаления элемента последовательности с заданным индексом с помощью выражения `del <Экземпляр класса>[<Ключ>]`. Метод не должен возвращать результата. В случае задания индекса недопустимого типа и отсутствия такого индекса в последовательности следует возбуждать те же исключения, что и в случае метода `__getitem__()`;
- `__contains__(self, <Значение>)` — вызывается при проверке существования заданного значения в последовательности с применением операторов `in` и `not in`. Метод должен возвращать `True`, если такое значение есть, и `False` — в противном случае.

В классе-последовательности мы можем дополнительно реализовать функциональность итератора (см. *разд. 15.1*), переопределив специальные методы `__iter__()`, `__next__()` и `__len__()`. Чаще всего так и поступают.

Мы уже давно знаем, что строки в Python являются неизменяемыми. Давайте же напишем класс `MutableString`, представляющий строку, которую можно изменять теми же способами, что и список (листинг 15.3).

Листинг 15.3. Класс `MutableString`

```
class MutableString:
    def __init__(self, s):
        self.__s = list(s)

    # Реализуем функциональность итератора
    def __iter__(self):
        self.__i = 0
        return self
    def __next__(self):
        if self.__i > len(self.__s) - 1:
            raise StopIteration
        else:
            a = self.__s[self.__i]
            self.__i = self.__i + 1
            return a
    def __len__(self):
        return len(self.__s)

    def __str__(self):
        return "".join(self.__s)

    # Определяем вспомогательный метод, который будет проверять
    # корректность индекса
    def __iscorrectindex(self, i):
        if type(i) == int or type(i) == slice:
            if type(i) == int and i > self.__len__() - 1:
                raise IndexError
        else:
            raise TypeError

    # Реализуем функциональность контейнера-списка
    def __getitem__(self, i):
        self.__iscorrectindex(i)
        return self.__s[i]
    def __setitem__(self, i, v):
        self.__iscorrectindex(i)
        self.__s[i] = v
    def __delitem__(self, i):
        self.__iscorrectindex(i)
        del self.__s[i]
    def __contains__(self, v):
        return v in self.__s
```

Проверим свеженаписанный класс в действии:

```
>>> s = MutableString("Python")
>>> print(s[-1])
n
>>> s[0] = "J"
>>> print(s)
Jython
>>> del s[2:4]
>>> print(s)
Juon
```

Теперь проверим, как наш класс обрабатывает нештатные ситуации. Введем вот такой код, обращающийся к элементу с несуществующим индексом:

```
>>> s[9] = "u"
```

В ответ интерпретатор Python выдаст вполне ожидаемое сообщение об ошибке:

```
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    s[9] = "u"
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/15/15.3.py", line 36, in __setitem__
    self.__iscorrectindex(i)
  File "D:/Data/Документы/Работа/Книги/Python 3 и PyQt 5 Разработка приложений
II/Примеры/15/15.3.py", line 27, in __iscorrectindex
    raise IndexError
IndexError
```

15.2.2. Контейнеры-словари

Класс, реализующий функциональность перечисления, должен переопределять уже знакомые нам методы: `__getitem__()`, `__setitem__()`, `__delitem__()` и `__contains__()`. Разумеется, при этом следует сделать поправку на то, что вместо индексов здесь будут использоваться ключи произвольного типа (как правило, строкового).

Давайте исключительно для практики напишем класс `Version`, который будет хранить номер версии интерпретатора Python, разбитый на части: старшая цифра, младшая цифра и подрелиз, при этом доступ к частям номера версии будет осуществляться по строковым ключам, как в обычном словаре Python (листинг 15.4). Ради простоты чтения кода функциональность итератора реализовывать не станем, а также заблокируем операцию удаления элемента словаря, возбудив в методе `__delitem__()` исключение `TypeError`.

Листинг 15.4. Класс `Version`

```
class Version:
    def __init__(self, major, minor, sub):
        self.__major = major                    # Старшая цифра
        self.__minor = minor                    # Младшая цифра
        self.__sub = sub                        # Подверсия
    def __str__(self):
        return str(self.__major) + "."  + str(self.__minor) + "." + ↳
        str(self.__sub)
```

```
    # Реализуем функциональность словаря
    def __getitem__(self, k):
        if k == "major":
            return self.__major
        elif k == "minor":
            return self.__minor
        elif k == "sub":
            return self.__sub
        else:
            raise IndexError
    def __setitem__(self, k, v):
        if k == "major":
            self.__major = v
        elif k == "minor":
            self.__minor = v
        elif k == "sub":
            self.__sub = v
        else:
            raise IndexError
    def __delitem__(self, k):
        raise TypeError
    def __contains__(self, v):
        return v == "major" or v == "minor" or v == "sub"
```

Чтобы наш новый класс не бездельничал, дадим ему работу, введя такой код:

```
>>> v = Version(3, 6, 3)
>>> print(v["major"])
3
>>> v["sub"] = 4
>>> print(str(v))
3.6.4
```

Как видим, все работает как надо.

15.3. Перечисления

Перечисление — это определенный самим программистом набор каких-либо именованных значений. Обычно они применяются для того, чтобы дать понятные имена каким-либо значениям, используемым в коде программы, — например, кодам ошибок, возвращаемых функциями Windows API.

Для создания перечислений применяются два класса, определенные в модуле `enum`:

♦ `Enum` — базовый класс для создания классов-перечислений, чьи элементы могут хранить значения произвольного типа.

  Для примера определим класс-перечисление `Versions`, имеющий два элемента: `V2_7` со значением `"2.7"` и `V3_6` со значением `"3.6"` (листинг 15.5). Отметим, что элементы перечислений представляют собой атрибуты объекта класса.

Листинг 15.5. Перечисление с элементами произвольного типа

```
from enum import Enum
class Versions(Enum):
    V2_7 = "2.7"
    V3_6 = "3.6"
```

- `IntEnum` — базовый класс для создания перечислений, способных хранить лишь целочисленные значения.

  Листинг 15.6 представляет код перечисления `Colors` с тремя элементами, хранящими целые числа.

Листинг 15.6. Перечисление с целочисленными элементами

```
from enum import IntEnum
class Colors(IntEnum):
    Red = 1
    Green = 2
    Blue = 3
```

Имена элементов перечислений должны быть уникальны (что и неудивительно — ведь фактически это атрибуты объекта класса). Однако разные элементы все же могут хранить одинаковые значения (листинг 15.7).

Листинг 15.7. Перечисление с элементами, хранящими одинаковые значения

```
from enum import Enum
class Versions(Enum):
    V2_7 = "2.7"
    V3_6 = "3.6"
    MostFresh = "3.6"
```

Чтобы объявить, что наше перечисление может хранить лишь уникальные значения, мы можем использовать декоратор `unique`, также определенный в модуле `enum` (листинг 15.8).

Листинг 15.8. Использование декоратора `unique`

```
from enum import Enum, unique
@unique
class Versions(Enum):
    V2_7 = "2.7"
    V3_6 = "3.6"
```

Если мы попытаемся определить в классе, для которого был указан декоратор `unique`, элементы с одинаковыми значениями, то получим сообщение об ошибке.

Определив перечисление, можно использовать его элементы в вычислениях:

```
>>> e = Versions.V3_6
>>> e
<Versions.V3_6: '3.6'>
```

```
>>> e.value
'3.6'
>>> e == Versions.V2_7
False
```

Отметим, что для этого нам не придется создавать экземпляр класса. Это сделает сам Python, неявно создав экземпляр с тем же именем, что мы дали классу (вся необходимая для этого функциональность определена в базовых классах перечислений `Enum` и `IntEnum`).

Все классы перечислений принадлежат типу `EnumMeta` из модуля `enum`:

```
>>> type(Colors)
<class 'enum.EnumMeta'>
>>> from enum import EnumMeta
>>> type(Colors) == EnumMeta
True
```

Однако элементы перечислений уже являются экземплярами их классов:

```
>>> type(Colors.Red)
<enum 'Colors'>
>>> type(Colors.Red) == Colors
True
```

Над элементами перечислений можно производить следующие операции:

- обращаться к ним по их именам, использовав знакомую нам запись с точкой:

```
>>> Versions.V3_6
<Versions.V3_6: '3.6'>
>>> e = Versions.V3_6
>>> e
<Versions.V3_6: '3.6'>
```

- обращаться к ним в стиле словарей, использовав в качестве ключа имя элемента:

```
>>> Versions["V3_6"]
<Versions.V3_6: '3.6'>
```

- обращаться к ним по их значениям, указав их в круглых скобках после имени класса перечисления:

```
>>> Versions("3.6")
<Versions.V3_6: '3.6'>
```

- получать имена соответствующих им атрибутов класса и их значения, воспользовавшись свойствами `name` и `value` соответственно:

```
>>> Versions.V2_7.name, Versions.V2_7.value
('V2_7', '2.7')
```

- использовать в качестве итератора (необходимая для этого функциональность определена в базовых классах):

```
>>> list(Colors)
[<Colors.Red: 1>, <Colors.Green: 2>, <Colors.Blue: 3>]
>>> for c in Colors: print(c.value, end = " ")
1 2 3
```

- использовать в выражениях с применением операторов равенства, неравенства, `in` и `not in`:

```
>>> e = Versions.V3_6
>>> e == Versions.V3_6
True
>>> e != Versions.V2_7
True
>>> e in Versions
True
>>> e in Colors
False
```

Отметим, что элементы разных перечислений всегда не равны друг другу, даже если хранят одинаковые значения;

♦ использовать элементы перечислений — подклассов `IntEnum` в арифметических выражениях и в качестве индексов перечислений. В этом случае они будут автоматически преобразовываться в целые числа, соответствующие их значениям:

```
>>> Colors.Red + 1              # Значение Colors.Red - 1
2
>>> Colors.Green != 3           # Значение Colors.Green - 2
True
>>> ["a", "b", "c"][Colors.Red]
'b'
```

Помимо элементов, классы перечислений могут включать атрибуты экземпляра класса и методы — как экземпляров, так и объектов класса. При этом методы экземпляра класса всегда вызываются у элемента перечисления (и, соответственно, первым параметром ему передается ссылка на экземпляр класса, представляющий элемент перечисления, у которого был вызван метод), а методы объекта класса — у самого класса перечисления. Для примера давайте рассмотрим код класса перечисления `VersionExtended` (листинг 15.9).

Листинг 15.9. Перечисление, включающее атрибуты и методы

```
from enum import Enum
class VersionExtended(Enum):
    V2_7 = "2.7"
    V3_6 = "3.6"

    # Методы экземпляра класса.
    # Вызываются у элемента перечисления
    def describe(self):
        return self.name, self.value
    def __str__(self):
        return str(__class__.__name__) + "." + self.name + ": " + ↳
        self.value

    # Метод объекта класса.
    # Вызывается у самого класса перечисления
    @classmethod
    def getmostfresh(cls):
        return cls.V3_6
```

В методе `__str__()` мы использовали встроенную переменную `__class__`, хранящую ссылку на объект текущего класса. Атрибут `__name__` этого объекта содержит имя класса в виде строки.

Осталось лишь проверить готовый класс в действии, для чего мы введем следующий код:

```
>>> d = VersionExtended.V2_7.describe()
>>> print(d[0] + ", " + d[1])
V2_7, 2.7
>>> print(VersionExtended.V2_7)
VersionExtended.V2_7: 2.7
>>> print(VersionExtended.getmostfresh())
VersionExtended.V3_6: 3.6
```

Осталось отметить одну важную деталь. На основе класса перечисления можно создавать подклассы только в том случае, если этот класс не содержит атрибутов объекта класса, т. е. собственно элементов перечисления. Если же класс перечисления содержит элементы, попытка определения его подкласса приведет к ошибке:

```
>>> class ExtendedColors(Colors):
        pass

Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    class ExtendedColors(Colors):
NameError: name 'Colors' is not defined
```

> **Примечание**
>
> В составе стандартной библиотеки Python уже давно присутствует модуль `struct`, позволяющий создавать нечто похожее на перечисления. Однако он не столь удобен в работе, как инструменты, предлагаемые модулем `enum`.

ГЛАВА 16

Работа с файлами и каталогами

Очень часто нужно сохранить какие-либо данные. Если эти данные имеют небольшой объем, их можно записать в файл.

16.1. Открытие файла

Прежде чем работать с файлом, необходимо создать объект файла с помощью функции `open()`. Функция имеет следующий формат:

```
open(<Путь к файлу>[, mode='r'][, buffering=-1][, encoding=None][,
     errors=None][, newline=None][, closefd=True])
```

В первом параметре указывается путь к файлу. Путь может быть абсолютным или относительным. При указании абсолютного пути в Windows следует учитывать, что в Python слэш является специальным символом. По этой причине слэш необходимо удваивать или вместо обычных строк использовать неформатированные строки:

```
>>> "C:\\temp\\new\\file.txt"    # Правильно
'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt"      # Правильно
'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt"       # Неправильно!!!
'C:\temp\new\x0cile.txt'
```

Обратите внимание на последний пример. В этом пути из-за того, что слэши не удвоены, возникло присутствие сразу трех специальных символов: `\t`, `\n` и `\f` (отображается как `\x0c`). После преобразования этих специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Если такую строку передать в функцию `open()`, это приведет к исключению `OSError`:

```
>>> open("C:\temp\new\file.txt")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    open("C:\temp\new\file.txt")
OSError: [Errno 22] Invalid argument: 'C:\temp\new\x0cile.txt'
```

Вместо абсолютного пути к файлу можно указать относительный путь, который определяется с учетом местоположения текущего рабочего каталога. Относительный путь будет авто-

матически преобразован в абсолютный путь с помощью функции `abspath()` из модуля `os.path`. Возможны следующие варианты:

- ♦ если открываемый файл находится в текущем рабочем каталоге, можно указать только имя файла:

```
>>> import os.path # Подключаем модуль
>>> # Файл в текущем рабочем каталоге (C:\book\)
>>> os.path.abspath(r"file.txt")
'C:\\book\\file.txt'
```

- ♦ если открываемый файл расположен во вложенной папке, перед именем файла через слэш указываются имена вложенных папок:

```
>>> # Открываемый файл в C:\book\folder1\
>>> os.path.abspath(r"folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> # Открываемый файл в C:\book\folder1\folder2\
>>> os.path.abspath(r"folder1/folder2/file.txt")
'C:\\book\\folder1\\folder2\\file.txt'
```

- ♦ если папка с файлом расположена ниже уровнем, перед именем файла указываются две точки и слэш (`"../"`):

```
>>> # Открываемый файл в C:\
>>> os.path.abspath(r"../file.txt")
'C:\\file.txt'
```

- ♦ если в начале пути расположен слэш, путь отсчитывается от корня диска. В этом случае местоположение текущего рабочего каталога не имеет значения:

```
>>> # Открываемый файл в C:\book\folder1\
>>> os.path.abspath(r"/book/folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> # Открываемый файл в C:\book\folder1\folder2\
>>> os.path.abspath(r"/book/folder1/folder2/file.txt")
'C:\\book\\folder1\\folder2\\file.txt'
```

Как можно видеть, в абсолютном и относительном путях можно указать как прямые, так и обратные слэши. Все они будут автоматически преобразованы с учетом значения атрибута `sep` из модуля `os.path`. Значение этого атрибута зависит от используемой операционной системы. Выведем значение атрибута `sep` в операционной системе Windows:

```
>>> os.path.sep
'\\'
>>> os.path.abspath(r"C:/book/folder1/file.txt")
'C:\\book\\folder1\\file.txt'
```

При использовании относительного пути необходимо учитывать местоположение текущего рабочего каталога, т. к. рабочий каталог не всегда совпадает с каталогом, в котором находится исполняемый файл. Если файл запускается с помощью двойного щелчка на его значке, то каталоги будут совпадать. Если же файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл.

Рассмотрим все это на примере, для чего в каталоге C:\book создадим следующую структуру файлов:

```
C:\book\
    test.py
    folder1\
        __init__.py
        module1.py
```

Содержимое файла C:\book\test.py приведено в листинге 16.1.

Листинг 16.1. Содержимое файла C:\book\test.py

```
# -*- coding: utf-8 -*-
import os, sys
print("%-25s%s" % ("Файл:", os.path.abspath(__file__)))
print("%-25s%s" % ("Текущий рабочий каталог:", os.getcwd()))
print("%-25s%s" % ("Каталог для импорта:", sys.path[0]))
print("%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt")))
print("-" * 40)
import folder1.module1 as m
m.get_cwd()
```

Файл C:\book\folder1__init__.py создаем пустым. Как вы уже знаете, этот файл указывает интерпретатору Python, что данный каталог является пакетом с модулями. Содержимое файла C:\book\folder1\module1.py приведено в листинге 16.2.

Листинг 16.2. Содержимое файла C:\book\folder1\module1.py

```
# -*- coding: utf-8 -*-
import os, sys
def get_cwd():
    print("%-25s%s" % ("Файл:", os.path.abspath(__file__)))
    print("%-25s%s" % ("Текущий рабочий каталог:", os.getcwd()))
    print("%-25s%s" % ("Каталог для импорта:", sys.path[0]))
    print("%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt")))
```

Запускаем командную строку, переходим в каталог C:\book и запускаем файл test.py:

```
C:\>cd C:\book
C:\book>test.py
Файл:                    C:\book\test.py
Текущий рабочий каталог: C:\book
Каталог для импорта:     C:\book
Путь к файлу:            C:\book\file.txt
----------------------------------------
Файл:                    C:\book\folder1\module1.py
Текущий рабочий каталог: C:\book
Каталог для импорта:     C:\book
Путь к файлу:            C:\book\file.txt
```

В этом примере текущий рабочий каталог совпадает с каталогом, в котором расположен файл test.py. Однако обратите внимание на текущий рабочий каталог внутри модуля module1.py. Если внутри этого модуля в функции `open()` указать имя файла без пути, поиск файла будет произведен в каталоге C:\book, а не C:\book\folder1.

Теперь перейдем в корень диска C: и опять запустим файл test.py:

```
C:\book>cd C:\
C:\>C:\book\test.py
Файл:                    C:\book\test.py
Текущий рабочий каталог: C:\
Каталог для импорта:     C:\book
Путь к файлу:            C:\file.txt
-----------------------------------------
Файл:                    C:\book\folder1\module1.py
Текущий рабочий каталог: C:\
Каталог для импорта:     C:\book
Путь к файлу:            C:\file.txt
```

В этом случае текущий рабочий каталог не совпадает с каталогом, в котором расположен файл test.py. Если внутри файлов test.py и module1.py в функции `open()` указать имя файла без пути, поиск файла будет производиться в корне диска C:, а не в каталогах с этими файлами.

Чтобы поиск файла всегда производился в каталоге с исполняемым файлом, необходимо этот каталог сделать текущим с помощью функции `chdir()` из модуля `os`. Для примера создадим файл test2.py (листинг 16.3).

Листинг 16.3. Пример использования функции `chdir()`

```
# -*- coding: utf-8 -*-
import os, sys
# Делаем каталог с исполняемым файлом текущим
os.chdir(os.path.dirname(os.path.abspath(__file__)))
print("%-25s%s" % ("Файл:", __file__))
print("%-25s%s" % ("Текущий рабочий каталог:", os.getcwd()))
print("%-25s%s" % ("Каталог для импорта:", sys.path[0]))
print("%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt")))
```

Обратите внимание на четвертую строку. С помощью атрибута `__file__` мы получаем путь к исполняемому файлу вместе с именем файла. Атрибут `__file__` не всегда содержит полный путь к файлу. Например, если запуск осуществляется следующим образом:

```
C:\book>C:\Python36\python test2.py
```

то атрибут будет содержать только имя файла без пути. Чтобы всегда получать полный путь к файлу, следует передать значение атрибута в функцию `abspath()` из модуля `os.path`. Далее мы извлекаем путь (без имени файла) с помощью функции `dirname()` и передаем его функции `chdir()`. Теперь, если в функции `open()` указать название файла без пути, поиск будет производиться в каталоге с этим файлом. Запустим файл test2.py с помощью командной строки:

```
C:\>C:\book\test2.py
Файл:                    C:\book\test2.py
Текущий рабочий каталог: C:\book
Каталог для импорта:     C:\book
Путь к файлу:            C:\book\file.txt
```

Функции, предназначенные для работы с каталогами, мы еще рассмотрим подробно в следующих разделах. Сейчас же важно запомнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог, в котором расположен исполняемый файл. Кроме того, пути поиска файлов не имеют никакого отношения к путям поиска модулей.

Необязательный параметр `mode` в функции `open()` может принимать следующие значения:

- `r` — только чтение (значение по умолчанию). После открытия файла указатель устанавливается на начало файла. Если файл не существует, возбуждается исключение `FileNotFoundError`;
- `r+` — чтение и запись. После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение `FileNotFoundError`;
- `w` — запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `w+` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `a` — запись. Если файл не существует, он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется;
- `a+` — чтение и запись. Если файл не существует, он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется;
- `x` — создание файла для записи. Если файл уже существует, возбуждается исключение `FileExistsError`;
- `x+` — создание файла для чтения и записи. Если файл уже существует, возбуждается исключение `FileExistsError`.

После указания режима может следовать модификатор:

- `b` — файл будет открыт в бинарном режиме. Файловые методы принимают и возвращают объекты типа `bytes`;
- `t` — файл будет открыт в текстовом режиме (значение по умолчанию в Windows). Файловые методы принимают и возвращают объекты типа `str`. В этом режиме будет автоматически выполняться обработка символа конца строки — так, в Windows при чтении вместо символов `\r\n` будет подставлен символ `\n`. Для примера создадим файл **file.txt** и запишем в него две строки:

```
>>> f = open(r"file.txt", "w")  # Открываем файл на запись
>>> f.write("String1\nString2") # Записываем две строки в файл
15
>>> f.close()                   # Закрываем файл
```

Поскольку мы указали режим `w`, то, если файл не существует, он будет создан, а если существует, то будет перезаписан.

Теперь выведем содержимое файла в бинарном и текстовом режимах:

```
>>> # Бинарный режим (символ \r остается)
>>> with open(r"file.txt", "rb") as f:
        for line in f:
            print(repr(line))

b'String1\r\n'
b'String2'
```

```
>>> # Текстовый режим (символ \r удаляется)
>>> with open(r"file.txt", "r") as f:
        for line in f:
            print(repr(line))
'String1\n'
'String2'
```

Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла или после вызова функции или метода `flush()`. В необязательном параметре `buffering` можно указать размер буфера. Если в качестве значения указан `0`, то данные будут сразу записываться в файл (значение допустимо только в бинарном режиме). Значение `1` используется при построчной записи в файл (значение допустимо только в текстовом режиме), другое положительное число задает примерный размер буфера, а отрицательное значение (или отсутствие значения) означает установку размера, применяемого в системе по умолчанию. По умолчанию текстовые файлы буферизуются построчно, а бинарные — частями, размер которых интерпретатор выбирает самостоятельно в диапазоне от 4096 до 8192 байтов.

При использовании текстового режима (задается по умолчанию) при чтении производится попытка преобразовать данные в кодировку Unicode, а при записи выполняется обратная операция — строка преобразуется в последовательность байтов в определенной кодировке. По умолчанию назначается кодировка, применяемая в системе. Если преобразование невозможно, возбуждается исключение. Указать кодировку, которая будет использоваться при записи и чтении файла, позволяет параметр `encoding`. Для примера запишем данные в кодировке UTF-8:

```
>>> f = open(r"file.txt", "w", encoding="utf-8")
>>> f.write("Строка")  # Записываем строку в файл
6
>>> f.close()          # Закрываем файл
```

Для чтения этого файла следует явно указать кодировку при открытии файла:

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(line)

Строка
```

При работе с файлами в кодировках UTF-8, UTF-16 и UTF-32 следует учитывать, что в начале файла могут присутствовать служебные символы, называемые сокращенно BOM (Byte Order Mark, метка порядка байтов). Для кодировки UTF-8 эти символы являются необязательными, и в предыдущем примере они не были добавлены в файл при записи. Чтобы символы были добавлены, в параметре `encoding` следует указать значение `utf-8-sig`. Запишем строку в файл в кодировке UTF-8 с BOM:

```
>>> f = open(r"file.txt", "w", encoding="utf-8-sig")
>>> f.write("Строка")  # Записываем строку в файл
6
>>> f.close()          # Закрываем файл
```

Теперь прочитаем файл с разными значениями в параметре `encoding`:

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(repr(line))
```

```
'\ufeffСтрока'
>>> with open(r"file.txt", "r", encoding="utf-8-sig") as f:
        for line in f:
            print(repr(line))

'Строка'
```

В первом примере мы указали значение `utf-8`, поэтому маркер BOM был прочитан из файла вместе с данными. Во втором примере указано значение `utf-8-sig`, поэтому маркер BOM не попал в результат. Если неизвестно, есть ли маркер в файле, и необходимо получить данные без маркера, то следует всегда указывать значение `utf-8-sig` при чтении файла в кодировке UTF-8.

Для кодировок UTF-16 и UTF-32 маркер BOM является обязательным. При указании значений `utf-16` и `utf-32` в параметре `encoding` обработка маркера производится автоматически. При записи данных маркер автоматически вставляется в начало файла, а при чтении он не попадает в результат. Запишем строку в файл, а затем прочитаем ее из файла:

```
>>> with open(r"file.txt", "w", encoding="utf-16") as f:
        f.write("Строка")

6
>>> with open(r"file.txt", "r", encoding="utf-16") as f:
        for line in f:
            print(repr(line))

'Строка'
```

При использовании значений `utf-16-le`, `utf-16-be`, `utf-32-le` и `utf-32-be` маркер BOM необходимо самим добавить в начало файла, а при чтении удалить его.

В параметре `errors` можно указать уровень обработки ошибок. Возможные значения: `"strict"` (при ошибке возбуждается исключение `ValueError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом вопроса или символом с кодом `\ufffd`), `"ignore"` (неизвестные символы игнорируются), `"xmlcharrefreplace"` (неизвестный символ заменяется последовательностью `&#xxxx;`) и `"backslashreplace"` (неизвестный символ заменяется последовательностью `\uxxxx`).

Параметр `newline` задает режим обработки символов конца строк. Поддерживаемые им значения таковы:

- `None` (значение по умолчанию) — выполняется стандартная обработка символов конца строки. Например, в Windows при чтении символы `\r\n` преобразуются в символ `\n`, а при записи производится обратное преобразование;
- `""` (пустая строка) — обработка символов конца строки не выполняется;
- `"<Специальный символ>"` — указанный специальный символ используется для обозначения конца строки, и никакая дополнительная обработка не выполняется. В качестве специального символа можно указать лишь `\r\n`, `\r` и `\n`.

16.2. Методы для работы с файлами

После открытия файла функция `open()` возвращает объект, с помощью которого производится дальнейшая работа с файлом. Тип объекта зависит от режима открытия файла и буферизации. Рассмотрим основные методы:

♦ `close()` — закрывает файл. Так как интерпретатор автоматически удаляет объект, когда на него отсутствуют ссылки, в небольших программах файл можно не закрывать явно. Тем не менее, явное закрытие файла является признаком хорошего стиля программирования. Кроме того, при наличии незакрытого файла генерируется предупреждающее сообщение: `"ResourceWarning: unclosed file"`.

Язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует закрытие файла вне зависимости от того, произошло исключение внутри блока кода или нет:

```
with open(r"file.txt", "w", encoding="cp1251") as f:
    f.write("Строка") # Записываем строку в файл
# Здесь файл уже закрыт автоматически
```

♦ `write(<Данные>)` — записывает данные в файл. Если в качестве параметра указана строка, файл должен быть открыт в текстовом режиме, а если указана последовательность байтов — в бинарном. Помните, что нельзя записывать строку в бинарном режиме и последовательность байтов в текстовом режиме. Метод возвращает количество записанных символов или байтов. Пример записи в файл:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.write("Строка1\nСтрока2") # Записываем строку в файл
15
>>> f.close()                   # Закрываем файл
>>> # Бинарный режим
>>> f = open(r"file.txt", "wb")
>>> f.write(bytes("Строка1\nСтрока2", "cp1251"))
15
>>> f.write(bytearray("\nСтрока3", "cp1251"))
8
>>> f.close()
```

♦ `writelines(<Последовательность>)` — записывает последовательность в файл. Если все элементы последовательности являются строками, файл должен быть открыт в текстовом режиме. Если все элементы являются последовательностями байтов, то файл должен быть открыт в бинарном режиме. Пример записи элементов списка:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.writelines(["Строка1\n", "Строка2"])
>>> f.close()
>>> # Бинарный режим
>>> f = open(r"file.txt", "wb")
>>> arr = [bytes("Строка1\n", "cp1251"), bytes("Строка2", "cp1251")]
>>> f.writelines(arr)
>>> f.close()
```

♦ `writable()` — возвращает `True`, если файл поддерживает запись, и `False` — в противном случае:

```
>>> f = open(r"file.txt", "r")        # Открываем файл для чтения
>>> f.writable()
False
```

```
>>> f = open(r"file.txt", "w")        # Открываем файл для записи
>>> f.writable()
True
```

- read([<Количество>]) — считывает данные из файла. Если файл открыт в текстовом режиме, возвращается строка, а если в бинарном — последовательность байтов. Если параметр не указан, возвращается содержимое файла от текущей позиции указателя до конца файла:

```
>>> # Текстовый режим
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.read()

'Строка1\nСтрока2'
>>> # Бинарный режим
>>> with open(r"file.txt", "rb") as f:
        f.read()

b'\xd1\xf2\xf0\xee\xea\xe01\n\xd1\xf2\xf0\xee\xea\xe02'
```

Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество символов или байтов. Когда достигается конец файла, метод возвращает пустую строку:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.read(8)                # Считываем 8 символов
'Строка1\n'
>>> f.read(8)                # Считываем 8 символов
'Строка2'
>>> f.read(8)                # Достигнут конец файла
''
>>> f.close()
```

- readline([<Количество>]) — считывает из файла одну строку при каждом вызове. Если файл открыт в текстовом режиме, возвращается строка, а если в бинарном — последовательность байтов. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода строки, то таковой добавлен не будет. При достижении конца файла возвращается пустая строка:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.readline(), f.readline()
('Строка1\n', 'Строка2')
>>> f.readline()             # Достигнут конец файла
''
>>> f.close()
>>> # Бинарный режим
>>> f = open(r"file.txt", "rb")
>>> f.readline(), f.readline()
(b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02')
>>> f.readline()             # Достигнут конец файла
b''
>>> f.close()
```

Если в необязательном параметре указано число, считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца файла или из файла не будет прочитано указанное количество символов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество символов, а если количество символов в строке больше, то возвращается указанное количество символов:

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.readline(2), f.readline(2)
('Ст', 'ро')
>>> f.readline(100) # Возвращается одна строка, а не 100 символов
'ка1\n'
>>> f.close()
```

- `readlines()` — считывает все содержимое файла в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода строки, таковой добавлен не будет. Если файл открыт в текстовом режиме, возвращается список строк, а если в бинарном — список объектов типа `bytes`:

```
>>> # Текстовый режим
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.readlines()
['Строка1\n', 'Строка2']
>>> # Бинарный режим
>>> with open(r"file.txt", "rb") as f:
        f.readlines()

[b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02']
```

- `__next__()` — считывает одну строку при каждом вызове. Если файл открыт в текстовом режиме, возвращается строка, а если в бинарном — последовательность байтов. При достижении конца файла возбуждается исключение `StopIteration`:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.__next__(), f.__next__()
('Строка1\n', 'Строка2')
>>> f.__next__() # Достигнут конец файла
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    f.__next__() # Достигнут конец файла
StopIteration
>>> f.close()
```

Благодаря методу `__next__()` мы можем перебирать файл построчно в цикле `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `__next__()`. Для примера выведем все строки, предварительно удалив символ перевода строки:

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> for line in f: print(line.rstrip("\n"), end=" ")

Строка1 Строка2
>>> f.close()
```

- `flush()` — принудительно записывает данные из буфера на диск;
- `fileno()` — возвращает целочисленный дескриптор файла. Возвращаемое значение всегда будет больше числа 2, т. к. число 0 закреплено за стандартным вводом `stdin`, 1 — за стандартным выводом `stdout`, а 2 — за стандартным выводом сообщений об ошибках `stderr`:

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.fileno()                    # Дескриптор файла
3
>>> f.close()
```

- `truncate([<Количество>])` — обрезает файл до указанного количества символов (если задан текстовый режим) или байтов (в случае бинарного режима). Метод возвращает новый размер файла:

```
>>> f = open(r"file.txt", "r+", encoding="cp1251")
>>> f.read()
'Строка1\nСтрока2'
>>> f.truncate(5)
5
>>> f.close()
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.read()

'Строк'
```

- `tell()` — возвращает позицию указателя относительно начала файла в виде целого числа. Обратите внимание на то, что в Windows метод `tell()` считает символ `\r` как дополнительный байт, хотя этот символ удаляется при открытии файла в текстовом режиме:

```
>>> with open(r"file.txt", "w", encoding="cp1251") as f:
        f.write("String1\nString2")


15
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.tell()      # Указатель расположен в начале файла
0
>>> f.readline()  # Перемещаем указатель
'String1\n'
>>> f.tell()      # Возвращает 9 (8 + '\r'), а не 8 !!!
9
>>> f.close()
```

Чтобы избежать этого несоответствия, следует открывать файл в бинарном режиме, а не в текстовом:

```
>>> f = open(r"file.txt", "rb")
>>> f.readline()  # Перемещаем указатель
b'String1\r\n'
>>> f.tell()      # Теперь значение соответствует
9
>>> f.close()
```

- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель в позицию, имеющую заданное `<Смещение>` относительно параметра `<Позиция>`. В качестве параметра `<Позиция>` могут быть указаны следующие атрибуты из модуля `io` или соответствующие им значения:
  - `io.SEEK_SET` или `0` — начало файла (значение по умолчанию);
  - `io.SEEK_CUR` или `1` — текущая позиция указателя. Положительное значение смещения вызывает перемещение к концу файла, отрицательное — к его началу;
  - `io.SEEK_END` или `2` — конец файла.

  Выведем значения этих атрибутов:

```
>>> import io
>>> io.SEEK_SET, io.SEEK_CUR, io.SEEK_END
(0, 1, 2)
```

  Пример использования метода `seek()`:

```
>>> import io
>>> f = open(r"file.txt", "rb")
>>> f.seek(9, io.SEEK_CUR) # 9 байтов от указателя
9
>>> f.tell()
9
>>> f.seek(0, io.SEEK_SET) # Перемещаем указатель в начало
0
>>> f.tell()
0
>>> f.seek(-9, io.SEEK_END) # -9 байтов от конца файла
7
>>> f.tell()
7
>>> f.close()
```

- `seekable()` — возвращает `True`, если указатель файла можно сдвинуть в другую позицию, и `False` — в противном случае:

```
>>> f = open(r"C:\temp\new\file.txt", "r")
>>> f.seekable()
True
```

Помимо методов, объекты файлов поддерживают несколько атрибутов:

- `name` — имя файла;
- `mode` — режим, в котором был открыт файл;
- `closed` — возвращает `True`, если файл был закрыт, и `False` — в противном случае:

```
>>> f = open(r"file.txt", "r+b")
>>> f.name, f.mode, f.closed
('file.txt', 'rb+', False)
>>> f.close()
>>> f.closed
True
```

- `encoding` — название кодировки, которая будет использоваться для преобразования строк перед записью в файл или при чтении. Атрибут доступен только в текстовом режиме. Обратите также внимание на то, что изменить значение атрибута нельзя, поскольку он доступен только для чтения:

```
>>> f = open(r"file.txt", "a", encoding="cp1251")
>>> f.encoding
'cp1251'
>>> f.close()
```

  Стандартный вывод `stdout` также является файловым объектом. Атрибут `encoding` этого объекта всегда содержит кодировку устройства вывода, поэтому строка преобразуется в последовательность байтов в правильной кодировке. Например, при запуске с помощью двойного щелчка на значке файла атрибут `encoding` будет иметь значение `"cp866"`, а при запуске в окне **Python Shell** редактора IDLE — значение `"cp1251"`:

```
>>> import sys
>>> sys.stdout.encoding
'cp1251'
```

- `buffer` — позволяет получить доступ к буферу. Атрибут доступен только в текстовом режиме. С помощью этого объекта можно записать последовательность байтов в текстовый поток:

```
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.buffer.write(bytes("Строка", "cp1251"))
6
>>> f.close()
```

16.3. Доступ к файлам с помощью модуля *os*

Модуль `os` содержит дополнительные низкоуровневые функции, позволяющие работать с файлами. Функциональность этого модуля зависит от используемой операционной системы. Получить название используемой версии модуля можно с помощью атрибута `name`. В любой из поддерживаемых Python версий операционной системы Windows этот атрибут возвращает значение `"nt"`:

```
>>> import os
>>> os.name                    # Значение в ОС Windows 8
'nt'
```

Для доступа к файлам предназначены следующие функции из модуля `os`:

- `open(<Путь к файлу>, <Режим>[, mode=0o777])` — открывает файл и возвращает целочисленный дескриптор, с помощью которого производится дальнейшая работа с файлом. Если файл открыть не удалось, возбуждается исключение `OSError` или одно из исключений, являющихся его подклассами (мы поговорим о них в конце этой главы). В параметре `<Режим>` в операционной системе Windows могут быть указаны следующие флаги (или их комбинация через символ `|`):
  - `os.O_RDONLY` — чтение;
  - `os.O_WRONLY` — запись;
  - `os.O_RDWR` — чтение и запись;
  - `os.O_APPEND` — добавление в конец файла;

- `os.O_CREAT` — создать файл, если он не существует и если не указан флаг `os.O_EXCL`;
- `os.O_EXCL` — при использовании совместно с `os.O_CREAT` указывает, что создаваемый файл изначально не должен существовать, в противном случае будет сгенерировано исключение `FileExistsError`;
- `os.O_TEMPORARY` — при использовании совместно с `os.O_CREAT` указывает, что создается временный файл, который будет автоматически удален сразу после закрытия;
- `os.O_SHORT_LIVED` — то же самое, что `os.O_TEMPORARY`, но созданный файл по возможности будет храниться лишь в оперативной памяти, а не на диске;
- `os.O_TRUNC` — очистить содержимое файла;
- `os.O_BINARY` — файл будет открыт в бинарном режиме;
- `os.O_TEXT` — файл будет открыт в текстовом режиме (в Windows файлы открываются в текстовом режиме по умолчанию).

Рассмотрим несколько примеров:

- откроем файл на запись и запишем в него одну строку. Если файл не существует, создадим его. Если файл существует, очистим его:

```
>>> import os                   # Подключаем модуль
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_TRUNC
>>> f = os.open(r"file.txt", mode)
>>> os.write(f, b"String1\n") # Записываем данные
8
>>> os.close(f)                # Закрываем файл
```

- добавим еще одну строку в конец файла:

```
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_APPEND
>>> f = os.open(r"file.txt", mode)
>>> os.write(f, b"String2\n") # Записываем данные
8
>>> os.close(f)                # Закрываем файл
```

- прочитаем содержимое файла в текстовом режиме:

```
>>> f = os.open(r"file.txt", os.O_RDONLY)
>>> os.read(f, 50)             # Читаем 50 байтов
b'String1\nString2\n'
>>> os.close(f)                # Закрываем файл
```

- теперь прочитаем содержимое файла в бинарном режиме:

```
>>> f = os.open(r"file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.read(f, 50)             # Читаем 50 байтов
b'String1\r\nString2\r\n'
>>> os.close(f)                # Закрываем файл
```

♦ `read(<Дескриптор>, <Количество байтов>)` — читает из файла указанное количество байтов. При достижении конца файла возвращается пустая строка:

```
>>> f = os.open(r"file.txt", os.O_RDONLY)
>>> os.read(f, 5), os.read(f, 5), os.read(f, 5), os.read(f, 5)
(b'Strin', b'g1\nS', b'tring', b'2\n')
```

```
>>> os.read(f, 5)              # Достигнут конец файла
b''
>>> os.close(f)                # Закрываем файл
```

- `write(<Дескриптор>, <Последовательность байтов>)` — записывает последовательность байтов в файл. Возвращает количество записанных байтов;
- `close(<Дескриптор>)` — закрывает файл;
- `lseek(<Дескриптор>, <Смещение>, <Позиция>)` — устанавливает указатель в позицию, имеющую заданное `<Смещение>` относительно параметра `<Позиция>`. Возвращает новую позицию указателя. В качестве параметра `<Позиция>` могут быть указаны следующие атрибуты или соответствующие им значения:
  - `os.SEEK_SET` или `0` — начало файла;
  - `os.SEEK_CUR` или `1` — текущая позиция указателя;
  - `os.SEEK_END` или `2` — конец файла.

  Пример:

```
>>> f = os.open(r"file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.lseek(f, 0, os.SEEK_END) # Перемещение в конец файла
18
>>> os.lseek(f, 0, os.SEEK_SET) # Перемещение в начало файла
0
>>> os.lseek(f, 9, os.SEEK_CUR) # Относительно указателя
9
>>> os.lseek(f, 0, os.SEEK_CUR) # Текущее положение указателя
9
>>> os.close(f)                 # Закрываем файл
```

- `dup(<Дескриптор>)` — возвращает дубликат файлового дескриптора;
- `fdopen(<Дескриптор>[, <Режим>[, <Размер буфера>]])` — возвращает файловый объект по указанному дескриптору. Параметры `<Режим>` и `<Размер буфера>` имеют тот же смысл, что и в функции `open()`:

```
>>> fd = os.open(r"file.txt", os.O_RDONLY)
>>> fd
3
>>> f = os.fdopen(fd, "r")
>>> f.fileno() # Объект имеет тот же дескриптор
3
>>> f.read()
'String1\nString2\n'
>>> f.close()
```

16.4. Классы *StringIO* и *BytesIO*

Класс `StringIO` из модуля `io` позволяет работать со строкой как с файловым объектом. Все операции с этим файловым объектом (будем называть его далее «файл») производятся в оперативной памяти. Формат конструктора класса:

```
StringIO([<Начальное значение>][, newline=None])
```

Если первый параметр не указан, то начальным значением будет пустая строка. После создания объекта указатель текущей позиции устанавливается на начало «файла». Объект, возвращаемый конструктором класса, имеет следующие методы:

- `close()` — закрывает «файл». Проверить, открыт «файл» или закрыт, позволяет атрибут `closed`. Атрибут возвращает `True`, если «файл» был закрыт, и `False` — в противном случае;
- `getvalue()` — возвращает содержимое «файла» в виде строки:

```
>>> import io               # Подключаем модуль
>>> f = io.StringIO("String1\n")
>>> f.getvalue()            # Получаем содержимое «файла»
'String1\n'
>>> f.close()               # Закрываем «файл»
```

- `tell()` — возвращает текущую позицию указателя относительно начала «файла»;
- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель в позицию, имеющую заданное `<Смещение>` относительно параметра `<Позиция>`. В качестве параметра `<Позиция>` могут быть указаны следующие значения:
  - `0` — начало «файла» (значение по умолчанию);
  - `1` — текущая позиция указателя;
  - `2` — конец «файла».

  Пример использования методов `seek()` и `tell()`:

```
>>> f = io.StringIO("String1\n")
>>> f.tell()         # Позиция указателя
0
>>> f.seek(0, 2)     # Перемещаем указатель в конец «файла»
8
>>> f.tell()         # Позиция указателя
8
>>> f.seek(0)        # Перемещаем указатель в начало «файла»
0
>>> f.tell()         # Позиция указателя
0
>>> f.close()        # Закрываем файл
```

- `write(<Строка>)` — записывает строку в «файл»:

```
>>> f = io.StringIO("String1\n")
>>> f.seek(0, 2)            # Перемещаем указатель в конец «файла»
8
>>> f.write("String2\n") # Записываем строку в «файл»
8
>>> f.getvalue()            # Получаем содержимое «файла»
'String1\nString2\n'
>>> f.close()               # Закрываем «файл»
```

- `writelines(<Последовательность>)` — записывает последовательность в «файл»:

```
>>> f = io.StringIO()
>>> f.writelines(["String1\n", "String2\n"])
```

```
>>> f.getvalue()          # Получаем содержимое «файла»
'String1\nString2\n'
>>> f.close()             # Закрываем «файл»
```

♦ `read([<Количество символов>])` — считывает данные из «файла». Если параметр не указан, возвращается содержимое «файла» от текущей позиции указателя до конца «файла». Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество символов. Когда достигается конец «файла», метод возвращает пустую строку:

```
>>> f = io.StringIO("String1\nString2\n")
>>> f.read()
'String1\nString2\n'
>>> f.seek(0) # Перемещаем указатель в начало «файла»
0
>>> f.read(5), f.read(5), f.read(5), f.read(5), f.read(5)
('Strin', 'g1\nSt', 'ring2', '\n', '')
>>> f.close() # Закрываем «файл»
```

♦ `readline([<Количество символов>])` — считывает из «файла» одну строку при каждом вызове. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода строки, таковой добавлен не будет. При достижении конца «файла» возвращается пустая строка:

```
>>> f = io.StringIO("String1\nString2")
>>> f.readline(), f.readline(), f.readline()
('String1\n', 'String2', '')
>>> f.close() # Закрываем «файл»
```

Если в необязательном параметре указано число, считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца «файла» или из «файла» не будет прочитано указанное количество символов. Иными словами, если количество символов в строке меньше значения параметра, будет считана одна строка, а не указанное количество символов. Если количество символов в строке больше, возвращается указанное количество символов:

```
>>> f = io.StringIO("String1\nString2\nString3\n")
>>> f.readline(5), f.readline(5)
('Strin', 'g1\n')
>>> f.readline(100) # Возвращается одна строка, а не 100 символов
'String2\n'
>>> f.close()       # Закрываем «файл»
```

♦ `readlines([<Примерное количество символов>])` — считывает все содержимое «файла» в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода строки, таковой добавлен не будет:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.readlines()
['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем «файл»
```

Если в необязательном параметре указано число, считывается указанное количество символов плюс фрагмент до символа конца строки `\n`. Затем эта строка разбивается и добавляется построчно в список:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.readlines(14)
['String1\n', 'String2\n']
>>> f.seek(0) # Перемещаем указатель в начало «файла»
0
>>> f.readlines(17)
['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем «файл»
```

- `__next__()` — считывает одну строку при каждом вызове. При достижении конца «файла» возбуждается исключение `StopIteration`:

```
>>> f = io.StringIO("String1\nString2")
>>> f.__next__(), f.__next__()
('String1\n', 'String2')
>>> f.__next__()
... Фрагмент опущен ...
StopIteration
>>> f.close() # Закрываем «файл»
```

  Благодаря методу `__next__()` мы можем перебирать файл построчно с помощью цикла `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `__next__()`:

```
>>> f = io.StringIO("String1\nString2")
>>> for line in f: print(line.rstrip())

String1
String2
>>> f.close() # Закрываем «файл»
```

- `flush()` — сбрасывает данные из буфера в «файл»;
- `truncate([<Количество символов>])` — обрезает «файл» до указанного количества символов:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.truncate(15)      # Обрезаем «файл»
15
>>> f.getvalue()        # Получаем содержимое «файла»
'String1\nString2'
>>> f.close()           # Закрываем «файл»
```

  Если параметр не указан, то «файл» обрезается до текущей позиции указателя:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.seek(15)          # Перемещаем указатель
15
>>> f.truncate()        # Обрезаем «файл» до указателя
15
>>> f.getvalue()        # Получаем содержимое «файла»
'String1\nString2'
>>> f.close()           # Закрываем «файл»
```

Описанные ранее методы `writable()` и `seekable()`, вызванные у объекта класса `StringIO`, всегда возвращают `True`.

Класс `StringIO` работает только со строками. Чтобы выполнять аналогичные операции с «файлами», представляющими собой последовательности байтов, следует использовать класс `BytesIO` из модуля `io`. Формат конструктора класса:

```
BytesIO([<Начальное значение>])
```

Класс `BytesIO` поддерживает такие же методы, что и класс `StringIO`, но в качестве значений методы принимают и возвращают последовательности байтов, а не строки. Рассмотрим основные операции на примере:

```
>>> import io              # Подключаем модуль
>>> f = io.BytesIO(b"String1\n")
>>> f.seek(0, 2)           # Перемещаем указатель в конец файла
8
>>> f.write(b"String2\n") # Пишем в файл
8
>>> f.getvalue()           # Получаем содержимое файла
b'String1\nString2\n'
>>> f.seek(0)              # Перемещаем указатель в начало файла
0
>>> f.read()               # Считываем данные
b'String1\nString2\n'
>>> f.close()              # Закрываем файл
```

Класс `BytesIO` поддерживает также метод `getbuffer()`, который возвращает ссылку на объект `memoryview`. С помощью этого объекта можно получать и изменять данные по индексу или срезу, преобразовывать данные в список целых чисел (с помощью метода `tolist()`) или в последовательность байтов (с помощью метода `tobytes()`):

```
>>> f = io.BytesIO(b"Python")
>>> buf = f.getbuffer()
>>> buf[0]                 # Получаем значение по индексу
b'P'
>>> buf[0] = b"J"          # Изменяем значение по индексу
>>> f.getvalue()           # Получаем содержимое
b'Jython'
>>> buf.tolist()           # Преобразуем в список чисел
[74, 121, 116, 104, 111, 110]
>>> buf.tobytes()          # Преобразуем в тип bytes
b'Jython'
>>> f.close()              # Закрываем файл
```

16.5. Права доступа к файлам и каталогам

В операционных системах семейства UNIX каждому объекту (файлу или каталогу) назначаются права доступа, предоставляемые той или иной разновидности пользователей: владельцу, группе и прочим. Могут быть назначены следующие права доступа:

- чтение;
- запись;
- выполнение.

Права доступа обозначаются буквами:

- `r` — файл можно читать, а содержимое каталога можно просматривать;
- `w` — файл можно модифицировать, удалять и переименовывать, а в каталоге можно создавать или удалять файлы. Каталог можно переименовать или удалить;
- `x` — файл можно выполнять, а в каталоге можно выполнять операции над файлами, в том числе производить в нем поиск файлов.

Права доступа к файлу определяются записью типа:

```
-rw-r--r--
```

Первый символ (`-`) означает, что это файл, и не задает никаких прав доступа. Далее три символа (`rw-`) задают права доступа для владельца: чтение и запись, символ (`-`) здесь означает, что права на выполнение нет. Следующие три символа задают права доступа для группы (`r--`) — здесь только чтение. Ну и последние три символа (`r--`) задают права для всех остальных пользователей — также только чтение.

Права доступа к каталогу определяются такой строкой:

```
drwxr-xr-x
```

Первая буква (`d`) означает, что это каталог. Владелец может выполнять в каталоге любые действия (`rwx`), а группа и все остальные пользователи — только читать и выполнять поиск (`r-x`). Для того чтобы каталог можно было просматривать, должны быть установлены права на выполнение (`x`).

Права доступа могут обозначаться и числом. Такие числа называются *маской прав доступа*. Число состоит из трех цифр: от 0 до 7. Первая цифра задает права для владельца, вторая — для группы, а третья — для всех остальных пользователей. Например, права доступа `-rw-r--r--` соответствуют числу 644. Сопоставим числам, входящим в маску прав доступа, двоичную и буквенную записи (табл. 16.1).

Таблица 16.1. *Права доступа в разных записях*

Восьмеричная цифра	Двоичная запись	Буквенная запись	Восьмеричная цифра	Двоичная запись	Буквенная запись
0	000	`---`	4	100	`r--`
1	001	`--x`	5	101	`r-x`
2	010	`-w-`	6	110	`rw-`
3	011	`-wx`	7	111	`rwx`

Теперь понятно, что, согласно данным этой таблицы, права доступа `rw-r--r--` можно записать так: 110 100 100, что и переводится в число 644. Таким образом, если право предоставлено, то в соответствующей позиции стоит 1, а если нет — то 0.

Для определения прав доступа к файлу или каталогу предназначена функция `access()` из модуля `os`. Функция имеет следующий формат:

```
access(<Путь>, <Режим>)
```

Функция возвращает `True`, если проверка прошла успешно, или `False` — в противном случае. В параметре `<Режим>` могут быть указаны следующие константы, определяющие тип проверки:

- `os.F_OK` — проверка наличия пути или файла:

```
>>> import os                      # Подключаем модуль os
>>> os.access(r"file.txt", os.F_OK) # Файл существует
True
>>> os.access(r"C:\book", os.F_OK)  # Каталог существует
True
>>> os.access(r"C:\book2", os.F_OK) # Каталог не существует
False
```

- `os.R_OK` — проверка на возможность чтения файла или каталога;
- `os.W_OK` — проверка на возможность записи в файл или каталог;
- `os.X_OK` — определение, является ли файл или каталог выполняемым.

Чтобы изменить права доступа из программы, необходимо воспользоваться функцией `chmod()` из модуля `os`. Функция имеет следующий формат:

```
chmod(<Путь>, <Права доступа>)
```

Права доступа задаются в виде числа, перед которым следует указать комбинацию символов `0o` (это соответствует восьмеричной записи числа):

```
>>> os.chmod(r"file.txt", 0o777) # Полный доступ к файлу
```

Вместо числа можно указать комбинацию констант из модуля `stat`. За дополнительной информацией обращайтесь к документации по модулю.

16.6. Функции для манипулирования файлами

Для копирования и перемещения файлов предназначены следующие функции из модуля `shutil`:

- `copyfile(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать содержимое файла в другой файл. Никакие метаданные (например, права доступа) не копируются. Если файл существует, он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса. В качестве результата возвращается путь файла, куда были скопированы данные:

```
>>> import shutil      # Подключаем модуль
>>> shutil.copyfile(r"file.txt", r"file2.txt")
>>> # Путь не существует:
>>> shutil.copyfile(r"file.txt", r"C:\book2\file2.txt")
... Фрагмент опущен ...
FileNotFoundError: [Errno 2] No such file or directory:
'C:\\book2\\file2.txt'
```

  Исключение `FileNotFoundError` является подклассом класса `OSError` и возбуждается, если указанный файл не найден. Более подробно классы исключений, возбуждаемых при файловых операциях, мы рассмотрим в конце этой главы;

- `copy(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать файл вместе с правами доступа. Если файл существует, он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса. В качестве результата возвращает путь скопированного файла:

```
>>> shutil.copy(r"file.txt", r"file3.txt")
```

♦ `copy2(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать файл вместе с метаданными. Если файл существует, он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса. В качестве результата возвращает путь скопированного файла:

```
>>> shutil.copy2(r"file.txt", r"file4.txt")
```

♦ `move(<Путь к файлу>, <Куда перемещаем>)` — перемещает файл в указанное место с удалением исходного файла. Если файл существует, он будет перезаписан. Если файл не удалось переместить, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса. В качестве результата возвращает путь перемещенного файла.

Пример перемещения файла file4.txt в каталог C:\book\test:

```
>>> shutil.move(r"file4.txt", r"C:\book\test")
```

Для переименования и удаления файлов предназначены следующие функции из модуля `os`:

♦ `rename(<Старое имя>, <Новое имя>)` — переименовывает файл. Если файл не удалось переименовать, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса.

Пример переименования файла с обработкой исключений:

```
import os # Подключаем модуль
try:
    os.rename(r"file3.txt", "file4.txt")
except OSError:
    print("Файл не удалось переименовать")
else:
    print("Файл успешно переименован")
```

♦ `remove(<Путь к файлу>)` и `unlink(<Путь к файлу>)` — позволяют удалить файл. Если файл не удалось удалить, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса:

```
>>> os.remove(r"file2.txt")
>>> os.unlink(r"file4.txt")
```

Модуль `os.path` содержит дополнительные функции, позволяющие проверить наличие файла, получить размер файла и др. Опишем эти функции:

♦ `exists(<Путь или дескриптор>)` — проверяет указанный путь на существование. В качестве параметра можно передать путь к файлу или целочисленный дескриптор открытого файла, возвращенный функцией `open()` из того же модуля `os`. Возвращает `True`, если путь существует, и `False` — в противном случае:

```
>>> import os.path
>>> os.path.exists(r"file.txt"), os.path.exists(r"file2.txt")
(True, False)
>>> os.path.exists(r"C:\book"), os.path.exists(r"C:\book2")
(True, False)
```

♦ `getsize(<Путь к файлу>)` — возвращает размер файла в байтах. Если файл не существует, возбуждается исключение `OSError`:

```
>>> os.path.getsize(r"file.txt")  # Файл существует
18
```

```
>>> os.path.getsize(r"file2.txt") # Файл не существует
... Фрагмент опущен ...
OSError: [Error 2] Не удается найти указанный файл: 'file2.txt'
```

♦ `getatime(<Путь к файлу>)` — возвращает время последнего доступа к файлу в виде количества секунд, прошедших с начала эпохи. Если файл не существует, возбуждается исключение `OSError`:

```
>>> import time    # Подключаем модуль time
>>> t = os.path.getatime(r"file.txt")
>>> t
1511773416.0529847
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'27.11.2017 12:03:36'
```

♦ `getctime(<Путь к файлу>)` — возвращает дату создания файла в виде количества секунд, прошедших с начала эпохи. Если файл не существует, возбуждается исключение `OSError`:

```
>>> t = os.path.getctime(r"file.txt")
>>> t
1511773416.0529847
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'27.11.2017 12:03:36'
```

♦ `getmtime(<Путь к файлу>)` — возвращает время последнего изменения файла в виде количества секунд, прошедших с начала эпохи. Если файл не существует, возбуждается исключение `OSError`:

```
>>> t = os.path.getmtime(r"file.txt")
>>> t
1511773609.980973
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'27.11.2017 12:06:49'
```

Получить размер файла и время создания, изменения и доступа к файлу, а также значения других метаданных, позволяет функция `stat()` из модуля `os`. В качестве значения функция возвращает объект `stat_result`, содержащий десять атрибутов: `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime` и `st_ctime`.

Пример использования функции `stat()`:

```
>>> import os, time
>>> s = os.stat(r"file.txt")
>>> s
os.stat_result(st_mode=33206, st_ino=5910974511035376, st_dev=2086732993,
st_nlink=1, st_uid=0, st_gid=0, st_size=15, st_atime=1511773416,
st_mtime=1511773609, st_ctime=1511773416)
>>> s.st_size      # Размер файла
15
>>> t = s.st_atime # Время последнего доступа к файлу
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'27.11.2017 12:03:36'
>>> t = s.st_ctime # Время создания файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'27.11.2017 12:03:36'
```

```
>>> t = s.st_mtime # Время последнего изменения файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'27.11.2017 12:06:49'
```

Обновить время последнего доступа и время изменения файла позволяет функция `utime()` из модуля `os`. Функция имеет два варианта формата:

```
utime(<Путь к файлу или его дескриптор>, None)
utime(<Путь к файлу или его дескриптор >, (<Последний доступ>, <Изменение файла>))
```

В качестве первого параметра можно указать как строковый путь, так и целочисленный дескриптор открытого файла, возвращенный функцией `open()` из модуля `os`. Если в качестве второго параметра указано значение `None`, то время доступа и изменения файла будет текущим. Во втором варианте формата функции `utime()` указывается кортеж из новых значений в виде количества секунд, прошедших с начала эпохи. Если файл не существует, возбуждается исключение `OSError`.

Пример использования функции `utime()`:

```
>>> import os, time
>>> os.stat(r"file.txt")          # Первоначальные значения
os.stat_result(st_mode=33206, st_ino=5910974511035376, st_dev=2086732993,
st_nlink=1, st_uid=0, st_gid=0, st_size=15, st_atime=1511773416,
st_mtime=1511773609, st_ctime=1511773416)
>>> t = time.time() - 600
>>> os.utime(r"file.txt", (t, t)) # Текущее время минус 600 сек
>>> os.stat(r"file.txt")
os.stat_result(st_mode=33206, st_ino=5910974511035376, st_dev=2086732993,
st_nlink=1, st_uid=0, st_gid=0, st_size=15, st_atime=1511790710,
st_mtime=1511790710, st_ctime=1511773416)
>>> os.utime(r"file.txt", None)   # Текущее время
>>> os.stat(r"file.txt")
os.stat_result(st_mode=33206, st_ino=5910974511035376, st_dev=2086732993,
st_nlink=1, st_uid=0, st_gid=0, st_size=15, st_atime=1511791343,
st_mtime=1511791343, st_ctime=1511773416)
```

16.7. Преобразование пути к файлу или каталогу

Преобразовать путь к файлу или каталогу позволяют следующие функции из модуля `os.path`:

- `abspath(<Относительный путь>)` — преобразует относительный путь в абсолютный, учитывая местоположение текущего рабочего каталога:

  ```
  >>> import os.path
  >>> os.path.abspath(r"file.txt")
  'C:\\book\\file.txt'
  >>> os.path.abspath(r"folder1/file.txt")
  'C:\\book\\folder1\\file.txt'
  >>> os.path.abspath(r"../file.txt")
  'C:\\file.txt'
  ```

  Как уже отмечалось ранее, в относительном пути можно указать как прямые, так и обратные слэши. Все они будут автоматически преобразованы с учетом значения атрибута

sep из модуля os.path. Значение этого атрибута зависит от используемой операционной системы. Выведем значение атрибута sep в операционной системе Windows:

```
>>> os.path.sep
'\\'
```

При указании пути в Windows следует учитывать, что слэш является специальным символом. По этой причине слэш необходимо удваивать (экранировать) или вместо обычных строк использовать неформатированные строки:

```
>>> "C:\\temp\\new\\file.txt"     # Правильно
'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt"       # Правильно
'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt"        # Неправильно!!!
'C:\temp\new\x0cile.txt'
```

Кроме того, если слэш расположен в конце строки, то его необходимо удваивать даже при использовании неформатированных строк:

```
>>> r"C:\temp\new\"               # Неправильно!!!
SyntaxError: EOL while scanning string literal
>>> r"C:\temp\new\\"
'C:\\temp\\new\\\\'
```

В первом случае последний слэш экранирует закрывающую кавычку, что приводит к синтаксической ошибке. Решить эту проблему можно, удвоив последний слэш. Однако посмотрите на результат — два слэша превратились в четыре. От одной проблемы ушли, а к другой пришли. Поэтому в данном случае лучше использовать обычные строки:

```
>>> "C:\\temp\\new\\"             # Правильно
'C:\\temp\\new\\'
>>> r"C:\temp\new\\"[:-1]         # Можно и удалить слэш
'C:\\temp\\new\\'
```

- isabs(<Путь>) — возвращает True, если путь является абсолютным, и False — в противном случае:

```
>>> os.path.isabs(r"C:\book\file.txt")
True
>>> os.path.isabs("file.txt")
False
```

- basename(<Путь>) — возвращает имя файла без пути к нему:

```
>>> os.path.basename(r"C:\book\folder1\file.txt")
'file.txt'
>>> os.path.basename(r"C:\book\folder")
'folder'
>>> os.path.basename("C:\\book\\folder\\")
''
```

- dirname(<Путь>) — возвращает путь к папке, где хранится файл:

```
>>> os.path.dirname(r"C:\book\folder\file.txt")
'C:\\book\\folder'
>>> os.path.dirname(r"C:\book\folder")
'C:\\book'
```

```
>>> os.path.dirname("C:\\book\\folder\\")
'C:\\book\\folder'
```

- `split(<Путь>)` — возвращает кортеж из двух элементов: пути к папке, где хранится файл, и имени файла:

```
>>> os.path.split(r"C:\book\folder\file.txt")
('C:\\book\\folder', 'file.txt')
>>> os.path.split(r"C:\book\folder")
('C:\\book', 'folder')
>>> os.path.split("C:\\book\\folder\\")
('C:\\book\\folder', '')
```

- `splitdrive(<Путь>)` — разделяет путь на имя диска и остальную часть пути. В качестве значения возвращается кортеж из двух элементов:

```
>>> os.path.splitdrive(r"C:\book\folder\file.txt")
('C:', '\\book\\folder\\file.txt')
```

- `splitext(<Путь>)` — возвращает кортеж из двух элементов: пути с именем файла, но без расширения, и расширения файла (фрагмент после последней точки):

```
>>> os.path.splitext(r"C:\book\folder\file.tar.gz")
('C:\\book\\folder\\file.tar', '.gz')
```

- `join(<Путь1>[, ..., <ПутьN>])` — соединяет указанные элементы пути, при необходимости вставляя между ними разделители:

```
>>> os.path.join("C:\\", "book\\folder", "file.txt")
'C:\\book\\folder\\file.txt'
>>> os.path.join(r"C:\\", "book/folder/", "file.txt")
'C:\\\\book/folder/file.txt'
```

Обратите внимание на последний пример — в пути используются разные слэши, и в результате получен некорректный путь. Чтобы этот путь сделать корректным, необходимо воспользоваться функцией `normpath()` из того же модуля `os.path`:

```
>>> p = os.path.join(r"C:\\", "book/folder/", "file.txt")
>>> os.path.normpath(p)
'C:\\book\\folder\\file.txt'
```

16.8. Перенаправление ввода/вывода

При рассмотрении методов для работы с файлами говорилось, что значение, возвращаемое методом `fileno()`, всегда будет больше числа 2, т. к. число 0 закреплено за стандартным вводом `stdin`, 1 — за стандартным выводом `stdout`, а 2 — за стандартным выводом сообщений об ошибках `stderr`. Все эти потоки имеют некоторое сходство с файловыми объектами. Например, потоки `stdout` и `stderr` поддерживают метод `write()`, предназначенный для вывода сообщений, а поток `stdin` — метод `readline()`, служащий для получения вводимых пользователем данных. Если этим потокам присвоить ссылку на объект, поддерживающий файловые методы, то можно перенаправить стандартные потоки в соответствующий файл. Для примера так и сделаем:

```
>>> import sys                    # Подключаем модуль sys
>>> tmp_out = sys.stdout          # Сохраняем ссылку на sys.stdout
```

```
>>> f = open(r"file.txt", "a")  # Открываем файл на дозапись
>>> sys.stdout = f              # Перенаправляем вывод в файл
>>> print("Пишем строку в файл")
>>> sys.stdout = tmp_out        # Восстанавливаем стандартный вывод
>>> print("Пишем строку в стандартный вывод")
Пишем строку в стандартный вывод
>>> f.close()                   # Закрываем файл
```

В этом примере мы вначале сохранили ссылку на стандартный вывод в переменной `tmp_out`. С помощью этой переменной можно в дальнейшем восстановить вывод в стандартный поток.

Функция `print()` напрямую поддерживает перенаправление вывода. Для этого используется параметр `file`, который по умолчанию ссылается на стандартный поток вывода. Например, записать строку в файл можно так:

```
>>> f = open(r"file.txt", "a")
>>> print("Пишем строку в файл", file=f)
>>> f.close()
```

Параметр `flush` позволяет указать, когда следует выполнять непосредственное сохранение данных из промежуточного буфера в файле. Если его значение равно `False` (это, кстати, значение по умолчанию), сохранение будет выполнено лишь после закрытия файла или после вызова метода `flush()`. Чтобы указать интерпретатору Python выполнять сохранение после каждого вызова функции `print()`, следует присвоить этому параметру значение `True`:

```
>>> f = open(r"file.txt", "a")
>>> print("Пишем строку в файл", file = f, flush = True)
>>> print("Пишем другую строку в файл", file = f, flush = True)
>>> f.close()
```

Стандартный ввод `stdin` также можно перенаправить. В этом случае функция `input()` будет читать одну строку из файла при каждом вызове. При достижении конца файла возбуждается исключение `EOFError`. Для примера выведем содержимое файла с помощью перенаправления потока ввода (листинг 16.4).

Листинг 16.4. Перенаправление потока ввода

```
# -*- coding: utf-8 -*-
import sys
tmp_in = sys.stdin             # Сохраняем ссылку на sys.stdin
f = open(r"file.txt", "r")     # Открываем файл на чтение
sys.stdin = f                  # Перенаправляем ввод
while True:
    try:
        line = input()         # Считываем строку из файла
        print(line)            # Выводим строку
    except EOFError:           # Если достигнут конец файла,
        break                  # выходим из цикла
sys.stdin = tmp_in             # Восстанавливаем стандартный ввод
f.close()                      # Закрываем файл
input()
```

Если необходимо узнать, ссылается ли стандартный ввод на терминал или нет, можно воспользоваться методом `isatty()`. Метод возвращает `True`, если объект ссылается на терминал, и `False` — в противном случае:

```
>>> tmp_in = sys.stdin          # Сохраняем ссылку на sys.stdin
>>> f = open(r"file.txt", "r")
>>> sys.stdin = f               # Перенаправляем ввод
>>> sys.stdin.isatty()          # Не ссылается на терминал
False
>>> sys.stdin = tmp_in          # Восстанавливаем стандартный ввод
>>> sys.stdin.isatty()          # Ссылается на терминал
True
>>> f.close()                   # Закрываем файл
```

Перенаправить стандартный ввод/вывод можно также с помощью командной строки. Для примера создадим в папке C:\book файл test3.py с кодом, приведенным в листинге 16.5.

Листинг 16.5. Содержимое файла test3.py

```
# -*- coding: utf-8 -*-
while True:
    try:
        line = input()
        print(line)
    except EOFError:
        break
```

Запускаем командную строку и переходим в папку со скриптом, выполнив команду: `cd C:\book`. Теперь выведем содержимое созданного ранее текстового файла file.txt (его содержимое может быть любым), выполнив команду:

```
C:\Python36\python.exe test3.py < file.txt
```

Перенаправить стандартный вывод в файл можно аналогичным образом. Только в этом случае символ < необходимо заменить символом >. Для примера создадим в папке C:\book файл test4.py с кодом из листинга 16.6.

Листинг 16.6. Содержимое файла test4.py

```
# -*- coding: utf-8 -*-
print("String")             # Эта строка будет записана в файл
```

Теперь перенаправим вывод в файл file.txt, выполнив команду:

```
C:\Python36\python.exe test4.py > file.txt
```

В этом режиме файл file.txt будет перезаписан. Если необходимо добавить результат в конец файла, следует использовать символы >>. Вот пример дозаписи в файл:

```
C:\Python36\python.exe test4.py >> file.txt
```

С помощью стандартного вывода `stdout` можно создать индикатор выполнения процесса непосредственно в окне консоли. Чтобы реализовать такой индикатор, нужно вспомнить, что

символ перевода строки в Windows состоит из двух символов: `\r` (перевод каретки) и `\n` (перевод строки). Таким образом, используя только символ перевода каретки `\r`, можно перемещаться в начало строки и перезаписывать ранее выведенную информацию. Рассмотрим вывод индикатора процесса на примере (листинг 16.7).

Листинг 16.7. Индикатор выполнения процесса

```
# -*- coding: utf-8 -*-
import sys, time
for i in range(5, 101, 5):
    sys.stdout.write("\r ... %s%%" % i) # Обновляем индикатор
    sys.stdout.flush()                  # Сбрасываем содержимое буфера
    time.sleep(1)                       # Засыпаем на 1 секунду
sys.stdout.write("\rПроцесс завершен\n")
input()
```

Сохраним код в файл и запустим его с помощью двойного щелчка. В окне консоли записи будут заменять друг друга на одной строке каждую секунду. Так как данные перед выводом будут помещаться в буфер, мы сбрасываем их на диск явным образом с помощью метода `flush()`.

16.9. Сохранение объектов в файл

Сохранить объекты в файл и в дальнейшем восстановить объекты из файла позволяют модули `pickle` и `shelve`. Модуль `pickle` предоставляет следующие функции:

- `dump(<Объект>, <Файл>[, <Протокол>][, fix_imports=True])` — производит сериализацию объекта и записывает данные в указанный файл. В параметре `<Файл>` указывается файловый объект, открытый на запись в бинарном режиме.

  Пример сохранения объекта в файл:

  ```
  >>> import pickle
  >>> f = open(r"file.txt", "wb")
  >>> obj = ["Строка", (2, 3)]
  >>> pickle.dump(obj, f)
  >>> f.close()
  ```

- `load()` — читает данные из файла и преобразует их в объект. Формат функции:

  ```
  load(<Файл>[, fix_imports=True][, encoding="ASCII"]
             [, errors="strict"])
  ```

  В параметре `<Файл>` указывается файловый объект, открытый на чтение в бинарном режиме.

  Пример восстановления объекта из файла:

  ```
  >>> f = open(r"file.txt", "rb")
  >>> obj = pickle.load(f)
  >>> obj
  ['Строка', (2, 3)]
  >>> f.close()
  ```

В один файл можно сохранить сразу несколько объектов, последовательно вызывая функцию `dump()`. Вот пример сохранения нескольких объектов:

```
>>> obj1 = ["Строка", (2, 3)]
>>> obj2 = (1, 2)
>>> f = open(r"file.txt", "wb")
>>> pickle.dump(obj1, f)          # Сохраняем первый объект
>>> pickle.dump(obj2, f)          # Сохраняем второй объект
>>> f.close()
```

Для восстановления объектов необходимо несколько раз вызвать функцию `load()`:

```
>>> f = open(r"file.txt", "rb")
>>> obj1 = pickle.load(f)        # Восстанавливаем первый объект
>>> obj2 = pickle.load(f)        # Восстанавливаем второй объект
>>> obj1, obj2
(['Строка', (2, 3)], (1, 2))
>>> f.close()
```

Сохранить объект в файл можно также с помощью метода `dump(<Объект>)` класса `Pickler`. Конструктор класса имеет следующий формат:

```
Pickler(<Файл>[, <Протокол>][, fix_imports=True])
```

Пример сохранения объекта в файл:

```
>>> f = open(r"file.txt", "wb")
>>> obj = ["Строка", (2, 3)]
>>> pkl = pickle.Pickler(f)
>>> pkl.dump(obj)
>>> f.close()
```

Восстановить объект из файла позволяет метод `load()` из класса `Unpickler`. Формат конструктора класса:

```
Unpickler(<Файл>[, fix_imports=True][, encoding="ASCII"]
                [, errors="strict"])
```

Пример восстановления объекта из файла:

```
>>> f = open(r"file.txt", "rb")
>>> obj = pickle.Unpickler(f).load()
>>> obj
['Строка', (2, 3)]
>>> f.close()
```

Модуль `pickle` позволяет также преобразовать объект в последовательность байтов и восстановить объект из таковой. Для этого предназначены две функции:

- `dumps(<Объект>[, <Протокол>][, fix_imports=True])` — производит сериализацию объекта и возвращает последовательность байтов специального формата. Формат зависит от указанного протокола — числа от `0` до значения `pickle.HIGHEST_PROTOCOL` в порядке от более старых к более новым и совершенным. По умолчанию в качестве номера протокола используется значение: `pickle.DEFAULT_PROTOCOL` (3).

  Пример преобразования списка и кортежа:

  ```
  >>> obj1 = [1, 2, 3, 4, 5]       # Список
  >>> obj2 = (6, 7, 8, 9, 10)      # Кортеж
  ```

```
>>> pickle.dumps(obj1)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
>>> pickle.dumps(obj2)
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

♦ `loads(<Последовательность байтов>[, fix_imports=True][, encoding="ASCII"][, errors="strict"])` — преобразует последовательность байтов специального формата в объект.

  Пример восстановления списка и кортежа:

```
>>> pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
[1, 2, 3, 4, 5]
>>> pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
(6, 7, 8, 9, 10)
```

Модуль `shelve` позволяет сохранять объекты под заданным строковым ключом и предоставляет интерфейс доступа, сходный со словарями, позволяя тем самым создать нечто, подобное базе данных. Для сериализации объекта используются возможности модуля `pickle`, а для записи получившейся строки по ключу в файл — модуль `dbm`. Все эти действия модуль `shelve` производит самостоятельно.

Открыть файл с набором объектов поможет функция `open()`. Функция имеет следующий формат:

```
open(<Путь к файлу>[, flag="c"][, protocol=None][, writeback=False])
```

В необязательном параметре `flag` можно указать один из режимов открытия файла:

♦ `r` — только чтение;

♦ `w` — чтение и запись;

♦ `c` — чтение и запись (значение по умолчанию). Если файл не существует, он будет создан;

♦ `n` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан.

Функция `open()` возвращает объект, с помощью которого производится дальнейшая работа с базой данных. Этот объект имеет следующие методы:

♦ `close()` — закрывает файл с базой данных. Для примера создадим файл и сохраним в нем список и кортеж:

```
>>> import shelve                     # Подключаем модуль
>>> db = shelve.open("db1")           # Открываем файл
>>> db["obj1"] = [1, 2, 3, 4, 5]      # Сохраняем список
>>> db["obj2"] = (6, 7, 8, 9, 10)     # Сохраняем кортеж
>>> db["obj1"], db["obj2"]            # Вывод значений
([1, 2, 3, 4, 5], (6, 7, 8, 9, 10))
>>> db.close()                        # Закрываем файл
```

♦ `keys()` — возвращает объект с ключами;

♦ `values()` — возвращает объект со значениями;

♦ `items()` — возвращает объект-итератор, который на каждой итерации генерирует кортеж, содержащий ключ и значение:

```
>>> db = shelve.open("db1")
>>> db.keys(), db.values()
(KeysView(<shelve.DbfilenameShelf object at 0x00FE81B0>),
 ValuesView(<shelve.DbfilenameShelf object at 0x00FE81B0>))
>>> list(db.keys()), list(db.values())
(['obj1', 'obj2'], [[1, 2, 3, 4, 5], (6, 7, 8, 9, 10)])
>>> db.items()
ItemsView(<shelve.DbfilenameShelf object at 0x00FE81B0>)
>>> list(db.items())
[('obj1', [1, 2, 3, 4, 5]), ('obj2', (6, 7, 8, 9, 10))]
>>> db.close()
```

- `get(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, возвращается значение `None` или значение, указанное во втором параметре;
- `setdefault(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует, метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, создается новый элемент со значением, указанным во втором параметре, и в качестве результата возвращается это значение. Если второй параметр не указан, значением нового элемента будет `None`;
- `pop(<Ключ>[, <Значение по умолчанию>])` — удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, возвращается значение из второго параметра. Если ключ отсутствует, и второй параметр не указан, возбуждается исключение `KeyError`;
- `popitem()` — удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если файл пустой, возбуждается исключение `KeyError`;
- `clear()` — удаляет все элементы. Метод ничего не возвращает в качестве значения;
- `update()` — добавляет элементы. Метод изменяет текущий объект и ничего не возвращает. Если элемент с указанным ключом уже присутствует, то его значение будет перезаписано. Форматы метода:

  ```
  update(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
  update(<Словарь>)
  update(<Список кортежей с двумя элементами>)
  update(<Список списков с двумя элементами>)
  ```

Помимо этих методов можно воспользоваться функцией `len()` для получения количества элементов и оператором `del` для удаления определенного элемента, а также операторами `in` и `not in` для проверки существования или несуществования ключа:

```
>>> db = shelve.open("db1")
>>> len(db)            # Количество элементов
2
>>> "obj1" in db
True
>>> del db["obj1"]     # Удаление элемента
>>> "obj1" in db
False
>>> "obj1" not in db
True
>>> db.close()
```

16.10. Функции для работы с каталогами

Для работы с каталогами используются следующие функции из модуля os:

- getcwd() — возвращает текущий рабочий каталог. От значения, возвращаемого этой функцией, зависит преобразование относительного пути в абсолютный. Кроме того, важно помнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог с исполняемым файлом:

```
>>> import os
>>> os.getcwd()              # Текущий рабочий каталог
'C:\\book'
```

- chdir(<Имя каталога>) — делает указанный каталог текущим:

```
>>> os.chdir("C:\\book\\folder1\\")
>>> os.getcwd()              # Текущий рабочий каталог
'C:\\book\\folder1'
```

- mkdir(<Имя каталога>[, <Права доступа>]) — создает новый каталог с правами доступа, указанными во втором параметре. Права доступа задаются восьмеричным числом (значение по умолчанию 0o777). Пример создания нового каталога в текущем рабочем каталоге:

```
>>> os.mkdir("newfolder")    # Создание каталога
```

- rmdir(<Имя каталога>) — удаляет пустой каталог. Если в каталоге есть файлы или указанный каталог не существует, возбуждается исключение — подкласс класса OSError. Удалим каталог newfolder:

```
>>> os.rmdir("newfolder") # Удаление каталога
```

- listdir(<Путь>) — возвращает список объектов в указанном каталоге:

```
>>> os.listdir("C:\\book\\folder1\\")
['file1.txt', 'file2.txt', 'file3.txt', 'folder1', 'folder2']
```

- walk() — позволяет обойти дерево каталогов. Формат функции:

```
walk(<Начальный каталог>[, topdown=True][, onerror=None]
     [, followlinks=False])
```

  В качестве значения функция walk() возвращает объект. На каждой итерации через этот объект доступен кортеж из трех элементов: текущего каталога, списка каталогов и списка файлов, находящихся в нем. Если произвести изменения в списке каталогов во время выполнения, это позволит изменить порядок обхода вложенных каталогов.

  Необязательный параметр topdown задает последовательность обхода каталогов. Если в качестве значения указано True (значение по умолчанию), последовательность обхода будет такой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\"): print(p)

C:\book\folder1\
C:\book\folder1\folder1_1
C:\book\folder1\folder1_1\folder1_1_1
C:\book\folder1\folder1_1\folder1_1_2
C:\book\folder1\folder1_2
```

Если в параметре `topdown` указано значение `False`, последовательность обхода будет другой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\", False):
        print(p)
C:\book\folder1\folder1_1\folder1_1_1
C:\book\folder1\folder1_1\folder1_1_2
C:\book\folder1\folder1_1
C:\book\folder1\folder1_2
C:\book\folder1\
```

Благодаря такой последовательности обхода каталогов можно удалить все вложенные файлы и каталоги. Это особенно важно при удалении каталога, т. к. функция `rmdir()` позволяет удалить только пустой каталог.

Пример очистки дерева каталогов:

```
import os
for (p, d, f) in os.walk("C:\\book\\folder1\\", False):
    for file_name in f: # Удаляем все файлы
        os.remove(os.path.join(p, file_name))
    for dir_name in d: # Удаляем все каталоги
        os.rmdir(os.path.join(p, dir_name))
```

> ***Внимание!***
>
> Очень осторожно используйте этот код. Если в качестве первого параметра в функции `walk()` указать корневой каталог диска, то все имеющиеся в нем файлы и каталоги будут удалены.

Удалить дерево каталогов позволяет также функция `rmtree()` из модуля `shutil`. Функция имеет следующий формат:

```
rmtree(<Путь>[, <Обработка ошибок>[, <Обработчик ошибок>]])
```

Если в параметре `<Обработка ошибок>` указано значение `True`, ошибки будут проигнорированы. Если указано значение `False` (значение по умолчанию), в третьем параметре можно задать ссылку на функцию, которая будет вызываться при возникновении исключения.

Пример удаления дерева каталогов вместе с начальным каталогом:

```
import shutil
shutil.rmtree("C:\\book\\folder1\\")
```

♦ `normcase(<Каталог>)` — преобразует заданный к каталогу путь к виду, подходящему для использования в текущей операционной системе. В Windows преобразует все прямые слэши в обратные. Также во всех системах приводит все буквы пути к нижнему регистру:

```
>>> from os.path import normcase
>>> normcase(r"c:/BoOk/fIlE.TxT")
'c:\\book\\file.txt'
```

Как вы уже знаете, функция `listdir()` возвращает список объектов в указанном каталоге. Проверить, на какой тип объекта ссылается элемент этого списка, можно с помощью следующих функций из модуля `os.path`:

♦ `isdir(<Объект>)` — возвращает `True`, если объект является каталогом, и `False` — в противном случае:

```
>>> import os.path
>>> os.path.isdir(r"C:\book\file.txt")
False
>>> os.path.isdir("C:\\book\\")
True
```

♦ `isfile(<Объект>)` — возвращает `True`, если объект является файлом, и `False` — в противном случае:

```
>>> os.path.isfile(r"C:\book\file.txt")
True
>>> os.path.isfile("C:\\book\\")
False
```

♦ `islink(<Объект>)` — возвращает `True`, если объект является символической ссылкой, и `False` — в противном случае. Если символические ссылки не поддерживаются, функция возвращает `False`.

Функция `listdir()` возвращает список всех объектов в указанном каталоге. Если необходимо ограничить список определенными критериями, следует воспользоваться функцией `glob(<Путь>)` из модуля `glob`. Функция `glob()` позволяет указать в пути следующие специальные символы:

♦ `?` — любой одиночный символ;

♦ `*` — любое количество символов;

♦ `[<Символы>]` — позволяет указать символы, которые должны быть на этом месте в пути. Можно задать символы или определить их диапазон через дефис.

В качестве значения функция возвращает список путей к объектам, совпадающим с шаблоном. Вот пример использования функции `glob()`:

```
>>> import os, glob
>>> os.listdir("C:\\book\\folder1\\")
['file.txt', 'file1.txt', 'file2.txt', 'folder1_1', 'folder1_2',
'index.html']
>>> glob.glob("C:\\book\\folder1\\*.txt")
['C:\\book\\folder1\\file.txt', 'C:\\book\\folder1\\file1.txt',
'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*.html") # Абсолютный путь
['C:\\book\\folder1\\index.html']
>>> glob.glob("folder1/*.html")            # Относительный путь
['folder1\\index.html']
>>> glob.glob("C:\\book\\folder1\\*[0-9].txt")
['C:\\book\\folder1\\file1.txt', 'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*\\*.html")
['C:\\book\\folder1\\folder1_1\\index.html',
'C:\\book\\folder1\\folder1_2\\test.html']
```

Обратите внимание на последний пример. Специальные символы могут быть указаны не только в названии файла, но и в именах каталогов в пути. Это позволяет просматривать сразу несколько каталогов в поисках объектов, соответствующих шаблону.

16.10.1. Функция *scandir()*

Начиная с Python 3.5, в модуле `os` появилась поддержка функции `scandir()` — более быстрого и развитого инструмента для просмотра содержимого каталогов. Формат функции:

```
os.scandir(<Путь>)
```

`<Путь>` можно указать как относительный, так и абсолютный. Если он не задан, будет использовано строковое значение `.` (точка), т. е. путь к текущему каталогу.

Функция `scandir()` возвращает итератор, на каждом проходе возвращающий очередной элемент — файл или каталог, что присутствует по указанному пути. Этот файл или каталог представляется экземпляром класса `DirEntry`, определенного в том же модуле `os`, который хранит всевозможные сведения о файле (каталоге).

Класс `DirEntry` поддерживает атрибуты:

- `name` — возвращает имя файла (каталога);
- `path` — возвращает путь к файлу (каталогу), составленный из пути, что был указан в вызове функции `scandir()`, и имени файла (каталога), хранящегося в свойстве `name`.

Для примера выведем список путей всех файлов и каталогов, находящихся в текущем каталоге (при вводе команд в **Python Shell** текущим станет каталог, где установлен Python):

```
>>> import os
>>> for entry in os.scandir():
        print(entry.name)

.\DLLs
.\Doc
.\include
# Часть вывода пропущена
.\python.exe
.\python3.dll
.\vcruntime140.dll
```

Видно, что путь, возвращаемый свойством `path`, составляется из пути, заданного в вызове функции `scandir()` (в нашем случае это используемый по умолчанию путь `.`), и имени файла (каталога). Теперь попробуем указать путь явно:

```
>>> for entry in os.scandir("c:\python36"):
        print(entry.path)

c:\python36\DLLs
c:\python36\Doc
c:\python36\include
# Часть вывода пропущена
c:\python36\python.exe
c:\python36\python3.dll
c:\python36\vcruntime140.dll
```

Помимо описанных ранее атрибутов, класс `DirEntry` поддерживает следующие методы:

- `is_file(follow_symlinks=True)` — возвращает `True`, если текущий элемент — файл, и `False` в противном случае. Если элемент представляет собой символическую ссылку, и для параметра `follow_symlinks` указано значение `True` (или если параметр вообще

опущен), проверяется элемент, на который указывает эта символическая ссылка. Если же для параметра `follow_symlinks` задано значение `False`, всегда возвращается `False`;

- `is_dir(follow_symlinks=True)` — возвращает `True`, если текущий элемент — каталог, и `False` в противном случае. Если элемент представляет собой символическую ссылку, и для параметра `follow_symlinks` указано значение `True` (или если параметр вообще опущен), проверяется элемент, на который указывает эта символическая ссылка. Если же для параметра `follow_symlinks` задано значение `False`, всегда возвращается `False`;
- `is_symlink()` — возвращает `True`, если текущий элемент — символическая ссылка, и `False` в противном случае;
- `stat(follow_symlinks=True)` — возвращает объект `stat_result`, хранящий сведения о файле (более подробно он был описан в *разд. 16.6*). Если элемент представляет собой символическую ссылку, и для параметра `follow_symlinks` указано значение `True` (или если параметр вообще опущен), возвращаются сведения об элементе, на который указывает эта символическая ссылка. Если же для параметра `follow_symlinks` задано значение `False`, возвращаются сведения о самой символической ссылке. В Windows атрибуты `st_ino`, `st_dev` и `st_nlink` объекта `stat_result`, возвращенного методом `stat()`, всегда хранят `0`, и для получения их значений следует воспользоваться функцией `stat()` из модуля `os`, описанной в *разд. 16.6*.

Рассмотрим пару примеров:

- для начала выведем список всех каталогов, что находятся в каталоге, где установлен Python, разделив их запятыми:

```
>>> for entry in os.scandir():
        if entry.is_dir():
            print(entry.name, end=", ")

DLLs, Doc, include, Lib, libs, Scripts, tcl, Tools,
```

- выведем список всех DLL-файлов, хранящихся в каталоге Windows, без обработки символических ссылок:

```
>>> for entry in os.scandir("c:\windows"):
        if entry.is_file(follow_symlinks=False) and entry.name.endswith(".dll"):
            print(entry.name, end=", ")
```

В Python 3.6 итератор, возвращаемый функцией `scandir()`, получил поддержку протокола менеджеров контекста (см. *разд. 16.2*). Так что мы можем выполнить просмотр содержимого какого-либо пути следующим способом:

```
>>> with os.scandir() as it:
        for entry in it:
            print(entry.name)
```

В том же Python 3.6 для Windows появилась возможность указывать путь в вызове функции `scandir()` в виде объекта `bytes`. Однако нужно иметь в виду, что в таком случае значения атрибутов `name` и `path` класса `DirEntry` также будут представлять собой объекты `bytes`, а не строки:

```
>>> with os.scandir(b"c:\python36") as it:
        for entry in it:
            print(entry.name)
```

```
b'DLLs'
b'Doc'
b'include'
# Часть вывода пропущена
b'python.exe'
b'python3.dll'
b'vcruntime140.dll'
```

16.11. Исключения, возбуждаемые файловыми операциями

В этой главе неоднократно говорилось, что функции и методы, осуществляющие файловые операции, при возникновении нештатных ситуаций возбуждают исключение класса `OSError` или одно из исключений, являющихся его подклассами. Настало время познакомиться с ними.

Исключений-подклассов класса `OSError` довольно много. Вот те из них, что затрагивают именно операции с файлами и каталогами:

- `BlockingIOError` — не удалось заблокировать объект (файл или поток ввода/вывода);
- `ConnectionError` — ошибка сетевого соединения. Может возникнуть при открытии файла по сети. Является базовым классом для ряда других исключений более высокого уровня, описанных в документации по Python;
- `FileExistsError` — файл или каталог с заданным именем уже существуют;
- `FileNotFoundError` — файл или каталог с заданным именем не обнаружены;
- `InterruptedError` — файловая операция неожиданно прервана по какой-либо причине;
- `IsADirectoryError` — вместо пути к файлу указан путь к каталогу;
- `NotADirectoryError` — вместо пути к каталогу указан путь к файлу;
- `PermissionError` — отсутствуют права на доступ к указанному файлу или каталогу;
- `TimeoutError` — истекло время, отведенное системой на выполнение операции.

Вот пример кода, обрабатывающего некоторые из указанных исключений:

```
. . .
try
    open("C:\temp\new\file.txt")
except FileNotFoundError:
    print("Файл отсутствует")
except IsADirectoryError:
    print("Это не файл, а каталог")
except PermissionError:
    print("Отсутствуют права на доступ к файлу")
except OSError:
    print("Неустановленная ошибка открытия файла")
. . .
```

ЧАСТЬ II

Библиотека PyQt 5

Глава 17. Знакомство с PyQt 5

Глава 18. Управление окном приложения

Глава 19. Обработка сигналов и событий

Глава 20. Размещение компонентов в окнах

Глава 21. Основные компоненты

Глава 22. Списки и таблицы

Глава 23. Работа с базами данных

Глава 24. Работа с графикой

Глава 25. Графическая сцена

Глава 26. Диалоговые окна

Глава 27. Создание SDI- и MDI-приложений

Глава 28. Мультимедиа

Глава 29. Печать документов

Глава 30. Взаимодействие с Windows

Глава 31. Сохранение настроек приложений

Глава 32. Приложение «Судоку»

ГЛАВА 17

Знакомство с PyQt 5

Итак, изучение основ языка Python закончено, и мы можем перейти к рассмотрению библиотеки PyQt, позволяющей разрабатывать приложения с графическим интерфейсом. Первые три главы второй части книги можно считать основными, поскольку в них описываются базовые возможности библиотеки и методы, которые наследуют все компоненты, так что материал этих глав нужно знать обязательно. Остальные главы содержат дополнительный справочный материал. В сопровождающий книгу электронный архив (см. *приложение*) включен файл PyQt.doc, который содержит более 750 дополнительных листингов, поясняющих материал второй части книги, — что позволило уменьшить ее объем, поскольку с этими листингами страниц книги было бы вдвое больше, чем всех страниц ее второй части.

17.1. Установка PyQt 5

Библиотека PyQt 5 не входит в комплект поставки Python, и прежде чем начать изучение ее основ, необходимо установить эту библиотеку на компьютер.

В настоящее время установка библиотеки PyQt 5 выполняется исключительно просто. Для этого достаточно запустить командную строку и отдать в ней команду:

```
pip3 install PyQt5
```

Утилита pip3, поставляемая в составе Python и предназначенная для установки дополнительных библиотек, самостоятельно загрузит последнюю версию PyQt 5 и установит ее по пути *<каталог, в котором установлен Python>*\lib\site-packages\PyQt5.

> ***Внимание!***
>
> При установке PyQt таким способом устанавливаются только компоненты библиотеки, необходимые для запуска программ. Средства разработчика (такие как программа Designer) и дополнительные компоненты, в частности клиентские части серверных СУБД, должны быть установлены отдельно.

Чтобы проверить правильность установки, выведем версии PyQt и Qt:

```
>>> from PyQt5 import QtCore
>>> QtCore.PYQT_VERSION_STR
'5.9.2'
>>> QtCore.QT_VERSION_STR
'5.9.3'
```

17.2. Первая программа

При изучении языков и технологий принято начинать с программы, выводящей надпись «Привет, мир!». Не станем нарушать традицию и напишем программу (листинг 17.1), создающую окно с приветствием и кнопкой для закрытия окна (рис. 17.1).

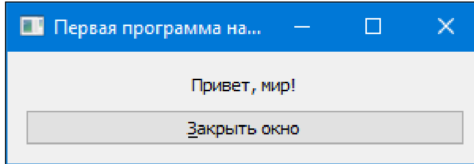

Рис. 17.1. Результат выполнения листинга 17.1

Листинг 17.1. Первая программа на PyQt

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Первая программа на PyQt")
window.resize(300, 70)
label = QtWidgets.QLabel("<center>Привет, мир!</center>")
btnQuit = QtWidgets.QPushButton("&Закрыть окно")
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
vbox.addWidget(btnQuit)
window.setLayout(vbox)
btnQuit.clicked.connect(app.quit)
window.show()
sys.exit(app.exec_())
```

Для создания файла с программой можно по-прежнему пользоваться редактором IDLE. Однако запуск оконного приложения из IDLE нажатием клавиши <F5> приводит к очень неприятным артефактам (в частности, при завершении программы ее главное окно остается на экране) и даже аварийному завершению работы редактора. Поэтому запускать оконные приложения следует двойным щелчком на значке файла.

До сих пор мы создавали файлы с расширением py и все результаты выполнения программы выводили в консоль. Оконное приложение также можно сохранить с расширением py, но тогда при его запуске, помимо основного окна, также будет выведено окно консоли, что, впрочем, на этапе разработки дает возможность вывести отладочную информацию (таким способом мы будем пользоваться в дальнейших примерах). Чтобы избавиться от окна консоли, следует сохранять файл с расширением pyw.

Попробуйте создать два файла с различными расширениями и запустить двойным щелчком каждый из них.

17.3. Структура PyQt-программы

Запускать программу мы научились. Теперь рассмотрим код из листинга 17.1 построчно.

В первой строке указывается кодировка файла. Поскольку в Python 3 по умолчанию для сохранения исходного кода используется кодировка UTF-8, эту строку можно и не указывать. Во второй строке импортируется модуль `QtWidgets` — он содержит классы, реализующие компоненты графического интерфейса: окна, надписи, кнопки, текстовые поля и др. В третьей строке импортируется модуль `sys`, из которого нам потребуется список параметров, переданных в командной строке (`argv`), а также функция `exit()`, позволяющая завершить выполнение программы.

Выражение:

```
app = QtWidgets.QApplication(sys.argv)
```

создает объект приложения в виде экземпляра класса `QApplication`. Конструктор этого класса принимает список параметров, переданных в командной строке. Следует помнить, что в программе всегда должен быть объект приложения, причем обязательно только один. Может показаться, что после создания объекта он в программе больше нигде не используется, однако надо понимать, что с его помощью осуществляется управление приложением незаметно для нас. Получить доступ к этому объекту из любого места в программе можно через атрибут `qApp` из модуля `QtWidgets`. Например, вывести список параметров, переданных в командной строке, можно так:

```
print(QtWidgets.qApp.argv())
```

Следующее выражение:

```
window = QtWidgets.QWidget()
```

создает объект окна в виде экземпляра класса `QWidget`. Этот класс наследуют практически все классы, реализующие компоненты графического интерфейса. И любой компонент, не имеющий родителя, обладает своим собственным окном.

Выражение:

```
window.setWindowTitle("Первая программа на PyQt")
```

задает текст, который будет выводиться в заголовке окна, для чего используется метод `setWindowTitle()`.

Очередное выражение:

```
window.resize(300, 70)
```

задает минимальные размеры окна. В первом параметре метода `resize()` указывается ширина окна, а во втором параметре — его высота. При этом надо учитывать, что метод `resize()` устанавливает размеры не самого окна, а его клиентской области, при этом размеры заголовка и ширина границ окна не учитываются. Также следует помнить, что эти размеры являются рекомендацией, — т. е., если компоненты не помещаются в окне, оно будет увеличено.

Выражение:

```
label = QtWidgets.QLabel("<center>Привет, мир!</center>")
```

создает объект надписи. Текст надписи задается в качестве параметра в конструкторе класса `QLabel`. Обратите внимание, что внутри строки мы указали HTML-теги, — а именно: с помощью тега `<center>` произвели выравнивание текста по центру компонента. Возможность

использования HTML-тегов и CSS-атрибутов является отличительной чертой библиотеки PyQt — например, внутри надписи можно вывести таблицу или отобразить изображение. Это очень удобно.

Следующее выражение:

```
btnQuit = QtWidgets.QPushButton("&Закрыть окно")
```

создает объект кнопки. Текст, который будет отображен на кнопке, задается в качестве параметра в конструкторе класса `QPushButton`. Обратите внимание на символ `&` перед буквой `З` — таким образом задаются клавиши быстрого доступа. Если нажать одновременно клавишу <Alt> и клавишу с буквой, перед которой в строке указан символ `&`, то кнопка сработает.

Выражение:

```
vbox = QtWidgets.QVBoxLayout()
```

создает вертикальный контейнер. Все компоненты, добавляемые в этот контейнер, будут располагаться по вертикали сверху вниз в порядке добавления, при этом размеры добавленных компонентов будут подогнаны под размеры контейнера. При изменении размеров контейнера будет произведено изменение размеров всех компонентов.

В следующих двух выражениях:

```
vbox.addWidget(label)
vbox.addWidget(btnQuit)
```

с помощью метода `addWidget()` производится добавление созданных ранее объектов надписи и кнопки в вертикальный контейнер. Так как объект надписи добавляется первым, он будет расположен над кнопкой. При добавлении компонентов в контейнер они автоматически становятся потомками контейнера.

Новое выражение:

```
window.setLayout(vbox)
```

добавляет контейнер в основное окно с помощью метода `setLayout()`. Таким образом, контейнер становится потомком основного окна.

Выражение:

```
btnQuit.clicked.connect(app.quit)
```

назначает обработчик сигнала `clicked()` кнопки, который генерируется при ее нажатии. Этот сигнал доступен через одноименный атрибут класса кнопки и поддерживает метод `connect()`, который и назначает для него обработчик, передаваемый первым параметром. Обработчик представляет собой метод `quit()` объекта приложения, выполняющий немедленное завершение его работы. Такой метод принято называть *слотом*.

> ***Пояснение***
>
> *Сигналом* в PyQt называется особое уведомление, генерируемое при наступлении какого-либо события в приложении: нажатия кнопки, ввода символа в текстовое поле, закрытия окна и пр.

Очередное выражение:

```
window.show()
```

выводит на экран окно и все компоненты, которые мы ранее в него добавили.

И, наконец, последнее выражение:

```
sys.exit(app.exec_())
```

запускает бесконечный цикл обработки событий в приложении.

Код, расположенный после вызова метода `exec_()`, будет выполнен только после завершения работы приложения, — поскольку результат выполнения метода `exec_()` мы передаем функции `exit()`, дальнейшее выполнение программы будет прекращено, а код возврата передан операционной системе.

17.4. ООП-стиль создания окна

Библиотека PyQt написана в объектно-ориентированном стиле (ООП-стиле) и содержит несколько сотен классов. Иерархия наследования всех классов имеет слишком большой размер, и приводить ее в книге возможности нет. Тем не менее, чтобы показать зависимости, при описании компонентов иерархия наследования конкретного класса будет показываться. В качестве примера выведем базовые классы класса `QWidget`:

```
>>> from PyQt5 import QtWidgets
>>> QtWidgets.QWidget.__bases__
(<class 'PyQt5.QtCore.QObject'>, <class 'PyQt5.QtGui.QPaintDevice'>)
```

Как видно из примера, класс `QWidget` наследует два класса: `QObject` и `QPaintDevice`. Класс `QObject` является классом верхнего уровня, и его в PyQt наследуют большинство классов. В свою очередь, класс `QWidget` является базовым классом для всех визуальных компонентов.

> ***Внимание!***
>
> В описании каждого класса PyQt приводятся лишь атрибуты, методы, сигналы и слоты, определенные непосредственно в описываемом классе. Атрибуты, методы, сигналы и слоты базовых классов там не описываются — присутствуют лишь ссылки на соответствующие страницы документации.

В своих программах вы можете наследовать стандартные классы и добавлять новую функциональность. В качестве примера переделаем соответствующим образом код из листинга 17.1 и создадим окно в ООП-стиле (листинг 17.2).

Листинг 17.2. ООП-стиль создания окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Привет, мир!")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.btnQuit = QtWidgets.QPushButton("&Закрыть окно")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnQuit)
        self.setLayout(self.vbox)
        self.btnQuit.clicked.connect(QtWidgets.qApp.quit)
```

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()   # Создаем экземпляр класса
    window.setWindowTitle("ООП-стиль создания окна")
    window.resize(300, 70)
    window.show()         # Отображаем окно
    sys.exit(app.exec_()) # Запускаем цикл обработки событий
```

В первых двух строках кода, как обычно, указывается кодировка файла и импортируются необходимые модули. На этот раз, помимо уже знакомого модуля `QtWidgets`, нам понадобится модуль `QtCore`, в котором объявлены атрибуты, задающие, в том числе, и режим выравнивания текста в объекте надписи.

Далее мы определяем класс `MyWindow`, который наследует класс `QWidget`:

```
class MyWindow(QtWidgets.QWidget):
```

Можно наследовать и другие классы, являющиеся наследниками `QWidget`, — например, `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При наследовании класса `QDialog` окно будет выравниваться по центру экрана (или по центру родительского окна) и иметь в заголовке окна только две кнопки: **Справка** и **Закрыть**. Кроме того, можно наследовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Наследование класса `QMainWindow` имеет свои отличия, которые мы рассмотрим в *главе 27*.

Выражение:

```
def __init__(self, parent=None):
```

определяет конструктор класса. В качестве параметров он принимает ссылки на экземпляр класса (`self`) и на родительский компонент (`parent`). Родительский компонент может отсутствовать, поэтому в определении конструктора параметру присваивается значение по умолчанию (`None`). Внутри метода `__init__()` вызывается конструктор базового класса, и ему передается ссылка на родительский компонент:

```
QtWidgets.QWidget.__init__(self, parent)
```

Следующие выражения внутри конструктора создают объекты надписи, кнопки и контейнера, затем добавляют компоненты в контейнер, а сам контейнер — в основное окно. Следует обратить внимание на то, что объекты надписи и кнопки сохраняются в атрибутах экземпляра класса. В дальнейшем из методов класса можно управлять этими объектами — например, изменять текст надписи. Если объекты не сохранить, то получить к ним доступ будет не так просто.

В предыдущем примере (см. листинг 17.1) мы выравнивали надпись с помощью HTML-тегов. Однако выравнивание можно задать и вызовом метода `setAlignment()`, которому следует передать атрибут `AlignHCenter` из модуля `QtCore`:

```
self.label.setAlignment(QtCore.Qt.AlignHCenter)
```

В выражении, назначающем обработчик сигнала:

```
self.btnQuit.clicked.connect(QtWidgets.qApp.quit)
```

мы получаем доступ к объекту приложения через рассмотренный ранее атрибут `qApp` модуля `QtWidgets`.

Создание объекта приложения и экземпляра класса `MyWindow` производится внутри условия:

```
if __name__ == "__main__":
```

Если вы внимательно читали первую часть книги, то уже знаете, что атрибут модуля `__name__` будет содержать значение `__main__` только в случае запуска модуля как главной программы. Если модуль импортировать, этот атрибут будет содержать другое значение. Поэтому весь последующий код создания объекта приложения и объекта окна выполняется только при запуске программы двойным щелчком на значке файла. Может возникнуть вопрос, зачем это нужно? Дело в том, что одним из преимуществ ООП-стиля программирования является повторное использование кода. Следовательно, можно импортировать модуль и использовать класс `MyWindow` в другом приложении.

Рассмотрим эту возможность на примере, для чего сохраним код из листинга 17.2 в файле с именем MyWindow.py, а затем создадим в той же папке еще один файл (например, с именем test.pyw) и вставим в него код из листинга 17.3.

Листинг 17.3. Повторное использование кода при ООП-стиле

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import MyWindow

class MyDialog(QtWidgets.QDialog):
    def __init__(self, parent=None):
        QtWidgets.QDialog.__init__(self, parent)
        self.myWidget = MyWindow.MyWindow()
        self.myWidget.vbox.setContentsMargins(0, 0, 0, 0)
        self.button = QtWidgets.QPushButton("&Изменить надпись")
        mainBox = QtWidgets.QVBoxLayout()
        mainBox.addWidget(self.myWidget)
        mainBox.addWidget(self.button)
        self.setLayout(mainBox)
        self.button.clicked.connect(self.on_clicked)

    def on_clicked(self):
        self.myWidget.label.setText("Новая надпись")
        self.button.setDisabled(True)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyDialog()
    window.setWindowTitle("Преимущество ООП-стиля")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

Теперь запустим файл test.pyw двойным щелчком — на экране откроется окно с надписью и двумя кнопками (рис. 17.2). По нажатию на кнопку **Изменить надпись** производится изме-

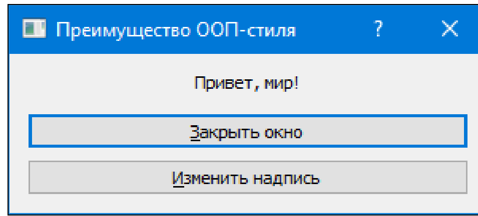

Рис. 17.2. Результат выполнения листинга 17.3

нение текста надписи, и кнопка делается неактивной. Нажатие кнопки **Закрыть окно** будет по-прежнему завершать выполнение приложения.

В этом примере мы создали класс `MyDialog`, который наследует класс `QDialog`. Поэтому при выводе окно автоматически выравнивается по центру экрана, а в заголовке окна выводятся только две кнопки: **Справка** и **Закрыть**. Внутри конструктора мы создаем экземпляр класса `MyWindow` и сохраняем его в атрибуте `myWidget`:

```
self.myWidget = MyWindow.MyWindow()
```

С его помощью позже мы получим доступ ко всем атрибутам класса `MyWindow`. Например, в следующей строке произведем изменение отступа между границами контейнера и границами соседних элементов:

```
self.myWidget.vbox.setContentsMargins(0, 0, 0, 0)
```

В следующих инструкциях внутри конструктора создаются кнопки и контейнер, затем экземпляр класса `MyWindow` и кнопка добавляются в контейнер, а сам контейнер помещается в основное окно.

Выражение:

```
self.button.clicked.connect(self.on_clicked)
```

назначает обработчик нажатия кнопки. В качестве параметра указывается ссылка на метод `on_clicked()`, внутри которого производится изменение текста надписи (с помощью метода `setText()`), и кнопка делается неактивной (с помощью метода `setDisabled()`). Внутри метода `on_clicked()` доступен указатель `self`, через который можно получить доступ к атрибутам классов `MyDialog` и `MyWindow`.

Вот так и производится повторное использование ранее написанного кода: мы создаем класс и сохраняем его внутри отдельного модуля, а чтобы протестировать модуль или использовать его как отдельное приложение, размещаем код создания объекта приложения и объекта окна внутри условия:

```
if __name__ == "__main__":
```

Тогда при запуске с помощью двойного щелчка на значке файла производится выполнение кода как отдельного приложения. Если модуль импортируется, то создание объекта приложения не производится, и мы можем использовать класс в других приложениях. Например, так, как это было сделано в листинге 17.3, или путем наследования класса и добавления или переопределения методов.

В некоторых случаях использование ООП-стиля является обязательным. Например, чтобы обработать нажатие клавиши на клавиатуре, необходимо наследовать какой-либо класс и переопределить в нем метод с предопределенным названием. Какие методы необходимо переопределять, мы рассмотрим при изучении обработки событий.

17.5. Создание окна с помощью программы Qt Designer

Если вы ранее пользовались Visual Studio или Delphi, то вспомните, как с помощью мыши размещали на форме компоненты: щелкали левой кнопкой мыши на нужной кнопке в панели инструментов и перетаскивали компонент на форму, затем с помощью инспектора свойств производили настройку значений некоторых свойств, а остальные свойства получали значения по умолчанию. При этом весь код генерировался автоматически. Произвести аналогичную операцию в PyQt позволяет программа Qt Designer, которая входит в состав этой библиотеки.

К огромному сожалению, в составе последних версий PyQt эта полезная программа отсутствует. Однако ее можно установить отдельно в составе программного пакета PyQt 5 Tools, отдав в командной строке команду:

```
pip3 install pyqt5-tools
```

17.5.1. Создание формы

Запустить Qt Designer можно щелчком на исполняемом файле designer.exe, который располагается по пути *<путь, по которому установлен Python>*\Lib\site-packages\pyqt5-tools. К сожалению, через меню **Пуск** это сделать не получится.

В окне **New Form** открывшегося окна (рис. 17.3) выбираем пункт **Widget** и нажимаем кнопку **Create** — откроется окно с пустой формой, на которую с помощью мыши можно перетаскивать компоненты из панели **Widget Box**.

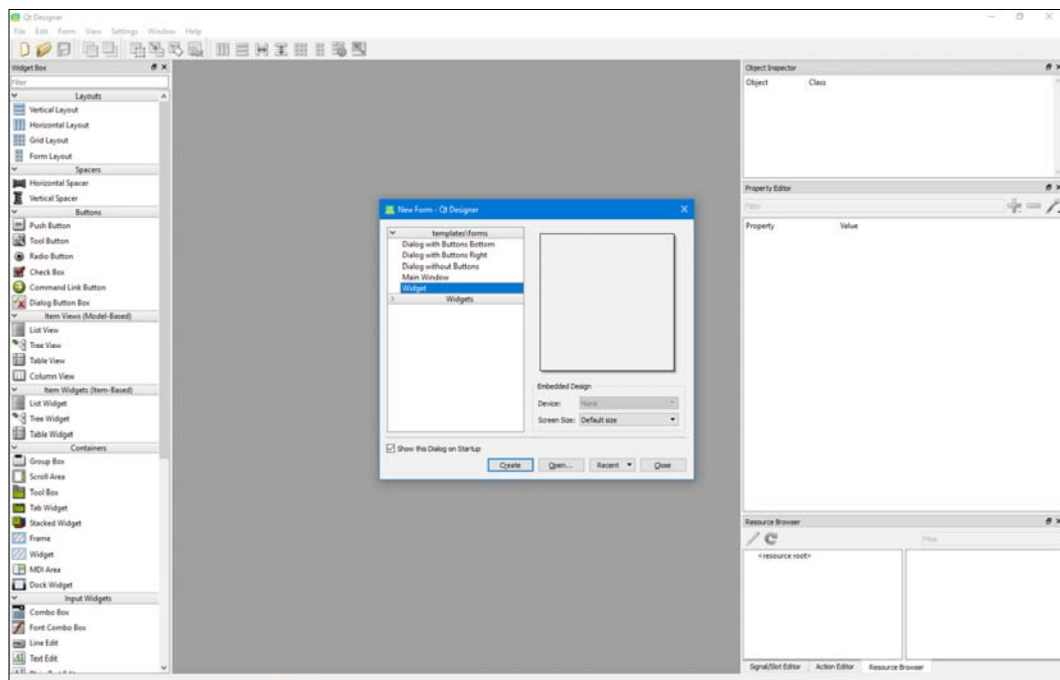

Рис. 17.3. Программа Qt Designer

В качестве примера добавим на форму надпись и кнопку. Для этого на панели **Widget Box** в группе **Display Widgets** щелкнем левой кнопкой мыши на пункте **Label** и, не отпуская кнопку мыши, перетащим компонент на форму. Затем проделаем аналогичную операцию с компонентом **Push Button**, находящимся в группе **Buttons**, и разместим его ниже надписи. Теперь выделим одновременно надпись и кнопку, щелкнем правой кнопкой мыши на любом компоненте и в контекстном меню выберем пункт **Lay out | Lay Out Vertically**. Чтобы компоненты занимали всю область формы, щелкнем правой кнопкой мыши на свободном месте формы и в контекстном меню выберем пункт **Lay out | Lay Out Horizontally**.

Теперь изменим некоторые свойства окна. Для этого в панели **Object Inspector** (рис. 17.4) выделим первый пункт (**Form**), перейдем в панель **Property Editor**, найдем свойство **objectName** и справа от свойства введем значение `MyForm`. Затем найдем свойство **geometry**, щелкнем мышью на значке уголка слева, чтобы отобразить скрытые свойства, и зададим ширину равной `300`, а высоту равной `70` (рис. 17.5), — размеры формы автоматически изменятся. Указать текст, который будет отображаться в заголовке окна, позволяет свойство **windowTitle**.

Чтобы изменить свойства надписи, следует выделить компонент с помощью мыши или выбрать соответствующий ему пункт в панели **Object Inspector**. Для примера изменим значение свойства **text** (оно задает текст надписи). После чего найдем свойство **alignment**, щелкнем мышью на значке уголка слева, чтобы отобразить скрытые свойства, и укажем для свойства **Horizontal** значение `AlignHCenter`. Теперь выделим кнопку и изменим значение свойства **objectName** на `btnQuit`, а в свойстве **text** укажем текст надписи, которая будет выводиться на кнопке. (Кстати, изменить текст надписи или кнопки также можно, выполнив двойной щелчок мышью на компоненте.)

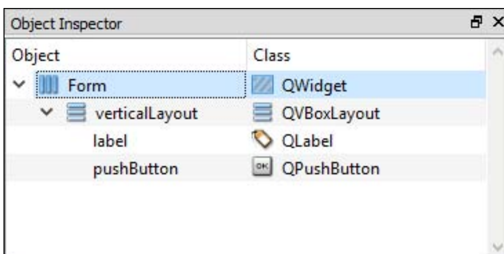

Рис. 17.4. Панель **Object Inspector**

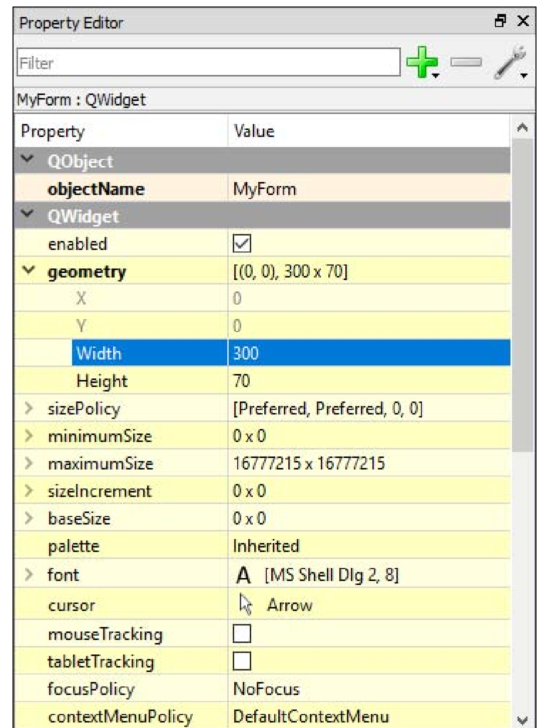

Рис. 17.5. Панель **Property Editor**

Закончив, выберем в меню **File** пункт **Save** и сохраним готовую форму в файл MyForm.ui. При необходимости внести в этот файл какие-либо изменения, его можно открыть в программе Qt Designer, выбрав в меню **File** пункт **Open**.

17.5.2. Использование UI-файла в программе

Как вы можете убедиться, внутри UI-файла содержится текст в XML-формате, а не программный код на языке Python. Следовательно, подключить файл с помощью инструкции `import` не получится. Чтобы использовать UI-файл внутри программы, следует воспользоваться модулем `uic`, который входит в состав библиотеки PyQt. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
from PyQt5 import uic
```

Для загрузки UI-файла предназначена функция `loadUi()`. Формат функции:

```
loadUi(<ui-файл>[, <Экземпляр класса>])
```

Если второй параметр не указан, функция возвращает ссылку на объект формы. С помощью этой ссылки можно получить доступ к компонентам формы и, например, назначить обработчики сигналов (листинг 17.4). Имена компонентов задаются в программе Qt Designer в свойстве **objectName**.

Листинг 17.4. Использование функции `loadUi()`. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets, uic
import sys
app = QtWidgets.QApplication(sys.argv)
window = uic.loadUi("MyForm.ui")
window.btnQuit.clicked.connect(app.quit)
window.show()
sys.exit(app.exec_())
```

Если во втором параметре указать ссылку на экземпляр класса, то все компоненты формы будут доступны через указатель `self` (листинг 17.5).

Листинг 17.5. Использование функции `loadUi()`. Вариант 2

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets, uic

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        uic.loadUi("MyForm.ui", self)
        self.btnQuit.clicked.connect(QtWidgets.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
```

```
    window.show()
    sys.exit(app.exec_())
```

Загрузить UI-файл позволяет также функция `loadUiType()` — она возвращает кортеж из двух элементов: ссылки на класс формы и ссылки на базовый класс. Так как функция возвращает ссылку на класс, а не на экземпляр класса, мы можем создать множество экземпляров класса. После создания экземпляра класса формы необходимо вызвать метод `setupUi()` и передать ему указатель `self` (листинг 17.6).

Листинг 17.6. Использование функции `loadUiType()`. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets, uic

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        Form, Base = uic.loadUiType("MyForm.ui")
        self.ui = Form()
        self.ui.setupUi(self)
        self.ui.btnQuit.clicked.connect(QtWidgets.qApp.quit)
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Загрузить UI-файл можно и вне класса, после чего указать класс формы во втором параметре в списке наследования, — в этом случае наш класс унаследует все методы класса формы (листинг 17.7).

Листинг 17.7. Использование функции `loadUiType()`. Вариант 2

```
from PyQt5 import QtWidgets, uic

Form, Base = uic.loadUiType("MyForm.ui")
class MyWindow(QtWidgets.QWidget, Form):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.btnQuit.clicked.connect(QtWidgets.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

17.5.3. Преобразование UI-файла в PY-файл

Вместо подключения UI-файла можно сгенерировать на его основе программный код на языке Python. Для этого служит утилита pyuic5, чей исполняемый файл располагается в каталоге *<путь, по которому установлен Python>*\Scripts. Запустим командную строку и перейдем в каталог, в котором находится UI-файл. Для генерации Python-программы выполним команду:

```
pyuic5 MyForm.ui -o ui_MyForm.py
```

В результате будет создан файл ui_MyForm.py, который мы уже можем подключить с помощью инструкции `import`. Внутри файла находится класс `Ui_MyForm` с методами `setupUi()` и `retranslateUi()`. При использовании процедурного стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему ссылку на экземпляр окна (листинг 17.8).

Листинг 17.8. Использование класса формы. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys, ui_MyForm
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
ui = ui_MyForm.Ui_MyForm()
ui.setupUi(window)
ui.btnQuit.clicked.connect(QtWidgets.qApp.quit)
window.show()
sys.exit(app.exec_())
```

При использовании ООП-стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему указатель `self` (листинг 17.9).

Листинг 17.9. Использование класса формы. Вариант 2

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import ui_MyForm

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.ui = ui_MyForm.Ui_MyForm()
        self.ui.setupUi(self)
        self.ui.btnQuit.clicked.connect(QtWidgets.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Класс формы можно указать во втором параметре в списке наследования — в этом случае он унаследует все методы класса формы (листинг 17.10).

Листинг 17.10. Использование класса формы. Вариант 3

```
from PyQt5 import QtWidgets
import ui_MyForm

class MyWindow(QtWidgets.QWidget, ui_MyForm.Ui_MyForm):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.btnQuit.clicked.connect(QtWidgets.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Как видите, в PyQt можно создавать формы, размещать компоненты с помощью мыши, а затем непосредственно подключать UI-файлы в программе или преобразовывать их в Python-код с помощью утилиты pyuic5, — все это очень удобно. Тем не менее, чтобы полностью овладеть программированием на PyQt, необходимо уметь создавать код вручную. Поэтому в оставшейся части книги мы больше не станем задействовать программу Qt Designer.

17.6. Модули PyQt 5

В состав библиотеки PyQt 5 входит множество модулей, объединенных в пакет `PyQt5`. Упомянем самые важные из них:

- `QtCore` — содержит классы, не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные модули;
- `QtGui` — содержит классы, реализующие низкоуровневую работу с оконными элементами, обработку сигналов, вывод двухмерной графики и текста и др.;
- `QtWidgets` — содержит классы, реализующие компоненты графического интерфейса: окна, диалоговые окна, надписи, кнопки, текстовые поля и др.;
- `QtWebEngineCore` — включает низкоуровневые классы для отображения веб-страниц;
- `QtWebEngineWidgets` — реализует высокоуровневые компоненты графического интерфейса, предназначенные для вывода веб-страниц и использующие модуль `QtWebEngineCore`;

> ***Примечание***
>
> Ранее для вывода веб-страниц использовались модули `QtWebKit` и `QtWebKitWidgets`. Однако в версии PyQt 5.5 они были объявлены нерекомендованными для использования, а в версии 5.6 удалены.

- `QtMultimedia` — включает низкоуровневые классы для работы с мультимедиа;

- `QtMultimediaWidgets` — реализует высокоуровневые компоненты графического интерфейса с мультимедиа, использующие модуль `QtMultimedia`;
- `QtPrintSupport` — содержит классы, обеспечивающие поддержку печати и предварительного просмотра документов;
- `QtSql` — включает поддержку работы с базами данных, а также реализацию SQLite;
- `QtSvg` — позволяет работать с векторной графикой (SVG);
- `QtNetwork` — содержит классы, предназначенные для работы с сетью;
- `QtXml` и `QtXmlPatterns` — предназначены для обработки XML;
- `QtHelp` — содержат инструменты для создания интерактивных справочных систем;
- `QtWinExtras` — включает поддержку специфических возможностей Microsoft Windows;
- `Qt` — включает классы из всех модулей сразу.

> ***Примечание***
>
> Модуль `QtOpenGL`, обеспечивающий поддержку OpenGL, в версии PyQt 5.9 был объявлен нерекомендованным к использованию и будет удален в одной из последующих версий этой библиотеки. Его функциональность перенесена в модуль `QtGui`.

Для подключения модулей используется следующий синтаксис:

```
from PyQt5 import <Названия модулей через запятую>
```

Так, например, можно подключить модули `QtCore` и `QtWidgets`:

```
from PyQt5 import QtCore, QtWidgets
```

В этой книге мы не станем рассматривать все упомянутые модули — чтобы получить информацию по не рассмотренным здесь модулям, обращайтесь к соответствующей документации.

17.7. Типы данных в PyQt

Библиотека PyQt является надстройкой над написанной на языке C++ библиотекой Qt. Последняя содержит множество классов, которые расширяют стандартные типы данных языка C++ и реализуют динамические массивы, ассоциативные массивы, множества и др. Все эти классы очень помогают при программировании на языке C++, но для языка Python они не представляют особого интереса, т. к. весь этот функционал содержат стандартные типы данных. Тем не менее, при чтении документации вы столкнетесь с ними, поэтому сейчас мы кратко рассмотрим основные типы:

- `QByteArray` — массив байтов. Преобразуется в тип `bytes`:

  ```
  >>> from PyQt5 import QtCore
  >>> arr = QtCore.QByteArray(bytes("str", "cp1251"))
  >>> arr
  PyQt5.QtCore.QByteArray(b'str')
  >>> bytes(arr)
  b'str'
  ```

- `QVariant` — может хранить данные любого типа. Создать экземпляр этого класса можно вызовом конструктора, передав ему нужное значение. А чтобы преобразовать данные, хранящиеся в экземпляре класса `QVariant`, в тип данных Python, нужно вызвать метод `value()`:

```
>>> from PyQt5 import QtCore
>>> n = QtCore.QVariant(10)
>>> n
<PyQt5.QtCore.QVariant object at 0x00FD8D50>
>>> n.value()
10
```

Также можно создать «пустой» экземпляр класса `QVariant`, вызвав конструктор без параметров:

```
>>> QtCore.QVariant() # Пустой объект
<PyQt5.QtCore.QVariant object at 0x00FD8420>
```

Если какой-либо метод ожидает данные типа `QVariant`, ему можно передать данные любого типа.

Еще этот класс поддерживает метод `typeName()`, возвращающий наименование типа хранящихся в экземпляре данных:

```
>>> from PyQt5 import QtCore
>>> n = QtCore.QVariant(10)
>>> n.typeName()
'int'
```

Кроме того, PyQt 5 поддерживает классы `QDate` (значение даты), `QTime` (значение времени), `QDateTime` (значение даты и времени), `QTextStream` (текстовый поток), `QUrl` (URL-адрес) и некоторые другие.

17.8. Управление основным циклом приложения

Для взаимодействия с системой и обработки возникающих сигналов предназначен основной цикл приложения. После вызова метода `exec_()` программа переходит в бесконечный цикл. Инструкции, расположенные после вызова этого метода, будут выполнены только после завершения работы всего приложения. Цикл автоматически прерывается после закрытия последнего открытого окна приложения. С помощью статического метода `setQuitOnLastWindowClosed()` класса `QApplication` это поведение можно изменить.

Чтобы завершить работу приложения, необходимо вызвать слот `quit()` или метод `exit([returnCode=0])` класса `QApplication`. Поскольку программа находится внутри цикла, вызвать эти методы можно лишь при наступлении какого-либо события, — например, при нажатии пользователем кнопки.

После возникновения любого сигнала основной цикл прерывается, и управление передается в обработчик этого сигнала. После завершения работы обработчика управление возвращается основному циклу приложения.

Если внутри обработчика выполняется длительная операция, программа перестает реагировать на события. В качестве примера изобразим длительный процесс с помощью функции `sleep()` из модуля `time` (листинг 17.11).

Листинг 17.11. Выполнение длительной операции

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys, time
```

```
def on_clicked():
    time.sleep(10) # "Засыпаем" на 10 секунд

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Запустить процесс")
button.resize(200, 40)
button.clicked.connect(on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере при нажатии кнопки **Запустить процесс** вызывается функция `on_clicked()`, внутри которой мы приостанавливаем выполнение программы на десять секунд и тем самым прерываем основной цикл. Попробуйте нажать кнопку, перекрыть окно другим окном, а затем заново его отобразить, — вам не удастся это сделать, поскольку окно перестает реагировать на любые события, пока не закончится выполнение процесса. Короче говоря, программа просто зависнет.

Длительную операцию можно разбить на несколько этапов и по завершении каждого этапа выходить в основной цикл с помощью статического метода `processEvents([flags=AllEvents])` класса `QCoreApplication`, от которого наследуется класс `QApplication`. Переделаем предыдущую программу, инсценировав с помощью цикла длительную операцию, которая выполняется 20 секунд (листинг 17.12).

Листинг 17.12. Использование метода `processEvents()`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys, time

def on_clicked():
    button.setDisabled(True)             # Делаем кнопку неактивной
    for i in range(1, 21):
        QtWidgets.qApp.processEvents()   # Запускаем оборот цикла
        time.sleep(1)                    # "Засыпаем" на 1 секунду
        print("step -", i)
    button.setDisabled(False)            # Делаем кнопку активной

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Запустить процесс")
button.resize(200, 40)
button.clicked.connect(on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере длительная операция разбита на одинаковые этапы, после выполнения каждого из которых выполняется выход в основной цикл приложения. Теперь при перекрытии окна и повторном его отображении оно будет перерисовано — таким образом, приложение по-прежнему будет взаимодействовать с системой, хотя и с некоторой задержкой.

17.9. Многопоточные приложения

При обработке больших объемов данных не всегда можно равномерно разбить операцию на небольшие по времени этапы, поэтому при использовании метода `processEvents()` возможны проблемы, и тогда имеет смысл вынести длительную операцию в отдельный поток, — в этом случае операция станет выполняться параллельно с основным циклом приложения и не будет его блокировать.

В одном процессе можно запустить сразу несколько независимых потоков, и если ваш компьютер оснащен многоядерным процессором, потоки будут равномерно распределены по его ядрам. За счет этого можно не только избежать блокировки GUI-потока приложения, в котором выполняется обновление его интерфейса, но и значительно увеличить эффективность выполнения кода. Завершение основного цикла приложения приводит к завершению работы всех потоков.

17.9.1. Класс *QThread*: создание потока

Для создания потока в PyQt предназначен класс `QThread`, который объявлен в модуле `QtCore` и наследует класс `QObject`. Конструктор класса `QThread` имеет следующий формат:

```
<Объект> = QThread([parent=None])
```

Чтобы использовать потоки, следует создать класс, который будет наследником класса `QThread`, и определить в нем метод `run()`. Код, расположенный в методе `run()`, будет выполняться в отдельном потоке, а после завершения выполнения метода `run()` этот поток прекратит свое существование. Затем нужно создать экземпляр класса и вызвать метод `start()`, который после запуска потока вызовет метод `run()`. Обратите внимание, что если напрямую вызвать метод `run()`, то код станет выполняться в основном, а не в отдельном потоке. Метод `start()` имеет следующий формат:

```
start([priority=QThread.InheritPriority])
```

Параметр `priority` задает приоритет выполнения потока по отношению к другим потокам. Следует учитывать, что при наличии потока с самым высоким приоритетом поток с самым низким приоритетом в некоторых операционных системах может быть просто проигнорирован. Приведем допустимые значения параметра (в порядке увеличения приоритета) и соответствующие им атрибуты из класса `QThread`:

- `0` — `IdlePriority` — самый низкий приоритет;
- `1` — `LowestPriority`;
- `2` — `LowPriority`;
- `3` — `NormalPriority`;
- `4` — `HighPriority`;
- `5` — `HighestPriority`;
- `6` — `TimeCriticalPriority` — самый высокий приоритет;
- `7` — `InheritPriority` — автоматический выбор приоритета (значение по умолчанию).

Задать приоритет потока также позволяет метод `setPriority(<Приоритет>)`. Узнать, какой приоритет использует запущенный поток, можно с помощью метода `priority()`.

После запуска потока генерируется сигнал `started()`, а после завершения — сигнал `finished()`. Назначив обработчики этим сигналам, можно контролировать статус потока из

основного цикла приложения. Если необходимо узнать текущий статус, следует воспользоваться методами `isRunning()` и `isFinished()`: метод `isRunning()` возвращает значение `True`, если поток выполняется, а метод `isFinished()` — значение `True`, если поток закончил выполнение.

Потоки выполняются внутри одного процесса и имеют доступ ко всем глобальным переменным. Однако следует учитывать, что из потока нельзя изменять что-либо в GUI-потоке приложения, — например, выводить текст на надпись. Для изменения данных в GUI-потоке нужно использовать сигналы. Внутри потока у нужного сигнала вызывается метод `emit()`, который, собственно, и выполняет его генерацию. В параметрах метода `emit()` можно указать данные, которые будут переданы обработчику сигнала. А внутри GUI-потока назначаем обработчик этого сигнала и в обработчике пишем код, который и будет обновлять интерфейс приложения.

Рассмотрим использование класса `QThread` на примере (листинг 17.13).

Листинг 17.13. Использование класса `QThread`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
class MyThread(QtCore.QThread):
    mysignal = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        for i in range(1, 21):
            self.sleep(3)              # "Засыпаем" на 3 секунды
            # Передача данных из потока через сигнал
            self.mysignal.emit("i = %s" % i)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button = QtWidgets.QPushButton("Запустить процесс")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.mythread = MyThread()     # Создаем экземпляр класса
        self.button.clicked.connect(self.on_clicked)
        self.mythread.started.connect(self.on_started)
        self.mythread.finished.connect(self.on_finished)
        self.mythread.mysignal.connect(self.on_change, QtCore.Qt.QueuedConnection)
    def on_clicked(self):
        self.button.setDisabled(True)  # Делаем кнопку неактивной
        self.mythread.start()          # Запускаем поток
    def on_started(self):              # Вызывается при запуске потока
        self.label.setText("Вызван метод on_started()")
```

```
    def on_finished(self):          # Вызывается при завершении потока
        self.label.setText("Вызван метод on_finished()")
        self.button.setDisabled(False) # Делаем кнопку активной
    def on_change(self, s):
        self.label.setText(s)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QThread")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec_())
```

Здесь мы создали класс `MyThread`, который является наследником класса `QThread`. В нем мы определили свой собственный сигнал `mysignal`, для чего создали атрибут с таким же именем и занесли в него значение, возвращенное функцией `pyqtSignal()` из модуля `QtCore`. Функции `pyqtSignal()` мы передали в качестве параметра тип `str` (строка Python), тем самым указав PyQt, что вновь определенный сигнал будет принимать единственный параметр строкового типа:

```
mysignal = QtCore.pyqtSignal(str)
```

В том же классе мы определили обязательный для потоков метод `run()` — в нем производится имитация процесса с помощью цикла `for` и метода `sleep()`: каждые три секунды выполняется генерация сигнала `mysignal` и передача текущего значения переменной `i` в составе строки:

```
self.mysignal.emit("i = %s" % i)
```

Внутри конструктора класса `MyWindow` мы назначили обработчик этого сигнала с помощью выражения:

```
self.mythread.mysignal.connect(self.on_change, QtCore.Qt.QueuedConnection)
```

Здесь все нам уже знакомо: у свойства `mysignal` потока, которое представляет одноименный сигнал, вызывается метод `connect()`, и ему первым параметром передается обработчик. Во втором параметре метода `connect()` с помощью атрибута `QueuedConnection` указывается, что сигнал помещается в очередь обработки событий, и обработчик должен выполняться в потоке приемника сигнала, т. е. в GUI-потоке. Из GUI-потока мы можем смело изменять свойства компонентов интерфейса.

Теперь рассмотрим код метода класса `MyWindow`, который станет обработчиком сигнала `mysignal`:

```
def on_change(self, s):
    self.label.setText(s)
```

Второй параметр этого метода служит для приема параметра, переданного этому сигналу. Значение этого параметра будет выведено в надписи с помощью метода `setText()`.

Еще в конструкторе класса `MyWindow` производится создание надписи и кнопки, а затем их размещение внутри вертикального контейнера. Далее выполняется создание экземпляра класса `MyThread` и сохранение его в атрибуте `mythread`. С помощью этого атрибута мы мо-

жем управлять потоком и назначить обработчики сигналов `started()`, `finished()` и `mysignal`. Запуск потока производится с помощью метода `start()` внутри обработчика нажатия кнопки. Чтобы исключить повторный запуск потока, мы с помощью метода `setDisabled()` делаем кнопку неактивной, а после окончания работы потока внутри обработчика сигнала `finished()` опять делаем кнопку активной.

Обратите внимание, что для имитации длительного процесса мы использовали статический метод `sleep()` из класса `QThread`, а не функцию `sleep()` из модуля `time`. Вообще, приостановить выполнение потока позволяют следующие статические методы класса `QThread`:

- `sleep()` — продолжительность задается в секундах:

  ```
  QtCore.QThread.sleep(3)          # "Засыпаем" на 3 секунды
  ```

- `msleep()` — продолжительность задается в миллисекундах:

  ```
  QtCore.QThread.msleep(3000)      # "Засыпаем" на 3 секунды
  ```

- `usleep()` — продолжительность задается в микросекундах:

  ```
  QtCore.QThread.usleep(3000000) # "Засыпаем" на 3 секунды
  ```

Еще один полезный статичный метод класса `QThread` — `yieldCurrentThread()` — немедленно приостанавливает выполнение текущего потока и передает управление следующему ожидающему выполнения потоку, если таковой есть:

```
QtCore.QThread.yieldCurrentThread()
```

17.9.2. Управление циклом внутри потока

Очень часто внутри потока одни и те же инструкции выполняются многократно. Например, при осуществлении мониторинга серверов в Интернете на каждой итерации цикла посылается запрос к одному и тому же серверу. При этом внутри метода `run()` используется бесконечный цикл, выход из которого производится после окончания опроса всех серверов. В некоторых случаях этот цикл необходимо прервать преждевременно по нажатию кнопки пользователем. Чтобы это стало возможным, в классе, реализующем поток, следует создать атрибут, который будет содержать флаг текущего состояния. Далее на каждой итерации цикла проверяется состояние флага и при его изменении прерывается выполнение цикла. Чтобы изменить значение атрибута, создаем обработчик и связываем его с сигналом `clicked()` соответствующей кнопки. При нажатии кнопки внутри обработчика производим изменение значения атрибута. Пример запуска и остановки потока с помощью кнопок приведен в листинге 17.14.

Листинг 17.14. Запуск и остановка потока

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyThread(QtCore.QThread):
    mysignal = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.running = False   # Флаг выполнения
        self.count = 0
    def run(self):
        self.running = True
```

```
        while self.running:    # Проверяем значение флага
            self.count += 1
            self.mysignal.emit("count = %s" % self.count)
            self.sleep(1)      # Имитируем процесс

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.btnStart = QtWidgets.QPushButton("Запустить поток")
        self.btnStop = QtWidgets.QPushButton("Остановить поток")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnStart)
        self.vbox.addWidget(self.btnStop)
        self.setLayout(self.vbox)
        self.mythread = MyThread()
        self.btnStart.clicked.connect(self.on_start)
        self.btnStop.clicked.connect(self.on_stop)
        self.mythread.mysignal.connect(self.on_change, QtCore.Qt.QueuedConnection)
    def on_start(self):
        if not self.mythread.isRunning():
            self.mythread.start()      # Запускаем поток
    def on_stop(self):
        self.mythread.running = False  # Изменяем флаг выполнения
    def on_change(self, s):
        self.label.setText(s)
    def closeEvent(self, event):       # Вызывается при закрытии окна
        self.hide()                    # Скрываем окно
        self.mythread.running = False  # Изменяем флаг выполнения
        self.mythread.wait(5000)       # Даем время, чтобы закончить
        event.accept()                 # Закрываем окно

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Запуск и остановка потока")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

В этом примере в конструкторе класса `MyThread` создается атрибут `running`, и ему присваивается значение `False`. При запуске потока внутри метода `run()` значение атрибута изменяется на `True`. Затем запускается цикл, в котором атрибут указывается в качестве условия. Как только значение атрибута станет равным значению `False`, цикл будет остановлен.

Внутри конструктора класса `MyWindow` производится создание надписи, двух кнопок и экземпляра класса `MyThread`. Далее назначаются обработчики сигналов. При нажатии кнопки

Запустить поток запустится метод `on_start()`, внутри которого с помощью метода `isRunning()` производится проверка текущего статуса потока. Если поток не запущен, выполняется его запуск вызовом метода `start()`. При нажатии кнопки **Остановить поток** запустится метод `on_stop()`, в котором атрибуту `running` присваивается значение `False`. Это значение является условием выхода из цикла внутри метода `run()`.

Путем изменения значения атрибута можно прервать выполнение цикла только в том случае, если закончилось выполнение очередной итерации. Если поток длительное время ожидает какого-либо события (например, ответа сервера), можно так и не дождаться завершения потока. Чтобы принудительно прервать выполнение потока, следует воспользоваться методом `terminate()`. Однако к этому методу рекомендуется прибегать только в крайнем случае, поскольку прерывание производится в любой части кода. При этом блокировки автоматически не снимаются, а кроме того, можно повредить данные, над которыми производились операции в момент прерывания. После вызова метода `terminate()` следует вызвать метод `wait()`.

При закрытии окна приложение завершает работу, что также приводит к завершению всех потоков. Чтобы предотвратить повреждение данных, следует перехватить событие закрытия окна и дождаться окончания выполнения. Чтобы перехватить событие, необходимо внутри класса создать метод с предопределенным названием, в нашем случае — с названием `closeEvent()`. Этот метод будет автоматически вызван при попытке закрыть окно. В качестве параметра метод принимает объект события `event`, через который можно получить дополнительную информацию о событии. Чтобы закрыть окно внутри метода `closeEvent()`, следует вызвать метод `accept()` объекта события. Если необходимо предотвратить закрытие окна, то следует вызвать метод `ignore()`.

Внутри метода `closeEvent()` мы присваиваем атрибуту `running` значение `False`. Далее с помощью метода `wait()` даем возможность потоку нормально завершить работу. В качестве параметра метод `wait()` принимает количество миллисекунд, по истечении которых управление будет передано следующей инструкции. Необходимо учитывать, что это максимальное время: если поток закончит работу раньше, то и метод закончит выполнение раньше. Метод `wait()` возвращает значение `True`, если поток успешно завершил работу, и `False` — в противном случае. Ожидание завершения потока занимает некоторое время, в течение которого окно будет по-прежнему видимым. Чтобы не вводить пользователя в заблуждение, в самом начале метода `closeEvent()` мы скрываем окно вызовом метода `hide()`.

Каждый поток может иметь собственный цикл обработки сигналов, который запускается с помощью метода `exec_()`. В этом случае потоки могут обмениваться сигналами между собой. Чтобы прервать цикл, следует вызвать слот `quit()` или метод `exit([returnCode=0])`. Рассмотрим обмен сигналами между потоками на примере (листинг 17.15).

Листинг 17.15. Обмен сигналами между потоками

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class Thread1(QtCore.QThread):
    s1 = QtCore.pyqtSignal(int)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.count = 0
```

```
    def run(self):
        self.exec_()          # Запускаем цикл обработки сигналов
    def on_start(self):
        self.count += 1
        self.s1.emit(self.count)

class Thread2(QtCore.QThread):
    s2 = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        self.exec_()          # Запускаем цикл обработки сигналов
    def on_change(self, i):
        i += 10
        self.s2.emit("%d" % i)
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button = QtWidgets.QPushButton("Сгенерировать сигнал")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.thread1 = Thread1()
        self.thread2 = Thread2()
        self.thread1.start()
        self.thread2.start()
        self.button.clicked.connect(self.thread1.on_start)
        self.thread1.s1.connect(self.thread2.on_change)
        self.thread2.s2.connect(self.on_thread2_s2)
    def on_thread2_s2(self, s):
        self.label.setText(s)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Обмен сигналами между потоками")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec_())
```

В этом примере мы создали классы `Thread1`, `Thread2` и `MyWindow`. Первые два класса представляют собой потоки. Внутри них в методе `run()` вызывается метод `exec_()`, который запускает цикл обработки событий. В конструкторе класса `MyWindow` производится создание надписи, кнопки и экземпляров классов `Thread1` и `Thread2`. Далее выполняется запуск сразу двух потоков.

В следующей инструкции сигнал нажатия кнопки соединяется с методом `on_start()` первого потока. Внутри этого метода производится какая-либо операция (в нашем случае — увеличение значения атрибута `count`), а затем с помощью метода `emit()` генерируется сигнал `s1`, и в параметре передается результат выполнения метода. Сигнал `s1` соединен с методом `on_change()` второго потока. Внутри этого метода также производится какая-либо операция, а затем генерируется сигнал `s2`, и передается результат выполнения метода. В свою очередь сигнал `s2` соединен со слотом `on_thread2_s2` объекта окна, который выводит в надпись значение, переданное с этим сигналом. Таким образом, при нажатии кнопки **Сгенерировать сигнал** вначале будет вызван метод `on_start()` из класса `Thread1`, затем метод `on_change()` из класса `Thread2`, а потом метод `on_thread2_s2` класса `MyWindow`, который выведет результат выполнения на экран.

17.9.3. Модуль *queue*: создание очереди заданий

В предыдущем разделе мы рассмотрели возможность обмена сигналами между потоками. Теперь предположим, что запущены десять потоков, которые ожидают задания в бесконечном цикле. Как передать задание одному потоку, а не всем сразу? И как определить, какому потоку передать задание? Можно, конечно, создать список в глобальном пространстве имен и добавлять задания в этот список, но в этом случае придется решать вопрос о совместном использовании одного ресурса сразу десятью потоками. Ведь если потоки будут получать задания одновременно, то одно задание могут получить сразу несколько потоков, и какому-либо потоку не хватит заданий, — возникнет исключительная ситуация. Попросту говоря, возникает ситуация, когда вы пытаетесь сесть на стул, а другой человек одновременно пытается вытащить его из-под вас. Думаете, что успеете сесть?

Модуль `queue`, входящий в состав стандартной библиотеки Python, позволяет решить эту проблему. Модуль содержит несколько классов, которые реализуют разного рода потокобезопасные очереди. Опишем эти классы:

- `Queue` — очередь (первым пришел, первым вышел). Формат конструктора:

  ```
  <Объект> = Queue([maxsize=0])
  ```

  Пример:

  ```
  >>> import queue
  >>> q = queue.Queue()
  >>> q.put_nowait("elem1")
  >>> q.put_nowait("elem2")
  >>> q.get_nowait()
  'elem1'
  >>> q.get_nowait()
  'elem2'
  ```

- `LifoQueue` — стек (последним пришел, первым вышел). Формат конструктора:

  ```
  <Объект> = LifoQueue([maxsize=0])
  ```

  Пример:

  ```
  >>> q = queue.LifoQueue()
  >>> q.put_nowait("elem1")
  >>> q.put_nowait("elem2")
  >>> q.get_nowait()
  'elem2'
  ```

```
>>> q.get_nowait()
'elem1'
```

- `PriorityQueue` — очередь с приоритетами. Элементы очереди должны быть кортежами, в которых первым элементом является число, означающее приоритет, а вторым — значение элемента. При получении значения возвращается элемент с наивысшим приоритетом (наименьшим значением в первом параметре кортежа). Формат конструктора класса:

  ```
  <Объект> = PriorityQueue([maxsize=0])
  ```

  Пример:

  ```
  >>> q = queue.PriorityQueue()
  >>> q.put_nowait((10, "elem1"))
  >>> q.put_nowait((3, "elem2"))
  >>> q.put_nowait((12, "elem3"))
  >>> q.get_nowait()
  (3, 'elem2')
  >>> q.get_nowait()
  (10, 'elem1')
  >>> q.get_nowait()
  (12, 'elem3')
  ```

Параметр `maxsize` во всех трех случаях задает максимальное количество элементов, которое может содержать очередь. Если параметр равен нулю (значение по умолчанию) или отрицательному значению, то размер очереди не ограничен.

Эти классы поддерживают следующие методы:

- `put(<Элемент>[, block=True][, timeout=None])` — добавляет элемент в очередь. Если в параметре `block` указано значение `True`, поток будет ожидать возможности добавления элемента, — при этом в параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось добавить, возбуждается исключение `queue.Full`. В случае передачи параметром `block` значения `False` очередь не будет ожидать, когда появится возможность добавить в нее новый элемент, и в случае невозможности сделать это возбудит исключение `queue.Full` немедленно;
- `put_nowait(<Элемент>)` — добавление элемента без ожидания. Эквивалентно:

  ```
  put(<Элемент>, False)
  ```
- `get([block=True][, timeout=None])` — возвращает элемент, при этом удаляя его из очереди. Если в параметре `block` указано значение `True`, поток будет ожидать возможности извлечения элемента, — при этом в параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось получить, возбуждается исключение `queue.Empty`. В случае передачи параметром `block` значения `False` очередь не будет ожидать, когда появится возможность извлечь из нее элемент, и в случае невозможности сделать это возбудит исключение `queue.Empty` немедленно;
- `get_nowait()` — извлечение элемента без ожидания. Эквивалентно вызову `get(False)`;
- `join()` — блокирует поток, пока не будут обработаны все задания в очереди. Другие потоки после обработки текущего задания должны вызывать метод `task_done()`. Как только все задания окажутся обработанными, поток будет разблокирован;
- `task_done()` — этот метод должны вызывать потоки после обработки задания;
- `qsize()` — возвращает приблизительное количество элементов в очереди. Так как доступ к очереди имеют сразу несколько потоков, доверять этому значению не следует — в любой момент времени количество элементов может измениться;

- `empty()` — возвращает `True`, если очередь пуста, и `False` — в противном случае;
- `full()` — возвращает `True`, если очередь содержит элементы, и `False` — в противном случае.

Рассмотрим использование очереди в многопоточном приложении на примере (листинг 17.16).

Листинг 17.16. Использование модуля `queue`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import queue
class MyThread(QtCore.QThread):
    task_done = QtCore.pyqtSignal(int, int, name = 'taskDone')
    def __init__(self, id, queue, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
        self.queue = queue
    def run(self):
        while True:
            task = self.queue.get()            # Получаем задание
            self.sleep(5)                      # Имитируем обработку
            self.task_done.emit(task, self.id) # Передаем данные обратно
            self.queue.task_done()

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
        QtWidgets.QPushButton.__init__(self)
        self.setText("Раздать задания")
        self.queue = queue.Queue()          # Создаем очередь
        self.threads = []
        for i in range(1, 3):               # Создаем потоки и запускаем
            thread = MyThread(i, self.queue)
            self.threads.append(thread)
            thread.task_done.connect(self.on_task_done, QtCore.Qt.QueuedConnection)
            thread.start()
        self.clicked.connect(self.on_add_task)
    def on_add_task(self):
        for i in range(0, 11):
            self.queue.put(i)               # Добавляем задания в очередь
    def on_task_done(self, data, id):
        print(data, "- id =", id)           # Выводим обработанные данные
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование модуля queue")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec_())
```

В этом примере конструктор класса MyThread принимает уникальный идентификатор (id) и ссылку на очередь (queue), которые сохраняются в одноименных атрибутах класса. В методе run() внутри бесконечного цикла производится получение элемента из очереди с помощью метода get(). Если очередь пуста, поток будет ожидать, пока не появится хотя бы один элемент. Далее производится обработка задания (в нашем случае — просто задержка), а затем обработанные данные передаются главному потоку через сигнал taskDone, принимающий два целочисленных параметра. В следующей инструкции с помощью метода task_done() указывается, что задание было обработано.

Отметим, что здесь в вызове функции pyqtSignal() присутствует именованный параметр name:

```
task_done = QtCore.pyqtSignal(int, int, name = 'taskDone')
```

Он задает имя сигнала и может быть полезен в том случае, если это имя отличается от имени атрибута класса, соответствующего сигналу, — как в нашем случае, где имя сигнала taskDone отличается от имени атрибута task_done. После чего мы можем обращаться к сигналу как по имени соответствующего ему атрибута:

```
self.task_done.emit(task, self.id)
```

так и по имени, заданному в параметре name функции pyqtSignal():

```
self.taskDone.emit(task, self.id)
```

Главный поток реализуется с помощью класса MyWindow. Обратите внимание, что наследуется класс QPushButton (кнопка), а не класс QWidget. Все визуальные компоненты являются наследниками класса QWidget, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В нашем случае используется только кнопка, поэтому можно сразу наследовать класс QPushButton.

Внутри конструктора класса MyWindow с помощью метода setText() задается текст надписи на кнопке, затем создается экземпляр класса Queue и сохраняется в атрибуте queue. В следующем выражении производится создание списка, в котором будут храниться ссылки на объекты потоков. Сами объекты потоков (в нашем случае их два) создаются внутри цикла и добавляются в список. Внутри цикла производится также назначение обработчика сигнала taskDone и запуск потока с помощью метода start(). Далее назначается обработчик нажатия кнопки.

При нажатии кнопки **Раздать задания** вызывается метод on_add_task(), внутри которого производится добавление заданий в очередь. После этого потоки выходят из цикла ожидания, и каждый из них получает одно уникальное задание. После окончания обработки потоки генерируют сигнал taskDone и вызывают метод task_done(), информирующий об окончании обработки задания. Главный поток получает сигнал и вызывает метод on_task_done(), внутри которого через параметры будут доступны обработанные данные. Так как метод расположен в GUI-потоке, мы можем изменять свойства компонентов и, например, добавить результат в список или таблицу. В нашем же примере результат просто выводится в окно консоли (чтобы увидеть сообщения, следует сохранить файл с расширением py, а не pyw). После окончания обработки задания потоки снова получают задания. Если очередь окажется пуста, потоки перейдут в режим ожидания заданий.

17.9.4. Классы *QMutex* и *QMutexLocker*

Как вы уже знаете, совместное использование одного ресурса сразу несколькими потоками может привести к непредсказуемому поведению программы или даже аварийному ее

завершению. То есть, доступ к ресурсу в один момент времени должен иметь лишь один поток. Следовательно, внутри программы необходимо предусмотреть возможность блокировки ресурса одним потоком и ожидание его разблокировки другим потоком.

Реализовать блокировку ресурса в PyQt позволяют классы `QMutex` и `QMutexLocker` из модуля `QtCore`.

Конструктор класса `QMutex` создает так называемый *мьютекс* и имеет следующий формат:

```
<Объект> = QMutex([mode=QtCore.QMutex.NonRecursive])
```

Необязательный параметр `mode` может принимать значения `NonRecursive` (поток может запросить блокировку только единожды, а после снятия блокировка может быть запрошена снова, — значение по умолчанию) и `Recursive` (поток может запросить блокировку несколько раз, и чтобы полностью снять блокировку, следует вызвать метод `unlock()` соответствующее количество раз).

Класс `QMutex` поддерживает следующие методы:

- `lock()` — устанавливает блокировку. Если ресурс был заблокирован другим потоком, работа текущего потока приостанавливается до снятия блокировки;
- `tryLock([timeout=0])` — устанавливает блокировку. Если блокировка была успешно установлена, метод возвращает значение `True`, если ресурс заблокирован другим потоком — значение `False` без ожидания возможности установить блокировку. Максимальное время ожидания в миллисекундах можно указать в качестве необязательного параметра `timeout`. Если в параметре указано отрицательное значение, то метод `tryLock()` ведет себя аналогично методу `lock()`;
- `unlock()` — снимает блокировку;
- `isRecursive()` — возвращает `True`, если конструктору было передано значение `Recursive`.

Рассмотрим использование класса `QMutex` на примере (листинг 17.17).

Листинг 17.17. Использование класса `QMutex`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyThread(QtCore.QThread):
    x = 10                               # Атрибут класса
    mutex = QtCore.QMutex()              # Мьютекс
    def __init__(self, id, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
    def run(self):
        self.change_x()
    def change_x(self):
        MyThread.mutex.lock()            # Блокируем
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
```

```
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        MyThread.mutex.unlock()          # Снимаем блокировку

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
        QtWidgets.QPushButton.__init__(self)
        self.setText("Запустить")
        self.thread1 = MyThread(1)
        self.thread2 = MyThread(2)
        self.clicked.connect(self.on_start)
    def on_start(self):
        if not self.thread1.isRunning(): self.thread1.start()
        if not self.thread2.isRunning(): self.thread2.start()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QMutex")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec_())
```

В этом примере в классе `MyThread` мы создали атрибут `x`, который доступен всем экземплярам класса. Изменение значения атрибута в одном потоке повлечет изменение значения и в другом потоке. Если потоки будут изменять значение одновременно, то предсказать текущее значение атрибута становится невозможным. Следовательно, изменять значение можно только после установки блокировки.

Чтобы обеспечить блокировку, внутри класса `MyThread` создается экземпляр класса `QMutex` и сохраняется в атрибуте `mutex`. Обратите внимание, что сохранение производится в атрибуте объекта класса, а не в атрибуте экземпляра класса. Чтобы блокировка сработала, необходимо, чтобы защищаемый атрибут и мьютекс находились в одной области видимости. Далее весь код метода `change_x()`, в котором производится изменение атрибута `x`, помещается между вызовами методов `lock()` и `unlock()` мьютекса, — таким образом гарантируется, что он будет выполнен сначала одним потоком и только потом — другим.

Внутри конструктора класса `MyWindow` производится создание двух экземпляров класса `MyThread` и назначение обработчика нажатия кнопки. По нажатию кнопки **Запустить** будет вызван метод `on_start()`, внутри которого производится запуск сразу двух потоков одновременно, — при условии, что потоки не были запущены ранее. В результате мы получим в окне консоли следующий результат:

```
x = 10 id = 1
x = 15 id = 1
x = 49 id = 1
x = 49 id = 2
x = 54 id = 2
x = 88 id = 2
```

Как можно видеть, сначала изменение атрибута произвел поток с идентификатором `1`, а лишь затем — поток с идентификатором `2`. Если блокировку не указать, то результат будет иным:

```
x = 10 id = 1
x = 15 id = 2
x = 20 id = 1
x = 54 id = 1
x = 54 id = 2
x = 88 id = 2
```

В этом случае поток с идентификатором `2` изменил значение атрибута `x` до окончания выполнения метода `change_x()` в потоке с идентификатором `1`.

При возникновении исключения внутри метода `change_x()` ресурс останется заблокированным, т. к. вызов метода `unlock()` не будет выполнен. Кроме того, можно по случайности забыть вызвать метод `unlock()`, что также приведет к вечной блокировке.

Исключить подобную ситуацию позволяет класс `QMutexLocker`. Конструктор этого класса принимает объект мьютекса и устанавливает блокировку. После выхода из области видимости будет вызван деструктор класса, внутри которого блокировка автоматически снимется. Следовательно, если создать экземпляр класса `QMutexLocker` в начале метода, то после выхода из метода блокировка будет снята. Переделаем метод `change_x()` из класса `MyThread` и используем класс `QMutexLocker` (листинг 17.18).

Листинг 17.18. Использование класса `QMutexLocker`

```
    def change_x(self):
        ml = QtCore.QMutexLocker(MyThread.mutex)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        # Блокировка автоматически снимется
```

При использовании класса `QMutexLocker` следует помнить о разнице между областями видимости в языках C++ и Python. В языке C++ область видимости ограничена блоком, которым может являться как функция, так и просто область, ограниченная фигурными скобками. Таким образом, если переменная объявлена внутри блока условного оператора, например, `if`, то при выходе из этого блока переменная уже не будет видна:

```
if (условие) {
    int x = 10; // Объявляем переменную
    // ...
}
// Здесь переменная x уже не видна!
```

В языке Python область видимости гораздо шире. Если мы объявим переменную внутри условного оператора, то она будет видна и после выхода из этого блока:

```
if условие:
    x = 10      # Объявляем переменную
    # ...
# Здесь переменная x еще видна
```

Таким образом, область видимости локальной переменной в языке Python ограничена функцией, а не любым блоком.

Класс `QMutexLocker` поддерживает протокол менеджеров контекста, который позволяет ограничить область видимости блоком инструкции `with...as`. Этот протокол гарантирует снятие блокировки, даже если внутри инструкции `with...as` будет возбуждено исключение. Переделаем метод `change_x()` из класса `MyThread` снова и используем в этот раз инструкцию `with...as` (листинг 17.19).

Листинг 17.19. Использование инструкции `with...as`

```
    def change_x(self):
        with QtCore.QMutexLocker(MyThread.mutex):
            print("x =", MyThread.x, "id =", self.id)
            MyThread.x += 5
            self.sleep(2)
            print("x =", MyThread.x, "id =", self.id)
            MyThread.x += 34
            print("x =", MyThread.x, "id =", self.id)
            # Блокировка автоматически снимется
```

Теперь, когда вы уже знаете о возможности блокировки ресурса, следует сделать несколько замечаний:

- установка и снятие блокировки занимают некоторый промежуток времени, тем самым снижая эффективность всей программы. Поэтому встроенные типы данных не обеспечивают безопасную работу в многопоточном приложении. И прежде чем использовать блокировки, подумайте — может быть, в вашем приложении они и не нужны;
- второе замечание относится к доступу к защищенному ресурсу из GUI-потока. Ожидание снятия блокировки может заблокировать GUI-поток, и приложение перестанет реагировать на события. Поэтому в таком случае следует использовать сигналы, а не прямой доступ;
- и последнее замечание относится к *взаимной блокировке*. Если первый поток, владея ресурсом `A`, захочет получить доступ к ресурсу `B`, а второй поток, владея ресурсом `B`, захочет получить доступ к ресурсу `A`, то оба потока будут ждать снятия блокировки вечно. В этой ситуации следует предусмотреть возможность временного освобождения ресурсов одним из потоков после превышения периода ожидания.

Класс `QMutexLocker` также поддерживает методы `unlock()` и `relock()`. Первый метод выполняет разблокировку мьютекса без уничтожения экземпляра класса `QMutexLocker`, а второй выполняет повторное наложение блокировки.

> ***Примечание***
>
> Для синхронизации и координации потоков предназначены также классы `QSemaphore` и `QWaitCondition`. За подробной информацией по этим классам обращайтесь к документации по PyQt. Следует также помнить, что в стандартную библиотеку языка Python входят модули `multiprocessing` и `threading`, которые позволяют работать с потоками в любом приложении. Однако при использовании PyQt нужно отдать предпочтение классу `QThread`, т. к. он позволяет работать с сигналами.

17.10. Вывод заставки

В больших приложениях загрузка начальных данных может занимать продолжительное время, в течение которого принято выводить окно-заставку, в котором отображается процесс загрузки. По окончании инициализации приложения окно-заставка скрывается и отображается главное окно.

Для вывода окна-заставки в PyQt предназначен класс `QSplashScreen` из модуля `QtWidgets`. Конструктор класса имеет следующие форматы:

```
<Объект> = QSplashScreen([<Изображение>][, flags=<Тип окна>])
<Объект> = QSplashScreen(<Родитель>[, <Изображение>][, flags=<Тип окна>])
```

Параметр `<Родитель>` позволяет указать ссылку на родительский компонент. В параметре `<Изображение>` указывается ссылка на изображение (экземпляр класса `QPixmap`, объявленного в модуле `QtGui`), которое будет отображаться на заставке. Конструктору класса `QPixmap` можно передать путь к файлу с изображением. Параметр `flags` предназначен для указания типа окна — например, чтобы заставка отображалась поверх всех остальных окон, следует передать флаг `WindowStaysOnTopHint`.

Класс `QSplashScreen` поддерживает следующие методы:

- `show()` — отображает заставку;
- `finish(<Ссылка на окно>)` — закрывает заставку. В качестве параметра указывается ссылка на главное окно приложения;
- `showMessage(<Сообщение>[, <Выравнивание>[, <Цвет>]])` — выводит сообщение. Во втором параметре указывается местоположение надписи в окне. По умолчанию надпись выводится в левом верхнем углу окна. В качестве значения можно через оператор `|` указать комбинацию следующих флагов: `AlignTop` (по верху), `AlignCenter` (по центру вертикали и горизонтали), `AlignBottom` (по низу), `AlignHCenter` (по центру горизонтали), `AlignVCenter` (по центру вертикали), `AlignLeft` (по левой стороне), `AlignRight` (по правой стороне). В третьем параметре указывается цвет текста. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black` (по умолчанию), `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.);
- `clearMessage()` — стирает надпись;
- `setPixmap(<Изображение>)` — позволяет изменить изображение в окне. В качестве параметра указывается экземпляр класса `QPixmap`;
- `pixmap()` — возвращает изображение в виде экземпляра класса `QPixmap`.

Пример кода, выводящего заставку, показан в листинге 17.20. А на рис. 17.6 можно увидеть эту заставку воочию.

Листинг 17.20. Вывод заставки

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtGui, QtWidgets
import time

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
```

```
        QtWidgets.QPushButton.__init__(self)
        self.setText("Закрыть окно")
        self.clicked.connect(QtWidgets.qApp.quit)
    def load_data(self, sp):
        for i in range(1, 11):              # Имитируем процесс
            time.sleep(2)                   # Что-то загружаем
            sp.showMessage("Загрузка данных... {0}%".format(i * 10), ↳
            QtCore.Qt.AlignHCenter | QtCore.Qt.AlignBottom, QtCore.Qt.black)
            QtWidgets.qApp.processEvents()  # Запускаем оборот цикла
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    splash = QtWidgets.QSplashScreen(QtGui.QPixmap("splashscreen.jpg"))
    splash.showMessage("Загрузка данных... 0%", ↳
    QtCore.Qt.AlignHCenter | QtCore.Qt.AlignBottom, QtCore.Qt.black)
    splash.show()                           # Отображаем заставку
    QtWidgets.qApp.processEvents()          # Запускаем оборот цикла
    window = MyWindow()
    window.setWindowTitle("Использование класса QSplashScreen")
    window.resize(300, 30)
    window.load_data(splash)                # Загружаем данные
    window.show()
    splash.finish(window)                   # Скрываем заставку
    sys.exit(app.exec_())
```

Рис. 17.6. Заставка, выводимая на экран кодом из листинга 17.20

17.11. Доступ к документации

Библиотека PyQt включает в себя несколько сотен классов. Понятно, что описать их все в одной книге не представляется возможным, поэтому мы рассмотрим здесь только наиболее часто используемые возможности библиотеки. А чтобы получить полную информацию, следует обратиться к документации.

Самая последняя версия документации в формате HTML доступна по интернет-адресу: **http://pyqt.sourceforge.net/Docs/PyQt5/**. Там рассматриваются основные вопросы PyQt-программирования: работа с сигналами, использование Qt Designer, отличия PyQt5 от PyQt 4 и пр.

> ***Примечание***
>
> Ранее эта документация поставлялась непосредственно в составе PyQt вместе с примерами программирования. Однако теперь, к сожалению, ни того, ни другого в поставке библиотеки нет.

В правом верхнем углу любой страницы документации находится гиперссылка **Classes**, ведущая на страницу со списком всех классов, что имеются в библиотеке. Название каждого класса является ссылкой на страницу с описанием этого класса, где имеется гиперссылка, в свою очередь ведущая на страницу сайта **http://doc.qt.io/** с полным описанием этого класса.

Также в правом верхнем углу любой страницы имеется гиперссылка **Modules**, ведущая на страницу со списком всех модулей, составляющих PyQt. Аналогично, каждое название модуля является гиперссылкой, ведущей на страницу со списком всех классов, которые определены в этом модуле. Названия классов также являются гиперссылками, ведущими на страницы с описаниями этих классов.

> ***Совет***
>
> Увы, но от документации, опубликованной на сайте **http://pyqt.sourceforge.net/Docs/PyQt5/**, немного толку. Поэтому лучше сразу же обратиться к сайту **http://doc.qt.io/**, где приводится полное описание библиотеки Qt, правда, рассчитанное на разработчиков, которые программируют на C++.

ГЛАВА 18

Управление окном приложения

Создать окно и управлять им позволяет класс `QWidget`. Он наследует классы `QObject` и `QPaintDevice` и, в свою очередь, является базовым классом для всех визуальных компонентов, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В этой главе мы рассмотрим методы класса `QWidget` применительно к окну верхнего уровня, однако следует помнить, что те же самые методы можно применять и к любым компонентам. Так, метод, позволяющий управлять размерами окна, можно использовать и для изменения размеров компонента, имеющего родителя. Тем не менее, некоторые методы имеет смысл использовать только для окон верхнего уровня, — например, метод, позволяющий изменить текст в заголовке окна, не имеет смысла использовать в обычных компонентах.

Для создания окна верхнего уровня, помимо класса `QWidget`, можно использовать и другие классы, которые являются его наследниками, — например, `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При использовании класса `QDialog` окно будет выравниваться по центру экрана или родительского окна и иметь в заголовке только две кнопки: **Справка** и **Закрыть**. Кроме того, можно использовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Использование классов `QDialog` и `QMainWindow` имеет свои различия, которые мы рассмотрим в отдельных главах.

18.1. Создание и отображение окна

Самый простой способ создать пустое окно показан в листинге 18.1.

Листинг 18.1. Создание и отображение окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()                 # Создаем окно
window.setWindowTitle("Заголовок окна")  # Указываем заголовок
window.resize(300, 50)                       # Минимальные размеры
window.show()                                # Отображаем окно
sys.exit(app.exec_())
```

Конструктор класса `QWidget` имеет следующий формат:

```
<Объект> = QWidget([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, компонент будет обладать своим собственным окном. Если в параметре `flags` указан тип окна, то компонент, имея родителя, также будет обладать своим собственным окном, но окажется привязан к родителю. Это позволяет, например, создать модальное окно, которое станет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `flags`, мы рассмотрим в следующем разделе.

Указать ссылку на родительский компонент и, возможно, тип окна уже после создания объекта позволяет метод `setParent()`. Формат метода:

```
setParent(<Родитель>[, <Тип окна>])
```

Получить ссылку на родительский компонент можно с помощью метода `parentWidget()`. Если компонент не имеет родителя, возвращается значение `None`.

Для изменения текста в заголовке окна предназначен метод `setWindowTitle()`. Формат метода:

```
setWindowTitle(<Текст, отображаемый в заголовке>)
```

Метод `windowTitle()` позволяет получить текст, выводящийся в заголовке окна.

После создания окна необходимо вызвать метод `show()`, чтобы вывести окно на экран. Для скрытия окна предназначен метод `hide()`. Для отображения и скрытия компонентов можно также пользоваться методом `setVisible(<Флаг>)`. Если параметром этого метода передано значение `True`, компонент будет отображен, а если значение `False` — скрыт. Пример отображения окна:

```
window.setVisible(True)
```

Проверить, видим компонент в настоящее время или нет, позволяет метод `isVisible()`, который возвращает `True`, если компонент видим, и `False` — в противном случае. Кроме того, можно воспользоваться методом `isHidden()` — он возвращает `True`, если компонент скрыт, и `False` — в противном случае.

18.2. Указание типа окна

При использовании класса `QWidget` окно по умолчанию создается с заголовком, в котором расположены значок, при нажатии на который выводится оконное меню, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Закрыть**. Указать другой тип создаваемого окна позволяет метод `setWindowFlags()` или параметр `flags` в конструкторе класса `QWidget`. Обратите внимание, что метод `setWindowFlags()` должен вызываться перед отображением окна. Формат метода:

```
setWindowFlags(<Тип окна>)
```

В параметре `<Тип окна>` можно указать следующие атрибуты из класса `QtCore.Qt`:

- `Widget` — тип по умолчанию для класса `QWidget`;
- `Window` — указывает, что компонент является окном, независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком, в котором расположены кнопки **Свернуть**, **Развернуть** и **Закрыть**. По умолчанию размеры окна можно изменять с помощью мыши;

♦ `Dialog` — диалоговое окно. Выводится с рамкой и заголовком, в котором расположены кнопки **Справка** и **Закрыть**. Размеры окна можно изменять с помощью мыши. Это значение по умолчанию для класса `QDialog`. Пример указания типа для диалогового окна:

```
window.setWindowFlags(QtCore.Qt.Dialog)
```

♦ `Sheet` и `Drawer` — окна в стиле Apple Macintosh;

♦ `Popup` — указывает, что окно представляет собой всплывающее меню. Оно выводится без рамки и заголовка и, кроме того, может отбрасывать тень. Изменить размеры окна с помощью мыши нельзя;

♦ `Tool` — сообщает, что окно представляет собой панель инструментов. Оно выводится с рамкой и заголовком (меньшим по высоте, чем обычное окно), в котором расположена кнопка **Закрыть**. Размеры окна можно изменять с помощью мыши;

♦ `ToolTip` — указывает, что окно представляет собой всплывающую подсказку. Оно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя;

♦ `SplashScreen` — сообщает, что окно представляет собой заставку. Оно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя. Это значение по умолчанию для класса `QSplashScreen`;

♦ `Desktop` — указывает, что окно представляет собой рабочий стол. Оно вообще не отображается на экране;

♦ `SubWindow` — сообщает, что окно представляет собой дочерний компонент, независимо от того, имеет он родителя или нет. Выводится оно с рамкой и заголовком (меньшим по высоте, чем у обычного окна), но без кнопок. Изменить размеры окна с помощью мыши нельзя;

♦ `ForeignWindow` — указывает, что окно создано другим процессом;

♦ `CoverWindow` — окно, представляющее минимизированное приложение на некоторых мобильных платформах.

Получить тип окна в программе позволяет метод `windowType()`.

Для окон верхнего уровня можно через оператор `|` дополнительно указать следующие атрибуты из класса `QtCore.Qt` (здесь упомянуты только наиболее часто используемые атрибуты, полный их список ищите в документации):

♦ `MSWindowsFixedSizeDialogHint` — запрещает изменение размеров окна. Кнопка **Развернуть** в заголовке окна становится неактивной;

♦ `FramelessWindowHint` — убирает рамку и заголовок окна. Изменять размеры окна и перемещать его нельзя;

♦ `NoDropShadowWindowHint` — убирает отбрасываемую окном тень;

♦ `CustomizeWindowHint` — убирает рамку и заголовок окна, но добавляет эффект объемности. Размеры окна можно изменять;

♦ `WindowTitleHint` — добавляет заголовок окна. Выведем для примера окно фиксированного размера с заголовком, в котором находится только текст:

```
window.setWindowFlags(QtCore.Qt.Window |
                      QtCore.Qt.FramelessWindowHint |
                      QtCore.Qt.WindowTitleHint)
```

♦ `WindowSystemMenuHint` — добавляет оконное меню и кнопку **Закрыть**;

- `WindowMinimizeButtonHint` — добавляет в заголовок кнопку **Свернуть**;
- `WindowMaximizeButtonHint` — добавляет в заголовок кнопку **Развернуть**;
- `WindowMinMaxButtonsHint` — добавляет в заголовок кнопки **Свернуть** и **Развернуть**;
- `WindowCloseButtonHint` — добавляет кнопку **Закрыть**;
- `WindowContextHelpButtonHint` — добавляет кнопку **Справка**;
- `WindowStaysOnTopHint` — сообщает системе, что окно всегда должно отображаться поверх всех других окон;
- `WindowStaysOnBottomHint` — сообщает системе, что окно всегда должно быть расположено позади всех других окон.

Получить все установленные флаги из программы позволяет метод `windowFlags()`.

18.3. Изменение и получение размеров окна

Для изменения размеров окна предназначены следующие методы:

- `resize(<Ширина>, <Высота>)` — изменяет текущий размер окна. Если содержимое окна не помещается в установленный размер, то размер будет выбран так, чтобы компоненты поместились без искажения при условии, что используются менеджеры геометрии. Следовательно, заданный размер может не соответствовать реальному размеру окна. Если используется абсолютное позиционирование, компоненты могут оказаться наполовину или полностью за пределами видимой части окна. В качестве параметра можно также указать экземпляр класса `QSize` из модуля `QtCore`:

```
window.resize(100, 70)
window.resize(QtCore.QSize(100, 70))
```

- `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — изменяет одновременно положение компонента и его текущий размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента), а третий и четвертый параметры — ширину и высоту. В качестве параметра можно также указать экземпляр класса `QRect` из модуля `QtCore`:

```
window.setGeometry(100, 100, 100, 70)
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
```

- `setFixedSize(<Ширина>, <Высота>)` — задает фиксированный размер. Изменить размеры окна с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setFixedSize(100, 70)
window.setFixedSize(QtCore.QSize(100, 70))
```

- `setFixedWidth(<Ширина>)` — задает фиксированный размер только по ширине. Изменить ширину окна с помощью мыши нельзя;
- `setFixedHeight(<Высота>)` — задает фиксированный размер только по высоте. Изменить высоту окна с помощью мыши нельзя;
- `setMinimumSize(<Ширина>, <Высота>)` — задает минимальные размеры окна. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setMinimumSize(100, 70)
window.setMinimumSize(QtCore.QSize(100, 70))
```

- `setMinimumWidth(<Ширина>)` и `setMinimumHeight(<Высота>)` — задают минимальную ширину и высоту соответственно;
- `setMaximumSize(<Ширина>, <Высота>)` — задает максимальный размер окна. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setMaximumSize(100, 70)
window.setMaximumSize(QtCore.QSize(100, 70))
```

- `setMaximumWidth(<Ширина>)` и `setMaximumHeight(<Высота>)` — задают максимальную ширину и высоту соответственно;
- `setBaseSize(<Ширина>, <Высота>)` — задает базовые размеры. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setBaseSize(500, 500)
window.setBaseSize(QtCore.QSize(500, 500))
```

- `adjustSize()` — подгоняет размеры компонента под содержимое. При этом учитываются рекомендуемые размеры, возвращаемые методом `sizeHint()`.

Получить размеры позволяют следующие методы:

- `width()` и `height()` — возвращают текущую ширину и высоту соответственно:

```
window.resize(50, 70)
print(window.width(), window.height())              # 50 70
```

- `size()` — возвращает экземпляр класса `QSize`, содержащий текущие размеры:

```
window.resize(50, 70)
print(window.size().width(), window.size().height()) # 50 70
```

- `minimumSize()` — возвращает экземпляр класса `QSize`, содержащий минимальные размеры;
- `minimumWidth()` и `minimumHeight()` — возвращают минимальную ширину и высоту соответственно;
- `maximumSize()` — возвращает экземпляр класса `QSize`, содержащий максимальные размеры;
- `maximumWidth()` и `maximumHeight()` — возвращают максимальную ширину и высоту соответственно;
- `baseSize()` — возвращает экземпляр класса `QSize`, содержащий базовые размеры;
- `sizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемые размеры компонента. Если таковые являются отрицательными, считается, что нет рекомендуемого размера;
- `minimumSizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый минимальный размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого минимального размера;
- `rect()` — возвращает экземпляр класса `QRect`, содержащий координаты и размеры прямоугольника, в который вписан компонент:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.rect()
print(rect.left(), rect.top())     # 0 0
print(rect.width(), rect.height()) # 100 70
```

- `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты относительно родительского компонента:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.geometry()
print(rect.left(), rect.top())     # 100 100
print(rect.width(), rect.height()) # 100 70
```

При изменении и получении размеров окна следует учитывать, что:

- размеры не включают высоту заголовка окна и ширину границ;
- размер компонентов может изменяться в зависимости от настроек стиля. Например, на разных компьютерах может быть задан шрифт разного наименования и размера, поэтому от указания фиксированных размеров лучше отказаться;
- размер окна может изменяться в промежутке между получением значения и действиями, выполняющими обработку этих значений в программе. Например, сразу после получения размера пользователь может изменить размеры окна с помощью мыши.

Чтобы получить размеры окна, включающие высоту заголовка и ширину границ, следует воспользоваться методом `frameSize()`, который возвращает экземпляр класса `QSize`. Обратите внимание, что полные размеры окна доступны только после его отображения, — до этого момента они совпадают с размерами клиентской области окна, без учета высоты заголовка и ширины границ. Пример получения полного размера окна:

```
window.resize(200, 70)                    # Задаем размеры
# ...
window.show()                             # Отображаем окно
print(window.width(), window.height())    # 200 70
print(window.frameSize().width(),
      window.frameSize().height())        # 208 104
```

Чтобы получить координаты окна с учетом высоты заголовка и ширины границ, следует воспользоваться методом `frameGeometry()`. И в этом случае полные размеры окна доступны только после отображения окна. Метод возвращает экземпляр класса `QRect`:

```
window.setGeometry(100, 100, 200, 70)
# ...
window.show()                             # Отображаем окно
rect = window.geometry()
print(rect.left(), rect.top())            # 100 100
print(rect.width(), rect.height())        # 200 70
rect = window.frameGeometry()
print(rect.left(), rect.top())            # 96 70
print(rect.width(), rect.height())        # 208 104
```

18.4. Местоположение окна на экране и управление им

Задать местоположение окна на экране монитора позволяют следующие методы:

- `move(<X>, <Y>)` — задает положение компонента относительно родителя с учетом высоты заголовка и ширины границ. В качестве параметра можно также указать экземпляр класса `QPoint` из модуля `QtCore`.

  Пример вывода окна в левом верхнем углу экрана:

  ```
  window.move(0, 0)
  window.move(QtCore.QPoint(0, 0))
  ```

- setGeometry(<X>, <Y>, <Ширина>, <Высота>) — изменяет одновременно положение компонента и его текущие размеры. Первые два параметра задают координаты левого верхнего угла относительно родительского компонента, а третий и четвертый параметры — ширину и высоту. Обратите внимание, что метод не учитывает высоту заголовка и ширину границ, поэтому, если указать координаты (0, 0), заголовок окна и левая граница окажутся за пределами экрана. В качестве параметра можно также задать экземпляр класса QRect из модуля QtCore:

```
window.setGeometry(100, 100, 100, 70)
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
```

> **Внимание!**
>
> Начало координат расположено в левом верхнем углу. Положительная ось X направлена вправо, а положительная ось Y — вниз.

Получить позицию окна позволяют следующие методы:

- x() и y() — возвращают координаты левого верхнего угла окна относительно родителя по осям X и Y соответственно. Методы учитывают высоту заголовка и ширину границ:

```
window.move(10, 10)
print(window.x(), window.y())              # 10 10
```

- pos() — возвращает экземпляр класса QPoint, содержащий координаты левого верхнего угла окна относительно родителя. Метод учитывает высоту заголовка и ширину границ:

```
window.move(10, 10)
print(window.pos().x(), window.pos().y()) # 10 10
```

- geometry() — возвращает экземпляр класса QRect, содержащий координаты относительно родительского компонента. Метод не учитывает высоту заголовка и ширину границ:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.geometry()
print(rect.left(), rect.top())            # 14 40
print(rect.width(), rect.height())        # 300 100
```

- frameGeometry() — возвращает экземпляр класса QRect, содержащий координаты с учетом высоты заголовка и ширины границ. Полные размеры окна доступны только после отображения окна:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.frameGeometry()
print(rect.left(), rect.top())            # 10 10
print(rect.width(), rect.height())        # 308 134
```

Для отображения окна по центру экрана, у правой или нижней его границы необходимо знать размеры экрана. Для получения размеров экрана вначале следует вызвать статический метод QApplication.desktop(), который возвращает ссылку на компонент рабочего стола, представленный экземпляром класса QDesktopWidget из модуля QtWidgets. Получить размеры экрана позволяют следующие методы этого класса:

- width() — возвращает ширину всего экрана в пикселах;
- height() — возвращает высоту всего экрана в пикселах.

Примеры:

```
desktop = QtGui.QApplication.desktop()
print(desktop.width(), desktop.height()) # 1440 900
```

- `screenGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты всего экрана:

```
desktop = QtGui.QApplication.desktop()
rect = desktop.screenGeometry()
print(rect.left(), rect.top())          # 0 0
print(rect.width(), rect.height())      # 1440 900
```

- `availableGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты только доступной части экрана (без размера панели задач):

```
desktop = QtGui.QApplication.desktop()
rect = desktop.availableGeometry()
print(rect.left(), rect.top())          # 0 0
print(rect.width(), rect.height())      # 1440 818
```

Пример отображения окна приблизительно по центру экрана показан в листинге 18.2.

Листинг 18.2. Вывод окна приблизительно по центру экрана

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
desktop = QtWidgets.QApplication.desktop()
x = (desktop.width() - window.width()) // 2
y = (desktop.height() - window.height()) // 2
window.move(x, y)
window.show()
sys.exit(app.exec_())
```

В этом примере мы воспользовались методами `width()` и `height()`, которые не учитывают высоту заголовка и ширину границ. В большинстве случаев этого способа достаточно. Если же при выравнивании необходима точность, то для получения размеров окна можно воспользоваться методом `frameSize()`. Однако этот метод возвращает корректные значения лишь после отображения окна. Если код выравнивания по центру расположить после вызова метода `show()`, окно вначале отобразится в одном месте экрана, а затем переместится в центр, что вызовет неприятное мелькание. Чтобы исключить такое мелькание, следует вначале отобразить окно за пределами экрана, а затем переместить его в центр экрана (листинг 18.3).

Листинг 18.3. Вывод окна точно по центру экрана

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
```

```
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
window.move(window.width() * -2, 0)
window.show()
desktop = QtWidgets.QApplication.desktop()
x = (desktop.width() - window.frameSize().width()) // 2
y = (desktop.height() - window.frameSize().height()) // 2
window.move(x, y)
sys.exit(app.exec_())
```

Этот способ можно также использовать для выравнивания окна по правому краю экрана. Например, чтобы расположить окно в правом верхнем углу экрана, необходимо заменить код из предыдущего примера, выравнивающий окно по центру, следующим кодом:

```
desktop = QtWidgets.QApplication.desktop()
x = desktop.width() - window.frameSize().width()
window.move(x, 0)
```

Если попробовать вывести окно в правом нижнем углу, может возникнуть проблема, поскольку в операционной системе Windows в нижней части экрана обычно располагается панель задач, и окно частично окажется под ней. Здесь нам пригодится метод `availableGeometry()`, позволяющий получить высоту панели задач, расположенной в нижней части экрана, следующим образом:

```
desktop = QtWidgets.QApplication.desktop()
taskBarHeight = (desktop.screenGeometry().height() -
                 desktop.availableGeometry().height())
```

Следует также заметить, что в некоторых операционных системах панель задач допускается прикреплять к любой стороне экрана. Кроме того, экран может быть разделен на несколько рабочих столов. Все это необходимо учитывать при размещении окна (за более подробной информацией обращайтесь к документации по классу `QDesktopWidget`).

18.5. Указание координат и размеров

В двух предыдущих разделах были упомянуты классы `QPoint`, `QSize` и `QRect`. Класс `QPoint` описывает координаты точки, класс `QSize` — размеры, а класс `QRect` — координаты и размеры прямоугольной области. Все эти классы определены в модуле `QtCore`. Рассмотрим их более подробно.

> ***Примечание***
>
> Классы `QPoint`, `QSize` и `QRect` предназначены для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать классы `QPointF`, `QSizeF` и `QRectF` соответственно. Эти классы также определены в модуле `QtCore`.

18.5.1. Класс *QPoint*: координаты точки

Класс `QPoint` описывает координаты точки. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
<Объект> = QPoint()
<Объект> = QPoint(<X>, <Y>)
<Объект> = QPoint(<QPoint>)
```

Первый конструктор создает экземпляр класса с нулевыми координатами:

```
>>> from PyQt5 import QtCore
>>> p = QtCore.QPoint()
>>> p.x(), p.y()
(0, 0)
```

Второй конструктор позволяет явно указать координаты точки:

```
>>> p = QtCore.QPoint(10, 88)
>>> p.x(), p.y()
(10, 88)
```

Третий конструктор создает новый экземпляр на основе другого экземпляра:

```
>>> p = QtCore.QPoint(QtCore.QPoint(10, 88))
>>> p.x(), p.y()
(10, 88)
```

Через экземпляр класса доступны следующие методы:

- `x()` и `y()` — возвращают координаты по осям `X` и `Y` соответственно;
- `setX(<X>)` и `setY(<Y>)` — задают координаты по осям `X` и `Y` соответственно;
- `isNull()` — возвращает `True`, если координаты равны нулю, и `False` — в противном случае:

  ```
  >>> p = QtCore.QPoint()
  >>> p.isNull()
  True
  >>> p.setX(10); p.setY(88)
  >>> p.x(), p.y()
  (10, 88)
  ```

- `manhattanLength()` — возвращает сумму абсолютных значений координат:

  ```
  >>> QtCore.QPoint(10, 88).manhattanLength()
  98
  ```

Над двумя экземплярами класса `QPoint` можно выполнять операции `+`, `+=`, `-` (минус), `-=`, `==` и `!=`. Для смены знака координат можно воспользоваться унарным оператором `-`. Кроме того, экземпляр класса `QPoint` можно умножить или разделить на вещественное число (операторами `*`, `*=`, `/` и `/=`):

```
>>> p1 = QtCore.QPoint(10, 20); p2 = QtCore.QPoint(5, 9)
>>> p1 + p2, p1 - p2
(PyQt5.QtCore.QPoint(15, 29), PyQt5.QtCore.QPoint(5, 11))
>>> p1 * 2.5, p1 / 2.0
(PyQt5.QtCore.QPoint(25, 50), PyQt5.QtCore.QPoint(5, 10))
>>> -p1, p1 == p2, p1 != p2
(PyQt5.QtCore.QPoint(-10, -20), False, True)
```

18.5.2. Класс *QSize*: размеры прямоугольной области

Класс `QSize` описывает размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
<Объект> = QSize()
<Объект> = QSize(<Ширина>, <Высота>)
<Объект> = QSize(<QSize>)
```

Первый конструктор создает экземпляр класса с отрицательной шириной и высотой. Второй конструктор позволяет явно указать ширину и высоту. Третий конструктор создает новый экземпляр на основе другого экземпляра:

```
>>> from PyQt5 import QtCore
>>> s1=QtCore.QSize(); s2=QtCore.QSize(10, 55); s3=QtCore.QSize(s2)
>>> s1
PyQt5.QtCore.QSize(-1, -1)
>>> s2, s3
(PyQt5.QtCore.QSize(10, 55), PyQt5.QtCore.QSize(10, 55))
```

Через экземпляр класса доступны следующие методы:

- `width()` и `height()` — возвращают ширину и высоту соответственно;
- `setWidth(<Ширина>)` и `setHeight(<Высота>)` — задают ширину и высоту соответственно.

  Примеры:

  ```
  >>> s = QtCore.QSize()
  >>> s.setWidth(10); s.setHeight(55)
  >>> s.width(), s.height()
  (10, 55)
  ```
- `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
- `isValid()` — возвращает `True`, если ширина и высота больше или равны нулю, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если один параметр (ширина или высота) меньше или равен нулю, и `False` — в противном случае;
- `scale()` — производит изменение размеров области в соответствии со значением параметра `<Тип преобразования>`. Метод изменяет текущий объект и ничего не возвращает. Форматы метода:

  ```
  scale(<QSize>, <Тип преобразования>)
  scale(<Ширина>, <Высота>, <Тип преобразования>)
  ```

  В параметре `<Тип преобразования>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

  - `IgnoreAspectRatio` — `0` — свободно изменяет размеры без сохранения пропорций сторон;
  - `KeepAspectRatio` — `1` — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
  - `KeepAspectRatioByExpanding` — `2` — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

Если новая ширина или высота имеет значение `0`, размеры изменяются непосредственно без сохранения пропорций, вне зависимости от значения параметра `<Тип преобразования>`.

Примеры:

```
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.IgnoreAspectRatio); s
PyQt5.QtCore.QSize(70, 60)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatio); s
PyQt5.QtCore.QSize(70, 28)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatioByExpanding); s
PyQt5.QtCore.QSize(150, 60)
```

- `scaled()` — то же самое, что `scale()`, но не изменяет сам объект, а возвращает новый экземпляр класса `QSize`, хранящий измененные размеры:

```
>>> s1 = QtCore.QSize(50, 20)
>>> s2 = s1.scaled(70, 60, QtCore.Qt.IgnoreAspectRatio)
>>> s1, s2
(PyQt5.QtCore.QSize(50, 20), PyQt5.QtCore.QSize(70, 60))
```

- `boundedTo(<QSize>)` — возвращает экземпляр класса `QSize`, который содержит минимальную ширину и высоту из текущих размеров и размеров, указанных в параметре:

```
>>> s = QtCore.QSize(50, 20)
>>> s.boundedTo(QtCore.QSize(400, 5))
PyQt5.QtCore.QSize(50, 5)
>>> s.boundedTo(QtCore.QSize(40, 50))
PyQt5.QtCore.QSize(40, 20)
```

- `expandedTo(<QSize>)` — возвращает экземпляр класса `QSize`, который содержит максимальную ширину и высоту из текущих размеров и размеров, указанных в параметре:

```
>>> s = QtCore.QSize(50, 20)
>>> s.expandedTo(QtCore.QSize(400, 5))
PyQt5.QtCore.QSize(400, 20)
>>> s.expandedTo(QtCore.QSize(40, 50))
PyQt5.QtCore.QSize(50, 50)
```

- `transpose()` — меняет значения местами. Метод изменяет текущий объект и ничего не возвращает:

```
>>> s = QtCore.QSize(50, 20)
>>> s.transpose(); s
PyQt5.QtCore.QSize(20, 50)
```

- `transposed()` — то же самое, что `transpose()`, но не изменяет сам объект, а возвращает новый экземпляр класса `QSize` с измененными размерами:

```
>>> s1 = QtCore.QSize(50, 20)
>>> s2 = s1.transposed()
>>> s1, s2
(PyQt5.QtCore.QSize(50, 20), PyQt5.QtCore.QSize(20, 50))
```

Над двумя экземплярами класса `QSize` можно выполнять операции `+`, `+=`, `–` (минус), `-=`, `==` и `!=`. Кроме того, экземпляр класса `QSize` можно умножить или разделить на вещественное число (операторами `*`, `*=`, `/` и `/=`):

```
>>> s1 = QtCore.QSize(50, 20); s2 = QtCore.QSize(10, 5)
>>> s1 + s2, s1 — s2
(PyQt5.QtCore.QSize(60, 25), PyQt5.QtCore.QSize(40, 15))
>>> s1 * 2.5, s1 / 2
(PyQt5.QtCore.QSize(125, 50), PyQt5.QtCore.QSize(25, 10))
>>> s1 == s2, s1 != s2
(False, True)
```

18.5.3. Класс *QRect*: координаты и размеры прямоугольной области

Класс `QRect` описывает координаты и размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструктора:

```
<Объект> = QRect()
<Объект> = QRect(<left>, <top>, <Ширина>, <Высота>)
<Объект> = QRect(<Координаты левого верхнего угла>, <Размеры>)
<Объект> = QRect(<Координаты левого верхнего угла>,
                 <Координаты правого нижнего угла>)
<Объект> = QRect(<QRect>)
```

Первый конструктор создает экземпляр класса со значениями по умолчанию. Второй и третий конструкторы позволяют указать координаты левого верхнего угла и размеры области. Во втором конструкторе значения указываются отдельно. В третьем конструкторе координаты задаются с помощью класса `QPoint`, а размеры — с помощью класса `QSize`. Четвертый конструктор позволяет указать координаты левого верхнего угла и правого нижнего угла. В качестве значений указываются экземпляры класса `QPoint`. Пятый конструктор создает новый экземпляр на основе другого экземпляра.

Примеры:

```
>>> from PyQt5 import QtCore
>>> r = QtCore.QRect()
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(0, 0, -1, -1, 0, 0)
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QSize(400, 300))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QPoint(409, 314))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> QtCore.QRect(r)
PyQt5.QtCore.QRect(10, 15, 400, 300)
```

Изменить значения уже после создания экземпляра позволяют следующие методы:

- `setLeft(<X1>)`, `setX(<X1>)`, `setTop(<Y1>)` и `setY(<Y1>)` — задают координаты левого верхнего угла по осям `X` и `Y`:

```
>>> r = QtCore.QRect()
>>> r.setLeft(10); r.setTop(55); r
PyQt5.QtCore.QRect(10, 55, -10, -55)
>>> r.setX(12); r.setY(81); r
PyQt5.QtCore.QRect(12, 81, -12, -81)
```

- `setRight(<X2>)` и `setBottom(<Y2>)` — задают координаты правого нижнего угла по осям `X` и `Y`:

```
>>> r = QtCore.QRect()
>>> r.setRight(12); r.setBottom(81); r
PyQt5.QtCore.QRect(0, 0, 13, 82)
```

- `setTopLeft(<QPoint>)` — задает координаты левого верхнего угла;
- `setTopRight(<QPoint>)` — задает координаты правого верхнего угла;
- `setBottomLeft(<QPoint>)` — задает координаты левого нижнего угла;
- `setBottomRight(<QPoint>)` — задает координаты правого нижнего угла.

Примеры:

```
>>> r = QtCore.QRect()
>>> r.setTopLeft(QtCore.QPoint(10, 5))
>>> r.setBottomRight(QtCore.QPoint(39, 19)); r
PyQt5.QtCore.QRect(10, 5, 30, 15)
>>> r.setTopRight(QtCore.QPoint(39, 5))
>>> r.setBottomLeft(QtCore.QPoint(10, 19)); r
PyQt5.QtCore.QRect(10, 5, 30, 15)
```

- `setWidth(<Ширина>)`, `setHeight(<Высота>)` и `setSize(<QSize>)` — задают ширину и высоту области;
- `setRect(<X1>, <Y1>, <Ширина>, <Высота>)` — задает координаты левого верхнего угла и размеры области;
- `setCoords(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты левого верхнего и правого нижнего углов.

Примеры:

```
>>> r = QtCore.QRect()
>>> r.setRect(10, 10, 100, 500); r
PyQt5.QtCore.QRect(10, 10, 100, 500)
>>> r.setCoords(10, 10, 109, 509); r
PyQt5.QtCore.QRect(10, 10, 100, 500)
```

- `marginsAdded(<QMargins>)` — возвращает новый экземпляр класса `QRect`, который представляет текущую область, увеличенную на заданные величины границ. Эти границы указываются в виде экземпляра класса `QMargins` из модуля `QtCore`, конструктор которого имеет следующий формат:

```
QMargins(<Граница слева>, <Граница сверху>, <Граница справа>, <Граница снизу>)
```

Текущая область при этом не изменяется:

```
>>> r1 = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(5, 2, 5, 2)
```

```
>>> r2 = r1.marginsAdded(m)
>>> r2
PyQt5.QtCore.QRect(5, 13, 410, 304)
>>> r1
PyQt5.QtCore.QRect(10, 15, 400, 300)
```

- `marginsRemoved()` — то же самое, что `marginsAdded()`, но уменьшает новую область на заданные величины границ:

```
>>> r1 = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(2, 10, 2, 10)
>>> r2 = r1.marginsRemoved(m)
>>> r2
PyQt5.QtCore.QRect(12, 25, 396, 280)
>>> r1
PyQt5.QtCore.QRect(10, 15, 400, 300)
```

Переместить область при изменении координат позволяют следующие методы:

- `moveTo(<X1>, <Y1>)`, `moveTo(<QPoint>)`, `moveLeft(<X1>)` и `moveTop(<Y1>)` — задают новые координаты левого верхнего угла:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTo(0, 0); r
PyQt5.QtCore.QRect(0, 0, 400, 300)
>>> r.moveTo(QtCore.QPoint(10, 10)); r
PyQt5.QtCore.QRect(10, 10, 400, 300)
>>> r.moveLeft(5); r.moveTop(0); r
PyQt5.QtCore.QRect(5, 0, 400, 300)
```

- `moveRight(<X2>)` и `moveBottom(<Y2>)` — задают новые координаты правого нижнего угла;
- `moveTopLeft(<QPoint>)` — задает новые координаты левого верхнего угла;
- `moveTopRight(<QPoint>)` — задает новые координаты правого верхнего угла;
- `moveBottomLeft(<QPoint>)` — задает новые координаты левого нижнего угла;
- `moveBottomRight(<QPoint>)` — задает новые координаты правого нижнего угла.

  Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTopLeft(QtCore.QPoint(0, 0)); r
PyQt5.QtCore.QRect(0, 0, 400, 300)
>>> r.moveBottomRight(QtCore.QPoint(599, 499)); r
PyQt5.QtCore.QRect(200, 200, 400, 300)
```

- `moveCenter(<QPoint>)` — задает новые координаты центра;
- `translate(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translate(<QPoint>)` — задают новые координаты левого верхнего угла относительно текущего значения координат:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.translate(20, 15); r
PyQt5.QtCore.QRect(20, 15, 400, 300)
>>> r.translate(QtCore.QPoint(10, 5)); r
PyQt5.QtCore.QRect(30, 20, 400, 300)
```

♦ `translated(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translated(<QPoint>)` — аналогичен методу `translate()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий;

♦ `adjust(<X1>, <Y1>, <X2>, <Y2>)` — задает новые координаты левого верхнего и правого нижнего углов относительно текущих значений координат:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.adjust(10, 5, 10, 5); r
PyQt5.QtCore.QRect(10, 5, 400, 300)
```

♦ `adjusted(<X1>, <Y1>, <X2>, <Y2>)` — аналогичен методу `adjust()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий.

Для получения параметров области предназначены следующие методы:

♦ `left()` и `x()` — возвращают координату левого верхнего угла по оси `X`;

♦ `top()` и `y()` — возвращают координату левого верхнего угла по оси `Y`;

♦ `right()` и `bottom()` — возвращают координаты правого нижнего угла по осям `X` и `Y` соответственно;

♦ `width()` и `height()` — возвращают ширину и высоту соответственно;

♦ `size()` — возвращает размеры в виде экземпляра класса `QSize`.

Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.x(), r.y(), r.right(), r.bottom()
(10, 15, 10, 15, 409, 314)
>>> r.width(), r.height(), r.size()
(400, 300, PyQt5.QtCore.QSize(400, 300))
```

♦ `topLeft()` — возвращает координаты левого верхнего угла;

♦ `topRight()` — возвращает координаты правого верхнего угла;

♦ `bottomLeft()` — возвращает координаты левого нижнего угла;

♦ `bottomRight()` — возвращает координаты правого нижнего угла.

Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.topLeft(), r.topRight()
(PyQt5.QtCore.QPoint(10, 15), PyQt5.QtCore.QPoint(409, 15))
>>> r.bottomLeft(), r.bottomRight()
(PyQt5.QtCore.QPoint(10, 314), PyQt5.QtCore.QPoint(409, 314))
```

♦ `center()` — возвращает координаты центра области. Например, вывести окно по центру доступной области экрана можно так:

```
desktop = QtWidgets.QApplication.desktop()
window.move(desktop.availableGeometry().center() -
            window.rect().center())
```

♦ `getRect()` — возвращает кортеж с координатами левого верхнего угла и размерами области;

♦ `getCoords()` — возвращает кортеж с координатами левого верхнего и правого нижнего углов:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.getRect(), r.getCoords()
((10, 15, 400, 300), (10, 15, 409, 314))
```

Прочие методы:

- `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
- `isValid()` — возвращает `True`, если `left() < right()` и `top() < bottom()`, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если `left() > right()` или `top() > bottom()`, и `False` — в противном случае;
- `normalized()` — исправляет ситуацию, при которой `left() > right()` или `top() > bottom()`, и возвращает новый экземпляр класса `QRect`:

```
>>> r = QtCore.QRect(QtCore.QPoint(409, 314), QtCore.QPoint(10, 15))
>>> r
PyQt5.QtCore.QRect(409, 314, -398, -298)
>>> r.normalized()
PyQt5.QtCore.QRect(10, 15, 400, 300)
```

- `contains(<QPoint>[, <Флаг>])` и `contains(<X>, <Y>[, <Флаг>])` — возвращает `True`, если точка с указанными координатами расположена внутри области или на ее границе, и `False` — в противном случае. Если во втором параметре указано значение `True`, то точка должна быть расположена только внутри области, а не на ее границе. Значение параметра по умолчанию — `False`:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(0, 10), r.contains(0, 10, True)
(True, False)
```

- `contains(<QRect>[, <Флаг>])` — возвращает `True`, если указанная область расположена внутри текущей области или на ее краю, и `False` — в противном случае. Если во втором параметре указано значение `True`, то указанная область должна быть расположена только внутри текущей области, а не на ее краю. Значение параметра по умолчанию — `False`:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(QtCore.QRect(0, 0, 20, 5))
True
>>> r.contains(QtCore.QRect(0, 0, 20, 5), True)
False
```

- `intersects(<QRect>)` — возвращает `True`, если указанная область пересекается с текущей областью, и `False` — в противном случае;
- `intersected(<QRect>)` — возвращает область, которая расположена на пересечении текущей и указанной областей:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.intersects(QtCore.QRect(10, 10, 20, 20))
True
>>> r.intersected(QtCore.QRect(10, 10, 20, 20))
PyQt5.QtCore.QRect(10, 10, 10, 10)
```

♦ `united(<QRect>)` — возвращает область, которая охватывает текущую и указанную области:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.united(QtCore.QRect(30, 30, 20, 20))
PyQt5.QtCore.QRect(0, 0, 50, 50)
```

Над двумя экземплярами класса `QRect` можно выполнять операции `&` и `&=` (пересечение), `|` и `|=` (объединение), `in` (проверка на вхождение), `==` и `!=`.

Пример:

```
>>> r1, r2 = QtCore.QRect(0, 0, 20, 20), QtCore.QRect(10, 10, 20, 20)
>>> r1 & r2, r1 | r2
(PyQt5.QtCore.QRect(10, 10, 10, 10), PyQt5.QtCore.QRect(0, 0, 30, 30))
>>> r1 in r2, r1 in QtCore.QRect(0, 0, 30, 30)
(False, True)
>>> r1 == r2, r1 != r2
(False, True)
```

Помимо этого, поддерживаются операторы `+` и `-`, выполняющие увеличение и уменьшение области на заданные величины границ, которые должны быть заданы в виде объекта класса `QMargins`:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(5, 15, 5, 15)
>>> r + m
PyQt5.QtCore.QRect(5, 0, 410, 330)
>>> r - m
PyQt5.QtCore.QRect(15, 30, 390, 270)
```

18.6. Разворачивание и сворачивание окна

В заголовке окна расположены кнопки **Свернуть** и **Развернуть**, с помощью которых можно свернуть окно в значок на панели задач или развернуть его на весь экран. Выполнить подобные действия из программы позволяют следующие методы класса `QWidget`:

♦ `showMinimized()` — сворачивает окно на панель задач. Эквивалентно нажатию кнопки **Свернуть** в заголовке окна;

♦ `showMaximized()` — разворачивает окно до максимального размера. Эквивалентно нажатию кнопки **Развернуть** в заголовке окна;

♦ `showFullScreen()` — включает полноэкранный режим отображения окна. Окно отображается без заголовка и границ;

♦ `showNormal()` — отменяет сворачивание, максимальный размер и полноэкранный режим, возвращая окно к изначальным размерам;

♦ `activateWindow()` — делает окно активным (т. е. имеющим фокус ввода). В Windows, если окно было ранее свернуто в значок на панель задач, оно не будет развернуто в изначальный вид;

♦ `setWindowState(<Флаги>)` — изменяет состояние окна в зависимости от переданных флагов. В качестве параметра указывается комбинация следующих атрибутов из класса `QtCore.Qt` через побитовые операторы:

- `WindowNoState` — нормальное состояние окна;
- `WindowMinimized` — окно свернуто;
- `WindowMaximized` — окно максимально развернуто;
- `WindowFullScreen` — полноэкранный режим;
- `WindowActive` — окно имеет фокус ввода, т. е. является активным.

Например, включить полноэкранный режим можно так:

```
window.setWindowState((window.windowState() &
    ~(QtCore.Qt.WindowMinimized | QtCore.Qt.WindowMaximized))
    | QtCore.Qt.WindowFullScreen)
```

Проверить текущий статус окна позволяют следующие методы:

- `isMinimized()` — возвращает `True`, если окно свернуто, и `False` — в противном случае;
- `isMaximized()` — возвращает `True`, если окно раскрыто до максимальных размеров, и `False` — в противном случае;
- `isFullScreen()` — возвращает `True`, если включен полноэкранный режим, и `False` — в противном случае;
- `isActiveWindow()` — возвращает `True`, если окно имеет фокус ввода, и `False` — в противном случае;
- `windowState()` — возвращает комбинацию флагов, обозначающих текущий статус окна.

Пример проверки использования полноэкранного режима:

```
if window.windowState() & QtCore.Qt.WindowFullScreen:
    print("Полноэкранный режим")
```

Пример разворачивания и сворачивания окна приведен в листинге 18.4.

Листинг 18.4. Разворачивание и сворачивание окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.btnMin = QtWidgets.QPushButton("Свернуть")
        self.btnMax = QtWidgets.QPushButton("Развернуть")
        self.btnFull = QtWidgets.QPushButton("Полный экран")
        self.btnNormal = QtWidgets.QPushButton("Нормальный размер")
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.btnMin)
        vbox.addWidget(self.btnMax)
        vbox.addWidget(self.btnFull)
        vbox.addWidget(self.btnNormal)
        self.setLayout(vbox)
        self.btnMin.clicked.connect(self.on_min)
        self.btnMax.clicked.connect(self.on_max)
        self.btnFull.clicked.connect(self.on_full)
        self.btnNormal.clicked.connect(self.on_normal)
```

```
    def on_min(self):
        self.showMinimized()
    def on_max(self):
        self.showMaximized()
    def on_full(self):
        self.showFullScreen()
    def on_normal(self):
        self.showNormal()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Разворачивание и сворачивание окна")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

18.7. Управление прозрачностью окна

Сделать окно полупрозрачным позволяет метод `setWindowOpacity()` класса `QWidget`. Формат метода:

```
setWindowOpacity(<Вещественное число от 0.0 до 1.0>)
```

Число `0.0` соответствует полностью прозрачному окну, а число `1.0` — отсутствию прозрачности. Для получения степени прозрачности окна из программы предназначен метод `windowOpacity()`. Выведем окно со степенью прозрачности `0.5` (листинг 18.5).

Листинг 18.5. Полупрозрачное окно

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Полупрозрачное окно")
window.resize(300, 100)
window.setWindowOpacity(0.5)
window.show()
print(window.windowOpacity()) # Выведет: 0.4980392156862745
sys.exit(app.exec_())
```

18.8. Модальные окна

Модальным называется окно, которое не позволяет взаимодействовать с другими окнами в том же приложении, — пока модальное окно не будет закрыто, сделать активным другое окно нельзя. Например, если в программе Microsoft Word выбрать пункт меню **Файл | Сохранить как**, откроется модальное диалоговое окно, позволяющее выбрать путь и название

файла, и, пока это окно не будет закрыто, вы не сможете взаимодействовать с главным окном приложения.

Указать, что окно является модальным, позволяет метод `setWindowModality(<Флаг>)` класса `QWidget`. В качестве параметра могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `NonModal` — `0` — окно не является модальным (поведение по умолчанию);
- `WindowModal` — `1` — окно блокирует только родительские окна в пределах иерархии;
- `ApplicationModal` — `2` — окно блокирует все окна в приложении.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение модальности позволяет метод `windowModality()`. Проверить, является ли окно модальным, можно с помощью метода `isModal()` — он возвращает `True`, если окно является модальным, и `False` — в противном случае.

Создадим два независимых окна. В первом окне разместим кнопку, по нажатию которой откроется модальное окно, — оно будет блокировать только первое окно, но не второе. При открытии модального окна отобразим его примерно по центру родительского окна (листинг 18.6).

Листинг 18.6. Модальные окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import sys

def show_modal_window():
    global modalWindow
    modalWindow = QtWidgets.QWidget(window1, QtCore.Qt.Window)
    modalWindow.setWindowTitle("Модальное окно")
    modalWindow.resize(200, 50)
    modalWindow.setWindowModality(QtCore.Qt.WindowModal)
    modalWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
    modalWindow.move(window1.geometry().center() - modalWindow.rect().center() - ↳
    QtCore.QPoint(4, 30))
    modalWindow.show()
app = QtWidgets.QApplication(sys.argv)
window1 = QtWidgets.QWidget()
window1.setWindowTitle("Обычное окно")
window1.resize(300, 100)
button = QtWidgets.QPushButton("Открыть модальное окно")
button.clicked.connect(show_modal_window)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(button)
window1.setLayout(vbox)
window1.show()

window2 = QtWidgets.QWidget()
window2.setWindowTitle("Это окно не будет блокировано при WindowModal")
```

```
window2.resize(500, 100)
window2.show()

sys.exit(app.exec_())
```

Если запустить приложение и нажать кнопку **Открыть модальное окно**, откроется окно, выровненное примерно по центру родительского окна (произвести точное выравнивание вы сможете самостоятельно). При этом получить доступ к родительскому окну можно только после закрытия модального окна, второе же окно блокировано не будет. Если заменить атрибут `WindowModal` атрибутом `ApplicationModal`, оба окна будут блокированы.

Обратите внимание, что в конструктор модального окна мы передали ссылку на первое окно и атрибут `Window`. Если не указать ссылку, то окно блокировано не будет, а если не указать атрибут, окно вообще не откроется. Кроме того, мы объявили переменную `modalWindow` глобальной, иначе при достижении конца функции переменная выйдет из области видимости, и окно будет автоматически удалено. Чтобы объект окна автоматически удалялся при закрытии окна, атрибуту `WA_DeleteOnClose` в методе `setAttribute()` было присвоено значение `True`.

Модальные окна в большинстве случаев являются диалоговыми. Для работы с такими окнами в PyQt предназначен класс `QDialog`, который автоматически выравнивает окно по центру экрана или родительского окна. Кроме того, этот класс предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и выполнить другие действия. Подробно класс `QDialog` мы рассмотрим в *главе 26*.

18.9. Смена значка в заголовке окна

По умолчанию в левом верхнем углу окна отображается стандартный значок. Отобразить другой значок позволяет метод `setWindowIcon()` класса `QWidget`. В качестве параметра метод принимает экземпляр класса `QIcon` из модуля `QtGui` (см. *разд. 24.3.4*).

Чтобы загрузить значок из файла, следует передать путь к файлу конструктору класса `QIcon`. Если указан относительный путь, поиск файла будет производиться относительно текущего рабочего каталога. Получить список поддерживаемых форматов файлов можно с помощью статического метода `supportedImageFormats()` класса `QImageReader`, объявленного в модуле `QtGui`. Метод возвращает список с экземплярами класса `QByteArray`. Получим список поддерживаемых форматов:

```
>>> from PyQt5 import QtGui
>>> for i in QtGui.QImageReader.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Вот результат выполнения этого примера на компьютере одного из авторов:

```
BMP CUR GIF ICNS ICO JPEG JPG PBM PGM PNG PPM SVG SVGZ TGA TIF TIFF WBMP WEBP XBM XPM
```

Если для окна не указать значок, будет использоваться значок приложения, установленный с помощью метода `setWindowIcon()` класса `QApplication`. В качестве параметра метод также принимает экземпляр класса `QIcon`.

Вместо загрузки значка из файла можно воспользоваться одним из встроенных значков. Загрузить стандартный значок позволяет следующий код:

```
ico = window.style().standardIcon(QtWidgets.QStyle.SP_MessageBoxCritical)
window.setWindowIcon(ico)
```

Посмотреть список всех встроенных значков можно в документации к классу `QStyle` (см. **https://doc.qt.io/qt-5/qstyle.html#StandardPixmap-enum**).

В качестве примера создадим значок размером 16 на 16 пикселов в формате PNG и сохраним его в одном каталоге с программой, после чего установим этот значок для окна и всего приложения (листинг 18.7).

Листинг 18.7. Смена значка в заголовке окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtGui, QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Смена значка в заголовке окна")
window.resize(300, 100)
ico = QtGui.QIcon("icon.png")
window.setWindowIcon(ico)              # Значок для окна
app.setWindowIcon(ico)                 # Значок приложения
window.show()
sys.exit(app.exec_())
```

18.10. Изменение цвета фона окна

Чтобы изменить цвет фона окна (или компонента), следует установить палитру с настроенной ролью `Window` (или `Background`). Цветовая палитра содержит цвета для каждой роли и состояния компонента. Указать состояние компонента позволяют следующие атрибуты из класса `QPalette` (модуль `QtGui`):

- `Active` и `Normal` — `0` — компонент активен (окно находится в фокусе ввода);
- `Disabled` — `1` — компонент недоступен;
- `Inactive` — `2` — компонент неактивен (окно находится вне фокуса ввода).

Получить текущую палитру компонента позволяет его метод `palette()`. Чтобы изменить цвет для какой-либо роли и состояния, следует воспользоваться методом `setColor()` класса `QPalette`. Формат метода:

```
setColor([<Состояние>, ]<Роль>, <Цвет>)
```

В параметре `<Роль>` указывается, для какого элемента изменяется цвет. Например, атрибут `Window` (или `Background`) изменяет цвет фона, а `WindowText` (или `Foreground`) — цвет текста. Полный список атрибутов имеется в документации по классу `QPalette` (см. **https://doc.qt.io/qt-5/qpalette.html**).

В параметре `<Цвет>` указывается цвет элемента. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.).

После настройки палитры необходимо вызвать метод `setPalette()` компонента и передать этому методу измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы вклю-

чить перерисовку, необходимо передать значение `True` методу `setAutoFillBackground()` окна.

Изменить цвет фона также можно с помощью CSS-атрибута `background-color`. Для этого следует передать таблицу стилей в метод `setStyleSheet()` компонента. Таблицы стилей могут быть внешними (подключение через командную строку), установленными на уровне приложения (с помощью метода `setStyleSheet()` класса `QApplication`) или установленными на уровне компонента (с помощью метода `setStyleSheet()` класса `QWidget`). Атрибуты, установленные последними, обычно перекрывают значения аналогичных атрибутов, указанных ранее. Если вы занимались веб-программированием, то язык CSS (каскадные таблицы стилей) вам уже знаком, а если нет, придется дополнительно его изучить.

Создадим окно с надписью. Для активного окна установим зеленый цвет, а для неактивного — красный. Цвет фона надписи сделаем белым. Для изменения фона окна используем палитру, а для изменения цвета фона надписи — CSS-атрибут `background-color` (листинг 18.8).

Листинг 18.8. Изменение цвета фона окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtGui, QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Изменение цвета фона окна")
window.resize(300, 100)
pal = window.palette()
pal.setColor(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QColor("#008800"))
pal.setColor(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
             QtGui.QColor("#ff0000"))
window.setPalette(pal)
label = QtWidgets.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignHCenter)
label.setStyleSheet("background-color: #ffffff;")
label.setAutoFillBackground(True)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

18.11. Вывод изображения в качестве фона

В качестве фона окна (или компонента) можно использовать изображение. Для этого необходимо получить текущую палитру компонента с помощью метода `palette()`, а затем вызвать метод `setBrush()` класса `QPalette`. Формат метода:

```
setBrush([<Состояние>, ]<Роль>, <QBrush>)
```

Первые два параметра аналогичны соответствующим параметрам в методе `setColor()`, который мы рассматривали в предыдущем разделе. В третьем параметре указывается кисть — экземпляр класса `QBrush` из модуля `QtGui`. Форматы конструктора класса:

```
<Объект> = QBrush(<Стиль кисти>)
<Объект> = QBrush(<Цвет>[, <Стиль кисти>=SolidPattern])
<Объект> = QBrush(<Цвет>, <QPixmap>)
<Объект> = QBrush(<QPixmap>)
<Объект> = QBrush(<QImage>)
<Объект> = QBrush(<QBrush>)
<Объект> = QBrush(<QGradient>)
```

В параметре `<Стиль кисти>` указываются атрибуты из класса `QtCore.Qt`, задающие стиль кисти, — например: `NoBrush`, `SolidPattern`, `Dense1Pattern`, `Dense2Pattern`, `Dense3Pattern`, `Dense4Pattern`, `Dense5Pattern`, `Dense6Pattern`, `Dense7Pattern`, `CrossPattern` и др. С помощью этого параметра можно сделать цвет сплошным (`SolidPattern`) или имеющим текстуру (например, атрибут `CrossPattern` задает текстуру в виде сетки).

В параметре `<Цвет>` указывается цвет кисти. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). При этом установка сплошного цвета фона окна может выглядеть так:

```
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
    QtGui.QBrush(QtGui.QColor("#008800"), QtCore.Qt.SolidPattern))
window.setPalette(pal)
```

Параметры `<QPixmap>` и `<QImage>` позволяют передать объекты изображений. Конструкторы этих классов принимают путь к файлу — абсолютный или относительный.

Параметр `<QBrush>` позволяет создать кисть на основе другой кисти, а параметр `<QGradient>` — на основе градиента, представленного объектом класса `QGradient` (см. *главу 24*).

После настройки палитры необходимо вызвать метод `setPalette()` и передать ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение `True` в метод `setAutoFillBackground()`.

Указать, какое изображение используется в качестве фона, также можно с помощью CSS-атрибутов `background` и `background-image`. С помощью CSS-атрибута `background-repeat` можно дополнительно указать режим повтора фонового рисунка. Он может принимать значения `repeat` (повтор по горизонтали и вертикали), `repeat-x` (повтор только по горизонтали), `repeat-y` (повтор только по вертикали) и `no-repeat` (не повторяется).

Создадим окно с надписью. Для активного окна установим одно изображение (с помощью изменения палитры), а для надписи — другое (с помощью CSS-атрибута `background-image`) (листинг 18.9).

Листинг 18.9. Использование изображения в качестве фона

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtGui, QtWidgets
import sys
```

```
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Изображение в качестве фона")
window.resize(300, 100)
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QBrush(QtGui.QPixmap("background1.jpg")))
window.setPalette(pal)
label = QtWidgets.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignCenter)
label.setStyleSheet("background-image: url(background2.jpg);")
label.setAutoFillBackground(True)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

18.12. Создание окна произвольной формы

Чтобы создать окно произвольной формы, нужно выполнить следующие шаги:

1. Создать изображение нужной формы с прозрачным фоном и сохранить его, например, в формате PNG.
2. Создать экземпляр класса `QPixmap`, передав конструктору класса абсолютный или относительный путь к изображению.
3. Установить изображение в качестве фона окна с помощью палитры.
4. Отделить альфа-канал с помощью метода `mask()` класса `QPixmap`.
5. Передать получившуюся маску в метод `setMask()` окна.
6. Убрать рамку окна, например, передав комбинацию следующих флагов:

   ```
   QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint
   ```

Если для создания окна используется класс `QLabel`, то вместо установки палитры можно передать экземпляр класса `QPixmap` в метод `setPixmap()`, а маску — в метод `setMask()`.

Для примера создадим круглое окно с кнопкой, с помощью которой можно закрыть окно. Для использования в качестве маски создадим изображение в формате PNG, установим для него прозрачный фон и нарисуем белый круг, занимающий все это изображение. Окно выведем без заголовка и границ (листинг 18.10).

Листинг 18.10. Создание окна произвольной формы

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtGui, QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowFlags(QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint)
```

```
window.setWindowTitle("Создание окна произвольной формы")
window.resize(300, 300)
pixmap = QtGui.QPixmap("mask.png")
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QBrush(pixmap))
pal.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
             QtGui.QBrush(pixmap))
window.setPalette(pal)
window.setMask(pixmap.mask())
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 135)
button.clicked.connect(QtWidgets.qApp.quit)
window.show()
sys.exit(app.exec_())
```

Получившееся окно можно увидеть на рис. 18.1.

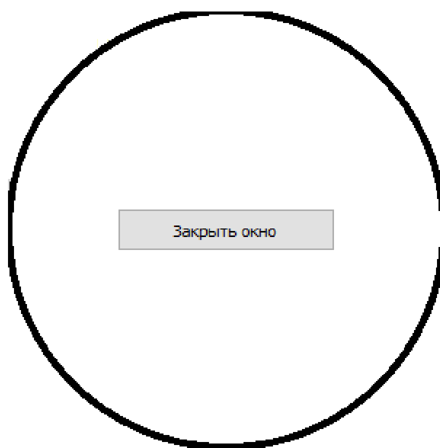

Рис. 18.1. Окно круглой формы

18.13. Всплывающие подсказки

При работе с программой у пользователя могут возникать вопросы о предназначении того или иного компонента. Обычно для информирования пользователя служат надписи, расположенные над компонентом или левее его. Но часто либо место в окне ограничено, либо вывод этих надписей портит весь дизайн окна. В таких случаях принято выводить текст подсказки в отдельном окне без рамки при наведении указателя мыши на компонент. Подсказка автоматически скроется после увода курсора мыши или спустя определенное время.

В PyQt нет необходимости создавать окно с подсказкой самому и следить за перемещениями указателя мыши — весь этот процесс автоматизирован и максимально упрощен. Чтобы создать всплывающие подсказки для окна или любого другого компонента и управлять ими, нужно воспользоваться следующими методами класса `QWidget`:

♦ `setToolTip(<Текст>)` — задает текст всплывающей подсказки. В качестве параметра можно указать простой текст или HTML-код. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;

♦ `toolTip()` — возвращает текст всплывающей подсказки;

♦ `setToolTipDuration(<Время>)` — задает время, в течение которого всплывающая подсказка будет присутствовать на экране. Значение должно быть указано в миллисекундах. Если задать значение `-1`, PyQt будет сама вычислять необходимое время, основываясь на длине текста подсказки (это поведение по умолчанию);

♦ `toolTipDuration()` — возвращает время, в течение которого всплывающая подсказка будет присутствовать на экране;

♦ `setWhatsThis(<Текст>)` — задает текст справки. Обычно этот метод используется для вывода информации большего объема, чем во всплывающей подсказке. У диалоговых окон в заголовке окна есть кнопка **Справка**, по нажатию которой курсор принимает вид стрелки со знаком вопроса, — чтобы в таком случае отобразить текст справки, следует нажать эту кнопку и щелкнуть на компоненте. Можно также сделать компонент активным и нажать комбинацию клавиш <Shift>+<F1>. В качестве параметра можно указать простой текст или HTML-код. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;

♦ `whatsThis()` — возвращает текст справки.

Создадим окно с кнопкой и зададим для них текст всплывающих подсказок и текст справки (листинг 18.11).

Листинг 18.11. Всплывающие подсказки

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Всплывающие подсказки")
window.resize(300, 70)
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.setToolTip("Это всплывающая подсказка для кнопки")
button.setToolTipDuration(3000)
window.setToolTip("Это всплывающая подсказка для окна")
button.setToolTipDuration(5000)
button.setWhatsThis("Это справка для кнопки")
window.setWhatsThis("Это справка для окна")
button.clicked.connect(QtWidgets.qApp.quit)
window.show()
sys.exit(app.exec_())
```

18.14. Программное закрытие окна

В предыдущих разделах для закрытия окна мы использовали слот `quit()` и метод `exit([returnCode=0])` объекта приложения. Однако эти методы не только закрывают текущее окно, но и завершают выполнение всего приложения. Чтобы закрыть только текущее окно, следует воспользоваться методом `close()` класса `QWidget`. Метод возвращает значение `True`, если окно успешно закрыто, и `False` — в противном случае. Закрыть сразу все окна приложения позволяет слот `closeAllWindows()` класса `QApplication`.

Если для окна атрибут `WA_DeleteOnClose` из класса `QtCore.Qt` установлен в значение `True`, после закрытия окна объект окна будет автоматически удален, в противном случае окно просто скрывается. Значение атрибута можно изменить с помощью метода `setAttribute()`:

```
window.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
```

После вызова метода `close()` или нажатия кнопки **Закрыть** в заголовке окна генерируется событие `QEvent.Close`. Если внутри класса определить метод с предопределенным названием `closeEvent()`, это событие можно перехватить и обработать. В качестве параметра метод принимает объект класса `QCloseEvent`, который поддерживает методы `accept()` (позволяет закрыть окно) и `ignore()` (запрещает закрытие окна). Вызывая эти методы, можно контролировать процесс закрытия окна.

В качестве примера закроем окно по нажатию кнопки (листинг 18.12).

Листинг 18.12. Программное закрытие окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Закрытие окна из программы")
window.resize(300, 70)
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.clicked.connect(window.close)
window.show()
sys.exit(app.exec_())
```

> ***Примечание***
>
> Закрыв последнее окно приложения, мы тем самым автоматически завершим и само приложение. Не забываем об этом.

18.15. Использование таблиц стилей CSS для оформления окон

Вернемся к методу `setStyleSheet()`, упомянутому в *разд. 18.10* и предназначенному для задания таблиц стилей у приложений и отдельных элементов управления. С помощью таблиц стилей можно задавать не только цвет фона и фоновое изображение, но и другие параметры оформления.

Метод `setStyleSheet()` поддерживается классами `QWidget` (и всеми его подклассами) и `QApplication`. Следовательно, его можно вызвать у:

- самого приложения — тогда заданные в таблице стилей параметры оформления будут применены ко всем элементам управления всех окон приложения;
- отдельного окна — тогда эти параметры будут действовать в пределах данного окна;
- отдельного элемента управления — тогда они будут действовать только на этот элемент управления.

При указании таблицы стилей у приложения и окна можно использовать привычный нам по CSS формат объявления стилей:

```
<Селектор> {<Определение стилей>}
```

`<Селектор>` записывается в следующем формате:

```
<Основной селектор>[<Дополнительный селектор>][<Псевдокласс>][<Псевдоселектор>]
```

Параметр `<Основной селектор>` указывает на класс элемента управления. Его можно указать в одном из следующих форматов:

- `*` (звездочка) — указывает на все элементы управления (*универсальный селектор*). Например, так можно задать для всех элементов управления зеленый цвет текста:

  ```
  * {color: green;}
  ```

- `<Класс>` — указывает на элементы управления, относящиеся к заданному `<Классу>` и его подклассам. Задание красного цвета текста для всех элементов управления, относящихся к классу `QAbstractButton` и его подклассам, т. е. для командных кнопок, флажков и переключателей, осуществляется так:

  ```
  QAbstractButton {color: red;}
  ```

- `.<Класс>` — указывает только на элементы управления, относящиеся к заданному `<Классу>`, но не к его подклассам. Указание полужирного шрифта для всех элементов управления, относящихся к классу `QPushButton` (командных кнопок), но не для его подклассов, осуществляется так:

  ```
  .QPushButton {font-weight: bold;}
  ```

Параметр `<Дополнительный селектор>` задает дополнительные параметры элемента управления. Его форматы:

- `[<Свойство>="<Значение>"]` — указанное `<Свойство>` элемента управления должно иметь заданное `<Значение>`. Так мы задаем полужирный шрифт для кнопки, чье свойство `default` имеет значение `true`, т. е. для кнопки по умолчанию:

  ```
  QPushButton[default="true"] {font-weight: bold;}
  ```

- `#<Имя>` — указывает на элемент управления, для которого было задано `<Имя>`. `<Имя>` можно задать вызовом у элемента управления метода `setObjectName(<Имя>)`, а получить — вызовом метода `objectName()`. Так выполняется указание красного цвета текста для кнопки с именем `btnRed`:

  ```
  QPushButton#btnRed {color: red;}
  ```

Параметр `<Псевдокласс>` указывает на отдельную составную часть сложного элемента управления. Он записывается в формате `::<Обозначение составной части>`. Вот пример указания графического изображения для кнопки разворачивания раскрывающегося списка (обозначение этой составной части — `down-arrow`):

```
QComboBox::down-arrow {image: url(arrow.png);}
```

Параметр `<Псевдоселектор>` указывает на состояние элемента управления (должна ли быть кнопка нажата, должен ли флажок быть установленным и т. п.). Он может быть записан в двух форматах:

- `:<Обозначение состояния>` — элемент управления должен находиться в указанном состоянии. Вот пример указания белого цвета фона для кнопки, когда она нажата (это состояние имеет обозначение `pressed`):

  ```
  QPushButton:pressed {background-color: white;}
  ```

- `:!<Обозначение состояния>` — элемент управления должен находиться в любом состоянии, кроме указанного. Вот пример указания желтого цвета фона для кнопки, когда она не нажата:

  ```
  QPushButton:!pressed {background-color: yellow;}
  ```

  Можно указать сразу несколько псевдоселекторов, расположив их непосредственно друг за другом — тогда селектор будет указывать на элемент управления, находящийся одновременно во всех состояниях, которые обозначены этими селекторами. Вот пример указания черного цвета фона и белого цвета текста для кнопки, которая нажата и над которой находится курсор мыши (обозначение — `hover`):

  ```
  QPushButton:pressed:hover {color: white; background-color: black;}
  ```

Если нужно указать стиль для элемента управления, вложенного в другой элемент управления, применяется следующий формат указания селектора:

```
<Селектор "внешнего" элемента><Разделитель><Селектор вложенного элемента>
```

Поддерживаются два варианта параметра `<Разделитель>`:

- пробел — `<Вложенный элемент>` не обязательно должен быть вложен непосредственно во `<"Внешний">`. Так мы указываем зеленый цвет фона для всех надписей (`QLabel`), вложенных в группу (`QGroupBox`) и вложенные в нее элементы:

  ```
  QGroupBox QLabel {background-color: green;}
  ```

- `>` — `<Вложенный элемент>` обязательно должен быть вложен непосредственно во `<"Внешний">`. Так мы укажем синий цвет текста для всех надписей, непосредственно вложенных в группу:

  ```
  QGroupBox>QLabel {color: blue;}
  ```

В стиле можно указать сразу несколько селекторов, записав их через запятую — тогда стиль будет применен к элементам управления, на которые указывают эти селекторы. Вот пример задания зеленого цвета фона для кнопок и надписей:

```
QLabel, QPushButton {background-color: green;}
```

В CSS элементы страницы наследуют параметры оформления от их родителей. Но в PyQt это не так. Скажем, если мы укажем для группы красный цвет текста:

```
app.setStyleSheet("QGroupBox {color: red;}")
```

вложенные в эту группу элементы не унаследуют его и будут иметь цвет текста, заданный по умолчанию. Нам придется задать для них нужный цвет явно:

```
app.setStyleSheet("QGroupBox, QGroupBox * {color: red;}")
```

Начиная с версии PyQt 5.7, поддерживается возможность указать библиотеке, что все элементы-потомки должны наследовать параметры оформления у родителя. Для этого достаточно вызвать у класса `QCoreApplication` статический метод `setAttribute`, передав ему

в качестве первого параметра значение атрибута `AA_UseStyleSheetPropagationInWidgetStyles` класса `QtCore.Qt`, а в качестве второго параметра — значение `True`:

```
QtCore.QCoreApplication.setAttribute(↳
QtCore.Qt.AA_UseStyleSheetPropagationInWidgetStyles, True)
```

Чтобы отключить такую возможность, достаточно вызвать этот метод еще раз, указав в нем вторым параметром `False`.

И, наконец, при вызове метода `setStyleSheet()` у элемента управления, для которого следует задать таблицу стилей, в последней не указываются ни селектор, ни фигурные скобки — они просто не нужны.

Отметим, что в случае PyQt, как и в CSS, также действуют правила каскадности. Так, таблица стилей, заданная для окна, имеет больший приоритет, нежели таковая, указанная для приложения, а стиль, что был задан для элемента управления, имеет наивысший приоритет. Помимо этого, более специфические стили имеют больший приоритет, чем менее специфические; так, стиль с селектором, в чей состав входит имя элемента управления, перекроет стиль с селектором любого другого типа.

За более подробным описанием поддерживаемых PyQt псевдоклассов, псевдоселекторов и особенностей указания стилей для отдельных классов элементов управления обращайтесь по интернет-адресу **https://doc.qt.io/qt-5/stylesheet-reference.html**.

Листинг 18.13 показывает пример задания таблиц стилей для элементов управления разными способами. Результат выполнения приведенного в нем кода можно увидеть на рис. 18.2.

Листинг 18.13. Использование таблиц стилей для указания оформления

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
# На уровне приложения задаем синий цвет текста для надписей, вложенных в группы,
# и курсивное начертание текста кнопок
app.setStyleSheet("QGroupBox QLabel {color: blue;} ↳
QPushButton {font-style: italic}")
window = QtWidgets.QWidget()
window.setWindowTitle("Таблицы стилей")
# На уровне окна задаем зеленый цвет текста для надписи с именем first и
# красный цвет текста для надписи, на которую наведен курсор мыши
window.setStyleSheet("QLabel#first {color: green;} QLabel:hover {color: red;}")
window.resize(200, 150)
# Создаем три надписи
lbl1 = QtWidgets.QLabel("Зеленый текст")
# Указываем для первой надписи имя first
lbl1.setObjectName("first")
lbl2 = QtWidgets.QLabel("Полужирный текст")
# Для второй надписи указываем полужирный шрифт
lbl2.setStyleSheet("font-weight: bold")
lbl3 = QtWidgets.QLabel("Синий текст")
# Создаем кнопку
btn = QtWidgets.QPushButton("Курсивный текст")
```

```
# Создаем группу
box = QtWidgets.QGroupBox("Группа")
# Создаем контейнер, помещаем в него третью надпись и вставляем в группу
bbox = QtWidgets.QVBoxLayout()
bbox.addWidget(lbl3)
box.setLayout(bbox)
# Создаем еще один контейнер, помещаем в него две первые надписи, группу и кнопку и
# вставляем в окно
mainbox = QtWidgets.QVBoxLayout()
mainbox.addWidget(lbl1)
mainbox.addWidget(lbl2)
mainbox.addWidget(box)
mainbox.addWidget(btn)
window.setLayout(mainbox)
# Выводим окно и запускаем приложение
window.show()
sys.exit(app.exec_())
```

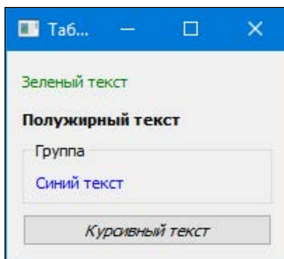

Рис. 18.2. Пример использования таблиц стилей

ГЛАВА 19

Обработка сигналов и событий

При взаимодействии пользователя с окном возникают *события* — своего рода извещения о том, что пользователь выполнил какое-либо действие или в самой системе возникло некоторое условие. В ответ на события система генерирует определенные *сигналы*, которые можно рассматривать как представления системных событий внутри библиотеки PyQt.

Сигналы являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик сигнала, как удалить его, а также уметь правильно обработать событие. Сигналы, которые генерирует тот или иной компонент, мы будем рассматривать при изучении конкретного компонента.

19.1. Назначение обработчиков сигналов

Чтобы обработать какой-либо сигнал, необходимо сопоставить ему функцию или метод класса, который будет вызван при возникновении события и станет его обработчиком.

Каждый сигнал, поддерживаемый классом, соответствует одноименному атрибуту этого класса (отметим, что это именно атрибуты класса, а не атрибуты экземпляра). Так, сигнал `clicked`, генерируемый при щелчке мышью, соответствует атрибуту `clicked`. Каждый такой атрибут хранит экземпляр особого класса, представляющего соответствующий сигнал.

Чтобы назначить сигналу обработчик, следует использовать метод `connect()`, унаследованный от класса `QObject`. Форматы вызова этого метода таковы:

```
<Компонент>.<Сигнал>.connect(<Обработчик>[, <Тип соединения>])
<Компонент>.<Сигнал>[<Тип>].connect(<Обработчик>[, <Тип соединения>])
```

Здесь мы назначаем `<Обработчик>` для параметра `<Сигнал>`, генерируемого параметром `<Компонент>`. В качестве обработчика можно указать:

- ссылку на пользовательскую функцию;
- ссылку на метод класса;
- ссылку на экземпляр класса, в котором определен метод `__call__()`;
- анонимную функцию;
- ссылку на слот класса.

Вот пример назначения функции `on_clicked_button()` в качестве обработчика сигнала `clicked` кнопки `button`:

```
button.clicked.connect(on_clicked_button)
```

Сигналы могут принимать произвольное число параметров, каждый из которых может относиться к любому типу данных. При этом бывает и так, что в классе существуют два сигнала с одинаковыми наименованиями, но разными наборами параметров. Тогда следует дополнительно в квадратных скобках указать `<Тип>` данных, принимаемых сигналом, — либо просто написав его наименование, либо задав его в виде строки. Например, оба следующих выражения назначают обработчик сигнала, принимающего один параметр логического типа:

```
button.clicked[bool].connect(on_clicked_button)
button.clicked["bool"].connect(on_clicked_button)
```

Одному и тому же сигналу можно назначить произвольное количество обработчиков. Это иллюстрирует код из листинга 19.1, где сигналу `clicked` кнопки назначены сразу четыре обработчика.

Листинг 19.1. Назначение сигналу нескольких обработчиков

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

def on_clicked():
    print("Кнопка нажата. Функция on_clicked()")

class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("Кнопка нажата. Метод MyClass.__call__()")
        print("x =", self.x)
    def on_clicked(self):
        print("Кнопка нажата. Метод MyClass.on_clicked()")

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")
# Назначаем обработчиком функцию
button.clicked.connect(on_clicked)
# Назначаем обработчиком метод класса
button.clicked.connect(obj.on_clicked)
# Назначаем обработчиком ссылку на экземпляр класса
button.clicked.connect(MyClass(10))
# Назначаем обработчиком анонимную функцию
button.clicked.connect(lambda: MyClass(5)())
button.show()
sys.exit(app.exec_())
```

В четвертом обработчике мы использовали анонимную функцию. В ней мы сначала создаем объект класса `MyClass`, передав ему в качестве параметра число `5`, после чего сразу же вызываем его как функцию, указав после конструктора пустые скобки:

```
button.clicked.connect(lambda: MyClass(5)())
```

Результат выполнения в окне консоли при щелчке на кнопке:

```
Кнопка нажата. Функция on_clicked()
Кнопка нажата. Метод MyClass.on_clicked()
Кнопка нажата. Метод MyClass.__call__()
x = 10
Кнопка нажата. Метод MyClass.__call__()
x = 5
```

Классы PyQt 5 поддерживают ряд методов, специально предназначенных для использования в качестве обработчиков сигналов. Такие методы называются *слотами*. Например, класс `QApplication` поддерживает слот `quit()`, завершающий текущее приложение. В листинге 19.2 показан код, использующий этот слот.

Листинг 19.2. Использование слота

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Завершить работу")
button.clicked.connect(app.quit)
button.show()
sys.exit(app.exec_())
```

Любой пользовательский метод можно сделать слотом, для чего необходимо перед его определением вставить декоратор `@pyqtSlot()`. Формат декоратора:

```
@QtCore.pyqtSlot(*<Типы данных>, name=None, result=None)
```

В параметре `<Типы данных>` через запятую указываются типы данных параметров, принимаемых слотом, — например: `bool` или `int`. При задании типа данных C++ его название необходимо указать в виде строки. Если метод не принимает параметров, параметр `<Типы данных>` не указывается. В именованном параметре `name` можно передать название слота в виде строки — это название станет использоваться вместо названия метода, а если параметр `name` не задан, название слота будет совпадать с названием метода. Именованный параметр `result` предназначен для указания типа данных, возвращаемых методом, — если параметр не задан, то метод ничего не возвращает. Чтобы создать перегруженную версию слота, декоратор указывается последовательно несколько раз с разными типами данных. Пример использования декоратора `@pyqtSlot()` приведен в листинге 19.3.

Листинг 19.3. Использование декоратора `@pyqtSlot()`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import sys

class MyClass(QtCore.QObject):
    def __init__(self):
        QtCore.QObject.__init__(self)
```

```
    @QtCore.pyqtSlot()
    def on_clicked(self):
        print("Кнопка нажата. Слот on_clicked()")
    @QtCore.pyqtSlot(bool, name="myslot")
    def on_clicked2(self, status):
        print("Кнопка нажата. Слот myslot(bool)", status)

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")
button.clicked.connect(obj.on_clicked)
button.clicked.connect(obj.myslot)
button.show()
sys.exit(app.exec_())
```

PyQt не требует обязательного превращения в слот метода, который будет использоваться как обработчик сигнала. Однако это рекомендуется сделать, т. к. вызов слота в этом случае выполняется быстрее, чем вызов обычного метода.

Необязательный параметр `<Тип соединения>` метода `connect()` определяет тип соединения между сигналом и обработчиком. На этот параметр следует обратить особое внимание при использовании в приложении нескольких потоков, т. к. изменять GUI-поток из другого потока нельзя. В параметре можно указать один из следующих атрибутов класса `QtCore.Qt`:

- `AutoConnection` — `0` — значение по умолчанию. Если источник сигнала и обработчик находятся в одном потоке, то оно эквивалентно значению `DirectConnection`, а если в разных потоках, то — `QueuedConnection`;
- `DirectConnection` — `1` — обработчик вызывается сразу после генерации сигнала и выполняется в потоке его источника;
- `QueuedConnection` — `2` — сигнал помещается в очередь обработки событий, а его обработчик выполняется в потоке приемника сигнала;
- `BlockingQueuedConnection` — `4` — аналогично значению `QueuedConnection` за тем исключением, что поток блокируется на время обработки сигнала. Обратите внимание, что источник и обработчик сигнала обязательно должны быть расположены в разных потоках;
- `UniqueConnection` — `0x80` — указывает, что обработчик можно назначить только один раз. Этот атрибут с помощью оператора `|` может быть объединен с любым из представленных ранее флагов:

```
# Эти два обработчика будут успешно назначены и выполнены
button.clicked.connect(on_clicked)
button.clicked.connect(on_clicked)
# А эти два обработчика назначены не будут
button.clicked.connect(on_clicked, Qt.Core.Qt.AutoConnection | ↳
QtCore.Qt.UniqueConnection)
button.clicked.connect(on_clicked, Qt.Core.Qt.AutoConnection | ↳
QtCore.Qt.UniqueConnection)
# Тем не менее, эти два обработчика будут назначены, поскольку они разные
button.clicked.connect(on_clicked, Qt.Core.Qt.AutoConnection | ↳
QtCore.Qt.UniqueConnection)
button.clicked.connect(obj.on_clicked, Qt.Core.Qt.AutoConnection | ↳
QtCore.Qt.UniqueConnection)
```

19.2. Блокировка и удаление обработчика

Для блокировки и удаления обработчиков предназначены следующие методы класса `QObject`:

- `blockSignals(<Флаг>)` — временно блокирует прием сигналов, если параметр имеет значение `True`, и снимает блокировку, если параметр имеет значение `False`. Метод возвращает логическое представление предыдущего состояния соединения;
- `signalsBlocked()` — возвращает значение `True`, если блокировка сигналов установлена, и `False` — в противном случае;
- `disconnect()` — удаляет обработчик. Форматы метода:

  ```
  <Компонент>.<Сигнал>.disconnect([<Обработчик>])
  <Компонент>.<Сигнал>[<Тип>].disconnect([<Обработчик>])
  ```

  Если параметр `<Обработчик>` не указан, удаляются все обработчики, назначенные ранее, в противном случае удаляется только указанный обработчик. Параметр `<Тип>` указывается лишь в том случае, если существуют сигналы с одинаковыми именами, но принимающие разные параметры:

  ```
  button.clicked.disconnect()
  button.clicked[bool].disconnect(on_clicked_button)
  button.clicked["bool"].disconnect(on_clicked_button)
  ```

Создадим окно с четырьмя кнопками (листинг 19.4). Для кнопки **Нажми меня** назначим обработчик сигнала `clicked`. Чтобы информировать о нажатии кнопки, выведем сообщение в окно консоли. Для кнопок **Блокировать**, **Разблокировать** и **Удалить обработчик** создадим обработчики, которые будут изменять статус обработчика для кнопки **Нажми меня**.

Листинг 19.4. Блокировка и удаление обработчика

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Блокировка и удаление обработчика")
        self.resize(300, 150)
        self.button1 = QtWidgets.QPushButton("Нажми меня")
        self.button2 = QtWidgets.QPushButton("Блокировать")
        self.button3 = QtWidgets.QPushButton("Разблокировать")
        self.button4 = QtWidgets.QPushButton("Удалить обработчик")
        self.button3.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        vbox.addWidget(self.button3)
        vbox.addWidget(self.button4)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
```

```
        self.button2.clicked.connect(self.on_clicked_button2)
        self.button3.clicked.connect(self.on_clicked_button3)
        self.button4.clicked.connect(self.on_clicked_button4)
    @QtCore.pyqtSlot()
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
    @QtCore.pyqtSlot()
    def on_clicked_button2(self):
        self.button1.blockSignals(True)
        self.button2.setEnabled(False)
        self.button3.setEnabled(True)
    @QtCore.pyqtSlot()
    def on_clicked_button3(self):
        self.button1.blockSignals(False)
        self.button2.setEnabled(True)
        self.button3.setEnabled(False)
    @QtCore.pyqtSlot()
    def on_clicked_button4(self):
        self.button1.clicked.disconnect(self.on_clicked_button1)
        self.button2.setEnabled(False)
        self.button3.setEnabled(False)
        self.button4.setEnabled(False)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Если нажать кнопку **Нажми меня**, в окно консоли будет выведена строка `Нажата кнопка button1`. Нажатие кнопки **Блокировать** производит блокировку обработчика — теперь при нажатии кнопки **Нажми меня** никаких сообщений в окно консоли не выводится. Отменить блокировку можно с помощью кнопки **Разблокировать**. Нажатие кнопки **Удалить обработчик** производит полное удаление обработчика — в этом случае, чтобы обрабатывать нажатие кнопки **Нажми меня**, необходимо заново назначить обработчик.

Также можно отключить генерацию сигнала, сделав компонент недоступным с помощью следующих методов из класса `QWidget`:

- `setEnabled(<Флаг>)` — если в параметре указано значение `False`, компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `True`;
- `setDisabled(<Флаг>)` — если в параметре указано значение `True`, компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `False`.

Проверить, доступен компонент или нет, позволяет метод `isEnabled()`. Он возвращает значение `True`, если компонент доступен, и `False` — в противном случае.

19.3. Генерация сигналов

В некоторых случаях необходимо сгенерировать сигнал программно. Например, при заполнении последнего текстового поля и нажатии клавиши <Enter> можно имитировать нажатие кнопки **OK** и тем самым выполнить подтверждение ввода пользователя. Осуществить генерацию сигнала из программы позволяет метод `emit()` класса `QObject`. Форматы этого метода:

```
<Компонент>.<Сигнал>.emit([<Данные>])
<Компонент>.<Сигнал>[<Тип>].emit([<Данные>])
```

Метод `emit()` всегда вызывается у объекта, которому посылается сигнал:

```
button.clicked.emit()
```

Сигналу и, соответственно, его обработчику можно передать данные, указав их в вызове метода `emit()`:

```
button.clicked[bool].emit(False)
button.clicked["bool"].emit(False)
```

Также мы можем создавать свои собственные сигналы. Для этого следует определить в классе атрибут, чье имя совпадет с наименованием сигнала. Отметим, что это должен быть атрибут класса, а не экземпляра. Далее мы присвоим вновь созданному атрибуту результат, возвращенный функцией `pyqtSignal()` из модуля `QtCore`. Формат функции:

```
<Объект сигнала> = pyqtSignal(*<Типы данных>[, name=<Имя сигнала>])
```

В параметре `<Типы данных>` через запятую указываются названия типов данных, передаваемых сигналу, — например: `bool` или `int`:

```
mysignal1 = QtCore.pyqtSignal(int)
mysignal2 = QtCore.pyqtSignal(int, str)
```

При использовании типа данных C++ его название необходимо указать в виде строки:

```
mysignal3 = QtCore.pyqtSignal("QDate")
```

Если сигнал не принимает параметров, параметр `<Типы данных>` не указывается.

Сигнал может иметь несколько перегруженных версий, различающихся количеством и типом принимаемых параметров. В этом случае типы параметров указываются внутри квадратных скобок. Вот пример сигнала, передающего данные типа `int` или `str`:

```
mysignal4 = QtCore.pyqtSignal([int], [str])
```

По умолчанию название создаваемого сигнала будет совпадать с названием атрибута класса. Однако мы можем указать для сигнала другое название, после чего он будет доступен под двумя названиями: совпадающим с именем атрибута класса и заданным нами. Для указания названия сигнала применяется параметр `name`:

```
mysignal = QtCore.pyqtSignal(int, name="mySignal")
```

В качестве примера создадим окно с двумя кнопками (листинг 19.5), которым назначим обработчики сигнала `clicked` (нажатие кнопки). Внутри обработчика щелчка на первой кнопке сгенерируем два сигнала: первый будет имитировать нажатие второй кнопки, а второй станет пользовательским, привязанным к окну. Внутри обработчиков выведем сообщения в окно консоли.

Листинг 19.5. Генерация сигнала из программы

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    mysignal = QtCore.pyqtSignal(int, int)
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Генерация сигнала из программы")
        self.resize(300, 100)
        self.button1 = QtWidgets.QPushButton("Нажми меня")
        self.button2 = QtWidgets.QPushButton("Кнопка 2")
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
        self.mysignal.connect(self.on_mysignal)
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
        # Генерируем сигналы
        self.button2.clicked[bool].emit(False)
        self.mysignal.emit(10, 20)
    def on_clicked_button2(self):
        print("Нажата кнопка button2")
    def on_mysignal(self, x, y):
        print("Обработан пользовательский сигнал mysignal()")
        print("x =", x, "y =", y)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Результат выполнения после нажатия первой кнопки:

```
Нажата кнопка button1
Нажата кнопка button2
Обработан пользовательский сигнал mysignal()
x = 10 y = 20
```

Вместо конкретного типа принимаемого сигналом параметра можно указать тип `QVariant` из модуля `QtCore`. В этом случае при генерации сигнала допускается передавать ему данные любого типа. Вот пример создания и генерирования сигнала, принимающего параметр любого типа:

```
mysignal = QtCore.pyqtSignal(QtCore.QVariant)
. . .
self.mysignal.emit(20)
```

```
self.mysignal.emit("Привет!")
self.mysignal.emit([1, "2"])
```

Сгенерировать сигнал можно не только с помощью метода `emit()`. Некоторые компоненты предоставляют методы, которые посылают сигнал. Например, у кнопок существует метод `click()`. Используя этот метод, инструкцию:

```
button.clicked.emit()
```

можно записать следующим образом:

```
button.click()
```

Более подробно такие методы мы будем рассматривать при изучении конкретных компонентов.

19.4. Передача данных в обработчик

При назначении обработчика в метод `connect()` передается ссылка на функцию или метод. Если после названия функции (метода) указать внутри круглых скобок какой-либо параметр, то это приведет к вызову функции (метода) и вместо ссылки будет передан результат ее выполнения, что вызовет ошибку. Передать данные в обработчик можно следующими способами:

♦ создать анонимную функцию и внутри ее выполнить вызов обработчика с параметрами. Вот пример передачи обработчику числа `10`:

  ```
  self.button1.clicked.connect(lambda : self.on_clicked_button1(10))
  ```

  Если передаваемое обработчику значение вычисляется в процессе выполнения кода, переменную, хранящую это значение, следует указывать в анонимной функции как значение по умолчанию, иначе функции будет передана ссылка на это значение, а не оно само:

  ```
  y = 10
  self.button1.clicked.connect(lambda x=y: self.on_clicked_button1(x))
  ```

♦ передать ссылку на экземпляр класса, внутри которого определен метод `__call__()`. Передаваемое значение указывается в качестве параметра конструктора этого класса:

  ```
  class MyClass():
      def __init__(self, x=0):
          self.x = x
      def __call__(self):
          print("x =", self.x)
  . . .
  self.button1.clicked.connect(MyClass(10))
  ```

♦ передать ссылку на обработчик и данные в функцию `partial()` из модуля `functools`. Формат функции:

  ```
  partial(<Функция>[, *<Неименованные параметры>][, **<Именованные параметры>])
  ```

  Пример передачи параметра в обработчик:

  ```
  from functools import partial
  self.button1.clicked.connect(partial(self.on_clicked_button1, 10))
  ```

Если при генерации сигнала передается предопределенное значение, то оно будет доступно в обработчике после остальных параметров. Назначим обработчик сигнала `clicked`, принимающего логический параметр, и дополнительно передадим число:

```
self.button1.clicked.connect(partial(self.on_clicked_button1, 10))
```

Обработчик будет иметь следующий вид:

```
def on_clicked_button1(self, x, status):
    print("Нажата кнопка button1", x, status)
```

Результат выполнения:

```
Нажата кнопка button1 10 False
```

19.5. Использование таймеров

Таймеры позволяют через заданный интервал времени выполнять метод с предопределенным названием `timerEvent()`. Для назначения таймера используется метод `startTimer()` класса `QObject`. Формат метода:

```
<Id> = <Объект>.startTimer(<Интервал>[, timerType=<Тип таймера>])
```

Параметр `<Интервал>` задает промежуток времени в миллисекундах, по истечении которого выполняется метод `timerEvent()`. Минимальное значение интервала зависит от операционной системы. Если в параметре `<Интервал>` указать значение `0`, таймер будет срабатывать много раз при отсутствии других необработанных событий.

Необязательный параметр `timerType` позволяет указать тип таймера в виде одного из атрибутов класса `QtCore.Qt`:

- `PreciseTimer` — точный таймер, обеспечивающий точность до миллисекунд;
- `CoarseTimer` — таймер, обеспечивающий точность в пределах 5% от заданного интервала (значение по умолчанию);
- `VeryCoarseTimer` — «приблизительный» таймер, обеспечивающий точность до секунд.

Метод `startTimer()` возвращает идентификатор таймера, с помощью которого впоследствии можно остановить таймер.

Формат метода `timerEvent()`:

```
timerEvent(self, <Объект класса QTimerEvent>)
```

Внутри него можно получить идентификатор таймера с помощью метода `timerId()` объекта класса `QTimerEvent`.

Чтобы остановить таймер, необходимо воспользоваться методом `killTimer()` класса `QObject`. Формат метода:

```
<Объект>.killTimer(<Id>)
```

В качестве параметра указывается идентификатор, возвращаемый методом `startTimer()`.

Создадим в окне часы, которые будут отображать текущее системное время с точностью до секунды, и добавим возможность запуска и остановки часов с помощью соответствующих кнопок (листинг 19.6).

Листинг 19.6. Вывод времени в окне с точностью до секунды

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import time

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Часы в окне")
        self.resize(200, 100)
        self.timer_id = 0
        self.label = QtWidgets.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button1 = QtWidgets.QPushButton("Запустить")
        self.button2 = QtWidgets.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
    def on_clicked_button1(self):
        # Задаем интервал в 1 секунду и "приближенный" таймер
        self.timer_id = self.startTimer(1000, timerType = QtCore.Qt.VeryCoarseTimer)
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        if self.timer_id:
            self.killTimer(self.timer_id)
            self.timer_id = 0
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def timerEvent(self, event):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Вместо методов `startTimer()` и `killTimer()` класса `QObject` можно воспользоваться классом `QTimer` из модуля `QtCore`. Конструктор класса имеет следующий формат:

```
<Объект> = QTimer([parent=None])
```

Методы класса:

- setInterval(<Интервал>) — задает промежуток времени в миллисекундах, по истечении которого генерируется сигнал timeout. Минимальное значение интервала зависит от операционной системы. Если в параметре <Интервал> указать значение 0, таймер будет срабатывать много раз при отсутствии других необработанных сигналов;
- start([<Интервал>]) — запускает таймер. В необязательном параметре можно указать промежуток времени в миллисекундах. Если параметр не указан, используется значение, заданное в вызове метода setInterval();
- stop() — останавливает таймер;
- isActive() — возвращает значение True, если таймер запущен, и False — в противном случае;
- timerId() — возвращает идентификатор таймера, если он запущен, и значение -1 — в противном случае;
- remainingTime() — возвращает время, оставшееся до очередного срабатывания таймера, в миллисекундах;
- interval() — возвращает установленный интервал;
- setSingleShot(<Флаг>) — если в параметре указано значение True, таймер сработает только один раз, в противном случае — будет срабатывать многократно;
- isSingleShot() — возвращает значение True, если таймер будет срабатывать только один раз, и False — в противном случае;
- setTimerType(<Тип таймера>) — задает тип таймера, который указывается в том же виде, что и в случае вызова метода startTimer();
- timerType() — возвращает тип таймера.

Переделаем предыдущий пример и используем класс QTimer вместо методов startTimer() и killTimer() (листинг 19.7).

Листинг 19.7. Использование класса QTimer

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import time

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Использование класса QTimer")
        self.resize(200, 100)
        self.label = QtWidgets.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button1 = QtWidgets.QPushButton("Запустить")
        self.button2 = QtWidgets.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
```

```
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.on_timeout);
    def on_clicked_button1(self):
        self.timer.start(1000) # 1 секунда
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        self.timer.stop()
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def on_timeout(self):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Статический метод `singleShot()` класса `QTimer` запускает таймер, настраивает его для однократного срабатывания и указывает функцию или метод, который будет вызван по истечении заданного интервала. Формат вызова этого метода следующий:

```
singleShot(<Интервал>[, <Тип таймера>], <Функция или метод>)
```

Примеры использования этого статического метода:

```
QtCore.QTimer.singleShot(1000, self.on_timeout)
QtCore.QTimer.singleShot(1000, QtWidgets.qApp.quit)
```

19.6. Перехват всех событий

В предыдущих разделах мы рассмотрели обработку сигналов, которые позволяют обмениваться сообщениями между компонентами. Обработка внешних событий — например, нажатий клавиш, — осуществляется несколько иначе. Чтобы обработать событие, необходимо наследовать класс и переопределить в нем метод со специальным названием, — так, чтобы обработать нажатие клавиши, следует переопределить метод `keyPressEvent()`. Специальные методы принимают объект, содержащий детальную информацию о событии, — например, код нажатой клавиши. Все эти объекты являются наследниками класса `QEvent` и наследуют следующие методы:

- `accept()` — устанавливает флаг, разрешающий дальнейшую обработку события. Скажем, если в методе `closeEvent()` вызвать метод `accept()` через объект события, окно будет закрыто. Этот флаг обычно установлен по умолчанию;
- `ignore()` — сбрасывает флаг, разрешающий дальнейшую обработку события. Так, если в методе `closeEvent()` вызвать метод `ignore()` через объект события, окно закрыто не будет;

- `setAccepted(<Флаг>)` — если в качестве параметра указано значение `True`, флаг, разрешающий дальнейшую обработку события, будет установлен (аналогично вызову метода `accept()`), а если `False` — сброшен (аналогично вызову метода `ignore()`);
- `isAccepted()` — возвращает текущее состояние флага, разрешающего дальнейшую обработку события;
- `spontaneous()` — возвращает `True`, если событие сгенерировано системой, и `False` — если внутри программы;
- `type()` — возвращает тип события. Приведем основные типы событий (полный их список содержится в документации по классу `QEvent` на странице **https://doc.qt.io/qt-5/qevent.html**):
  - `0` — нет события;
  - `1` — `Timer` — событие таймера;
  - `2` — `MouseButtonPress` — нажата кнопка мыши;
  - `3` — `MouseButtonRelease` — отпущена кнопка мыши;
  - `4` — `MouseButtonDblClick` — двойной щелчок мышью;
  - `5` — `MouseMove` — перемещение мыши;
  - `6` — `KeyPress` — клавиша на клавиатуре нажата;
  - `7` — `KeyRelease` — клавиша на клавиатуре отпущена;
  - `8` — `FocusIn` — получен фокус ввода с клавиатуры;
  - `9` — `FocusOut` — потерян фокус ввода с клавиатуры;
  - `10` — `Enter` — указатель мыши входит в область компонента;
  - `11` — `Leave` — указатель мыши покидает область компонента;
  - `12` — `Paint` — перерисовка компонента;
  - `13` — `Move` — позиция компонента изменилась;
  - `14` — `Resize` — изменился размер компонента;
  - `17` — `Show` — компонент отображен;
  - `18` — `Hide` — компонент скрыт;
  - `19` — `Close` — окно закрыто;
  - `24` — `WindowActivate` — окно стало активным;
  - `25` — `WindowDeactivate` — окно стало неактивным;
  - `26` — `ShowToParent` — дочерний компонент отображен;
  - `27` — `HideToParent` — дочерний компонент скрыт;
  - `31` — `Wheel` — прокручено колесико мыши;
  - `40` — `Clipboard` — содержимое буфера обмена изменено;
  - `60` — `DragEnter` — указатель мыши входит в область компонента при операции перетаскивания;
  - `61` — `DragMove` — производится операция перетаскивания;
  - `62` — `DragLeave` — указатель мыши покидает область компонента при операции перетаскивания;

* 63 — Drop — операция перетаскивания завершена;
* 68 — ChildAdded — добавлен дочерний компонент;
* 69 — ChildPolished — производится настройка дочернего компонента;
* 71 — ChildRemoved — удален дочерний компонент;
* 74 — PolishRequest — компонент настроен;
* 75 — Polish — производится настройка компонента;
* 82 — ContextMenu — событие контекстного меню;
* 99 — ActivationChange — изменился статус активности окна верхнего уровня;
* 103 — WindowBlocked — окно блокировано модальным окном;
* 104 — WindowUnblocked — текущее окно разблокировано после закрытия модального окна;
* 105 — WindowStateChange — статус окна изменился;
* 121 — ApplicationActivate — приложение стало доступно пользователю;
* 122 — ApplicationDeactivate — приложение стало недоступно пользователю;
* 1000 — User — пользовательское событие;
* 65535 — MaxUser — максимальный идентификатор пользовательского события.

Статический метод registerEventType([<Число>]) позволяет зарегистрировать пользовательский тип события, возвращая идентификатор зарегистрированного события. В качестве параметра можно указать значение в пределах от QEvent.User (1000) до QEvent.MaxUser (65535).

Перехват всех событий осуществляется с помощью метода с предопределенным названием event(self, <event>). Через параметр <event> доступен объект с дополнительной информацией о событии. Этот объект различен для разных типов событий — например, для события MouseButtonPress объект будет экземпляром класса QMouseEvent, а для события KeyPress — экземпляром класса QKeyEvent. Методы, поддерживаемые всеми этими классами, мы рассмотрим в следующих разделах.

Из метода event() следует вернуть в качестве результата значение True, если событие было обработано, и False — в противном случае. Если возвращается значение True, то родительский компонент не получит событие. Чтобы продолжить распространение события, необходимо вызвать метод event() базового класса и передать ему текущий объект события. Обычно это делается так:

```
return QtWidgets.QWidget.event(self, e)
```

В этом случае пользовательский класс является наследником класса QWidget и переопределяет метод event(). Если вы наследуете другой класс, следует вызывать метод именно этого класса. Например, при наследовании класса QLabel инструкция будет выглядеть так:

```
return QtWidgets.QLabel.event(self, e)
```

Пример перехвата нажатия клавиши, щелчка мышью и закрытия окна показан в листинге 19.8.

Листинг 19.8. Перехват всех событий

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
```

```
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def event(self, e):
        if e.type() == QtCore.QEvent.KeyPress:
            print("Нажата клавиша на клавиатуре")
            print("Код:", e.key(), ", текст:", e.text())
        elif e.type() == QtCore.QEvent.Close:
            print("Окно закрыто")
        elif e.type() == QtCore.QEvent.MouseButtonPress:
            print("Щелчок мышью. Координаты:", e.x(), e.y())
        return QtWidgets.QWidget.event(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.7. События окна

Перехватывать все события следует только в самом крайнем случае. В обычных ситуациях нужно использовать методы, предназначенные для обработки определенного события, — например, чтобы обработать закрытие окна, достаточно переопределить метод `closeEvent()`. Методы, которые требуется переопределять для обработки событий окна, мы сейчас и рассмотрим.

19.7.1. Изменение состояния окна

Отследить изменение состояния окна (сворачивание, разворачивание, скрытие и отображение) позволяют следующие методы:

- `changeEvent(self, <event>)` — вызывается при изменении состояния окна, приложения или компонента, заголовка окна, его палитры, статуса активности окна верхнего уровня, языка, локали и др. (полный список смотрите в документации). При обработке события `WindowStateChange` через параметр `<event>` доступен экземпляр класса `QWindowStateChangeEvent`. Этот класс поддерживает только метод `oldState()`, с помощью которого можно получить предыдущее состояние окна;
- `showEvent(self, <event>)` — вызывается при отображении компонента. Через параметр `<event>` доступен экземпляр класса `QShowEvent`;
- `hideEvent(self, <event>)` — вызывается при скрытии компонента. Через параметр `<event>` доступен экземпляр класса `QHideEvent`.

Для примера выведем в консоль текущее состояние окна при его сворачивании, разворачивании, скрытии и отображении (листинг 19.9).

Листинг 19.9. Отслеживание состояния окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def changeEvent(self, e):
        if e.type() == QtCore.QEvent.WindowStateChange:
            if self.isMinimized():
                print("Окно свернуто")
            elif self.isMaximized():
                print("Окно раскрыто до максимальных размеров")
            elif self.isFullScreen():
                print("Полноэкранный режим")
            elif self.isActiveWindow():
                print("Окно находится в фокусе ввода")
        QtWidgets.QWidget.changeEvent(self, e) # Отправляем дальше
    def showEvent(self, e):
        print("Окно отображено")
        QtWidgets.QWidget.showEvent(self, e)   # Отправляем дальше
    def hideEvent(self, e):
        print("Окно скрыто")
        QtWidgets.QWidget.hideEvent(self, e)   # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.7.2. Изменение положения и размеров окна

При перемещении и изменении размеров окна вызываются следующие методы:

- `moveEvent(self, <event>)` — непрерывно вызывается при перемещении окна. Через параметр `<event>` доступен экземпляр класса `QMoveEvent`. Получить координаты окна позволяют следующие методы этого класса:
  - `pos()` — возвращает экземпляр класса `QPoint` с текущими координатами;
  - `oldPos()` — возвращает экземпляр класса `QPoint` с предыдущими координатами;
- `resizeEvent(self, <event>)` — непрерывно вызывается при изменении размеров окна. Через параметр `<event>` доступен экземпляр класса `QResizeEvent`. Получить размеры окна позволяют следующие методы этого класса:
  - `size()` — возвращает экземпляр класса `QSize` с текущими размерами;
  - `oldSize()` — возвращает экземпляр класса `QSize` с предыдущими размерами.

Пример обработки изменения положения окна и его размера показан в листинге 19.10.

Листинг 19.10. Отслеживание смены положения и размеров окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def moveEvent(self, e):
        print("x = {0}; y = {1}".format(e.pos().x(), e.pos().y()))
        QtWidgets.QWidget.moveEvent(self, e)   # Отправляем дальше
    def resizeEvent(self, e):
        print("w = {0}; h = {1}".format(e.size().width(),
                                        e.size().height()))
        QtWidgets.QWidget.resizeEvent(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.7.3. Перерисовка окна или его части

Когда компонент (или часть компонента) становится видимым, требуется выполнить его перерисовку. В этом случае вызывается метод с названием `paintEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QPaintEvent`, который поддерживает следующие методы:

- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, которую требуется перерисовать;
- `region()` — возвращает экземпляр класса `QRegion` с регионом, требующим перерисовки.

С помощью этих методов можно получить координаты области, которая, например, была ранее перекрыта другим окном и теперь вновь оказалась в зоне видимости. Перерисовывая только область, а не весь компонент, можно заметно повысить быстродействие приложения. Следует также заметить, что в целях эффективности последовательность событий перерисовки может быть объединена в одно событие с общей областью перерисовки.

В некоторых случаях перерисовку окна необходимо выполнить вне зависимости от внешних действий системы или пользователя — например, при изменении каких-либо значений требуется обновить график. Вызвать событие перерисовки компонента позволяют следующие методы класса `QWidget`:

- `repaint()` — немедленно вызывает метод `paintEvent()` для перерисовки компонента при условии, что таковой не скрыт, и обновление не было запрещено вызовом метода `setUpdatesEnabled()`. Форматы метода:

```
repaint()
repaint(<X>, <Y>, <Ширина>, <Высота>)
repaint(<QRect>)
repaint(<QRegion>)
```

Первый формат вызова выполняет перерисовку всего компонента, а остальные — только области с указанными координатами;

- `update()` — посылает сообщение о необходимости перерисовки компонента при условии, что компонент не скрыт и обновление не запрещено. Событие будет обработано на следующей итерации основного цикла приложения. Если посылаются сразу несколько сообщений, они объединяются в одно, благодаря чему можно избежать неприятного мерцания. Рекомендуется использовать этот метод вместо метода `repaint()`. Форматы вызова:

```
update()
update(<X>, <Y>, <Ширина>, <Высота>)
update(<QRect>)
update(<QRegion>)
```

19.7.4. Предотвращение закрытия окна

При закрытии окна нажатием кнопки **Закрыть** в его заголовке или вызовом метода `close()` в коде выполняется метод `closeEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QCloseEvent`. Чтобы предотвратить закрытие окна, у объекта события следует вызвать метод `ignore()`, в противном случае — метод `accept()`.

В качестве примера по нажатию кнопки **Закрыть** выведем стандартное диалоговое окно с запросом подтверждения закрытия окна (листинг 19.11). Если пользователь нажмет кнопку **Да**, закроем окно, а если щелкнет кнопку **Нет** или просто закроет диалоговое окно, не будем его закрывать.

Листинг 19.11. Обработка закрытия окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def closeEvent(self, e):
        result = QtWidgets.QMessageBox.question(self,
                    "Подтверждение закрытия окна",
                    "Вы действительно хотите закрыть окно?",
                    QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No,
                    QtWidgets.QMessageBox.No)
        if result == QtWidgets.QMessageBox.Yes:
            e.accept()
            QtWidgets.QWidget.closeEvent(self, e)
        else:
            e.ignore()
```

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.8. События клавиатуры

События клавиатуры обрабатываются очень часто. Например, при нажатии клавиши <F1> выводится справочная информация, при нажатии клавиши <Enter> в однострочном текстовом поле фокус ввода переносится на другой компонент, и т. д. Рассмотрим события клавиатуры подробно.

19.8.1. Установка фокуса ввода

В текущий момент времени только один компонент (или вообще ни одного) может иметь фокус ввода. Для управления фокусом ввода предназначены следующие методы класса `QWidget`:

- `setFocus([<Причина>])` — устанавливает фокус ввода, если компонент находится в активном окне. В параметре `<Причина>` можно указать причину изменения фокуса ввода в виде одного из следующих атрибутов класса `QtCore.Qt`:
  - `MouseFocusReason` — `0` — фокус изменен с помощью мыши;
  - `TabFocusReason` — `1` — нажата клавиша <Tab>;
  - `BacktabFocusReason` — `2` — нажата комбинация клавиш <Shift>+<Tab>;
  - `ActiveWindowFocusReason` — `3` — окно стало активным или неактивным;
  - `PopupFocusReason` — `4` — открыто или закрыто всплывающее окно;
  - `ShortcutFocusReason` — `5` — нажата комбинация клавиш быстрого доступа;
  - `MenuBarFocusReason` — `6` — фокус изменился из-за меню;
  - `OtherFocusReason` — `7` — другая причина;
- `clearFocus()` — убирает фокус ввода с компонента;
- `hasFocus()` — возвращает значение `True`, если компонент имеет фокус ввода, и `False` — в противном случае;
- `focusWidget()` — возвращает ссылку на последний компонент, для которого вызывался метод `setFocus()`. Для компонентов верхнего уровня возвращается ссылка на компонент, который получит фокус после того, как окно станет активным;
- `setFocusProxy(<QWidget>)` — позволяет указать ссылку на компонент, который будет получать фокус ввода вместо текущего компонента;
- `focusProxy()` — возвращает ссылку на компонент, который обрабатывает фокус ввода вместо текущего компонента. Если такого компонента нет, метод возвращает значение `None`;
- `focusNextChild()` — находит следующий компонент, которому можно передать фокус, и передает фокус ему. Фактически работает аналогично нажатию клавиши <Tab>. Возвращает значение `True`, если компонент найден, и `False` — в противном случае;

- focusPreviousChild() — находит предыдущий компонент, которому можно передать фокус, и передает фокус ему. Работает аналогично нажатию комбинации клавиш <Shift>+<Tab>. Возвращает значение True, если компонент найден, и False — в противном случае;
- focusNextPrevChild(<Флаг>) — если в параметре указано значение True, работает аналогично методу focusNextChild(), если указано False — аналогично методу focusPreviousChild(). Возвращает значение True, если компонент найден, и False — в противном случае;
- setTabOrder(<Компонент1>, <Компонент2>) — позволяет задать последовательность смены фокуса при нажатии клавиши <Tab>. Метод является статическим. В параметре <Компонент2> указывается ссылка на компонент, на который переместится фокус с компонента <Компонент1>. Если компонентов много, метод вызывается несколько раз. Вот пример указания цепочки перехода widget1 -> widget2 -> widget3 -> widget4:

```
QtWidgets.QWidget.setTabOrder(widget1, widget2)
QtWidgets.QWidget.setTabOrder(widget2, widget3)
QtWidgets.QWidget.setTabOrder(widget3, widget4)
```

- setFocusPolicy(<Способ>) — задает способ получения фокуса компонентом в виде одного из следующих атрибутов класса QtCore.Qt:
  - NoFocus — 0 — компонент не может получать фокус;
  - TabFocus — 1 — получает фокус с помощью клавиши <Tab>;
  - ClickFocus — 2 — получает фокус с помощью щелчка мышью;
  - StrongFocus — 11 — получает фокус с помощью клавиши <Tab> и щелчка мышью;
  - WheelFocus — 15 — получает фокус с помощью клавиши <Tab>, щелчка мышью и колесика мыши;
- focusPolicy() — возвращает текущий способ получения фокуса;
- grabKeyboard() — захватывает ввод с клавиатуры. Другие компоненты не будут получать события клавиатуры, пока не будет вызван метод releaseKeyboard();
- releaseKeyboard() — освобождает захваченный ранее ввод с клавиатуры.

Получить ссылку на компонент, находящийся в фокусе ввода, позволяет статический метод focusWidget() класса QApplication. Если ни один компонент не имеет фокуса ввода, метод возвращает значение None. Не путайте этот метод с одноименным методом из класса QWidget.

Обработать получение и потерю фокуса ввода позволяют следующие методы класса QWidget:

- focusInEvent(self, <event>) — вызывается при получении фокуса ввода;
- focusOutEvent(self, <event>) — вызывается при потере фокуса ввода.

Через параметр <event> доступен экземпляр класса QFocusEvent, который поддерживает следующие методы:

- gotFocus() — возвращает значение True, если тип события QEvent.FocusIn (получение фокуса ввода), и False — в противном случае;
- lostFocus() — возвращает значение True, если тип события QEvent.FocusOut (потеря фокуса ввода), и False — в противном случае;

- `reason()` — возвращает причину установки фокуса. Значение аналогично значению параметра `<Причина>` в методе `setFocus()`.

Создадим окно с кнопкой и двумя однострочными полями ввода (листинг 19.12). Для полей ввода обработаем получение и потерю фокуса ввода, а по нажатию кнопки установим фокус ввода на второе поле. Кроме того, зададим последовательность перехода при нажатии клавиши <Tab>.

Листинг 19.12. Установка фокуса ввода

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets

class MyLineEdit(QtWidgets.QLineEdit):
    def __init__(self, id, parent=None):
        QtWidgets.QLineEdit.__init__(self, parent)
        self.id = id
    def focusInEvent(self, e):
        print("Получен фокус полем", self.id)
        QtWidgets.QLineEdit.focusInEvent(self, e)
    def focusOutEvent(self, e):
        print("Потерян фокус полем", self.id)
        QtWidgets.QLineEdit.focusOutEvent(self, e)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.button = QtWidgets.QPushButton("Установить фокус на поле 2")
        self.line1 = MyLineEdit(1)
        self.line2 = MyLineEdit(2)
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.button)
        self.vbox.addWidget(self.line1)
        self.vbox.addWidget(self.line2)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
        # Задаем порядок обхода с помощью клавиши <Tab>
        QtWidgets.QWidget.setTabOrder(self.line1, self.line2)
        QtWidgets.QWidget.setTabOrder(self.line2, self.button)
    def on_clicked(self):
        self.line2.setFocus()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.8.2. Назначение клавиш быстрого доступа

Клавиши быстрого доступа (иногда их также называют «горячими» клавишами) позволяют установить фокус ввода с помощью нажатия специальной (например, <Alt> или <Ctrl>) и какой-либо дополнительной клавиши. Если после нажатия клавиш быстрого доступа в фокусе окажется кнопка (или пункт меню), она будет нажата.

Чтобы задать клавиши быстрого доступа, следует в тексте надписи указать символ `&` перед буквой. В этом случае буква, перед которой указан символ `&`, будет — в качестве подсказки пользователю — подчеркнута. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент окажется в фокусе ввода. Некоторые компоненты, например текстовое поле, не имеют надписи. Чтобы задать клавиши быстрого доступа для таких компонентов, необходимо отдельно создать надпись и связать ее с компонентом с помощью метода `setBuddy(<Компонент>)` класса `QLabel`. Если же создание надписи не представляется возможным, можно воспользоваться следующими методами класса `QWidget`:

- `grabShortcut(<Клавиши>[, <Контекст>])` — регистрирует клавиши быстрого доступа и возвращает идентификатор, с помощью которого можно управлять ими в дальнейшем. В параметре `<Клавиши>` указывается экземпляр класса `QKeySequence` из модуля `QtGui`. Создать экземпляр этого класса для комбинации клавиш <Alt>+<E> можно, например, так:

  ```
  QtGui.QKeySequence.mnemonic("&e")
  QtGui.QKeySequence("Alt+e")
  QtGui.QKeySequence(QtCore.Qt.ALT + QtCore.Qt.Key_E)
  ```

  В параметре `<Контекст>` можно указать атрибуты `WidgetShortcut`, `WidgetWithChildrenShortcut`, `WindowShortcut` (значение по умолчанию) и `ApplicationShortcut` класса `QtCore.Qt`;
- `releaseShortcut(<ID>)` — удаляет комбинацию с идентификатором `<ID>`;
- `setShortcutEnabled(<ID>[, <Флаг>])` — если в качестве параметра `<Флаг>` указано `True` (значение по умолчанию), клавиша быстрого доступа с идентификатором `<ID>` разрешена. Значение `False` запрещает использование клавиши быстрого доступа.

При нажатии клавиш быстрого доступа генерируется событие `QEvent.Shortcut`, которое можно обработать в методе `event(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QShortcutEvent`, поддерживающий следующие методы:

- `shortcutId()` — возвращает идентификатор комбинации клавиш;
- `isAmbiguous()` — возвращает значение `True`, если событие отправлено сразу нескольким компонентам, и `False` — в противном случае;
- `key()` — возвращает экземпляр класса `QKeySequence`, представляющий нажатую клавишу быстрого доступа.

Создадим окно с надписью, двумя однострочными текстовыми полями и кнопкой (листинг 19.13). Для первого текстового поля назначим комбинацию клавиш <Alt>+<B> через надпись, а для второго поля — комбинацию <Alt>+<E> с помощью метода `grabShortcut()`. Для кнопки назначим комбинацию клавиш <Alt>+<У> обычным образом — через надпись на кнопке.

Листинг 19.13. Назначение клавиш быстрого доступа разными способами

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtGui, QtWidgets
```

```
class MyLineEdit(QtWidgets.QLineEdit):
    def __init__(self, parent=None):
        QtWidgets.QLineEdit.__init__(self, parent)
        self.id = None
    def event(self, e):
        if e.type() == QtCore.QEvent.Shortcut:
            if self.id == e.shortcutId():
                self.setFocus(QtCore.Qt.ShortcutFocusReason)
                return True
        return QtWidgets.QLineEdit.event(self, e)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.label = QtWidgets.QLabel("Устано&вить фокус на поле 1")
        self.lineEdit1 = QtWidgets.QLineEdit()
        self.label.setBuddy(self.lineEdit1)
        self.lineEdit2 = MyLineEdit()
        self.lineEdit2.id = self.lineEdit2.grabShortcut(
                            QtGui.QKeySequence.mnemonic("&e"))
        self.button = QtWidgets.QPushButton("&Убрать фокус с поля 1")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.lineEdit1)
        self.vbox.addWidget(self.lineEdit2)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
    def on_clicked(self):
        self.lineEdit1.clearFocus()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Помимо рассмотренных способов, для назначения клавиш быстрого доступа можно воспользоваться классом `QShortcut` из модуля `QtWidgets`. В этом случае назначение клавиш для второго текстового поля будет выглядеть так:

```
self.lineEdit2 = QtWidgets.QLineEdit()
self.shc = QtWidgets.QShortcut(QtGui.QKeySequence.mnemonic("&e"), self)
self.shc.setContext(QtCore.Qt.WindowShortcut)
self.shc.activated.connect(self.lineEdit2.setFocus)
```

Назначить комбинацию быстрых клавиш также позволяет класс `QAction` из модуля `QtWidgets`. Назначение клавиш для второго текстового поля выглядит следующим образом:

```
self.lineEdit2 = QtWidgets.QLineEdit()
self.act = QtWidgets.QAction(self)
self.act.setShortcut(QtGui.QKeySequence.mnemonic("&e"))
self.act.triggered.connect(self.lineEdit2.setFocus)
self.addAction(self.act)
```

19.8.3. Нажатие и отпускание клавиши на клавиатуре

При нажатии и отпускании клавиши вызываются следующие методы:

- `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши на клавиатуре. Если клавишу удерживать нажатой, метод будет вызываться многократно, пока клавишу не отпустят;
- `keyReleaseEvent(self, <event>)` — вызывается при отпускании нажатой ранее клавиши.

Через параметр `<event>` доступен экземпляр класса `QKeyEvent`, хранящий дополнительную информацию о событии. Он поддерживает следующие полезные для нас методы (полный их список приведен в документации по классу `QKeyEvent` на странице **https://doc.qt.io/qt-5/qkeyevent.html**):

- `key()` — возвращает код нажатой клавиши. Пример определения клавиши:

  ```
  if e.key() == QtCore.Qt.Key_B:
      print("Нажата клавиша <B>")
  ```

- `text()` — возвращает текстовое представление введенного символа в кодировке Unicode или пустую строку, если была нажата специальная клавиша;
- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с клавишей. Может содержать значения следующих атрибутов из класса `QtCore.Qt` или их комбинацию:
  - `NoModifier` — модификаторы не были нажаты;
  - `ShiftModifier` — была нажата клавиша <Shift>;
  - `ControlModifier` — была нажата клавиша <Ctrl>;
  - `AltModifier` — была нажата клавиша <Alt>;
  - `MetaModifier` — была нажата клавиша <Meta>;
  - `KeypadModifier` — была нажата любая клавиша на дополнительной клавиатуре;
  - `GroupSwitchModifier` — была нажата клавиша <Mode_switch> (только в X11).

  Вот пример определения, была ли нажата клавиша-модификатор <Shift>:

  ```
  if e.modifiers() & QtCore.Qt.ShiftModifier:
      print("Нажата клавиша-модификатор <Shift>")
  ```

- `isAutoRepeat()` — возвращает `True`, если событие было вызвано удержанием клавиши нажатой, и `False` — в противном случае;
- `matches(<QKeySequence.StandardKey>)` — возвращает значение `True`, если была нажата специальная комбинация клавиш, соответствующая указанному значению, и `False` — в противном случае. В качестве значения указываются атрибуты из класса `QKeySequence` — например, `QKeySequence.Copy` для комбинации клавиш <Ctrl>+<C> (Копировать):

```
if e.matches(QtGui.QKeySequence.Copy):
    print("Нажата комбинация <Ctrl>+<C>")
```

Полный список атрибутов содержится в документации по классу QKeySequence (см. **https://doc.qt.io/qt-5/qkeysequence.html#StandardKey-enum**).

При обработке нажатия клавиш следует учитывать, что:

- компонент должен иметь возможность принимать фокус ввода. Некоторые компоненты по умолчанию не могут принимать фокус ввода — например, надпись. Чтобы изменить способ получения фокуса, следует воспользоваться методом setFocusPolicy(<Способ>), который мы рассматривали в *разд. 19.8.1*;
- чтобы захватить эксклюзивный ввод с клавиатуры, следует воспользоваться методом grabKeyboard(), а чтобы освободить ввод — методом releaseKeyboard();
- можно перехватить нажатие любых клавиш, кроме клавиши <Tab> и комбинации <Shift>+<Tab>. Эти клавиши используются для передачи фокуса следующему и предыдущему компоненту соответственно. Перехватить нажатие этих клавиш можно только в методе event(self, <event>);
- если событие обработано, следует вызвать метод accept() объекта события. Чтобы родительский компонент смог получить событие, вместо метода accept() необходимо вызвать метод ignore().

19.9. События мыши

События мыши обрабатываются не реже, чем события клавиатуры. С помощью специальных методов можно обработать нажатие и отпускание кнопки мыши, перемещение указателя, вхождение указателя в область компонента и выхода из этой области. В зависимости от ситуации можно изменить вид указателя — например, при выполнении длительной операции отобразить указатель в виде песочных часов. В этом разделе мы рассмотрим изменение вида указателя мыши как для отдельного компонента, так и для всего приложения.

19.9.1. Нажатие и отпускание кнопки мыши

При нажатии и отпускании кнопки мыши вызываются следующие методы:

- mousePressEvent(self, <event>) — вызывается при нажатии кнопки мыши;
- mouseReleaseEvent(self, <event>) — вызывается при отпускании ранее нажатой кнопки мыши;
- mouseDoubleClickEvent(self, <event>) — вызывается при двойном щелчке мышью в области компонента. Следует учитывать, что двойному щелчку предшествуют другие события. Последовательность событий при двойном щелчке выглядит так:

  ```
  MouseButtonPress
  MouseButtonRelease
  MouseButtonDblClick
  MouseButtonPress
  MouseButtonRelease
  ```

  Задать интервал двойного щелчка позволяет метод setDoubleClickInterval() класса QApplication, а получить его текущее значение — метод doubleClickInterval() того же класса.

Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, хранящий дополнительную информацию о событии. Он поддерживает такие методы:

- `x()` и `y()` — возвращают координаты по осям `X` и `Y` соответственно в пределах области компонента;
- `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами в пределах области компонента;
- `localPos()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах области компонента;
- `globalX()` и `globalY()` — возвращают координаты по осям `X` и `Y` соответственно в пределах экрана;
- `globalPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана;
- `windowPos()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах окна;
- `screenPos()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах экрана;
- `button()` — позволяет определить, щелчок какой кнопкой мыши вызвал событие. Возвращает значение одного из следующих атрибутов класса `QtCore.Qt` (здесь указаны не все атрибуты, полный их список приведен на странице **https://doc.qt.io/qt-5/qt.html#MouseButton-enum**):
  - `NoButton` — `0` — кнопки не нажаты. Это значение возвращается методом `button()` при перемещении указателя мыши;
  - `LeftButton` — `1` — нажата левая кнопка мыши;
  - `RightButton` — `2` — нажата правая кнопка мыши;
  - `MidButton` и `MiddleButton` — `4` — нажата средняя кнопка мыши;
  - `XButton1`, `ExtraButton1` и `BackButton` — `8` — нажата первая из дополнительных кнопок мыши;
  - `XButton2`, `ExtraButton2` и `ForwardButton` — `16` — нажата вторая из дополнительных кнопок мыши;
- `buttons()` — позволяет определить все кнопки, которые нажаты одновременно. Возвращает комбинацию упомянутых ранее атрибутов. Вот пример определения нажатой кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:
    print("Нажата левая кнопка мыши")
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в *разд. 19.8.3*;
- `timestamp()` — возвращает в виде числа отметку системного времени, в которое возникло событие.

Если событие было успешно обработано, следует вызвать метод `accept()` объекта события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` нужно вызвать метод `ignore()`.

Если у компонента атрибут `WA_NoMousePropagation` класса `QtCore.Qt` установлен в `True`, событие мыши не будет передаваться родительскому компоненту. Значение атрибута можно изменить с помощью метода `setAttribute()`, вызванного у этого компонента:

```
button.setAttribute(QtCore.Qt.WA_NoMousePropagation, True)
```

По умолчанию событие мыши перехватывает компонент, над которым был произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне компонента, следует захватить мышь вызовом метода `grabMouse()`. Освободить захваченную ранее мышь позволяет метод `releaseMouse()`.

19.9.2. Перемещение указателя мыши

Чтобы обработать перемещение указателя мыши, необходимо переопределить метод `mouseMoveEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, содержащий дополнительную информацию о событии. Методы этого класса мы уже рассматривали в предыдущем разделе. Следует учитывать, что метод `button()` при перемещении мыши возвращает значение `QtCore.Qt.NoButton`.

По умолчанию метод `mouseMoveEvent()` вызывается только в том случае, если при перемещении удерживается нажатой какая-либо кнопка мыши. Это сделано специально, чтобы не создавать лишних событий при обычном перемещении указателя мыши. Если необходимо обрабатывать любые перемещения указателя в пределах компонента, следует вызвать у этого компонента метод `setMouseTracking()`, которому передать значение `True`. Чтобы обработать все перемещения внутри окна, нужно дополнительно захватить мышь вызовом метода `grabMouse()`.

Метод `pos()` объекта события возвращает позицию точки в системе координат текущего компонента. Чтобы преобразовать координаты точки в систему координат родительского компонента или в глобальную систему координат, нужно воспользоваться следующими методами класса `QWidget`:

- `mapToGlobal(<QPoint>)` — преобразует координаты точки из системы координат компонента в глобальную систему координат. Возвращает экземпляр класса `QPoint`;
- `mapFromGlobal(<QPoint>)` — преобразует координаты точки из глобальной в систему координат компонента. Возвращает экземпляр класса `QPoint`;
- `mapToParent(<QPoint>)` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента. Если компонент не имеет родителя, действует как метод `mapToGlobal()`. Возвращает экземпляр класса `QPoint`;
- `mapFromParent(<QPoint>)` — преобразует координаты точки из системы координат родительского компонента в систему координат текущего компонента. Если компонент не имеет родителя, работает подобно методу `mapFromGlobal()`. Возвращает экземпляр класса `QPoint`;
- `mapTo(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат текущего компонента в систему координат родительского компонента `<QWidget>`. Возвращает экземпляр класса `QPoint`;
- `mapFrom(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат родительского компонента `<QWidget>` в систему координат текущего компонента. Возвращает экземпляр класса `QPoint`.

19.9.3. Наведение и увод указателя

Обработать наведение указателя мыши на компонент и увод его с компонента позволяют следующие методы:

- `enterEvent(self, <event>)` — вызывается при наведении указателя мыши на область компонента;
- `leaveEvent(self, <event>)` — вызывается, когда указатель мыши покидает область компонента.

Через параметр `<event>` доступен экземпляр класса `QEvent`, не несущий никакой дополнительной информации. Вполне достаточно знать, что указатель попал в область компонента или покинул ее.

19.9.4. Прокрутка колесика мыши

Все современные мыши комплектуются колесиком, обычно используемым для прокрутки содержимого компонента. Обработать поворот колесика позволяет метод `wheelEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QWheelEvent`, который позволяет получить дополнительную информацию о событии.

Класс `QWheelEvent` поддерживает методы:

- `angleDelta()` — возвращает угол поворота колесика в градусах, умноженный на 8, в виде экземпляра класса `QPoint`. Это значение может быть положительным или отрицательным — в зависимости от направления поворота. Вот пример определения угла поворота колесика:

  ```
  angle = e.angleDelta() / 8
  angleX = angle.x()
  angleY = angle.y()
  ```

- `pixelDelta()` — возвращает величину поворота колесика в пикселах в виде экземпляра класса `QPoint`. Это значение может быть положительным или отрицательным — в зависимости от направления поворота;
- `x()` и `y()` — возвращают координаты указателя в момент события по осям `X` и `Y` соответственно в пределах области компонента;
- `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами указателя в момент события в пределах области компонента;
- `posF()` — возвращает экземпляр класса `QPointF` с вещественными координатами указателя в момент события в пределах области компонента;
- `globalX()` и `globalY()` — возвращают координаты указателя в момент события по осям `X` и `Y` соответственно в пределах экрана;
- `globalPos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами указателя в момент события в пределах экрана;
- `globalPosF()` — возвращает экземпляр класса `QPointF` с вещественными координатами указателя в момент события в пределах экрана;
- `buttons()` — позволяет определить кнопки, которые нажаты одновременно с поворотом колесика. Возвращает комбинацию значений атрибутов, указанных в описании метода `buttons()` (см. *разд. 19.9.1*). Вот пример определения нажатой кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:
    print("Нажата левая кнопка мыши")
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты одновременно с поворотом колесика. Возможные значения мы уже рассматривали в *разд. 19.8.3*;
- `timestamp()` — возвращает в виде числа отметку системного времени, в которое возникло событие.

Если событие было успешно обработано, необходимо вызвать метод `accept()` объекта события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

19.9.5. Изменение внешнего вида указателя мыши

Для изменения внешнего вида указателя мыши при вхождении его в область компонента предназначены следующие методы класса `QWidget`:

- `setCursor(<Курсор>)` — задает внешний вид указателя мыши для компонента. В качестве параметра указывается экземпляр класса `QCursor` или следующие атрибуты из класса `QtCore.Qt`: `ArrowCursor` (стандартная стрелка), `UpArrowCursor` (стрелка, направленная вверх), `CrossCursor` (крестообразный указатель), `WaitCursor` (песочные часы), `IBeamCursor` (I-образный указатель), `SizeVerCursor` (стрелки, направленные вверх и вниз), `SizeHorCursor` (стрелки, направленные влево и вправо), `SizeBDiagCursor` (стрелки, направленные в правый верхний угол и левый нижний угол), `SizeFDiagCursor` (стрелки, направленные в левый верхний угол и правый нижний угол), `SizeAllCursor` (стрелки, направленные вверх, вниз, влево и вправо), `SplitVCursor` (указатель изменения высоты), `SplitHCursor` (указатель изменения ширины), `PointingHandCursor` (указатель в виде руки), `ForbiddenCursor` (перечеркнутый круг), `OpenHandCursor` (разжатая рука), `ClosedHandCursor` (сжатая рука), `WhatsThisCursor` (стрелка с вопросительным знаком) и `BusyCursor` (стрелка с песочными часами):

  ```
  self.setCursor(QtCore.Qt.WaitCursor)
  ```
- `unsetCursor()` — отменяет установку указателя для компонента. В результате внешний вид указателя мыши будет наследоваться от родительского компонента;
- `cursor()` — возвращает экземпляр класса `QCursor`, представляющий текущий указатель.

Управлять видом указателя для всего приложения сразу можно с помощью следующих статических методов из класса `QApplication`:

- `setOverrideCursor(<Курсор>)` — задает внешний вид указателя мыши для всего приложения. В качестве параметра указывается экземпляр класса `QCursor` или один из ранее упомянутых специальных атрибутов класса `QtCore.Qt`. Для отмены установки необходимо вызвать метод `restoreOverrideCursor()`;
- `restoreOverrideCursor()` — отменяет изменение внешнего вида указателя мыши для всего приложения:

  ```
  QtWidgets.QApplication.setOverrideCursor(QtCore.Qt.WaitCursor)
  # Выполняем длительную операцию
  QtWidgets.QApplication.restoreOverrideCursor()
  ```
- `changeOverrideCursor(<Курсор>)` — изменяет внешний вид указателя мыши для всего приложения. Если до вызова этого метода не вызывался метод `setOverrideCursor()`,

значение будет проигнорировано. В качестве параметра указывается экземпляр класса `QCursor` или один из специальных атрибутов класса `QtCore.Qt`;

- `overrideCursor()` — возвращает экземпляр класса `QCursor`, представляющий текущий указатель, или значение `None`, если таковой не был изменен.

Изменять внешний вид указателя мыши для всего приложения принято на небольшой промежуток времени — обычно на время выполнения какой-либо операции, в процессе которой приложение не может нормально реагировать на действия пользователя. Чтобы информировать об этом пользователя, указатель принято выводить в виде песочных часов (атрибут `WaitCursor`) или стрелки с песочными часами (атрибут `BusyCursor`).

Метод `setOverrideCursor()` может быть вызван несколько раз. В этом случае курсоры помещаются в стек. Каждый вызов метода `restoreOverrideCursor()` удаляет последний курсор, добавленный в стек. Для нормальной работы приложения необходимо вызывать методы `setOverrideCursor()` и `restoreOverrideCursor()` одинаковое количество раз.

Класс `QCursor` позволяет создать объект курсора с изображением любой формы. Чтобы загрузить изображение, следует передать конструктору класса `QPixmap` путь к файлу изображения. Для создания объекта курсора необходимо передать конструктору класса `QCursor` в первом параметре экземпляр класса `QPixmap`, а во втором и третьем параметрах — координаты «горячей» точки будущего курсора. Вот пример создания и установки пользовательского курсора:

```
self.setCursor(QtGui.QCursor(QtGui.QPixmap("cursor.png"), 0, 0))
```

Класс `QCursor` также поддерживает два статических метода:

- `pos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши относительно экрана:

  ```
  p = QtGui.QCursor.pos()
  print(p.x(), p.y())
  ```

- `setPos()` — позволяет задать позицию указателя мыши. Метод имеет два формата: `setPos(<X>, <Y>)` и `setPos(<QPoint>)`.

19.10. Технология drag & drop

Технология drag & drop позволяет обмениваться данными различных типов между компонентами как одного приложения, так и разных приложений, путем перетаскивания и сбрасывания объектов с помощью мыши. Типичным примером использования технологии служит перемещение файлов в Проводнике Windows. Чтобы переместить файл в другой каталог, достаточно нажать левую кнопку мыши над значком файла и, не отпуская кнопки, перетащить файл на значок каталога, а затем отпустить кнопку мыши. Если необходимо скопировать файл, следует дополнительно удерживать нажатой клавишу <Ctrl>.

19.10.1. Запуск перетаскивания

Операция перетаскивания состоит из двух частей: первая часть запускает процесс, а вторая обрабатывает момент сброса объекта. Обе части могут обрабатываться как одним, так и двумя разными приложениями. Запуск перетаскивания осуществляется следующим образом:

1. Внутри метода `mousePressEvent()` запоминаются координаты указателя мыши в момент щелчка ее левой кнопкой.

2. Внутри метода mouseMoveEvent() вычисляется пройденное расстояние или измеряется время операции. Это необходимо для того, чтобы предотвратить случайное перетаскивание. Управлять задержкой позволяют следующие статические методы класса QApplication:
   - setStartDragDistance(<Дистанция>) — задает минимальное расстояние, после прохождения которого будет запущена операция перетаскивания;
   - startDragDistance() — возвращает это расстояние;
   - setStartDragTime(<Время>) — задает время задержки в миллисекундах перед запуском операции перетаскивания;
   - startDragTime() — возвращает это время.
3. Если пройдено минимальное расстояние или истек минимальный промежуток времени, создается экземпляр класса QDrag, и у него вызывается метод exec(), который после завершения операции возвращает действие, выполненное с данными (например, их копирование или перемещение).

Создать экземпляр класса QDrag можно так:

```
<Объект> = QtGui.QDrag(<Ссылка на компонент>)
```

Класс QDrag поддерживает следующие методы:

- exec() — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. Метод имеет два формата:

  ```
  exec([<Действия>=MoveAction])
  exec(<Действия>, <Действие по умолчанию>)
  ```

  В параметре <Действия> указывается комбинация допустимых действий, а в параметре <Действие по умолчанию> — действие, которое осуществляется, если в процессе выполнения операции не были нажаты клавиши-модификаторы. Возможные действия могут быть заданы следующими атрибутами класса QtCore.Qt: CopyAction (1, копирование), MoveAction (2, перемещение), LinkAction (4, создание ссылки), IgnoreAction (0, действие игнорировано), TargetMoveAction (32770):

  ```
  act = drag.exec(QtCore.Qt.MoveAction | QtCore.Qt.CopyAction,
                  QtCore.Qt.MoveAction)
  ```

  Вместо метода exec() можно использовать аналогичный метод exec_(), сохраненный в PyQt 5 для совместимости с кодом, написанным под библиотеку PyQt 4;
- setMimeData(<QMimeData>) — позволяет задать перемещаемые данные. В качестве значения указывается экземпляр класса QMimeData. Вот пример передачи текста:

  ```
  data = QtCore.QMimeData()
  data.setText("Перетаскиваемый текст")
  drag = QtGui.QDrag(self)
  drag.setMimeData(data)
  ```

- mimeData() — возвращает экземпляр класса QMimeData с перемещаемыми данными;
- setPixmap(<QPixmap>) — задает изображение, которое будет перемещаться вместе с указателем мыши. В качестве параметра указывается экземпляр класса QPixmap:

  ```
  drag.setPixmap(QtGui.QPixmap("dd_representer.png"))
  ```

- pixmap() — возвращает экземпляр класса QPixmap с изображением, которое перемещается вместе с указателем;

- `setHotSpot(<QPoint>)` — задает координаты «горячей» точки на перемещаемом изображении. В качестве параметра указывается экземпляр класса `QPoint`:

```
drag.setHotSpot(QtCore.QPoint(20, 20))
```

- `hotSpot()` — возвращает экземпляр класса `QPoint` с координатами «горячей» точки на перемещаемом изображении;
- `setDragCursor(<QPixmap>, <Действие>)` — позволяет изменить внешний вид указателя мыши для действия, указанного во втором параметре. Первым параметром передается экземпляр класса `QPixmap`, который, собственно, станет указателем мыши. Если в первом параметре указан пустой объект класса `QPixmap`, ранее установленный указатель для действия будет отменен. Вот пример изменения указателя для перемещения:

```
drag.setDragCursor(QtGui.QPixmap("move_cursor.png"),
                   QtCore.Qt.MoveAction)
```

- `dragCursor(<Действие>)` — возвращает экземпляр класса `QPixmap`, представляющий указатель мыши для заданного действия;
- `source()` — возвращает ссылку на компонент-источник;
- `target()` — возвращает ссылку на компонент-приемник или значение `None`, если компонент находится в другом приложении;
- `supportedActions()` — возвращает значение, представляющее комбинацию допустимых в текущей операции действий. Возможные действия обозначаются упомянутыми ранее атрибутами класса `QtCore.Qt`;
- `defaultAction()` — возвращает действие по умолчанию в виде одного из перечисленных ранее атрибутов класса `QtCore.Qt`.

Класс `QDrag` поддерживает два сигнала:

- `actionChanged(<Действие>)` — генерируется при изменении действия. Новое действие представляется одним из упомянутых ранее атрибутов класса `QtCore.Qt`;
- `targetChanged(<Компонент>)` — генерируется при изменении принимающего компонента, который представляется экземпляром соответствующего класса.

Вот пример назначения обработчиков сигналов:

```
drag.actionChanged.connect(self.on_action_changed)
drag.targetChanged.connect(self.on_target_changed)
```

19.10.2. Класс *QMimeData*

Перемещаемые данные и сведения о MIME-типе должны быть представлены экземпляром класса `QMimeData`. Его следует передать в метод `setMimeData()` класса `QDrag`. Выражение, создающее экземпляр класса `QMimeData`, выглядит так:

```
data = QtCore.QMimeData()
```

Класс `QMimeData` поддерживает следующие полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-5/qmimedata.html**):

- `setText(<Текст>)` — устанавливает текстовые данные (MIME-тип `text/plain`):

```
data.setText("Перетаскиваемый текст")
```

- `text()` — возвращает текстовые данные;

- `hasText()` — возвращает значение `True`, если объект содержит текстовые данные, и `False` — в противном случае;
- `setHtml(<HTML-текст>)` — устанавливает текстовые данные в формате HTML (MIME-тип `text/html`):

```
data.setHtml("<b>Перетаскиваемый HTML-текст</b>")
```

- `html()` — возвращает текстовые данные в формате HTML;
- `hasHtml()` — возвращает значение `True`, если объект содержит текстовые данные в формате HTML, и `False` — в противном случае;
- `setUrls(<Список URI-адресов>)` — устанавливает список URI-адресов (MIME-тип `text/uri-list`). В качестве значения указывается список с экземплярами класса `QUrl`. С помощью этого MIME-типа можно обработать перетаскивание файлов:

```
data.setUrls([QtCore.QUrl("https://www.google.ru/")])
```

- `urls()` — возвращает список URI-адресов:

```
uri = e.mimeData().urls()[0].toString()
```

- `hasUrls()` — возвращает значение `True`, если объект содержит список URI-адресов, и `False` — в противном случае;
- `setImageData(<Объект изображения>)` — устанавливает изображение (MIME-тип `application/x-qt-image`). В качестве значения можно указать, например, экземпляр класса `QImage` или `QPixmap`:

```
data.setImageData(QtGui.QImage("pixmap.png"))
data.setImageData(QtGui.QPixmap("pixmap.png"))
```

- `imageData()` — возвращает объект изображения (тип возвращаемого объекта зависит от типа объекта, указанного в методе `setImageData()`);
- `hasImage()` — возвращает значение `True`, если объект содержит изображение, и `False` — в противном случае;
- `setData(<MIME-тип>, <Данные>)` — позволяет установить данные произвольного MIME-типа. В первом параметре указывается MIME-тип в виде строки, а во втором параметре — экземпляр класса `QByteArray` с данными. Метод можно вызвать несколько раз с различными MIME-типами. Вот пример передачи текстовых данных:

```
data.setData("text/plain",
             QtCore.QByteArray(bytes("Данные", "utf-8")))
```

- `data(<MIME-тип>)` — возвращает экземпляр класса `QByteArray` с данными, соответствующими указанному MIME-типу;
- `hasFormat(<MIME-тип>)` — возвращает значение `True`, если объект содержит данные указанного MIME-типа, и `False` — в противном случае;
- `formats()` — возвращает список с поддерживаемыми объектом MIME-типами;
- `removeFormat(<MIME-тип>)` — удаляет данные, соответствующие указанному MIME-типу;
- `clear()` — удаляет все данные.

Если необходимо перетаскивать данные какого-либо специфического типа, нужно наследовать класс `QMimeData` и переопределить в нем методы `retrieveData()` и `formats()`. За подробной информацией по этому вопросу обращайтесь к документации.

19.10.3. Обработка сброса

Прежде чем обрабатывать перетаскивание и сбрасывание объекта, необходимо сообщить системе, что компонент может обрабатывать эти события. Для этого внутри конструктора компонента следует вызвать метод `setAcceptDrops()`, унаследованный от класса `QWidget`, и передать этому методу `True`:

```
self.setAcceptDrops(True)
```

Обработка перетаскивания и сброса объекта выполняется следующим образом:

1. Внутри метода `dragEnterEvent()` компонента проверяется MIME-тип перетаскиваемых данных и действие. Если компонент способен обработать сброс этих данных и соглашается с предложенным действием, необходимо вызвать метод `acceptProposedAction()` объекта события. Если нужно изменить действие, методу `setDropAction()` объекта события передается новое действие, а затем у того же объекта вызывается метод `accept()` вместо `acceptProposedAction()`.
2. Если необходимо ограничить область сброса некоторым участком компонента, следует дополнительно определить в нем метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области компонента. При достижении указателем мыши нужного участка компонента следует вызвать метод `accept()` и передать ему экземпляр класса `QRect` с координатами и размером этого участка. В этом случае при перетаскивании внутри участка метод `dragMoveEvent()` повторно вызываться не будет.
3. Внутри метода `dropEvent()` компонента производится обработка сброса.

Обработать события, возникающие в процессе перетаскивания и сбрасывания, позволяют следующие методы класса `QWidget`:

- `dragEnterEvent(self, <event>)` — вызывается, когда перетаскиваемый объект входит в область компонента. Через параметр `<event>` доступен экземпляр класса `QDragEnterEvent`;
- `dragLeaveEvent(self, <event>)` — вызывается, когда перетаскиваемый объект покидает область компонента. Через параметр `<event>` доступен экземпляр класса `QDragLeaveEvent`;
- `dragMoveEvent(self, <event>)` — вызывается при перетаскивании объекта внутри области компонента. Через параметр `<event>` доступен экземпляр класса `QDragMoveEvent`;
- `dropEvent(self, <event>)` — вызывается при сбрасывании объекта в области компонента. Через параметр `<event>` доступен экземпляр класса `QDropEvent`.

Класс `QDragLeaveEvent` наследует класс `QEvent` и не несет никакой дополнительной информации. Достаточно просто знать, что перетаскиваемый объект покинул область компонента.

Цепочка наследования остальных классов выглядит так:

```
QEvent — QDropEvent — QDragMoveEvent — QDragEnterEvent
```

Класс `QDragEnterEvent` не содержит собственных методов, но наследует все методы классов `QDropEvent` и `QDragMoveEvent`.

Класс `QDropEvent` поддерживает следующие методы:

- `mimeData()` — возвращает экземпляр класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе;
- `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами сбрасывания объекта;

- posF() — возвращает экземпляр класса QPointF с вещественными координатами сбрасывания объекта;
- possibleActions() — возвращает комбинацию возможных действий при сбрасывании. Вот пример определения значений:

```
if e.possibleActions() & QtCore.Qt.MoveAction:
    print("MoveAction")
if e.possibleActions() & QtCore.Qt.CopyAction:
    print("CopyAction")
```

- proposedAction() — возвращает действие по умолчанию при сбрасывании;
- acceptProposedAction() — сообщает о готовности принять переносимые данные и согласии с действием, возвращаемым методом proposedAction(). Метод acceptProposedAction() (или метод accept(), поддерживаемый классом QDragMoveEvent) необходимо вызвать внутри метода dragEnterEvent(), иначе метод dropEvent() вызван не будет;
- setDropAction(<Действие>) — позволяет указать другое действие при сбрасывании. После изменения действия следует вызвать метод accept(), а не acceptProposedAction();
- dropAction() — возвращает действие, которое должно быть выполнено при сбрасывании. Оно может не совпадать со значением, возвращаемым методом proposedAction(), если действие было изменено с помощью метода setDropAction();
- keyboardModifiers() — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в *разд. 19.8.3*;
- mouseButtons() — позволяет определить кнопки мыши, которые были нажаты в процессе переноса данных;
- source() — возвращает ссылку на компонент внутри приложения, являющийся источником события, или значение None, если данные переносятся из другого приложения.

Теперь рассмотрим методы класса QDragMoveEvent:

- accept([<QRect>]) — сообщает о согласии с дальнейшей обработкой события. В качестве параметра можно указать экземпляр класса QRect с координатами и размерами прямоугольной области, в которой будет доступно сбрасывание;
- ignore([<QRect>]) — отменяет операцию переноса данных. В качестве параметра можно указать экземпляр класса QRect с координатами и размерами прямоугольной области, в которой сбрасывание запрещено;
- answerRect() — возвращает экземпляр класса QRect с координатами и размерами прямоугольной области, в которой произойдет сбрасывание, если событие будет принято.

Некоторые компоненты в PyQt уже поддерживают технологию drag & drop — так, в однострочное текстовое поле можно перетащить текст из другого приложения. Поэтому, прежде чем изобретать свой «велосипед», убедитесь, что поддержка технологии в компоненте не реализована.

19.11. Работа с буфером обмена

Помимо технологии drag & drop, для обмена данными между приложениями используется буфер обмена — одно приложение помещает данные в буфер обмена, а второе приложение

(или то же самое) может их извлечь. Получить ссылку на глобальный объект буфера обмена позволяет статический метод `clipboard()` класса `QApplication`:

```
clipboard = QtWidgets.QApplication.clipboard()
```

Класс `QClipboard` поддерживает следующие методы:

- `setText(<Текст>)` — заносит текст в буфер обмена:

  ```
  clipboard.setText("Текст")
  ```

- `text()` — возвращает из буфера обмена текст или пустую строку;
- `text(<Тип>)` — возвращает кортеж из двух строк: первая хранит текст из буфера обмена, вторая — название типа. В параметре `<Тип>` могут быть указаны значения `"plain"` (простой текст), `"html"` (HTML-код) или пустая строка (любой тип);
- `setImage(<QImage>)` — заносит в буфер обмена изображение, представленное экземпляром класса `QImage`:

  ```
  clipboard.setImage(QtGui.QImage("image.jpg"))
  ```

- `image()` — возвращает из буфера обмена изображение, представленное экземпляром класса `QImage`, или пустой экземпляр этого класса;
- `setPixmap(<QPixmap>)` — заносит в буфер обмена изображение, представленное экземпляром класса `QPixmap`:

  ```
  clipboard.setPixmap(QtGui.QPixmap("image.jpg"))
  ```

- `pixmap()` — возвращает из буфера обмена изображение, представленное экземпляром класса `QPixmap`, или пустой экземпляр этого класса;
- `setMimeData(<QMimeData>)` — позволяет сохранить в буфере данные любого типа, представленные экземпляром класса `QMimeData` (см. *разд. 19.10.2*);
- `mimeData([<Режим>])` — возвращает данные, представленные экземпляром класса `QMimeData`;
- `clear()` — очищает буфер обмена.

Отследить изменение состояния буфера обмена позволяет сигнал `dataChanged`. Назначить обработчик этого сигнала можно так:

```
QtWidgets.qApp.clipboard().dataChanged.connect(on_change_clipboard)
```

19.12. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы компоненту. Для этого необходимо создать класс, который является наследником класса `QObject`, и переопределить в нем метод `eventFilter(self, <Объект>, <event>)`. Через параметр `<Объект>` доступна ссылка на компонент, а через параметр `<event>` — на объект с дополнительной информацией о событии. Этот объект различен для разных типов событий — так, для события `MouseButtonPress` объект будет экземпляром класса `QMouseEvent`, а для события `KeyPress` — экземпляром класса `QKeyEvent`. Из метода `eventFilter()` следует вернуть значение `True`, если событие не должно быть передано дальше, и `False` — в противном случае. Вот пример такого класса-фильтра, перехватывающего нажатие клавиши <B>:

```
class MyFilter(QtCore.QObject):
    def __init__(self, parent=None):
        QtCore.QObject.__init__(self, parent)
```

```
    def eventFilter(self, obj, e):
        if e.type() == QtCore.QEvent.KeyPress:
            if e.key() == QtCore.Qt.Key_B:
                print("Событие от клавиши <B> не дойдет до компонента")
                return True
        return QtCore.QObject.eventFilter(self, obj, e)
```

Далее следует создать экземпляр этого класса, передав в конструктор ссылку на компонент, а затем вызвать у того же компонента метод `installEventFilter()`, передав в качестве единственного параметра ссылку на объект фильтра. Вот пример установки фильтра для надписи:

```
self.label.installEventFilter(MyFilter(self.label))
```

Метод `installEventFilter()` можно вызвать несколько раз, передавая ссылку на разные объекты фильтров. В этом случае первым будет вызван фильтр, который был добавлен последним. Кроме того, один фильтр можно установить сразу в нескольких компонентах. Ссылка на компонент, который является источником события, доступна через второй параметр метода `eventFilter()`.

Удалить фильтр позволяет метод `removeEventFilter(<Фильтр>)`, вызываемый у компонента, для которого был назначен этот фильтр. Если таковой не был установлен, при вызове метода ничего не произойдет.

19.13. Искусственные события

Для создания искусственных событий применяются следующие статические методы из класса `QCoreApplication`:

- `sendEvent(<QObject>, <QEvent>)` — немедленно посылает событие компоненту и возвращает результат выполнения обработчика;
- `postEvent(<QObject>, <QEvent>[, priority=NormalEventPriority])` — добавляет событие в очередь. Параметром `priority` можно передать приоритет события, использовав один из следующих атрибутов класса `QtCore.Qt`: `HighEventPriority` (`1`, высокий приоритет), `NormalEventPriority` (`0`, обычный приоритет — значение по умолчанию) и `LowEventPriority` (`-1`, низкий приоритет). Этот метод является потокобезопасным, следовательно, его можно использовать в многопоточных приложениях для обмена событиями между потоками.

В параметре `<QObject>` указывается ссылка на объект, которому посылается событие, а в параметре `<QEvent>` — объект события. Последний может быть экземпляром как стандартного (например, `QMouseEvent`), так и пользовательского класса, являющегося наследником класса `QEvent`. Вот пример отправки события `QEvent.MouseButtonPress` компоненту `label`:

```
e = QtGui.QMouseEvent(QtCore.QEvent.MouseButtonPress,
                      QtCore.QPointF(5, 5), QtCore.Qt.LeftButton,
                      QtCore.Qt.LeftButton, QtCore.Qt.NoModifier)
QtCore.QCoreApplication.sendEvent(self.label, e)
```

Для отправки пользовательского события необходимо создать класс, наследующий `QEvent`. В этом классе следует зарегистрировать пользовательское событие с помощью статического метода `registerEventType()` и сохранить идентификатор события в атрибуте класса:

```
class MyEvent(QtCore.QEvent):
    idType = QtCore.QEvent.registerEventType()
    def __init__(self, data):
        QtCore.QEvent.__init__(self, MyEvent.idType)
        self.data = data
    def get_data(self):
        return self.data
```

Вот пример отправки события класса `MyEvent` компоненту `label`:

```
QtCore.QCoreApplication.sendEvent(self.label, MyEvent("512"))
```

Обработать пользовательское событие можно с помощью методов `event(self, <event>)` или `customEvent(self, <event>)`:

```
def customEvent(self, e):
    if e.type() == MyEvent.idType:
        self.setText("Получены данные: {0}".format(e.get_data()))
```

ГЛАВА 20

Размещение компонентов в окнах

При размещении в окне нескольких компонентов возникает вопрос их взаимного расположения и минимальных размеров. Следует помнить, что по умолчанию размеры окна можно изменять, а значит, необходимо перехватывать событие изменения размеров и производить перерасчет позиции и размера каждого компонента. Библиотека PyQt избавляет нас от лишних проблем и предоставляет множество компонентов-контейнеров, которые производят такой перерасчет автоматически. Все, что от нас требуется, это выбрать нужный контейнер, добавить в него компоненты в определенном порядке, а затем поместить контейнер в окно или в другой контейнер.

20.1. Абсолютное позиционирование

Прежде чем изучать контейнеры, рассмотрим возможность абсолютного позиционирования компонентов в окне. Итак, если при создании компонента указана ссылка на родительский компонент, то он выводится в позицию с координатами `(0, 0)`. Иными словами, если мы добавим несколько компонентов, то все они отобразятся в одной и той же позиции, наложившись друг на друга. Последний добавленный компонент окажется на вершине этой кучи, а остальные компоненты станут видны лишь частично или вообще не видны. Размеры добавляемых компонентов будут соответствовать их содержимому.

Для перемещения компонента можно воспользоваться методом `move()`, а для изменения размеров — методом `resize()`. Выполнить одновременное изменение позиции и размеров позволяет метод `setGeometry()`. Все эти методы, а также множество других, позволяющих изменять позицию и размеры, мы уже рассматривали в *разд. 18.3* и *18.4*. Если компонент не имеет родителя, эти методы изменяют характеристики окна, а если родительский компонент был указан, они изменяют характеристики только самого компонента.

Для примера выведем внутри окна надпись и кнопку, указав позицию и размеры для каждого компонента (листинг 20.1).

Листинг 20.1. Абсолютное позиционирование

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Абсолютное позиционирование")
```

```
window.resize(300, 120)
label = QtWidgets.QLabel("Текст надписи", window)
button = QtWidgets.QPushButton("Текст на кнопке", window)
label.setGeometry(10, 10, 280, 60)
button.resize(280, 30)
button.move(10, 80)
window.show()
sys.exit(app.exec_())
```

Абсолютное позиционирование имеет следующие недостатки:

- при изменении размеров окна необходимо самостоятельно пересчитывать и изменять характеристики всех компонентов в программном коде;
- при указании фиксированных размеров надписи на компонентах могут выходить за их пределы. Помните, что в разных операционных системах используются разные стили оформления, в том числе и характеристики шрифта. Подогнав размеры в одной операционной системе, можно прийти в ужас при виде приложения в другой операционной системе, где размер шрифта в два раза больше. Поэтому лучше вообще отказаться от указания фиксированных размеров или задавать размер и название шрифта для каждого компонента явно. Кроме того, приложение может поддерживать несколько языков интерфейса, а поскольку длина слов в разных языках различается, это также станет причиной искажения компонентов.

20.2. Горизонтальное и вертикальное выравнивание

Компоненты-контейнеры (их еще называют *менеджерами компоновки* и *менеджерами геометрии*) лишены недостатков абсолютного позиционирования. При изменении размеров окна производится автоматическое изменение характеристик всех компонентов, добавленных в контейнер. Настройки шрифта при этом также учитываются, поэтому изменение размеров шрифта в два раза приведет только к увеличению компонентов и окон.

Для автоматического выравнивания компонентов используются два класса:

- `QHBoxLayout` — выстраивает все добавляемые компоненты по горизонтали (по умолчанию — слева направо). Конструктор класса имеет следующий формат:

  ```
  <Объект> = QHBoxLayout([<Родитель>])
  ```

- `QVBoxLayout` — выстраивает все добавляемые компоненты по вертикали (по умолчанию — сверху вниз). Формат конструктора класса:

  ```
  <Объект> = QVBoxLayout([<Родитель>])
  ```

Иерархия наследования для этих классов выглядит так:

```
(QObject, QLayoutItem) — QLayout — QBoxLayout — QHBoxLayout
(QObject, QLayoutItem) — QLayout — QBoxLayout — QVBoxLayout
```

Обратите внимание, что указанные классы не являются наследниками класса `QWidget`, а следовательно, не обладают собственным окном и не могут использоваться отдельно. Поэтому контейнеры обязательно должны быть привязаны к родительскому компоненту. Передать ссылку на родительский компонент можно в конструкторе классов `QHBoxLayout` и `QVBoxLayout`. Кроме того, можно передать ссылку на контейнер в метод `setLayout()` роди-

тельского компонента. После этого все компоненты, добавленные в контейнер, автоматически привязываются к родительскому компоненту.

Типичный пример использования класса `QHBoxLayout` показан в листинге 20.2, а увидеть результат выполнения этого кода можно на рис. 20.1.

Листинг 20.2. Использование контейнера `QHBoxLayout`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()              # Родительский компонент — окно
window.setWindowTitle("QHBoxLayout")
window.resize(300, 60)
button1 = QtWidgets.QPushButton("1")
button2 = QtWidgets.QPushButton("2")
hbox = QtWidgets.QHBoxLayout()            # Создаем контейнер
hbox.addWidget(button1)                   # Добавляем компоненты
hbox.addWidget(button2)
window.setLayout(hbox)                    # Передаем ссылку родителю
window.show()
sys.exit(app.exec_())
```

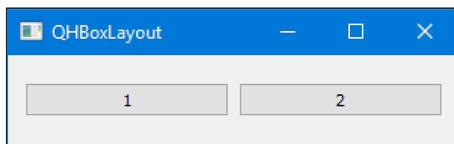

Рис. 20.1. Контейнер `QHBoxLayout` с двумя кнопками

Добавить компоненты в контейнер, удалить их и заменить другими позволяют следующие методы:

- `addWidget()` — добавляет компонент в конец контейнера. Формат метода:

  ```
  addWidget(<Компонент>[, stretch=0][, alignment=0])
  ```

  В первом параметре указывается ссылка на компонент. Необязательный параметр `stretch` задает фактор растяжения для ячейки, а параметр `alignment` — выравнивание компонента внутри ячейки. Два последних параметра можно задавать в порядке следования или по именам в произвольном порядке:

  ```
  hbox.addWidget(button1, 10, QtCore.Qt.AlignRight)
  hbox.addWidget(button2, stretch=10)
  hbox.addWidget(button3, alignment=QtCore.Qt.AlignRight)
  ```

- `insertWidget()` — добавляет компонент в указанную позицию контейнера. Формат метода:

  ```
  insertWidget(<Индекс>, <Компонент>[, stretch=0][, alignment=0])
  ```

  Если в первом параметре указано значение `0`, компонент будет добавлен в начало контейнера, а если — отрицательное значение, компонент добавляется в конец контейнера.

Иное значение указывает определенную позицию. Остальные параметры аналогичны параметрам метода `addWidget()`:

```
hbox.addWidget(button1)
hbox.insertWidget(-1, button2) # Добавление в конец
hbox.insertWidget(0, button3)  # Добавление в начало
```

- `removeWidget(<Компонент>)` — удаляет указанный компонент из контейнера;
- `replaceWidget()` — заменяет присутствующий в контейнере компонент другим. Формат метода:

  ```
  replaceWidget(<Заменяемый компонент>, <Заменяющий компонент>
                            [, options= FindChildrenRecursively])
  ```

  В необязательном параметре `options` можно задать режим поиска заменяемого компонента с помощью одного из атрибутов класса `QtCore.Qt`: `FindDirectChildrenOnly` (`0`, искать только среди содержимого текущего контейнера) и `FindChildrenRecursively` (`1`, искать среди содержимого текущего и всех вложенных в него контейнеров — поведение по умолчанию);
- `addLayout()` — добавляет другой контейнер в конец текущего контейнера. С помощью этого метода можно вкладывать один контейнер в другой, создавая таким образом структуру любой сложности. Формат метода:

  ```
  addLayout(<Контейнер>[, stretch=0])
  ```
- `insertLayout()` — добавляет другой контейнер в указанную позицию текущего контейнера. Если в первом параметре задано отрицательное значение, контейнер добавляется в конец. Формат метода:

  ```
  insertLayout(<Индекс>, <Контейнер>[, stretch=0])
  ```
- `addSpacing(<Размер>)` — добавляет пустое пространство указанного размера в конец контейнера. Размер пустого пространства задается в пикселах:

  ```
  hbox.addSpacing(100)
  ```
- `insertSpacing(<Индекс>, <Размер>)` — добавляет пустое пространство указанного размера в определенную позицию. Размер пустого пространства задается в пикселах. Если первым параметром передается отрицательное значение, то пространство добавляется в конец;
- `addStretch([stretch=0])` — добавляет пустое растягиваемое пространство с нулевым минимальным размером и фактором растяжения `stretch` в конец контейнера. Это пространство можно сравнить с пружиной, вставленной между компонентами, а параметр `stretch` — с жесткостью пружины;
- `insertStretch(<Индекс>[, stretch=0])` — метод аналогичен методу `addStretch()`, но добавляет растягиваемое пространство в указанную позицию. Если в первом параметре задано отрицательное значение, пространство добавляется в конец контейнера.

Параметр `alignment` в методах `addWidget()` и `insertWidget()` задает выравнивание компонента внутри ячейки. В этом параметре можно указать следующие атрибуты класса `QtCore.Qt`:

- `AlignLeft` — `1` — горизонтальное выравнивание по левому краю;
- `AlignRight` — `2` — горизонтальное выравнивание по правому краю;
- `AlignHCenter` — `4` — горизонтальное выравнивание по центру;

- `AlignJustify` — `8` — заполнение всего пространства;
- `AlignTop` — `32` — вертикальное выравнивание по верхнему краю;
- `AlignBottom` — `64` — вертикальное выравнивание по нижнему краю;
- `AlignVCenter` — `128` — вертикальное выравнивание по центру;
- `AlignBaseline` — `256` — вертикальное выравнивание по базовой линии;
- `AlignCenter` — `AlignVCenter | AlignHCenter` — горизонтальное и вертикальное выравнивание по центру;
- `AlignAbsolute` — `16` — если в методе `setLayoutDirection()` из класса `QWidget` указан атрибут `QtCore.Qt.RightToLeft`, атрибут `AlignLeft` задает выравнивание по правому краю, а атрибут `AlignRight` — по левому краю. Чтобы атрибут `AlignLeft` всегда соответствовал именно левому краю, необходимо указать комбинацию `AlignAbsolute | AlignLeft`. Аналогично следует поступить с атрибутом `AlignRight`.

Можно задавать и комбинации атрибутов. В них может присутствовать только один атрибут горизонтального выравнивания и только один атрибут вертикального выравнивания. Например, комбинация `AlignLeft | AlignTop` задает выравнивание по левому и верхнему краям. Противоречивые значения приводят к непредсказуемым результатам.

Помимо рассмотренных, контейнеры поддерживают также следующие методы (здесь приведены только основные — полный их список ищите в документации):

- `setDirection(<Направление>)` — задает направление вывода компонентов. В параметре можно указать следующие атрибуты из класса `QBoxLayout`:
  - `LeftToRight` — `0` — слева направо (значение по умолчанию для горизонтального контейнера);
  - `RightToLeft` — `1` — справа налево;
  - `TopToBottom` — `2` — сверху вниз (значение по умолчанию для вертикального контейнера);
  - `BottomToTop` — `3` — снизу вверх;
- `setContentsMargins()` — задает величины отступов от границ контейнера до компонентов. Форматы метода:

  ```
  setContentsMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
  setContentsMargins(<QMargins>)
  ```

  Примеры:

  ```
  hbox.setContentsMargins(2, 4, 2, 4)
  m = QtCore.QMargins(4, 2, 4, 2)
  hbox.setContentsMargins(m)
  ```

- `setSpacing(<Расстояние>)` — задает расстояние между компонентами.

20.3. Выравнивание по сетке

Помимо выравнивания компонентов по горизонтали и вертикали, существует возможность размещения компонентов внутри ячеек сетки. Для этого предназначен класс `QGridLayout`. Иерархия его наследования:

```
(QObject, QLayoutItem) — QLayout — QGridLayout
```

Создать экземпляр класса `QGridLayout` можно следующим образом:

```
<Объект> = QGridLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент. Если таковой не указан, следует передать ссылку на сетку в метод `setLayout()` родительского компонента. Код, иллюстрирующий типичный пример использования класса `QGridLayout`, представлен в листинге 20.3, а результат его выполнения — на рис. 20.2.

Листинг 20.3. Использование контейнера `QGridLayout`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()                # Родительский компонент — окно
window.setWindowTitle("QGridLayout")
window.resize(150, 100)
button1 = QtWidgets.QPushButton("1")
button2 = QtWidgets.QPushButton("2")
button3 = QtWidgets.QPushButton("3")
button4 = QtWidgets.QPushButton("4")
grid = QtWidgets.QGridLayout()              # Создаем сетку
grid.addWidget(button1, 0, 0)               # Добавляем компоненты
grid.addWidget(button2, 0, 1)
grid.addWidget(button3, 1, 0)
grid.addWidget(button4, 1, 1)
window.setLayout(grid)                      # Передаем ссылку родителю
window.show()
sys.exit(app.exec_())
```

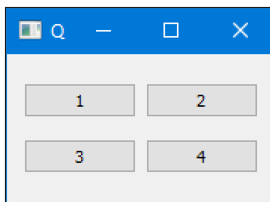

Рис. 20.2. Компонент `QGridLayout` с четырьмя кнопками

Добавить компоненты позволяют следующие методы:

- `addWidget()` — добавляет компонент в указанную ячейку сетки. Метод имеет следующие форматы:

  ```
  addWidget(<Компонент>, <Строка>, <Столбец>[, alignment=0])
  addWidget(<Компонент>, <Строка>, <Столбец>, <Количество строк>,
            <Количество столбцов>[, alignment=0])
  ```

  В первом параметре указывается ссылка на компонент, во втором параметре — индекс строки, а в третьем — индекс столбца. Нумерация строк и столбцов начинается с нуля. Параметр `<Количество строк>` задает количество занимаемых компонентом ячеек по

вертикали, а параметр <Количество столбцов> — по горизонтали. Параметр alignment задает выравнивание компонента внутри ячейки. Значения, которые можно указать в этом параметре, мы рассматривали в предыдущем разделе.

Пример:

```
grid = QtGui.QGridLayout()
grid.addWidget(button1, 0, 0, alignment=QtCore.Qt.AlignLeft)
grid.addWidget(button2, 0, 1, QtCore.Qt.AlignRight)
grid.addWidget(button3, 1, 0, 1, 2)
```

- addLayout() — добавляет контейнер в указанную ячейку сетки. Метод имеет следующие форматы:

```
addLayout(<Контейнер>, <Строка>, <Столбец>[, alignment=0])
addLayout(<Контейнер>, <Строка>, <Столбец>, <Количество строк>,
          <Количество столбцов>[, alignment=0])
```

  В первом параметре указывается ссылка на контейнер. Остальные параметры аналогичны параметрам метода addWidget().

Для удаления и замены компонентов следует пользоваться методами removeWidget() и replaceWidget(), описанными в *разд. 20.2*.

Класс QGridLayout поддерживает следующие методы (здесь приведены только основные методы — полный их список смотрите на странице **https://doc.qt.io/qt-5/qgridlayout.html**):

- setRowMinimumHeight(<Индекс>, <Высота>) — задает минимальную высоту строки с индексом <Индекс>;
- setColumnMinimumWidth(<Индекс>, <Ширина>) — задает минимальную ширину столбца с индексом <Индекс>;
- setRowStretch(<Индекс>, <Фактор растяжения>) — задает фактор растяжения по вертикали для строки с индексом <Индекс>;
- setColumnStretch(<Индекс>, <Фактор растяжения>) — задает фактор растяжения по горизонтали для столбца с индексом <Индекс>;
- setContentsMargins() — задает величины отступов от границ сетки до компонентов. Форматы метода:

```
setContentsMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
setContentsMargins(<QMargins>)
```

- setSpacing(<Значение>) — задает расстояние между компонентами по горизонтали и вертикали;
- setHorizontalSpacing(<Значение>) — задает расстояние между компонентами по горизонтали;
- setVerticalSpacing(<Значение>) — задает расстояние между компонентами по вертикали;
- rowCount() — возвращает количество строк сетки;
- columnCount() — возвращает количество столбцов сетки;
- cellRect(<Индекс строки>, <Индекс колонки>) — возвращает экземпляр класса QRect, который хранит координаты и размеры ячейки, расположенной на пересечении строки и колонки с указанными индексами.

20.4. Выравнивание компонентов формы

Класс QFormLayout позволяет выравнивать компоненты формы. Контейнер по умолчанию состоит из двух столбцов: первый предназначен для вывода надписи, а второй — для вывода самого компонента. При этом надпись связывается с компонентом, что позволяет назначать клавиши быстрого доступа, указав символ & перед буквой внутри текста надписи. По нажатию комбинации клавиш быстрого доступа (комбинация <Alt>+<буква>) в фокусе окажется компонент, расположенный справа от надписи. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QFormLayout
```

Создать экземпляр класса QFormLayout можно следующим образом:

```
<Объект> = QFormLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент. Если таковой не указан, необходимо передать ссылку на контейнер в метод setLayout() родительского компонента.

В листинге 20.4 показан код, создающий форму с контейнером QFormLayout. Результат выполнения этого кода можно увидеть на рис. 20.3.

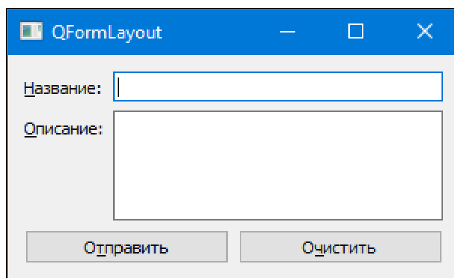

Рис. 20.3. Пример использования контейнера QFormLayout

Листинг 20.4. Использование контейнера QFormLayout

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QFormLayout")
window.resize(300, 150)
lineEdit = QtWidgets.QLineEdit()
textEdit = QtWidgets.QTextEdit()
button1 = QtWidgets.QPushButton("О&тправить")
button2 = QtWidgets.QPushButton("О&чистить")
hbox = QtWidgets.QHBoxLayout()
hbox.addWidget(button1)
hbox.addWidget(button2)
form = QtWidgets.QFormLayout()
form.addRow("&Название:", lineEdit)
```

```
form.addRow("&Описание:", textEdit)
form.addRow(hbox)
window.setLayout(form)
window.show()
sys.exit(app.exec_())
```

Класс `QFormLayout` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qformlayout.html**):

- `addRow()` — добавляет строку в конец контейнера. Форматы метода:

  ```
  addRow(<Текст надписи>, <Компонент> | <Контейнер>)
  addRow(<QLabel>, <Компонент> | <Контейнер>)
  addRow(<Компонент> | <Компонент>)
  ```

  Первый формат позволяет задать текст надписи, которая будет помещена в первую колонку формы, и компонент или контейнер, помещаемый во вторую колонку. В тексте надписи можно указать символ `&`, который пометит клавишу быстрого доступа для этого компонента (контейнера). Второй формат использует в качестве надписи компонент класса `QLabel` (он представляет надпись) — в этом случае связь с компонентом (контейнером) необходимо устанавливать вручную, передав ссылку на него в метод `setBuddy()`. Третий формат заставляет компонент (контейнер) занять сразу обе колонки формы;

- `insertRow()` — добавляет строку в указанную позицию контейнера. Если в первом параметре задано отрицательное значение, компонент добавляется в конец контейнера. Форматы метода:

  ```
  insertRow(<Индекс>, <Текст надписи>, <Компонент> | <Контейнер>)
  insertRow(<Индекс>, <QLabel>, <Компонент> | <Контейнер>)
  insertRow(<Индекс>, <Компонент> | <Контейнер>)
  ```

- `setFormAlignment(<Режим>)` — задает режим выравнивания формы (допустимые значения мы рассматривали в *разд. 20.2*):

  ```
  form.setFormAlignment(
      QtCore.Qt.AlignRight | QtCore.Qt.AlignBottom)
  ```

- `setLabelAlignment(<Режим>)` — задает режим выравнивания надписи (допустимые значения мы рассматривали в *разд. 20.2*). Вот пример выравнивания по правому краю:

  ```
  form.setLabelAlignment(QtCore.Qt.AlignRight)
  ```

- `setRowWrapPolicy(<Режим>)` — задает местоположение надписей. В качестве параметра указываются следующие атрибуты класса `QFormLayout`:
  - `DontWrapRows` — `0` — надписи расположены слева от компонентов;
  - `WrapLongRows` — `1` — длинные надписи могут находиться выше компонентов, а короткие надписи — слева от компонентов;
  - `WrapAllRows` — `2` — надписи всегда расположены выше компонентов;

- `setFieldGrowthPolicy(<Режим>)` — задает режим управления размерами компонентов. В качестве параметра указываются следующие атрибуты класса `QFormLayout`:
  - `FieldsStayAtSizeHint` — `0` — компоненты всегда будут принимать рекомендуемые (возвращаемые методом `sizeHint()`) размеры;
  - `ExpandingFieldsGrow` — `1` — компоненты, для которых установлена политика изменения размеров `QSizePolicy.Expanding` или `QSizePolicy.MinimumExpanding`, займут

всю доступную ширину. Размеры остальных компонентов всегда будут соответствовать рекомендуемым;

  • `AllNonFixedFieldsGrow` — `2` — все компоненты (если это возможно) займут всю доступную ширину;

- `setContentsMargins()` — задает величины отступов от границ сетки до компонентов. Форматы метода:

  ```
  setContentsMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
  setContentsMargins(<QMargins>)
  ```

- `setSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали и вертикали;
- `setHorizontalSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали;
- `setVerticalSpacing(<Значение>)` — задает расстояние между компонентами по вертикали.

Для удаления и замены компонентов следует пользоваться методами `removeWidget()` и `replaceWidget()`, описанными в *разд. 20.2*.

20.5. Классы *QStackedLayout* и *QStackedWidget*

Класс `QStackedLayout` реализует стек компонентов — в один момент времени показывается только один компонент, а вывод другого компонента выполняется программно. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QStackedLayout
```

Создать экземпляр класса `QStackedLayout` можно следующим образом:

```
<Объект> = QStackedLayout([<Родитель>])
```

В необязательном параметре можно задать ссылку на родительский компонент или контейнер. Если параметр не указан, следует передать ссылку на контейнер в метод `setLayout()` родительского компонента.

Класс `QStackedLayout` поддерживает следующие методы:

- `addWidget(<Компонент>)` — добавляет компонент в конец контейнера. Метод возвращает индекс добавленного компонента;
- `insertWidget(<Индекс>, <Компонент>)` — добавляет компонент в указанную позицию контейнера. Метод возвращает индекс добавленного компонента;
- `setCurrentIndex(<Индекс>)` — делает видимым компонент с указанным индексом. Метод является слотом;
- `currentIndex()` — возвращает индекс видимого компонента;
- `setCurrentWidget(<Компонент>)` — делает видимым указанный компонент. Метод является слотом;
- `currentWidget()` — возвращает ссылку на видимый компонент;
- `setStackingMode(<Режим>)` — задает режим отображения компонентов. В параметре могут быть указаны следующие атрибуты из класса `QStackedLayout`:
  • `StackOne` — `0` — только один компонент видим (значение по умолчанию);
  • `StackAll` — `1` — видны все компоненты;

- `stackingMode()` — возвращает режим отображения компонентов;
- `count()` — возвращает количество компонентов внутри контейнера;
- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`.

Для удаления и замены компонентов следует пользоваться методами `removeWidget()` и `replaceWidget()`, описанными в *разд. 20.2*.

Класс `QStackedLayout` поддерживает следующие сигналы:

- `currentChanged(<Индекс>)` — генерируется при изменении видимого компонента. Через параметр внутри обработчика доступен целочисленный индекс нового компонента;
- `widgetRemoved(<Индекс>)` — генерируется при удалении компонента из контейнера. Через параметр внутри обработчика доступен целочисленный индекс удаленного компонента.

Класс `QStackedWidget` также реализует стек компонентов, но представляет собой компонент, а не контейнер. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QStackedWidget
```

Создать экземпляр этого класса можно следующим образом:

```
<Объект> = QStackedWidget([<Родитель>])
```

Класс `QStackedWidget` поддерживает методы `addWidget()`, `insertWidget()`, `removeWidget()`, `replaceWidget()`, `count()`, `currentIndex()`, `currentWidget()`, `widget()`, `setCurrentIndex()` и `setCurrentWidget()`, которые выполняют те же действия, что и одноименные методы в классе `QStackedLayout`. Кроме того, класс `QStackedWidget` наследует все методы из базовых классов и определяет два дополнительных:

- `indexOf(<Компонент>)` — возвращает индекс компонента, ссылка на который указана в параметре;
- `__len__()` — возвращает количество компонентов. Метод вызывается при использовании функции `len()`, а также для проверки объекта на логическое значение.

Чтобы отследить изменения внутри компонента, следует назначить обработчики сигналов `currentChanged` и `widgetRemoved`.

20.6. Класс *QSizePolicy*

Если в вертикальный контейнер большой высоты добавить надпись и кнопку, то кнопка займет пространство, совпадающее с рекомендуемыми размерами (которые возвращает метод `sizeHint()`), а под надпись будет выделено все остальное место. Управление размерами компонентов внутри контейнера определяется правилами, установленными с помощью класса `QSizePolicy`. Установить эти правила для компонента можно с помощью метода `setSizePolicy(<QSizePolicy>)` класса `QWidget`, а получить их — с помощью метода `sizePolicy()`.

Создать экземпляр класса `QSizePolicy` (он определен в модуле `QtWidgets`) можно следующим способом:

```
<Объект> = QSizePolicy([<Правило для горизонтали>,
                        <Правило для вертикали>[, <Тип компонента>]])
```

Если параметры не заданы, размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`. В первом и втором параметрах указывается один из следующих атрибутов класса `QSizePolicy`:

- ♦ `Fixed` — размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`;
- ♦ `Minimum` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента и может быть увеличен при необходимости;
- ♦ `Maximum` — размер, возвращаемый методом `sizeHint()`, является максимальным для компонента и может быть уменьшен при необходимости;
- ♦ `Preferred` — размер, возвращаемый методом `sizeHint()`, является предпочтительным и может быть как увеличен, так и уменьшен;
- ♦ `Expanding` — размер, возвращаемый методом `sizeHint()`, может быть как увеличен, так и уменьшен. Компонент займет все свободное пространство в контейнере;
- ♦ `MinimumExpanding` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента. Компонент займет все свободное пространство в контейнере;
- ♦ `Ignored` — размер, возвращаемый методом `sizeHint()`, игнорируется. Компонент займет все свободное пространство в контейнере.

Изменить правила управления размерами уже после создания экземпляра класса `QSizePolicy` позволяют методы `setHorizontalPolicy(<Правило для горизонтали>)` и `setVerticalPolicy(<Правило для вертикали>)`.

С помощью методов `setHorizontalStretch(<Фактор для горизонтали>)` и `setVerticalStretch(<Фактор для вертикали>)` можно указать фактор растяжения. Чем больше указанное значение относительно значения, заданного в других компонентах, тем больше места будет выделяться под компонент. Этот параметр можно сравнить с жесткостью пружины.

Можно указать, что минимальная высота компонента зависит от его ширины. Для этого необходимо передать значение `True` в метод `setHeightForWidth(<Флаг>)`. Кроме того, следует в классе компонента переопределить метод `heightForWidth(<Ширина>)` — переопределенный метод должен возвращать высоту компонента для указанной в параметре ширины.

20.7. Объединение компонентов в группу

Состояние одних компонентов может зависеть от состояния других — например, из нескольких переключателей можно выбрать только один. Кроме того, некоторый набор компонентов может использоваться для ввода связанных данных — например, имени, отчества и фамилии пользователя. В этом случае компоненты объединяют в группу.

Группа компонентов отображается внутри рамки, на верхней границе которой выводится текст заголовка. Реализовать группу позволяет класс `QGroupBox`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QGroupBox
```

Создать экземпляр класса `QGroupBox` можно следующим образом:

```
<Объект> = QGroupBox([<Родитель>])
<Объект> = QGroupBox(<Текст>[, <Родитель>])
```

В необязательном параметре <Родитель> можно указать ссылку на родительский компонент. Параметр <Текст> задает текст заголовка, который отобразится на верхней границе рамки. Внутри текста заголовка символ &, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет — в качестве подсказки пользователю — подчеркнута. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы первый компонент из группы окажется в фокусе ввода.

После создания экземпляра класса QGroupBox следует добавить компоненты в какой-либо контейнер, а затем передать ссылку на контейнер в метод setLayout() группы.

Типичный пример использования класса QGroupBox представлен кодом из листинга 20.5. Созданная им группа показана на рис. 20.4.

Листинг 20.5. Пример использования компонента QGroupBox

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QGroupBox")
window.resize(200, 80)
mainbox = QtWidgets.QVBoxLayout()
radio1 = QtWidgets.QRadioButton("&Да")
radio2 = QtWidgets.QRadioButton("&Нет")
box = QtWidgets.QGroupBox("&Вы знаете язык Python?") # Объект группы
hbox = QtWidgets.QHBoxLayout() # Контейнер для группы
hbox.addWidget(radio1)          # Добавляем компоненты
hbox.addWidget(radio2)
box.setLayout(hbox)             # Передаем ссылку на контейнер
mainbox.addWidget(box)          # Добавляем группу в главный контейнер
window.setLayout(mainbox)       # Передаем ссылку на главный контейнер в окно
radio1.setChecked(True)         # Выбираем первый переключатель
window.show()
sys.exit(app.exec_())
```

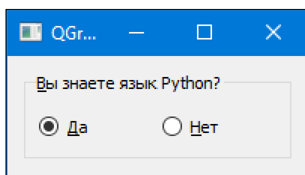

Рис. 20.4. Пример использования компонента QGroupBox

Класс QGroupBox поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qgroupbox.html**):

- setTitle(<Текст>) — задает текст заголовка;
- title() — возвращает текст заголовка;

- setAlignment(<Выравнивание>) — задает горизонтальное выравнивание текста заголовка. В параметре указываются следующие атрибуты класса QtCore.Qt: AlignLeft, AlignHCenter или AlignRight:

  ```
  box.setAlignment(QtCore.Qt.AlignRight)
  ```

- alignment() — возвращает горизонтальное выравнивание текста заголовка;
- setCheckable(<Флаг>) — если в параметре указать значение True, то перед текстом заголовка будет отображен флажок. Если флажок установлен, группа станет активной, а если флажок снят, все компоненты внутри группы окажутся неактивными. По умолчанию флажок не отображается;
- isCheckable() — возвращает значение True, если перед заголовком выводится флажок, и False — в противном случае;
- setChecked(<Флаг>) — если в параметре указать значение True, флажок, отображаемый перед текстом подсказки, будет установлен. Значение False сбрасывает флажок. Метод является слотом;
- isChecked() — возвращает значение True, если флажок, отображаемый перед текстом заголовка, установлен, и False — в противном случае;
- setFlat(<Флаг>) — если в параметре указано значение True, будет отображаться только верхняя граница рамки, а если False — то все границы рамки;
- isFlat() — возвращает значение True, если отображается только верхняя граница рамки, и False — если все границы рамки.

Класс QGroupBox поддерживает сигналы:

- clicked(<Состояние флажка>) — генерируется при щелчке мышью на флажке, выводимом перед текстом заголовка. Если состояние флажка изменяется с помощью метода setChecked(), сигнал не генерируется. Через параметр внутри обработчика доступно значение True, если флажок установлен, и False — если сброшен;
- toggled(<Состояние флажка>) — генерируется при изменении состояния флажка, выводимого перед текстом заголовка. Через параметр внутри обработчика доступно значение True, если флажок установлен, и False — если сброшен.

20.8. Панель с рамкой

Класс QFrame расширяет возможности класса QWidget за счет добавления рамки вокруг компонента. Этот класс, в свою очередь, наследуют некоторые компоненты, например надписи, многострочные текстовые поля и др. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame
```

Конструктор класса QFrame имеет следующий формат:

```
<Объект> = QFrame([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, компонент будет обладать своим собственным окном. Если в параметре flags указан тип окна, компонент, имея родителя, будет обладать своим собственным окном, но окажется привязан к родителю. Это позволяет, например, создать модальное окно, которое будет блокировать только окно родителя, но не все окна приложения. Какие именно значения можно указать в параметре flags, мы уже рассматривали в *разд. 18.2*.

Класс `QFrame` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qframe.html**):

- `setFrameShape(<Форма>)` — задает форму рамки. Могут быть указаны следующие атрибуты класса `QFrame`:
  - `NoFrame` — `0` — нет рамки;
  - `Box` — `1` — прямоугольная рамка;
  - `Panel` — `2` — панель, которая может быть выпуклой или вогнутой;
  - `WinPanel` — `3` — панель в стиле Windows 2000, которая может быть выпуклой или вогнутой. Ширина границы — 2 пиксела. Этот атрибут присутствует для совместимости со старыми версиями Qt;
  - `HLine` — `4` — горизонтальная линия. Используется как разделитель;
  - `VLine` — `5` — вертикальная линия без содержимого;
  - `StyledPanel` — `6` — панель, внешний вид которой зависит от текущего стиля. Она может быть выпуклой или вогнутой. Это рекомендуемая форма рамки для панелей;
- `setFrameShadow(<Тень>)` — задает стиль тени. Могут быть указаны следующие атрибуты класса `QFrame`:
  - `Plain` — `16` — нет тени;
  - `Raised` — `32` — панель отображается выпуклой;
  - `Sunken` — `48` — панель отображается вогнутой;
- `setFrameStyle(<Стиль>)` — задает форму рамки и стиль тени одновременно. В качестве значения через оператор `|` указывается комбинация приведенных ранее атрибутов класса `QFrame`:

  ```
  frame.setFrameStyle(QtWidgets.QFrame.Panel | QtWidgets.QFrame.Raised)
  ```
- `setLineWidth(<Ширина>)` — задает ширину линий у рамки;
- `setMidLineWidth(<Ширина>)` — задает ширину средней линии у рамки. Средняя линия используется для создания эффекта выпуклости и вогнутости и доступна только для форм рамки `Box`, `HLine` и `VLine`.

20.9. Панель с вкладками

Для создания панели с вкладками («блокнота») предназначен класс `QTabWidget`. Панель состоит из области заголовка с ярлыками и набора вкладок с различными компонентами. В один момент времени показывается содержимое только одной вкладки. Щелчок мышью на ярлыке в области заголовка приводит к отображению содержимого соответствующей вкладки.

Иерархия наследования для класса `QTabWidget` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QTabWidget
```

Конструктор класса `QTabWidget` имеет следующий формат:

```
<Объект> = QTabWidget([<Родитель>])
```

В параметре `<Родитель>` указывается ссылка на родительский компонент. Если он не указан, компонент будет обладать своим собственным окном.

Пример кода, создающего компонент `QTabWidget`, приведен в листинге 20.6. Сама панель с вкладками показана на рис. 20.5.

Листинг 20.6. Пример использования компонента `QTabWidget`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QTabWidget")
window.resize(400, 100)
tab = QtWidgets.QTabWidget()
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 1"), "Вкладка &1")
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 2"), "Вкладка &2")
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 3"), "Вкладка &3")
tab.setCurrentIndex(0)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(tab)
window.setLayout(vbox)
window.show()
window.show()
sys.exit(app.exec_())
```

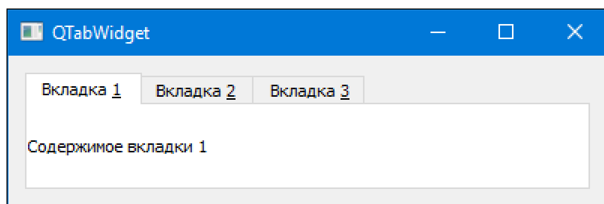

Рис. 20.5. Панель с вкладками `QTabWidget`

Класс `QTabWidget` поддерживает следующие методы (здесь приведены только основные — полное описание класса содержится на странице **https://doc.qt.io/qt-5/qtabwidget.html**):

- `addTab()` — добавляет вкладку в конец контейнера и возвращает ее индекс. Форматы метода:

  ```
  addTab(<Компонент>, <Текст заголовка>)
  addTab(<Компонент>, <QIcon>, <Текст заголовка>)
  ```

  В параметре `<Компонент>` указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр `<Текст заголовка>` задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет — в качестве подсказки пользователю — подчеркнута. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы откроется соответствующая

вкладка. Параметр `<QIcon>` позволяет указать значок (экземпляр класса `QIcon`), который отобразится перед текстом в области заголовка.

* Пример указания стандартного значка:

```
style = window.style()
icon = style.standardIcon(QtWidgets.QStyle.SP_DriveNetIcon)
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 1"), icon,
           "Вкладка &1")
```

* Пример загрузки значка из файла:

```
icon = QtGui.QIcon("icon.png")
tab.addTab(QtWidgets.QLabel("Содержимое вкладки 1"), icon,
           "Вкладка &1")
```

♦ `insertTab()` — добавляет вкладку в указанную позицию и возвращает индекс этой вкладки. Форматы метода:

```
insertTab(<Индекс>, <Компонент>, <Текст заголовка>)
insertTab(<Индекс>, <Компонент>, <QIcon>, <Текст заголовка>)
```

♦ `removeTab(<Индекс>)` — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;

♦ `clear()` — удаляет все вкладки, при этом компоненты, которые отображались на вкладках, не удаляются;

♦ `setTabText(<Индекс>, <Текст заголовка>)` — задает текст заголовка для вкладки с указанным индексом;

♦ `setElideMode(<Режим>)` — задает режим обрезки текста в названии вкладки, если он не помещается в отведенную область (в месте пропуска выводится троеточие). Могут быть указаны следующие атрибуты класса `QtCore.Qt`:

* `ElideLeft` — `0` — текст обрезается слева;
* `ElideRight` — `1` — текст обрезается справа;
* `ElideMiddle` — `2` — текст вырезается посередине;
* `ElideNone` — `3` — текст не обрезается;

♦ `tabText(<Индекс>)` — возвращает текст заголовка вкладки с указанным индексом;

♦ `setTabIcon(<Индекс>, <QIcon>)` — устанавливает значок перед текстом в заголовке вкладки с указанным индексом. Во втором параметре указывается экземпляр класса `QIcon`;

♦ `setTabPosition(<Позиция>)` — задает позицию области заголовка. Могут быть указаны следующие атрибуты класса `QTabWidget`:

* `North` — `0` — сверху;
* `South` — `1` — снизу;
* `West` — `2` — слева;
* `East` — `3` — справа.

Пример:

```
tab.setTabPosition(QtWidgets.QTabWidget.South)
```

- setTabShape(<Форма>) — задает форму углов ярлыка вкладки в области заголовка. Могут быть указаны следующие атрибуты класса QTabWidget:
  - Rounded — 0 — скругленные углы (значение по умолчанию);
  - Triangular — 1 — треугольная форма;
- setTabsClosable(<Флаг>) — если в качестве параметра указано значение True, то после текста заголовка вкладки будет отображена кнопка ее закрытия. По нажатию этой кнопки генерируется сигнал tabCloseRequested;
- setMovable(<Флаг>) — если в качестве параметра указано значение True, ярлыки вкладок можно перемещать с помощью мыши;
- setDocumentMode(<Флаг>) — если в качестве параметра указано значение True, область компонента не будет отображаться как панель;
- setUsesScrollButtons(<Флаг>) — если в качестве параметра указано значение True, то, если все ярлыки вкладок не помещаются в область заголовка панели, появятся две кнопки, с помощью которых можно прокручивать область заголовка, тем самым отображая только часть ярлыков. Значение False запрещает скрытие ярлыков;
- setTabToolTip(<Индекс>, <Текст>) — задает текст всплывающей подсказки для ярлыка вкладки с указанным индексом;
- setTabWhatsThis(<Индекс>, <Текст>) — задает текст справки для ярлыка вкладки с указанным индексом;
- setTabEnabled(<Индекс>, <Флаг>) — если вторым параметром передано значение False, вкладка с указанным в первом параметре индексом станет недоступной. Значение True делает вкладку доступной;
- isTabEnabled(<Индекс>) — возвращает значение True, если вкладка с указанным индексом доступна, и False — в противном случае;
- setCurrentIndex(<Индекс>) — делает видимой вкладку с указанным в параметре индексом. Метод является слотом;
- currentIndex() — возвращает индекс видимой вкладки;
- setCurrentWidget(<Компонент>) — делает видимой вкладку с указанным компонентом. Метод является слотом;
- currentWidget() — возвращает ссылку на компонент, расположенный на видимой вкладке;
- widget(<Индекс>) — возвращает ссылку на компонент, который расположен по указанному индексу, или значение None;
- indexOf(<Компонент>) — возвращает индекс вкладки, на которой расположен компонент. Если компонент не найден, возвращается значение -1;
- count() — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции len():

```
print(tab.count(), len(tab))
```

Класс QTabWidget поддерживает такие сигналы:

- currentChanged(<Индекс>) — генерируется при переключении на другую вкладку. Через параметр внутри обработчика доступен целочисленный индекс новой вкладки;

- `tabCloseRequested(<Индекс>)` — генерируется при нажатии кнопки закрытия вкладки. Через параметр внутри обработчика доступен целочисленный индекс закрываемой вкладки.

20.10. Компонент «аккордеон»

Класс `QToolBox` позволяет создать «аккордеон» — компонент с несколькими вкладками, в котором изначально отображается содержимое только одной вкладки, а у остальных видимы лишь заголовки. После щелчка мышью на заголовке вкладки она открывается, а остальные сворачиваются. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QToolBox
```

Конструктор класса `QToolBox` имеет следующий формат:

```
<Объект> = QToolBox([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если он не указан или имеет значение `None`, компонент будет обладать своим собственным окном. В параметре `flags` может быть указан тип окна.

Пример кода, создающего компонент класса `QToolBox`, представлен в листинге 20.7. Созданный им «аккордеон» показан на рис. 20.6.

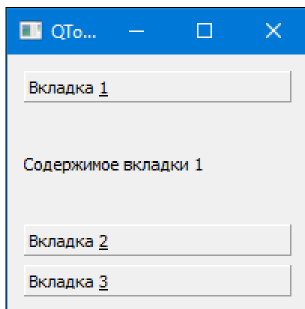

Рис. 20.6. Пример использования класса `QToolBox`

Листинг 20.7. Пример использования класса `QToolBox`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QToolBox")
window.resize(200, 100)
toolBox = QtWidgets.QToolBox()
toolBox.addItem(QtWidgets.QLabel("Содержимое вкладки 1"), "Вкладка &1")
toolBox.addItem(QtWidgets.QLabel("Содержимое вкладки 2"), "Вкладка &2")
toolBox.addItem(QtWidgets.QLabel("Содержимое вкладки 3"), "Вкладка &3")
toolBox.setCurrentIndex(0)
```

```
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(toolBox)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

Класс `QToolBox` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qtoolbox.html**):

- `addItem()` — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Форматы метода:

  ```
  addItem(<Компонент>, <Текст заголовка>)
  addItem(<Компонент>, <QIcon>, <Текст заголовка>)
  ```

  В параметре `<Компонент>` указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр `<Текст заголовка>` задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет — в качестве подсказки пользователю — подчеркнута. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы откроется соответствующая вкладка. Параметр `<QIcon>` позволяет указать значок (экземпляр класса `QIcon`), который отобразится перед текстом в области заголовка;

- `insertItem()` — добавляет вкладку в указанную позицию. Метод возвращает индекс добавленной вкладки. Форматы метода:

  ```
  insertItem(<Индекс>, <Компонент>, <Текст заголовка>)
  insertItem(<Индекс>, <Компонент>, <QIcon>, <Текст заголовка>)
  ```

- `removeItem(<Индекс>)` — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;
- `setItemText(<Индекс>, <Текст заголовка>)` — задает текст заголовка для вкладки с указанным индексом;
- `itemText(<Индекс>)` — возвращает текст заголовка вкладки с указанным индексом;
- `setItemIcon(<Индекс>, <QIcon>)` — устанавливает значок перед текстом в заголовке вкладки с указанным индексом. Во втором параметре указывается экземпляр класса `QIcon`;
- `setItemToolTip(<Индекс>, <Текст>)` — задает текст всплывающей подсказки для ярлыка вкладки с указанным индексом;
- `setItemEnabled(<Индекс>, <Флаг>)` — если вторым параметром передается значение `False`, вкладка с указанным в первом параметре индексом станет недоступной. Значение `True` делает вкладку доступной;
- `isItemEnabled(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом доступна, и `False` — в противном случае;
- `setCurrentIndex(<Индекс>)` — делает видимой вкладку с указанным индексом. Метод является слотом;
- `currentIndex()` — возвращает индекс видимой вкладки;
- `setCurrentWidget(<Компонент>)` — делает видимым вкладку с указанным компонентом. Метод является слотом;

♦ `currentWidget()` — возвращает ссылку на компонент, который расположен на видимой вкладке;

♦ `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;

♦ `indexOf(<Компонент>)` — возвращает индекс вкладки, на которой расположен компонент. Если компонент не найден, возвращается значение `-1`;

♦ `count()` — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции `len()`:

```
print(toolBox.count(), len(toolBox))
```

При переключении на другую вкладку генерируется сигнал `currentChanged(<Индекс>)`. Через параметр внутри обработчика доступен целочисленный индекс вкладки, на которую было выполнено переключение.

20.11. Панели с изменяемым размером

Класс `QSplitter` позволяет изменять размеры добавленных компонентов, взявшись мышью за границу между компонентами. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QSplitter
```

Конструктор класса `QSplitter` имеет два формата:

```
<Объект> = QSplitter([parent=<Родитель>])
<Объект> = QSplitter(<Ориентация>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если таковой не указан или имеет значение `None`, компонент будет обладать своим собственным окном. Параметр `<Ориентация>` задает ориентацию размещения компонентов. Могут быть заданы атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) класса `QtCore.Qt`. Если параметр не указан, компоненты размещаются по горизонтали.

Пример использования класса `QSplitter` показан в листинге 20.8, а результат можно увидеть на рис. 20.7.

Листинг 20.8. Пример использования класса `QSplitter`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QSplitter")
window.resize(200, 200)
splitter = QtWidgets.QSplitter(QtCore.Qt.Vertical)
label1 = QtWidgets.QLabel("Содержимое компонента 1")
label2 = QtWidgets.QLabel("Содержимое компонента 2")
label1.setFrameStyle(QtWidgets.QFrame.Box | QtWidgets.QFrame.Plain)
label2.setFrameStyle(QtWidgets.QFrame.Box | QtWidgets.QFrame.Plain)
splitter.addWidget(label1)
splitter.addWidget(label2)
```

```
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(splitter)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

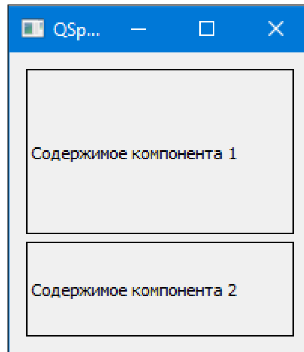

Рис. 20.7. Пример использования класса `QSplitter`

Класс `QSplitter` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qsplitter.html**):

- `addWidget(<Компонент>)` — добавляет компонент в конец контейнера;
- `insertWidget(<Индекс>, <Компонент>)` — добавляет компонент в указанную позицию. Если компонент был добавлен ранее, он будет перемещен в новую позицию;
- `setOrientation(<Ориентация>)` — задает ориентацию размещения компонентов. Могут быть заданы атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) класса `QtCore.Qt`;
- `setHandleWidth(<Ширина>)` — задает ширину компонента-разделителя, взявшись за который мышью, можно изменить размер области;
- `saveState()` — возвращает экземпляр класса `QByteArray` с размерами всех областей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода `restoreState()`;
- `restoreState(<QByteArray>)` — восстанавливает размеры областей компонента из экземпляра класса, возвращенного методом `saveState()`;
- `setChildrenCollapsible(<Флаг>)` — если в параметре указано значение `False`, пользователь не сможет уменьшить размеры всех компонентов до нуля. По умолчанию размер может быть нулевым, даже если для какого-либо компонента установлены минимальные размеры;
- `setCollapsible(<Индекс>, <Флаг>)` — значение `False` в параметре `<Флаг>` запрещает уменьшение размеров до нуля для компонента с указанным индексом;
- `setOpaqueResize(<Флаг>)` — если в качестве параметра указано значение `False`, размеры компонентов изменятся только после окончания перемещения границы и отпускания кнопки мыши. В процессе перемещения мыши вместе с ней будет перемещаться специальный компонент в виде линии;
- `setStretchFactor(<Индекс>, <Фактор>)` — задает фактор растяжения для компонента с указанным индексом;

- setSizes(<Список>) — задает размеры всех компонентов. Для горизонтального контейнера указывается список со значениями ширины каждого компонента, а для вертикального контейнера — список со значениями высоты каждого компонента;
- sizes() — возвращает список с размерами (шириной или высотой):

```
print(splitter.sizes()) # Результат: [308, 15]
```

- count() — возвращает количество компонентов. Получить количество компонентов можно также с помощью функции len():

```
print(splitter.count(), len(splitter))
```

- widget(<Индекс>) — возвращает ссылку на компонент, который расположен по указанному индексу, или значение None;
- indexOf(<Компонент>) — возвращает индекс области, в которой расположен компонент. Если таковой не найден, возвращается значение -1.

При изменении размеров генерируется сигнал splitterMoved(<Позиция>, <Индекс>). Через первый параметр внутри обработчика доступна новая позиция, а через второй параметр — индекс перемещаемого разделителя; оба параметра целочисленные.

20.12. Область с полосами прокрутки

Класс QScrollArea реализует область с полосами прокрутки. Если компонент не помещается в размеры области, полосы прокрутки будут отображены автоматически. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QScrollArea
```

Конструктор класса QScrollArea имеет следующий формат:

```
<Объект> = QScrollArea([<Родитель>])
```

Класс QScrollArea поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qscrollarea.html**):

- setWidget(<Компонент>) — помещает компонент в область прокрутки;
- setWidgetResizable(<Флаг>) — если в качестве параметра указано значение True, при изменении размеров области будут изменяться и размеры компонента. Значение False запрещает изменение размеров компонента;
- setAlignment(<Выравнивание>) — задает местоположение компонента внутри области, когда размеры области больше размеров компонента:

```
scrollArea.setAlignment(QtCore.Qt.AlignCenter)
```

- ensureVisible(<X>, <Y>[, xMargin=50][, yMargin=50]) — прокручивает область к точке с координатами (<X>, <Y>) и полями xMargin и yMargin;
- ensureWidgetVisible(<Компонент>[, xMargin=50][, yMargin=50]) — прокручивает область таким образом, чтобы <Компонент> был видим;
- widget() — возвращает ссылку на компонент, который расположен внутри области, или значение None;
- takeWidget() — удаляет компонент из области и возвращает ссылку на него. Сам компонент не удаляется.

Класс `QScrollArea` наследует следующие методы из класса `QAbstractScrollArea` (здесь перечислены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qabstractscrollarea.html**):

- `horizontalScrollBar()` — возвращает ссылку на горизонтальную полосу прокрутки (экземпляр класса `QScrollBar`);
- `verticalScrollBar()` — возвращает ссылку на вертикальную полосу прокрутки (экземпляр класса `QScrollBar`);
- `setHorizontalScrollBarPolicy(<Режим>)` — устанавливает режим отображения горизонтальной полосы прокрутки;
- `setVerticalScrollBarPolicy(<Режим>)` — устанавливает режим отображения вертикальной полосы прокрутки.

  В параметре `<Режим>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

  - `ScrollBarAsNeeded` — `0` — полоса прокрутки отображается только в том случае, если размеры компонента больше размеров области;
  - `ScrollBarAlwaysOff` — `1` — полоса прокрутки никогда не отображается;
  - `ScrollBarAlwaysOn` — `2` — полоса прокрутки отображается всегда.

ГЛАВА 21

Основные компоненты

Практически все компоненты графического интерфейса определены в модуле `QtWidgets` (за исключением `QWebEngineWidgets`) и наследуют класс `QWidget`. Следовательно, методы этих классов, которые мы рассматривали в предыдущих главах, доступны всем компонентам. Если компонент не имеет родителя, он обладает собственным окном, и его положение отсчитывается, например, относительно экрана. Если же компонент имеет родителя, его положение отсчитывается относительно родительского компонента. Это обстоятельство важно учитывать при работе с компонентами. Обращайте внимание на иерархию наследования, которую мы будем показывать для каждого компонента.

21.1. Надпись

Надпись применяется для вывода подсказки пользователю, информирования пользователя о ходе выполнения операции, назначении клавиш быстрого доступа и т. п. Кроме того, надписи позволяют отображать отформатированный с помощью CSS текст в формате HTML, что позволяет реализовать простейший веб-браузер. В библиотеке PyQt 5 надпись реализуется с помощью класса `QLabel`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLabel
```

Конструктор класса `QLabel` имеет два формата:

```
<Объект> = QLabel([parent=<Родитель>][, flags=<Тип окна>])
<Объект> = QLabel(<Текст>[, parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если он не указан или имеет значение `None`, компонент будет обладать своим собственным окном, тип которого можно задать с помощью параметра `flags`. Параметр `<Текст>` позволяет задать текст, который будет отображен на надписи:

```
label = QtWidgets.QLabel("Текст надписи", flags=QtCore.Qt.Window)
label.resize(300, 50)
label.show()
```

Класс `QLabel` поддерживает следующие основные методы (полный их список смотрите на странице **https://doc.qt.io/qt-5/qlabel.html**):

- ♦ `setText(<Текст>)` — задает текст, который будет отображен на надписи. Можно указать как обычный текст, так и содержащий CSS-форматирование текст в формате HTML:

  ```
  label.setText("Текст <b>полужирный</b>")
  ```

Перевод строки в простом тексте осуществляется с помощью символа `\n`, а в тексте в формате HTML — с помощью тега `<br>`:

```
label.setText("Текст\nна двух строках")
```

Внутри текста символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет — в качестве подсказки пользователю — подчеркнута. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент, ссылка на который передана в метод `setBuddy()`, окажется в фокусе ввода. Чтобы вывести сам символ `&`, необходимо его удвоить. Если надпись не связана с другим компонентом, символ `&` выводится в составе текста:

```
label = QtWidgets.QLabel("&Пароль")
lineEdit = QtWidgets.QLineEdit()
label.setBuddy(lineEdit)
```

Метод является слотом;

- `setNum(<Число>)` — преобразует целое или вещественное число в строку и отображает ее на надписи. Метод является слотом;
- `setWordWrap(<Флаг>)` — если в параметре указано значение `True`, текст может переноситься на другую строку. По умолчанию перенос строк не осуществляется;
- `text()` — возвращает текст надписи;
- `setTextFormat(<Режим>)` — задает режим отображения текста. Могут быть указаны следующие атрибуты класса `QtCore.Qt`:
  - `PlainText` — `0` — простой текст;
  - `RichText` — `1` — текст, отформатированный тегами HTML;
  - `AutoText` — `2` — автоматическое определение (режим по умолчанию). Если текст содержит HTML-теги, то используется режим `RichText`, в противном случае — режим `PlainText`;
- `setAlignment(<Режим>)` — задает режим выравнивания текста внутри надписи (допустимые значения мы рассматривали в *разд. 20.2*):

```
label.setAlignment(QtCore.Qt.AlignRight | QtCore.Qt.AlignBottom)
```

- `setOpenExternalLinks(<Флаг>)` — если в качестве параметра указано значение `True`, теги `<a>`, присутствующие в тексте, будут преобразованы в гиперссылки:

```
label.setText('<a href="https://www.google.ru/">Это гиперссылка</a>')
label.setOpenExternalLinks(True)
```

- `setBuddy(<Компонент>)` — позволяет связать надпись с другим компонентом. В этом случае в тексте надписи можно задавать клавиши быстрого доступа, указав символ `&` перед буквой или цифрой. После нажатия комбинации клавиш в фокусе ввода окажется компонент, ссылка на который передана в качестве параметра;
- `setPixmap(<QPixmap>)` — позволяет вывести изображение на надпись. В качестве параметра указывается экземпляр класса `QPixmap`:

```
label.setPixmap(QtGui.QPixmap("picture.jpg"))
```

Метод является слотом;

- `setPicture(<QPicture>)` — позволяет вывести рисунок. В качестве параметра указывается экземпляр класса `QPicture`. Метод является слотом;

- `setMovie(<QMovie>)` — позволяет вывести анимацию. В качестве параметра указывается экземпляр класса `QMovie`. Метод является слотом;
- `setScaledContents(<Флаг>)` — если в параметре указано значение `True`, то при изменении размеров надписи размер содержимого также будет изменяться. По умолчанию изменение размеров содержимого не осуществляется;
- `setMargin(<Отступ>)` — задает отступы от границ компонента до его содержимого;
- `setIndent(<Отступ>)` — задает отступ от рамки до текста надписи в зависимости от значения выравнивания. Если выравнивание производится по левой стороне, то задает отступ слева, если по правой стороне, то справа;
- `clear()` — удаляет содержимое надписи. Метод является слотом;
- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом надписи. Можно указать следующие атрибуты (или их комбинацию через оператор `|`) класса `QtCore.Qt`:
  - `NoTextInteraction` — `0` — пользователь не может взаимодействовать с текстом надписи;
  - `TextSelectableByMouse` — `1` — текст можно выделить мышью, чтобы, например, скопировать его в буфер обмена;
  - `TextSelectableByKeyboard` — `2` — текст можно выделить с помощью клавиш на клавиатуре. Внутри надписи будет отображен текстовый курсор;
  - `LinksAccessibleByMouse` — `4` — на гиперссылках, присутствующих в тексте надписи, можно щелкать мышью;
  - `LinksAccessibleByKeyboard` — `8` — с гиперссылками, присутствующими в тексте надписи, допускается взаимодействовать с помощью клавиатуры: перемещаться между гиперссылками можно с помощью клавиши <Tab>, а переходить по гиперссылке — по нажатию клавиши <Enter>;
  - `TextEditable` — `16` — текст надписи можно редактировать;
  - `TextEditorInteraction` — комбинация `TextSelectableByMouse | TextSelectableByKeyboard | TextEditable`;
  - `TextBrowserInteraction` — комбинация `TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard`;
- `setSelection(<Индекс>, <Длина>)` — выделяет фрагмент длиной `<Длина>`, начиная с позиции `<Индекс>`;
- `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено;
- `selectedText()` — возвращает выделенный текст или пустую строку, если ничего не выделено;
- `hasSelectedText()` — возвращает значение `True`, если фрагмент текста надписи выделен, и `False` — в противном случае.

Класс `QLabel` поддерживает следующие сигналы:

- `linkActivated(<URL>)` — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен URL-адрес, заданный в виде строки;

♦ `linkHovered(<URL>)` — генерируется при наведении указателя мыши на гиперссылку. Через параметр внутри обработчика доступен URL-адрес в виде строки или пустая строка.

21.2. Командная кнопка

Командная кнопка используется для запуска какой-либо операции. Кнопка реализуется с помощью класса `QPushButton`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QPushButton
```

Конструктор класса `QPushButton` имеет три формата:

```
<Объект> = QPushButton([parent=<Родитель>])
<Объект> = QPushButton(<Текст>[, parent=<Родитель>])
<Объект> = QPushButton(<QIcon>, <Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если таковой не задан или имеет значение `None`, компонент будет обладать своим собственным окном. Параметр `<Текст>` позволяет задать текст, который отобразится на кнопке, а параметр `<QIcon>` — добавить перед текстом значок.

Класс `QPushButton` наследует следующие методы из класса `QAbstractButton` (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qabstractbutton.html**):

♦ `setText(<Текст>)` — задает текст, который будет отображен на кнопке. Внутри текста символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет — в качестве подсказки пользователю — подчеркнута. Одновременное нажатие клавиши <Alt> и подчеркнутой буквы приведет к нажатию этой кнопки. Чтобы вывести сам символ `&`, необходимо его удвоить;

♦ `text()` — возвращает текст, отображаемый на кнопке;

♦ `setShortcut(<QKeySequence>)` — задает комбинацию клавиш быстрого доступа. Вот примеры указания значения:

```
button.setShortcut("Alt+B")
button.setShortcut(QtGui.QKeySequence.mnemonic("&B"))
button.setShortcut(QtGui.QKeySequence("Alt+B"))
button.setShortcut(
        QtGui.QKeySequence(QtCore.Qt.ALT + QtCore.Qt.Key_E))
```

♦ `setIcon(<QIcon>)` — вставляет значок перед текстом кнопки;

♦ `setIconSize(<QSize>)` — задает размеры значка в виде экземпляра класса `QSize`. Метод является слотом;

♦ `setAutoRepeat(<Флаг>)` — если в качестве параметра указано значение `True`, сигнал `clicked` будет периодически генерироваться, пока кнопка находится в нажатом состоянии. Примером являются кнопки, изменяющие значение полосы прокрутки;

♦ `animateClick([<Интервал>])` — имитирует нажатие кнопки пользователем. После нажатия кнопка находится в этом состоянии указанный промежуток времени, по истечении которого отпускается. Значение указывается в миллисекундах. Если параметр не указан, то интервал равен 100 миллисекундам. Метод является слотом;

- `click()` — имитирует нажатие кнопки без анимации. Метод является слотом;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка является переключателем, который может находиться в двух состояниях: установленном и неустановленном;
- `setChecked(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка-переключатель будет находиться в установленном состоянии. Метод является слотом;
- `isChecked()` — возвращает значение `True`, если кнопка находится в установленном состоянии, и `False` — в противном случае;
- `toggle()` — переключает кнопку. Метод является слотом;
- `setAutoExclusive(<Флаг>)` — если в качестве параметра указано значение `True`, внутри контейнера может быть установлена только одна кнопка-переключатель;
- `setDown(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка будет находиться в нажатом состоянии;
- `isDown()` — возвращает значение `True`, если кнопка находится в нажатом состоянии, и `False` — в противном случае.

Кроме указанных состояний, кнопка может находиться в неактивном состоянии. Для этого необходимо передать значение `False` в метод `setEnabled()`, унаследованный от класса `QWidget`. Проверить, активна ли кнопка, позволяет метод `isEnabled()`, возвращающий значение `True`, если кнопка находится в активном состоянии, и `False` — в противном случае. Это же касается и всех прочих компонентов, порожденных от класса `QWidget`.

Класс `QAbstractButton` поддерживает следующие сигналы:

- `pressed` — генерируется при нажатии кнопки;
- `released` — генерируется при отпускании ранее нажатой кнопки;
- `clicked(<Состояние>)` — генерируется при нажатии, а затем отпускании кнопки мыши над кнопкой. Именно для этого сигнала обычно назначают обработчики. Передаваемый обработчику параметр имеет значение `True`, если кнопка-переключатель установлена, и `False`, если она сброшена или это обычная кнопка, а не переключатель;
- `toggled(<Состояние>)` — генерируется при изменении состояния кнопки-переключателя. Через параметр доступно новое состояние кнопки.

Класс `QPushButton` определяет свои собственные методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qpushbutton.html**):

- `setFlat(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка будет отображаться без рамки;
- `setAutoDefault(<Флаг>)` — если в качестве параметра указано значение `True`, кнопка может быть нажата с помощью клавиши <Enter>, при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>. В диалоговых окнах для всех кнопок по умолчанию указано значение `True`, а для остальных окон — значение `False`;
- `setDefault(<Флаг>)` — задает кнопку по умолчанию. Метод работает только в диалоговых окнах. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент, — например, на текстовое поле;
- `setMenu(<QMenu>)` — устанавливает всплывающее меню, которое будет отображаться при нажатии кнопки. В качестве параметра указывается экземпляр класса `QMenu`;

- `menu()` — возвращает ссылку на всплывающее меню или значение `None`;
- `showMenu()` — отображает всплывающее меню. Метод является слотом.

21.3. Переключатель

Переключатели (иногда их называют *радиокнопками*) всегда используются группами. В такой группе может быть установлен только один переключатель — при попытке установить другой переключатель ранее установленный сбрасывается. Для объединения переключателей в группу можно воспользоваться классом `QGroupBox`, который мы уже рассматривали в *разд. 20.7*, а также классом `QButtonGroup`.

Переключатель реализуется классом `QRadioButton`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QRadioButton
```

Конструктор класса `QRadioButton` имеет два формата:

```
<Объект> = QRadioButton([parent=<Родитель>])
<Объект> = QRadioButton(<Текст>[, parent=<Родитель>])
```

Класс `QRadioButton` наследует все методы класса `QAbstractButton` (см. *разд. 21.2*). Установить или сбросить переключатель позволяет метод `setChecked()`, а проверить его текущее состояние можно с помощью метода `isChecked()`. Отследить изменение состояния можно в обработчике сигнала `toggled(<Состояние>)`, в параметре которого передается логическая величина, указывающая новое состояние переключателя.

21.4. Флажок

Флажок предназначен для включения или выключения какой-либо опции и может находиться в нескольких состояниях: установленном, сброшенном и промежуточном (неопределенном) — последнее состояние может быть запрещено программно. Флажок реализуется с помощью класса `QCheckBox`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QCheckBox
```

Конструктор класса `QCheckBox` имеет два формата:

```
<Объект> = QCheckBox([parent=<Родитель>])
<Объект> = QCheckBox(<Текст>[, parent=<Родитель>])
```

Класс `QCheckBox` наследует все методы класса `QAbstractButton` (см. *разд. 21.2*), а также добавляет несколько новых:

- `setCheckState(<Статус>)` — задает состояние флажка. Могут быть указаны следующие атрибуты класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок находится в промежуточном состоянии;
  - `Checked` — `2` — флажок установлен;
- `checkState()` — возвращает текущее состояние флажка;
- `setTristate([<Флаг>=True])` — если в качестве параметра указано значение `True` (значение по умолчанию), флажок может находиться во всех трех состояниях. По умолчанию поддерживаются только установленное и сброшенное состояния;

♦ `isTristate()` — возвращает значение `True`, если флажок поддерживает три состояния, и `False` — если только два.

Чтобы перехватить изменение состояния флажка, следует назначить обработчик сигнала `stateChanged(<Состояние>)`. Через параметр внутри обработчика доступно новое состояние флажка, заданное в виде целого числа.

Если используется флажок, поддерживающий только два состояния, установить или сбросить его позволяет метод `setChecked()`, а проверить текущее состояние — метод `isChecked()`. Обработать изменение состояния можно в обработчике сигнала `toggled(<Состояние>)`, параметр которого имеет логический тип.

21.5. Однострочное текстовое поле

Однострочное текстовое поле предназначено для ввода и редактирования текста небольшого объема. С его помощью можно также отобразить вводимые символы в виде звездочек (чтобы скрыть пароль) или вообще не отображать их (что позволит скрыть длину пароля). Поле поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое.

Однострочное текстовое поле реализуется классом `QLineEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QLineEdit
```

Конструктор класса `QLineEdit` имеет два формата:

```
<Объект> = QLineEdit([parent=<Родитель>])
<Объект> = QLineEdit(<Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если родитель не указан или имеет значение `None`, компонент будет обладать своим собственным окном. Параметр `<Текст>` позволяет задать текст, который будет отображен в текстовом поле.

21.5.1. Основные методы и сигналы

Класс `QLineEdit` поддерживает следующие методы (полный их список смотрите на странице **https://doc.qt.io/qt-5/qlineedit.html**):

♦ `setText(<Текст>)` — помещает указанный текст в поле. Метод является слотом;

♦ `insert(<Текст>)` — вставляет текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, он будет удален;

♦ `text()` — возвращает текст, содержащийся в текстовом поле;

♦ `displayText()` — возвращает текст, который видит пользователь. Результат зависит от режима отображения, заданного с помощью метода `setEchoMode()`, — например, в режиме `Password` строка будет состоять из звездочек;

♦ `clear()` — удаляет весь текст из поля. Метод является слотом;

♦ `backspace()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, удаляет символ, стоящий слева от текстового курсора;

♦ `del()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, удаляет символ, стоящий справа от текстового курсора;

- ♦ `setSelection(<Индекс>, <Длина>)` — выделяет фрагмент длиной `<Длина>`, начиная с позиции `<Индекс>`. Во втором параметре можно указать отрицательное значение;
- ♦ `selectedText()` — возвращает выделенный фрагмент или пустую строку, если ничего не выделено;
- ♦ `selectAll()` — выделяет весь текст в поле. Метод является слотом;
- ♦ `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено;
- ♦ `hasSelectedText()` — возвращает значение `True`, если поле содержит выделенный фрагмент, и `False` — в противном случае;
- ♦ `deselect()` — снимает выделение;
- ♦ `isModified()` — возвращает `True`, если текст в поле был изменен пользователем, и `False` — в противном случае. Отметьте, что вызов метода `setText()` помечает поле как неизмененное;
- ♦ `setModified(<Флаг>)` — если передано значение `True`, поле ввода помечается как измененное, если `False` — как неизмененное;
- ♦ `setEchoMode(<Режим>)` — задает режим отображения текста. Могут быть указаны следующие атрибуты класса `QLineEdit`:
  - `Normal` — `0` — показывать символы как они были введены;
  - `NoEcho` — `1` — не показывать вводимые символы;
  - `Password` — `2` — вместо символов выводить звездочки (`*`);
  - `PasswordEchoOnEdit` — `3` — показывать символы при вводе, а после потери фокуса вместо них отображать звездочки (`*`);
- ♦ `setCompleter(<QCompleter>)` — позволяет предлагать возможные варианты значений, начинающиеся с введенных пользователем символов. В качестве параметра указывается экземпляр класса `QCompleter`:

```
lineEdit = QtWidgets.QLineEdit()
arr = ["кадр", "каменный", "камень", "камера"]
completer = QtWidgets.QCompleter(arr, parent=window)
lineEdit.setCompleter(completer)
```

- ♦ `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, поле будет доступно только для чтения;
- ♦ `isReadOnly()` — возвращает значение `True`, если поле доступно только для чтения, и `False` — в противном случае;
- ♦ `setAlignment(<Выравнивание>)` — задает выравнивание текста внутри поля;
- ♦ `setMaxLength(<Количество>)` — задает максимальное количество символов;
- ♦ `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, поле будет отображаться без рамки;
- ♦ `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, режим перетаскивания текста из текстового поля с помощью мыши будет включен. По умолчанию однострочное текстовое поле только принимает перетаскиваемый текст;
- ♦ `setPlaceholderText(<Текст>)` — задает текст подсказки, который будет выводиться в поле, когда оно не содержит значения и не имеет фокуса ввода;

- `setTextMargins()` — задает величины отступов от границ компонента до находящегося в нем текста. Форматы метода:

```
setTextMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)
setTextMargins(<QMargins>)
```

- `setCursorPosition(<Индекс>)` — задает положение текстового курсора;
- `cursorPosition()` — возвращает текущее положение текстового курсора;
- `cursorForward(<Флаг>[, steps=1])` — перемещает текстовый курсор вперед на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, выполняется выделение фрагмента;
- `cursorBackward(<Флаг>[, steps=1])` — перемещает текстовый курсор назад на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, выполняется выделение фрагмента;
- `cursorWordForward(<Флаг>)` — перемещает текстовый курсор вперед на одно слово. Если в параметре указано значение `True`, выполняется выделение фрагмента;
- `cursorWordBackward(<Флаг>)` — перемещает текстовый курсор назад на одно слово. Если в параметре указано значение `True`, выполняется выделение фрагмента;
- `home(<Флаг>)` — перемещает текстовый курсор в начало поля. Если в параметре указано значение `True`, выполняется выделение фрагмента;
- `end(<Флаг>)` — перемещает текстовый курсор в конец поля. Если в параметре указано значение `True`, выполняется выделение фрагмента;
- `cut()` — копирует выделенный текст в буфер обмена и удаляет его из поля при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора при условии, что поле доступно для редактирования. Метод является слотом;
- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;
- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `createStandardContextMenu()` — создает стандартное меню, которое отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню, следует создать класс, наследующий класс `QLineEdit`, и переопределить в нем метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню;
- `setClearButtonEnabled(<Флаг>)` — если передано `True`, в левой части непустого поля будет выводиться кнопка, нажатием которой можно очистить это поле, если `False`, кнопка очистки выводиться не будет.

Класс `QLineEdit` поддерживает следующие сигналы:

- `cursorPositionChanged(<Старая позиция>, <Новая позиция>)` — генерируется при перемещении текстового курсора. Внутри обработчика через первый параметр доступна старая позиция курсора, а через второй параметр — новая позиция. Оба параметра являются целочисленными;
- `editingFinished` — генерируется при нажатии клавиши <Enter> или потере полем фокуса ввода;
- `returnPressed` — генерируется при нажатии клавиши <Enter>;
- `selectionChanged` — генерируется при изменении выделения;
- `textChanged(<Новый текст>)` — генерируется при изменении текста внутри поля пользователем или программно. Внутри обработчика через параметр доступен новый текст в виде строки;
- `textEdited(<Новый текст>)` — генерируется при изменении текста внутри поля пользователем. При задании текста вызовом метода `setText()` не генерируется. Внутри обработчика через параметр доступен новый текст в виде строки.

21.5.2. Ввод данных по маске

С помощью метода `setInputMask(<Маска>)` можно ограничить ввод символов допустимым диапазоном значений. В качестве параметра указывается строка, имеющая следующий формат:

```
"<Последовательность символов>[;<Символ-заполнитель>]"
```

В первом параметре указывается комбинация из следующих специальных символов:

- `9` — обязательна цифра от `0` до `9`;
- `0` — разрешена, но не обязательна цифра от `0` до `9`;
- `D` — обязательна цифра от `1` до `9`;
- `d` — разрешена, но не обязательна цифра от `1` до `9`;
- `B` — обязательна цифра `0` или `1`;
- `b` — разрешена, но не обязательна цифра `0` или `1`;
- `H` — обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `h` — разрешен, но не обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `#` — разрешена, но не обязательна цифра, знак плюс или минус;
- `A` — обязательна буква в любом регистре;
- `a` — разрешена, но не обязательна буква;
- `N` — обязательна буква в любом регистре или цифра от `0` до `9`;
- `n` — разрешена, но не обязательна буква или цифра от `0` до `9`;
- `X` — обязателен любой символ;
- `x` — разрешен, но не обязателен любой символ;
- `>` — все последующие буквы переводятся в верхний регистр;
- `<` — все последующие буквы переводятся в нижний регистр;

- `!` — отключает изменение регистра;
- `\` — используется для отмены действия спецсимволов.

Все остальные символы трактуются как есть. В необязательном параметре `<Символ-заполнитель>` можно указать символ, который будет отображаться в поле, обозначая место ввода. Если параметр не указан, заполнителем будет служить пробел:

```
lineEdit.setInputMask("Дата: 99.B9.9999;_")  # Дата: __.__.____
lineEdit.setInputMask("Дата: 99.B9.9999;#")  # Дата: ##.##.####
lineEdit.setInputMask("Дата: 99.B9.9999 г.") # Дата:   .  .     г.
```

Проверить соответствие введенных данных маске позволяет метод `hasAcceptableInput()`. Если данные соответствуют маске, метод возвращает значение `True`, а в противном случае — `False`.

21.5.3. Контроль ввода

Контролировать ввод данных позволяет метод `setValidator(<QValidator>)`. В качестве параметра указывается экземпляр класса, наследующего класс `QValidator` из модуля `QtGui`. Существуют следующие стандартные классы, позволяющие контролировать ввод данных:

- `QIntValidator` — допускает ввод только целых чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

  ```
  QIntValidator([parent=None])
  QIntValidator(<Минимальное значение>, <Максимальное значение>
                [, parent=None])
  ```

  Пример ограничения ввода диапазоном целых чисел от `0` до `100`:

  ```
  lineEdit.setValidator(QtGui.QIntValidator(0, 100, parent=window))
  ```

- `QDoubleValidator` — допускает ввод только вещественных чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

  ```
  QDoubleValidator([parent=None])
  QDoubleValidator(<Минимальное значение>, <Максимальное значение>,
                   <Количество цифр после точки>[, parent=None])
  ```

  Пример ограничения ввода диапазоном вещественных чисел от `0.0` до `100.0`. и двумя цифрами после десятичной точки:

  ```
  lineEdit.setValidator(
           QtGui.QDoubleValidator(0.0, 100.0, 2, parent=window))
  ```

  Чтобы позволить вводить числа в экспоненциальной форме, необходимо передать значение атрибута `ScientificNotation` в метод `setNotation()`. Если передать значение атрибута `StandardNotation`, будет разрешено вводить числа только в десятичной форме:

  ```
  validator = QtGui.QDoubleValidator(0.0, 100.0, 2, parent=window)
  validator.setNotation(QtGui.QDoubleValidator.StandardNotation)
  lineEdit.setValidator(validator)
  ```

- `QRegExpValidator` — позволяет проверить данные на соответствие регулярному выражению. Форматы конструктора:

  ```
  QRegExpValidator([parent=None])
  QRegExpValidator(<QRegExp>[, parent=None])
  ```

Пример ввода только цифр от 0 до 9:

```
validator = QtGui.QRegExpValidator(
                 QtCore.QRegExp("[0-9]+"), parent=window)
lineEdit.setValidator(validator)
```

Обратите внимание, что здесь производится проверка полного соответствия шаблону, поэтому символы ^ и $ явным образом указывать не нужно.

Проверить соответствие введенных данных условию позволяет метод hasAcceptableInput(). Если данные соответствуют условию, метод возвращает значение True, а в противном случае — False.

21.6. Многострочное текстовое поле

Многострочное текстовое поле предназначено для ввода и редактирования как простого текста, так и текста в формате HTML. Поле поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Многострочное текстовое поле реализуется с помощью класса QTextEdit. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QTextEdit
```

Конструктор класса QTextEdit имеет два формата вызова:

```
<Объект> = QTextEdit([parent=<Родитель>])
<Объект> = QTextEdit(<Текст>[, parent=<Родитель>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, компонент будет обладать своим собственным окном. Параметр <Текст> позволяет задать текст в формате HTML, который будет отображен в текстовом поле.

> ***Примечание***
>
> Класс QTextEdit предназначен для отображения как простого текста, так и текста в формате HTML. Если поддержка HTML не нужна, то следует воспользоваться классом QPlainTextEdit, который оптимизирован для работы с простым текстом большого объема.

21.6.1. Основные методы и сигналы

Класс QTextEdit поддерживает следующие основные методы (полный их список смотрите на странице **https://doc.qt.io/qt-5/qtextedit.html**):

- setText(<Текст>) — помещает указанный текст в поле. Текст может быть простым или в формате HTML. Метод является слотом;
- setPlainText(<Текст>) — помещает в поле простой текст. Метод является слотом;
- setHtml(<Текст>) — помещает в поле текст в формате HTML. Метод является слотом;
- insertPlainText(<Текст>) — вставляет простой текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, он будет удален. Метод является слотом;
- insertHtml(<Текст>) — вставляет текст в формате HTML в текущую позицию текстового курсора. Если в поле был выделен фрагмент, он будет удален. Метод является слотом;

- `append(<Текст>)` — добавляет новый абзац с указанным текстом в формате HTML в конец поля. Метод является слотом;
- `setDocumentTitle(<Текст>)` — задает текст заголовка документа (для тега `<title>`);
- `documentTitle()` — возвращает текст заголовка (из тега `<title>`);
- `toPlainText()` — возвращает простой текст, содержащийся в текстовом поле;
- `toHtml()` — возвращает текст в формате HTML;
- `clear()` — удаляет весь текст из поля. Метод является слотом;
- `selectAll()` — выделяет весь текст в поле. Метод является слотом;
- `zoomIn([range=1])` — увеличивает размер шрифта. Метод является слотом;
- `zoomOut([range=1])` — уменьшает размер шрифта. Метод является слотом;
- `cut()` — копирует выделенный текст в буфер обмена и удаляет его из поля при условии, что есть выделенный фрагмент. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена при условии, что есть выделенный фрагмент. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора при условии, что поле доступно для редактирования. Метод является слотом;
- `canPaste()` — возвращает `True`, если из буфера обмена можно вставить текст, и `False` — в противном случае;
- `setAcceptRichText(<Флаг>)` — если в качестве параметра указано значение `True`, в поле можно будет ввести, вставить из буфера обмена или при помощи перетаскивания текст в формате HTML. Значение `False` дает возможность заносить в поле лишь обычный текст;
- `acceptRichText()` — возвращает значение `True`, если в поле можно занести текст в формате HTML, и `False` — если доступно занесение лишь обычного текста;
- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве значения указано значение `True`, операции отмены и повтора действий разрешены, а если `False` — то запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `createStandardContextMenu([<QPoint>])` — создает стандартное меню, которое отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню, следует создать класс, наследующий класс `QTextEdit`, и переопределить в нем метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню;
- `ensureCursorVisible()` — прокручивает область таким образом, чтобы текстовый курсор оказался в зоне видимости;
- `find()` — производит поиск фрагмента (по умолчанию в прямом направлении без учета регистра символов) в текстовом поле. Если фрагмент найден, он выделяется, и метод возвращает значение `True`, в противном случае — значение `False`. Форматы метода:

```
find(<Искомый текст>[, <Режим>])
find(<QRegExp>[, <Режим>])
```

Искомый текст можно указать либо строкой, либо регулярным выражением, представленным экземпляром класса `QRegExp`. В необязательном параметре `<Режим>` можно указать комбинацию (через оператор `|`) следующих атрибутов класса `QTextDocument` из модуля `QtGui`:

- `FindBackward` — 1 — поиск в обратном, а не в прямом направлении;
- `FindCaseSensitively` — 2 — поиск с учетом регистра символов;
- `FindWholeWords` — 4 — поиск целых слов, а не фрагментов;

- `print(<QPagedPaintDevice>)` — отправляет содержимое текстового поля на печать. В качестве параметра указывается экземпляр одного из классов, порожденных от `QPagedPaintDevice`: `QPrinter` или `QPdfWriter`. Вот пример вывода документа в файл в формате PDF:

```
pdf = QtGui.QPdfWriter("document.pdf")
textEdit.print(pdf)
```

Класс `QTextEdit` поддерживает следующие сигналы:

- `currentCharFormatChanged(<QTextCharFormat>)` — генерируется при изменении формата текста. Внутри обработчика через параметр доступен новый формат;
- `cursorPositionChanged` — генерируется при изменении положения текстового курсора;
- `selectionChanged` — генерируется при изменении выделения текста;
- `textChanged` — генерируется при изменении текста в поле;
- `copyAvailable(<Флаг>)` — генерируется при выделении текста или, наоборот, снятии выделения. Значение параметра `True` указывает, что фрагмент выделен, и его можно скопировать, значение `False` — обратное;
- `undoAvailable(<Флаг>)` — генерируется при изменении возможности отменить операцию ввода. Значение параметра `True` указывает, что операция ввода может быть отменена, значение `False` говорит об обратном;
- `redoAvailable(<Флаг>)` — генерируется при изменении возможности повторить отмененную операцию ввода. Значение параметра `True` обозначает возможность повтора отмененной операции, а значение `False` — невозможность сделать это.

21.6.2. Изменение параметров поля

Задать другие параметры поля можно вызовами следующих методов класса `QTextEdit` (полный их список смотрите на странице **https://doc.qt.io/qt-5/qtextedit.html**):

- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом. Можно указать следующие атрибуты (или их комбинацию через оператор `|`) класса `QtCore.Qt`:
  - `NoTextInteraction` — 0 — пользователь не может взаимодействовать с текстом;
  - `TextSelectableByMouse` — 1 — текст можно выделить мышью;
  - `TextSelectableByKeyboard` — 2 — текст можно выделить с помощью клавиатуры. Внутри поля будет отображен текстовый курсор;

   - `LinksAccessibleByMouse` — `4` — на гиперссылках, присутствующих в тексте, можно щелкать мышью;
   - `LinksAccessibleByKeyboard` — `8` — с гиперссылками, присутствующими в тексте, можно взаимодействовать с клавиатуры: перемещаться между гиперссылками — с помощью клавиши <Tab>, а переходить по гиперссылке — нажав клавишу <Enter>;
   - `TextEditable` — `16` — текст можно редактировать;
   - `TextEditorInteraction` — комбинация `TextSelectableByMouse | TextSelectableByKeyboard | TextEditable`;
   - `TextBrowserInteraction` — комбинация `TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard`;
- `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, поле будет доступно только для чтения;
- `isReadOnly()` — возвращает значение `True`, если поле доступно только для чтения, и `False` — в противном случае;
- `setLineWrapMode(<Режим>)` — задает режим переноса строк. В качестве значения могут быть указаны следующие атрибуты класса `QTextEdit`:
   - `NoWrap` — `0` — перенос строк не производится;
   - `WidgetWidth` — `1` — перенос строк при достижении ими ширины поля;
   - `FixedPixelWidth` — `2` — перенос строк при достижении ими фиксированной ширины в пикселах, которую можно задать с помощью метода `setLineWrapColumnOrWidth()`;
   - `FixedColumnWidth` — `3` — перенос строк при достижении ими фиксированной ширины в символах, которую можно задать с помощью метода `setLineWrapColumnOrWidth()`;
- `setLineWrapColumnOrWidth(<Значение>)` — задает фиксированную ширину строк, при достижении которой будет выполняться перенос;
- `setWordWrapMode(<Режим>)` — задает режим переноса по словам. В качестве значения могут быть указаны следующие атрибуты класса `QTextOption` из модуля `QtGui`:
   - `NoWrap` — `0` — перенос по словам не производится;
   - `WordWrap` — `1` — перенос строк только по словам;
   - `ManualWrap` — `2` — аналогичен режиму `NoWrap`;
   - `WrapAnywhere` — `3` — перенос строки может быть внутри слова;
   - `WrapAtWordBoundaryOrAnywhere` — `4` — по возможности перенос по словам, но может быть выполнен и перенос внутри слова;
- `setOverwriteMode(<Флаг>)` — если в качестве параметра указано значение `True`, вводимый текст будет замещать ранее введенный. Значение `False` отключает замещение;
- `overwriteMode()` — возвращает значение `True`, если вводимый текст замещает ранее введенный, и `False` — в противном случае;
- `setAutoFormatting(<Режим>)` — задает режим автоматического форматирования. В качестве значения могут быть указаны следующие атрибуты класса `QTextEdit`:
   - `AutoNone` — автоматическое форматирование не используется;
   - `AutoBulletList` — автоматическое создание маркированного списка при вводе пользователем в начале строки символа `*`;

- AutoAll — включить все режимы. На данный момент эквивалентно режиму AutoBulletList;

- setCursorWidth(<Ширина>) — задает ширину текстового курсора;
- setTabChangesFocus(<Флаг>) — если параметром передать значение False, то с помощью нажатия клавиши <Tab> можно вставить в поле символ табуляции. Если указано значение True, клавиша <Tab> используется для передачи фокуса следующему компоненту;
- setTabStopWidth(<Ширина>) — задает ширину табуляции в пикселах;
- tabStopWidth() — возвращает ширину табуляции в пикселах.

21.6.3. Указание параметров текста и фона

Для изменения параметров текста и фона предназначены следующие методы класса QTextEdit (полный их список смотрите на странице **https://doc.qt.io/qt-5/qtextedit.html**):

- setCurrentFont(<QFont>) — задает текущий шрифт. Метод является слотом. В качестве параметра указывается экземпляр класса QFont из модуля QtGui. Конструктор этого класса имеет следующий формат:

```
<Шрифт> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1]
               [, italic=False])
```

  В первом параметре задается название шрифта в виде строки. Необязательный параметр pointSize устанавливает размер шрифта. В параметре weight можно указать степень жирности шрифта: число от 0 до 99 или значение атрибутов Light, Normal, DemiBold, Bold или Black класса QFont. Если в параметре italic указано значение True, шрифт будет курсивным;
- currentFont() — возвращает экземпляр класса QFont с текущими характеристиками шрифта;
- setFontFamily(<Название шрифта>) — задает название текущего шрифта. Метод является слотом;
- fontFamily() — возвращает название текущего шрифта;
- setFontPointSize(<Размер>) — задает размер текущего шрифта. Метод является слотом;
- fontPointSize() — возвращает размер текущего шрифта;
- setFontWeight(<Жирность>) — задает жирность текущего шрифта. Метод является слотом;
- fontWeight() — возвращает жирность текущего шрифта;
- setFontItalic(<Флаг>) — если в качестве параметра указано значение True, шрифт будет курсивным. Метод является слотом;
- fontItalic() — возвращает True, если шрифт курсивный, и False — в противном случае;
- setFontUnderline(<Флаг>) — если в качестве параметра указано значение True, текст будет подчеркнутым. Метод является слотом;
- fontUnderline() — возвращает True, если текст подчеркнутый, и False — в противном случае;
- setTextColor(<QColor>) — задает цвет текущего текста. В качестве значения можно указать атрибут класса QtCore.Qt (например, black, white и т. д.) или экземпляр класса

QColor из модуля QtGui (например, QColor("red"), QColor("#ff0000"), QColor(255, 0, 0) и др.). Метод является слотом;

- textColor() — возвращает экземпляр класса QColor с цветом текущего текста;
- setTextBackgroundColor(<QColor>) — задает цвет фона. В качестве значения можно указать атрибут из класса QtCore.Qt (например, black, white и т. д.) или экземпляр класса QColor (например, QColor("red"), QColor("#ff0000"), QColor(255, 0, 0) и др.). Метод является слотом;
- textBackgroundColor() — возвращает экземпляр класса QColor с цветом фона;
- setAlignment(<Выравнивание>) — задает горизонтальное выравнивание текста внутри абзаца (допустимые значения мы рассматривали в *разд. 20.2*). Метод является слотом;
- alignment() — возвращает значение выравнивания текста внутри абзаца.

Задать формат символов можно также с помощью класса QTextCharFormat, который определен в модуле QtGui и поддерживает дополнительные настройки. После создания экземпляра класса его следует передать в метод setCurrentCharFormat(<QTextCharFormat>). Получить экземпляр класса с текущими настройками позволяет метод currentCharFormat(). За подробной информацией по классу QTextCharFormat обращайтесь к странице **https://doc.qt.io/qt-5/qtextcharformat.html**.

21.6.4. Класс *QTextDocument*

Класс QTextDocument из модуля QtGui представляет документ, который отображается в многострочном текстовом поле. Получить ссылку на текущий документ позволяет метод document() класса QTextEdit. Установить новый документ можно с помощью метода setDocument(<QTextDocument>). Иерархия наследования:

```
QObject — QTextDocument
```

Конструктор класса QTextDocument имеет два формата:

```
<Объект> = QTextDocument([parent=<Родитель>])
<Объект> = QTextDocument(<Текст>[, parent=<Родитель>])
```

В параметре parent указывается ссылка на родительский компонент. Параметр <Текст> позволяет задать простой текст (не в HTML-формате), который будет отображен в текстовом поле.

Класс QTextDocument поддерживает следующий набор методов (полный их список смотрите на странице **https://doc.qt.io/qt-5/qtextdocument.html**):

- setPlainText(<Текст>) — помещает в документ простой текст;
- setHtml(<Текст>) — помещает в документ текст в формате HTML;
- toPlainText() — возвращает простой текст, содержащийся в документе;
- toHtml([<QByteArray>]) — возвращает текст в формате HTML. В качестве параметра можно указать кодировку документа, которая будет выведена в теге <meta>;
- clear() — удаляет весь текст из документа;
- isEmpty() — возвращает значение True, если документ пустой, и False — в противном случае;
- setModified(<Флаг>) — если передано значение True, документ помечается как измененный, если False — как неизмененный. Метод является слотом;

- `isModified()` — возвращает значение `True`, если документ был изменен, и `False` — в противном случае;
- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;
- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то операции отмены и повтора действий разрешены, а если `False` — то запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `availableUndoSteps()` — возвращает количество возможных операций отмены;
- `availableRedoSteps()` — возвращает количество возможных повторов отмененных операций;
- `clearUndoRedoStacks([stacks=UndoAndRedoStacks])` — очищает список возможных отмен и/или повторов. В качестве параметра можно указать следующие атрибуты класса `QTextDocument`:
  - `UndoStack` — только список возможных отмен;
  - `RedoStack` — только список возможных повторов;
  - `UndoAndRedoStacks` — очищаются оба списка;
- `print(<QPagedPaintDevice>)` — отправляет содержимое документа на печать. В качестве параметра указывается экземпляр одного из классов, порожденных от `QPagedPaintDevice`: `QPrinter` или `QPdfWriter`;
- `find()` — производит поиск фрагмента в документе. Метод возвращает экземпляр класса `QTextCursor` из модуля `QtGui`. Если фрагмент не найден, то возвращенный экземпляр объекта будет нулевым. Проверить успешность операции можно с помощью метода `isNull()` класса `QTextCursor`. Форматы метода:

  ```
  find(<Текст>[, position=0][, options=0])
  find(<QRegExp>[, position=0][, options=0])
  find(<Текст>, <QTextCursor>[, options=0])
  find(<QRegExp>, <QTextCursor>[, options=0])
  ```

  Параметр `<Текст>` задает искомый фрагмент, а параметр `<QRegExp>` позволяет указать регулярное выражение. По умолчанию обычный поиск производится без учета регистра символов в прямом направлении, начиная с позиции `position` или от текстового курсора, указанного в параметре `<QTextCursor>`. Поиск по регулярному выражению по умолчанию производится с учетом регистра символов. Чтобы поиск производился без учета регистра, необходимо передать атрибут `QtCore.Qt.CaseInsensitive` в метод `setCaseSensitivity()` регулярного выражения. В необязательном параметре `options` можно указать комбинацию (через оператор `|`) следующих атрибутов класса `QTextDocument`:

- `FindBackward` — 1 — поиск в обратном, а не в прямом направлении;
- `FindCaseSensitively` — 2 — поиск с учетом регистра символов. При использовании регулярного выражения значение игнорируется;
- `FindWholeWords` — 4 — поиск целых слов, а не фрагментов;

♦ `setDefaultFont(<QFont>)` — задает шрифт по умолчанию для документа. В качестве параметра указывается экземпляр класса `QFont` из модуля `QtGui`. Конструктор класса `QFont` имеет следующий формат:

```
<Шрифт> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1]
                [, italic=False])
```

В первом параметре указывается название шрифта в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно выставить степень жирности шрифта: число от `0` до `99` или значение атрибутов `Light`, `Normal`, `DemiBold`, `Bold` или `Black` класса `QFont`. Если в параметре `italic` указано значение `True`, шрифт будет курсивным;

♦ `setDefaultStyleSheet(<CSS>)` — устанавливает для документа таблицу стилей CSS по умолчанию;

♦ `setDocumentMargin(<Отступ>)` — задает отступ от краев поля до текста;

♦ `documentMargin()` — возвращает величину отступа от краев поля до текста;

♦ `setMaximumBlockCount(<Количество>)` — задает максимальное количество текстовых блоков в документе. Если количество блоков становится больше указанного значения, первый блок будет удален;

♦ `maximumBlockCount()` — возвращает максимальное количество текстовых блоков;

♦ `characterCount()` — возвращает количество символов в документе;

♦ `lineCount()` — возвращает количество абзацев в документе;

♦ `blockCount()` — возвращает количество текстовых блоков в документе;

♦ `firstBlock()` — возвращает экземпляр класса `QTextBlock`, объявленного в модуле `QtGui`, который содержит первый текстовый блок документа;

♦ `lastBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит последний текстовый блок документа;

♦ `findBlock(<Индекс символа>)` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа, включающий символ с указанным индексом;

♦ `findBlockByLineNumber(<Индекс абзаца>)` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа, включающий абзац с указанным индексом;

♦ `findBlockByNumber(<Индекс блока>)` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа с указанным индексом.

Класс `QTextDocument` поддерживает сигналы:

♦ `undoAvailable(<Флаг>)` — генерируется при изменении возможности отменить операцию ввода. Значение параметра `True` обозначает наличие возможности отменить операцию ввода, а `False` — отсутствие такой возможности;

♦ `redoAvailable(<Флаг>)` — генерируется при изменении возможности повторить отмененную операцию ввода. Значение параметра `True` обозначает наличие возможности повторить отмененную операцию ввода, а `False` — отсутствие такой возможности;

- ♦ `undoCommandAdded` — генерируется при добавлении операции ввода в список возможных отмен;
- ♦ `blockCountChanged(<Новое количество блоков>)` — генерируется при изменении количества текстовых блоков. Внутри обработчика через параметр доступно новое количество текстовых блоков, заданное целым числом;
- ♦ `cursorPositionChanged(<QTextCursor>)` — генерируется при изменении позиции текстового курсора из-за операции редактирования. При простом перемещении текстового курсора сигнал не генерируется;
- ♦ `contentsChange(<Позиция курсора>, <Количество добавленных символов>, <Количество удаленных символов>)` — генерируется при изменении текста. Все три параметра целочисленные;
- ♦ `contentsChanged` — генерируется при любом изменении документа;
- ♦ `modificationChanged(<Флаг>)` — генерируется при изменении состояния документа: из неизмененного в измененное или наоборот. Значение параметра `True` обозначает, что документ помечен как измененный, значение `False` — что он теперь неизмененный.

21.6.5. Класс *QTextCursor*

Класс `QTextCursor` из модуля `QtGui` предоставляет инструмент для доступа к документу, представленному экземпляром класса `QTextDocument`, и для его правки, — иными словами, текстовый курсор. Конструктор класса `QTextCursor` поддерживает следующие форматы:

```
<Объект> = QTextCursor()
<Объект> = QTextCursor(<QTextDocument>)
<Объект> = QTextCursor(<QTextFrame>)
<Объект> = QTextCursor(<QTextBlock>)
<Объект> = QTextCursor(<QTextCursor>)
```

Создать текстовый курсор, установить его в документе и управлять им позволяют следующие методы класса `QTextEdit`:

- ♦ `textCursor()` — возвращает видимый в данный момент текстовый курсор (экземпляр класса `QTextCursor`). Чтобы изменения затронули текущий документ, необходимо передать этот объект в метод `setTextCursor()`;
- ♦ `setTextCursor(<QTextCursor>)` — устанавливает текстовый курсор, ссылка на который указана в качестве параметра;
- ♦ `cursorForPosition(<QPoint>)` — возвращает текстовый курсор, который соответствует позиции, указанной в качестве параметра. Позиция задается с помощью экземпляра класса `QPoint` в координатах области;
- ♦ `moveCursor(<Позиция>[, mode=MoveAnchor])` — перемещает текстовый курсор внутри документа. В первом параметре можно указать следующие атрибуты класса `QTextCursor`:
  - `NoMove` — `0` — не перемещать курсор;
  - `Start` — `1` — в начало документа;
  - `Up` — `2` — на одну строку вверх;
  - `StartOfLine` — `3` — в начало текущей строки;
  - `StartOfBlock` — `4` — в начало текущего текстового блока;

- `StartOfWord` — `5` — в начало текущего слова;
- `PreviousBlock` — `6` — в начало предыдущего текстового блока;
- `PreviousCharacter` — `7` — на предыдущий символ;
- `PreviousWord` — `8` — в начало предыдущего слова;
- `Left` — `9` — сдвинуть на один символ влево;
- `WordLeft` — `10` — влево на одно слово;
- `End` — `11` — в конец документа;
- `Down` — `12` — на одну строку вниз;
- `EndOfLine` — `13` — в конец текущей строки;
- `EndOfWord` — `14` — в конец текущего слова;
- `EndOfBlock` — `15` — в конец текущего текстового блока;
- `NextBlock` — `16` — в начало следующего текстового блока;
- `NextCharacter` — `17` — на следующий символ;
- `NextWord` — `18` — в начало следующего слова;
- `Right` — `19` — сдвинуть на один символ вправо;
- `WordRight` — `20` — в начало следующего слова.

Помимо указанных, существуют также атрибуты `NextCell`, `PreviousCell`, `NextRow` и `PreviousRow`, позволяющие перемещать текстовый курсор внутри таблицы. В необязательном параметре `mode` можно указать следующие атрибуты из класса `QTextCursor`:

- `MoveAnchor` — `0` — если существует выделенный фрагмент, выделение будет снято, и текстовый курсор переместится в новое место (значение по умолчанию);
- `KeepAnchor` — `1` — фрагмент текста от старой позиции курсора до новой будет выделен.

Класс `QTextCursor` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qtextcursor.html**):

- `isNull()` — возвращает значение `True`, если объект курсора является нулевым (создан с помощью конструктора без параметра), и `False` — в противном случае;
- `setPosition(<Позиция>[, mode=MoveAnchor])` — перемещает текстовый курсор внутри документа. В первом параметре указывается позиция внутри документа. Необязательный параметр `mode` аналогичен одноименному параметру в методе `moveCursor()` класса `QTextEdit`;
- `movePosition(<Позиция>[, mode=MoveAnchor][, n=1])` — перемещает текстовый курсор внутри документа. Параметры `<Позиция>` и `mode` аналогичны одноименным параметрам в методе `moveCursor()` класса `QTextEdit`. Необязательный параметр `n` позволяет указать количество перемещений — например, переместить курсор на 10 символов вперед можно так:

```
cur = textEdit.textCursor()
cur.movePosition(QtGui.QTextCursor.NextCharacter,
                 mode=QtGui.QTextCursor.MoveAnchor, n=10)
textEdit.setTextCursor(cur)
```

Метод `movePosition()` возвращает значение `True`, если операция успешно выполнена указанное количество раз. Если было выполнено меньшее количество перемещений (например, из-за достижения конца документа), метод возвращает значение `False`;

- ♦ `position()` — возвращает позицию текстового курсора внутри документа;
- ♦ `positionInBlock()` — возвращает позицию текстового курсора внутри блока;
- ♦ `block()` — возвращает экземпляр класса `QTextBlock`, который описывает текстовый блок, содержащий курсор;
- ♦ `blockNumber()` — возвращает номер текстового блока, содержащего курсор;
- ♦ `atStart()` — возвращает значение `True`, если текстовый курсор находится в начале документа, и `False` — в противном случае;
- ♦ `atEnd()` — возвращает значение `True`, если текстовый курсор находится в конце документа, и `False` — в противном случае;
- ♦ `atBlockStart()` — возвращает значение `True`, если текстовый курсор находится в начале блока, и `False` — в противном случае;
- ♦ `atBlockEnd()` — возвращает значение `True`, если текстовый курсор находится в конце блока, и `False` — в противном случае;
- ♦ `select(<Режим>)` — выделяет фрагмент в документе в соответствии с указанным режимом. В качестве параметра можно указать следующие атрибуты класса `QTextCursor`:
  - `WordUnderCursor` — `0` — выделяет слово, в котором расположен курсор;
  - `LineUnderCursor` — `1` — выделяет строку, в которой расположен курсор;
  - `BlockUnderCursor` — `2` — выделяет текстовый блок, в котором находится курсор;
  - `Document` — `3` — выделяет весь документ;
- ♦ `hasSelection()` — возвращает значение `True`, если существует выделенный фрагмент, и `False` — в противном случае;
- ♦ `hasComplexSelection()` — возвращает значение `True`, если выделенный фрагмент содержит сложное форматирование, а не просто текст, и `False` — в противном случае;
- ♦ `clearSelection()` — снимает выделение;
- ♦ `selectionStart()` — возвращает начальную позицию выделенного фрагмента;
- ♦ `selectionEnd()` — возвращает конечную позицию выделенного фрагмента;
- ♦ `selectedText()` — возвращает текст выделенного фрагмента;

> ***Внимание!***
>
> Если выделенный фрагмент занимает несколько строк, то вместо символа перевода строки вставляется символ с кодом `\u2029`. Попытка вывести этот символ в окно консоли приведет к исключению, поэтому следует произвести замену символа с помощью метода `replace()`:
>
> ```
> print(cur.selectedText().replace("\u2029", "\n"))
> ```

- ♦ `selection()` — возвращает экземпляр класса `QTextDocumentFragment`, который описывает выделенный фрагмент. Получить текст позволяют методы `toPlainText()` (возвращает простой текст) и `toHtml()` (возвращает текст в формате HTML) этого класса;
- ♦ `removeSelectedText()` — удаляет выделенный фрагмент;

- `deleteChar()` — если нет выделенного фрагмента, удаляет символ справа от курсора, в противном случае удаляет выделенный фрагмент;
- `deletePreviousChar()` — если нет выделенного фрагмента, удаляет символ слева от курсора, в противном случае удаляет выделенный фрагмент;
- `beginEditBlock()` и `endEditBlock()` — задают начало и конец блока инструкций. Эти инструкции могут быть отменены или повторены как единое целое с помощью методов `undo()` и `redo()`;
- `joinPreviousEditBlock()` — делает последующие инструкции частью предыдущего блока инструкций;
- `setKeepPositionOnInsert(<Флаг>)` — если в качестве параметра указано значение `True`, то после операции вставки курсор сохранит свою предыдущую позицию. По умолчанию позиция курсора при вставке изменяется;
- `insertText(<Текст>[, <QTextCharFormat>])` — вставляет простой текст;
- `insertHtml(<Текст>)` — вставляет текст в формате HTML.

С помощью методов `insertBlock()`, `insertFragment()`, `insertFrame()`, `insertImage()`, `insertList()` и `insertTable()` можно вставить различные элементы: изображения, списки и др. Изменить формат выделенного фрагмента позволяют методы `mergeBlockCharFormat()`, `mergeBlockFormat()` и `mergeCharFormat()`. За подробной информацией по этим методам обращайтесь к странице документации **https://doc.qt.io/qt-5/qtextcursor.html**.

21.7. Текстовый браузер

Класс `QTextBrowser` расширяет возможности класса `QTextEdit` и реализует текстовый браузер с возможностью перехода по гиперссылкам. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QTextEdit — QTextBrowser
```

Формат конструктора класса `QTextBrowser`:

```
<Объект> = QTextBrowser([parent=<Родитель>])
```

Класс `QTextBrowser` поддерживает следующие основные методы (полный их список смотрите на странице **https://doc.qt.io/qt-5/qtextbrowser.html**):

- `setSource(<QUrl>)` — загружает ресурс. В качестве параметра указывается экземпляр класса `QUrl` из модуля `QtCore`:

  ```
  # Загружаем и выводим содержимое текстового файла
  url = QtCore.QUrl("text.txt")
  browser.setSource(url)
  ```

  Метод является слотом;
- `source()` — возвращает экземпляр класса `QUrl` с адресом текущего ресурса;
- `reload()` — перезагружает текущий ресурс. Метод является слотом;
- `home()` — загружает первый ресурс из списка истории. Метод является слотом;
- `backward()` — загружает предыдущий ресурс из списка истории. Метод является слотом;
- `forward()` — загружает следующий ресурс из списка истории. Метод является слотом;

- `backwardHistoryCount()` — возвращает количество предыдущих ресурсов из списка истории;
- `forwardHistoryCount()` — возвращает количество следующих ресурсов из списка истории;
- `isBackwardAvailable()` — возвращает значение `True`, если существует предыдущий ресурс в списке истории, и `False` — в противном случае;
- `isForwardAvailable()` — возвращает значение `True`, если существует следующий ресурс в списке истории, и `False` — в противном случае;
- `clearHistory()` — очищает список истории;
- `historyTitle(<Количество позиций>)` — если в качестве параметра указано отрицательное число, возвращает заголовок предыдущего ресурса, отстоящего от текущего на заданное число позиций, если `0` — заголовок текущего ресурса, а если положительное число — заголовок следующего ресурса, также отстоящего от текущего на заданное число позиций;
- `historyUrl(<Количество позиций>)` — то же самое, что `historyTitle()`, но возвращает адрес ресурса в виде экземпляра класса `QUrl`;
- `setOpenLinks(<Флаг>)` — если в качестве параметра указано значение `True`, то автоматический переход по гиперссылкам будет разрешен (значение по умолчанию). Значение `False` запрещает переход.

Класс `QTextBrowser` поддерживает сигналы:

- `anchorClicked(<QUrl>)` — генерируется при переходе по гиперссылке. Внутри обработчика через параметр доступен адрес (URL) гиперссылки;
- `backwardAvailable(<Признак>)` — генерируется при изменении статуса списка предыдущих ресурсов. Внутри обработчика через параметр доступно значение `True`, если в списке истории имеются предыдущие ресурсы, и `False` — в противном случае;
- `forwardAvailable(<Признак>)` — генерируется при изменении статуса списка следующих ресурсов. В обработчике через параметр доступно значение `True`, если в списке истории имеются следующие ресурсы, и `False` — в противном случае;
- `highlighted(<QUrl>)` — генерируется при наведении указателя мыши на гиперссылку и выведении его. Внутри обработчика через параметр доступен адрес (URL) ссылки или пустой объект;
- `highlighted(<Адрес>)` — генерируется при наведении указателя мыши на гиперссылку и выведении его. Внутри обработчика через параметр доступен адрес (URL) ссылки в виде строки или пустая строка;
- `historyChanged` — генерируется при изменении списка истории;
- `sourceChanged(<Адрес>)` — генерируется при загрузке нового ресурса. Внутри обработчика через параметр доступен адрес (URL) загруженного ресурса.

21.8. Поля для ввода целых и вещественных чисел

Для ввода целых чисел предназначен класс `QSpinBox`, для ввода вещественных чисел — класс `QDoubleSpinBox`. Эти поля могут содержать две кнопки, которые позволяют щелчками

мыши увеличивать и уменьшать значение внутри поля. Пример такого поля ввода можно увидеть на рис. 21.1. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QSpinBox
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDoubleSpinBox
```

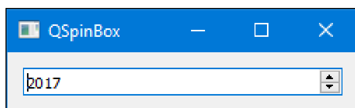

Рис. 21.1. Компонент QSpinBox

Форматы конструкторов классов QSpinBox и QDoubleSpinBox:

```
<Объект> = QSpinBox([parent=<Родитель>])
<Объект> = QDoubleSpinBox([parent=<Родитель>])
```

Классы QSpinBox и QDoubleSpinBox наследуют следующие методы из класса QAbstractSpinBox (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qabstractspinbox.html**):

- setButtonSymbols(<Режим>) — задает режим отображения кнопок, предназначенных для изменения значения поля с помощью мыши. Можно указать следующие атрибуты класса QAbstractSpinBox:
  - UpDownArrows — 0 — отображаются кнопки со стрелками;
  - PlusMinus — 1 — отображаются кнопки с символами + и –. Обратите внимание, что при использовании некоторых стилей это значение может быть проигнорировано;
  - NoButtons — 2 — кнопки не отображаются;
- setAlignment(<Режим>) — задает режим выравнивания значения внутри поля;
- setWrapping(<Флаг>) — если в качестве параметра указано значение True, то значение внутри поля будет при нажатии кнопок изменяться по кругу: максимальное значение сменится минимальным и наоборот;
- setSpecialValueText(<Строка>) — позволяет задать строку, которая будет отображаться внутри поля вместо минимального значения;
- setReadOnly(<Флаг>) — если в качестве параметра указано значение True, поле будет доступно только для чтения;
- setFrame(<Флаг>) — если в качестве параметра указано значение False, поле будет отображаться без рамки;
- stepDown() — уменьшает значение на одно приращение. Метод является слотом;
- stepUp() — увеличивает значение на одно приращение. Метод является слотом;
- stepBy(<Количество>) — увеличивает (при положительном значении) или уменьшает (при отрицательном значении) значение поля на указанное количество приращений;
- text() — возвращает текст, содержащийся внутри поля;
- clear() — очищает поле. Метод является слотом;
- selectAll() — выделяет все содержимое поля. Метод является слотом.

Класс QAbstractSpinBox поддерживает сигнал editingFinished, который генерируется при потере полем фокуса ввода или при нажатии клавиши <Enter>.

Классы `QSpinBox` и `QDoubleSpinBox` поддерживают следующие методы (здесь приведены только основные — полные их списки доступны на страницах **https://doc.qt.io/qt-5/qspinbox.html** и **https://doc.qt.io/qt-5/qdoublespinbox.html** соответственно):

- `setValue(<Число>)` — задает значение поля. Метод является слотом, принимающим, в зависимости от компонента, целое или вещественное значение;
- `value()` — возвращает целое или вещественное число, содержащееся в поле;
- `cleanText()` — возвращает целое или вещественное число в виде строки;
- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное допустимые значения;
- `setPrefix(<Текст>)` — задает текст, который будет отображаться внутри поля перед значением;
- `setSuffix(<Текст>)` — задает текст, который будет отображаться внутри поля после значения;
- `setSingleStep(<Число>)` — задает число, которое будет прибавляться или вычитаться из текущего значения поля на каждом шаге.

Класс `QDoubleSpinBox` также поддерживает метод `setDecimals(<Количество>)`, который задает количество цифр после десятичной точки.

Классы `QSpinBox` и `QDoubleSpinBox` поддерживают сигналы `valueChanged(<Целое число>)` (только в классе `QSpinBox`), `valueChanged(<Вещественное число>)` (только в классе `QDoubleSpinBox`) и `valueChanged(<Строка>)`, которые генерируются при изменении значения внутри поля. Внутри обработчика через параметр доступно новое значение в виде числа или строки в зависимости от типа параметра.

21.9. Поля для ввода даты и времени

Для ввода даты и времени предназначены классы `QDateTimeEdit` (ввод даты и времени), `QDateEdit` (ввод даты) и `QTimeEdit` (ввод времени). Поля могут содержать кнопки, которые позволяют щелчками мыши увеличивать и уменьшать значение внутри поля. Пример такого поля показан на рис. 21.2.

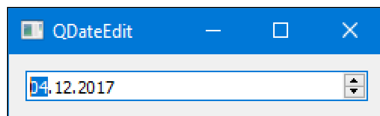

Рис. 21.2. Компонент `QDateEdit` с кнопками-стрелками

Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit — QDateEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit — QTimeEdit
```

Форматы конструкторов классов:

```
<Объект> = QDateTimeEdit([parent=<Родитель>])
<Объект> = QDateTimeEdit(<QDateTime>[, parent=<Родитель>])
<Объект> = QDateTimeEdit(<QDate>[, parent=<Родитель>])
```

```
<Объект> = QDateTimeEdit(<QTime>[, parent=<Родитель>])
<Объект> = QDateEdit([parent=<Родитель>])
<Объект> = QDateEdit(<QDate>[, parent=<Родитель>])
<Объект> = QTimeEdit([parent=<Родитель>])
<Объект> = QTimeEdit(<QTime>[, parent=<Родитель>])
```

В параметре `<QDateTime>` можно указать экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python. Преобразовать экземпляр класса `QDateTime` в экземпляр класса `datetime` позволяет метод `toPyDateTime()` класса `QDateTime`:

```
>>> from PyQt5 import QtCore
>>> d = QtCore.QDateTime()
>>> d
PyQt5.QtCore.QDateTime()
>>> d.toPyDateTime()
datetime.datetime(0, 0, 0, 255, 255, 255, 16776216)
```

В качестве параметра `<QDate>` можно указать экземпляр класса `QDate` или экземпляр класса `date` из языка Python. Преобразовать экземпляр класса `QDate` в экземпляр класса `date` позволяет метод `toPyDate()` класса `QDate`.

В параметре `<QTime>` можно указать экземпляр класса `QTime` или экземпляр класса `time` из языка Python. Преобразовать экземпляр класса `QTime` в экземпляр класса `time` позволяет метод `toPyTime()` класса `QTime`.

Классы `QDateTime`, `QDate` и `QTime` определены в модуле `QtCore`.

Класс `QDateTimeEdit` наследует все методы из класса `QAbstractSpinBox` (см. *разд. 21.8*) и дополнительно реализует следующие методы (здесь приведены только самые полезные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qdatetimeedit.html**):

- `setDateTime(<QDateTime>)` — устанавливает дату и время. В качестве параметра указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python. Метод является слотом;
- `setDate(<QDate>)` — устанавливает дату. В качестве параметра указывается экземпляр класса `QDate` или экземпляр класса `date` языка Python. Метод является слотом;
- `setTime(<QTime>)` — устанавливает время. В качестве параметра указывается экземпляр класса `QTime` или экземпляр класса `time` из языка Python. Метод является слотом;
- `dateTime()` — возвращает экземпляр класса `QDateTime` с датой и временем;
- `date()` — возвращает экземпляр класса `QDate` с датой;
- `time()` — возвращает экземпляр класса `QTime` со временем;
- `setDateTimeRange(<Минимум>, <Максимум>)`, `setMinimumDateTime(<Минимум>)` и `setMaximumDateTime(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты и времени. В параметрах указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python;
- `setDateRange(<Минимум>, <Максимум>)`, `setMinimumDate(<Минимум>)` и `setMaximumDate(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты. В параметрах указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python;
- `setTimeRange(<Минимум>, <Максимум>)`, `setMinimumTime(<Минимум>)` и `setMaximumTime(<Максимум>)` — задают минимальное и максимальное допустимые значения для време-

ни. В параметрах указывается экземпляр класса `QTime` или экземпляр класса `time` из языка Python;

- `setDisplayFormat(<Формат>)` — задает формат отображения даты и времени. В качестве параметра указывается строка, содержащая специальные символы. Пример задания строки формата:

```
dateTimeEdit.setDisplayFormat("dd.MM.yyyy HH:mm:ss")
```

- `setTimeSpec(<Зона>)` — задает зону времени. В качестве параметра можно указать атрибуты `LocalTime`, `UTC` или `OffsetFromUTC` класса `QtCore.Qt`;
- `setCalendarPopup(<Флаг>)` — если в качестве параметра указано значение `True`, то дату можно будет выбрать с помощью календаря, который появится на экране при щелчке на кнопке с направленной вниз стрелкой, выведенной вместо стандартных кнопок-стрелок (рис. 21.3);

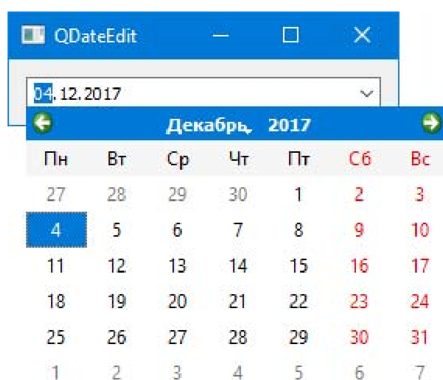

Рис. 21.3. Компонент `QDateEdit` с открытым календарем

- `setSelectedSection(<Секция>)` — выделяет указанную секцию. В качестве параметра можно задать атрибуты `NoSection`, `DaySection`, `MonthSection`, `YearSection`, `HourSection`, `MinuteSection`, `SecondSection`, `MSecSection` или `AmPmSection` класса `QDateTimeEdit`;
- `setCurrentSection(<Секция>)` — делает указанную секцию текущей;
- `setCurrentSectionIndex(<Индекс>)` — делает секцию с указанным индексом текущей;
- `currentSection()` — возвращает тип текущей секции;
- `currentSectionIndex()` — возвращает индекс текущей секции;
- `sectionCount()` — возвращает количество секций внутри поля;
- `sectionAt(<Индекс>)` — возвращает тип секции по указанному индексу;
- `sectionText(<Секция>)` — возвращает текст указанной секции.

При изменении значений даты или времени генерируются сигналы `timeChanged(<QTime>)`, `dateChanged(<QDate>)` и `dateTimeChanged(<QDateTime>)`. Внутри обработчиков через параметр доступно новое значение.

Классы `QDateEdit` (поле для ввода даты) и `QTimeEdit` (поле для ввода времени) созданы для удобства и отличаются от класса `QDateTimeEdit` только форматом отображаемых данных. Эти классы наследуют методы базовых классов и не добавляют никаких своих методов.

21.10. Календарь

Класс QCalendarWidget реализует календарь с возможностью выбора даты и перемещения по месяцам с помощью мыши и клавиатуры (рис. 21.4). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QCalendarWidget
```

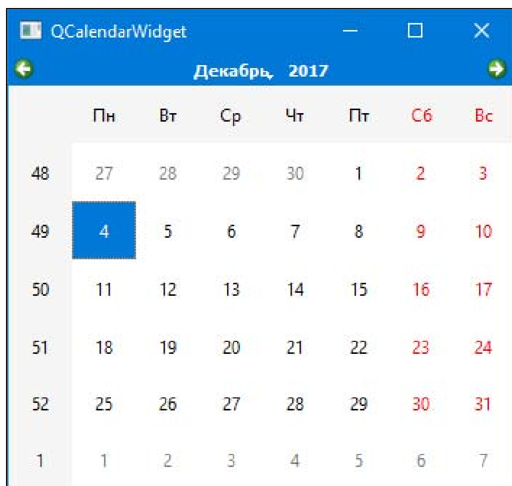

Рис. 21.4. Компонент QCalendarWidget

Формат конструктора класса QCalendarWidget:

```
<Объект> = QCalendarWidget([parent=<Родитель>])
```

Класс QCalendarWidget поддерживает следующие методы (здесь представлены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qcalendarwidget.html**):

- setSelectedDate(<QDate>) — устанавливает дату, заданную в качестве параметра экземпляром класса QDate или экземпляром класса date языка Python. Метод является слотом;
- selectedDate() — возвращает экземпляр класса QDate с выбранной датой;
- setDateRange(<Минимум>, <Максимум>), setMinimumDate(<Минимум>) и setMaximumDate(<Максимум>) — задают минимальное и максимальное допустимые значения для даты. В параметрах указывается экземпляр класса QDate или экземпляр класса date из языка Python. Метод setDateRange() является слотом;
- setCurrentPage(<Год>, <Месяц>) — делает текущей страницу календаря с указанными годом и месяцем, которые задаются целыми числами. Выбранная дата при этом не изменяется. Метод является слотом;
- monthShown() — возвращает месяц (число от 1 до 12), отображаемый на текущей странице;
- yearShown() — возвращает год, отображаемый на текущей странице;
- showSelectedDate() — отображает страницу с выбранной датой. Выбранная дата при этом не изменяется. Метод является слотом;
- showToday() — отображает страницу с сегодняшней датой. Выбранная дата при этом не изменяется. Метод является слотом;

- `showPreviousMonth()` — отображает страницу с предыдущим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showNextMonth()` — отображает страницу со следующим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showPreviousYear()` — отображает страницу с текущим месяцем в предыдущем году. Выбранная дата не изменяется. Метод является слотом;
- `showNextYear()` — отображает страницу с текущим месяцем в следующем году. Выбранная дата при этом не изменяется. Метод является слотом;
- `setFirstDayOfWeek(<День>)` — задает первый день недели. По умолчанию используется воскресенье. Чтобы первым днем недели сделать понедельник, следует в качестве параметра указать атрибут `Monday` класса `QtCore.Qt`;
- `setNavigationBarVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то панель навигации выводиться не будет. Метод является слотом;
- `setHorizontalHeaderFormat(<Формат>)` — задает формат горизонтального заголовка. В качестве параметра можно указать следующие атрибуты класса `QCalendarWidget`:
  - `NoHorizontalHeader` — `0` — заголовок не отображается;
  - `SingleLetterDayNames` — `1` — отображается только первая буква из названия дня недели;
  - `ShortDayNames` — `2` — отображается сокращенное название дня недели;
  - `LongDayNames` — `3` — отображается полное название дня недели;
- `setVerticalHeaderFormat(<Формат>)` — задает формат вертикального заголовка. В качестве параметра можно указать следующие атрибуты класса `QCalendarWidget`:
  - `NoVerticalHeader` — `0` — заголовок не отображается;
  - `ISOWeekNumbers` — `1` — отображается номер недели в году;
- `setGridVisible(<Флаг>)` — если в качестве параметра указано значение `True`, линии сетки будут отображены. Метод является слотом;
- `setSelectionMode(<Режим>)` — задает режим выделения даты. В качестве параметра можно указать следующие атрибуты класса `QCalendarWidget`:
  - `NoSelection` — `0` — дата не может быть выбрана пользователем;
  - `SingleSelection` — `1` — может быть выбрана одна дата;
- `setHeaderTextFormat(<QTextCharFormat>)` — задает формат ячеек заголовка. В параметре указывается экземпляр класса `QTextCharFormat` из модуля `QtGui`;
- `setWeekdayTextFormat(<День недели>, <QTextCharFormat>)` — определяет формат ячеек для указанного дня недели. В первом параметре задаются атрибуты `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday` или `Sunday` класса `QtCore.Qt`, а во втором параметре — экземпляр класса `QTextCharFormat`;
- `setDateTextFormat(<QDate>, <QTextCharFormat>)` — задает формат ячейки с указанной датой. В первом параметре указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python, а во втором параметре — экземпляр класса `QTextCharFormat`.

Класс `QCalendarWidget` поддерживает такие сигналы:

- `activated(<QDate>)` — генерируется при двойном щелчке мышью или нажатии клавиши <Enter>. Внутри обработчика через параметр доступна текущая дата;

- `clicked(<QDate>)` — генерируется при щелчке мышью на доступной дате. Внутри обработчика через параметр доступна выбранная дата;
- `currentPageChanged(<Год>, <Месяц>)` — генерируется при изменении страницы. Внутри обработчика через первый параметр доступен год, а через второй — месяц. Обе величины задаются целыми числами;
- `selectionChanged` — генерируется при изменении выбранной даты пользователем или из программного кода.

21.11. Электронный индикатор

Класс `QLCDNumber` реализует электронный индикатор, в котором цифры и буквы отображаются отдельными сегментами — как на электронных часах или дисплее калькулятора (рис. 21.5). Индикатор позволяет отображать числа в двоичной, восьмеричной, десятичной и шестнадцатеричной системах счисления. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLCDNumber
```

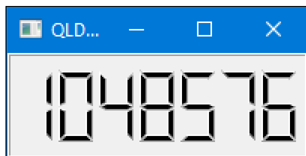

Рис. 21.5. Компонент `QLCDNumber`

Форматы конструктора класса `QLCDNumber`:

```
<Объект> = QLCDNumber([parent=<Родитель>])
<Объект> = QLCDNumber(<Количество цифр>[, parent=<Родитель>])
```

В параметре `<Количество цифр>` указывается количество отображаемых цифр — если оно не указано, используется значение `5`.

Класс `QLCDNumber` поддерживает следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qlcdnumber.html**):

- `display(<Значение>)` — задает новое значение. В качестве параметра можно указать целое число, вещественное число или строку:

  ```
  lcd.display(1048576)
  ```

  Метод является слотом;
- `checkOverflow(<Число>)` — возвращает значение `True`, если целое или вещественное число, указанное в параметре, не может быть отображено индикатором. В противном случае возвращает значение `False`;
- `intValue()` — возвращает значение индикатора в виде целого числа;
- `value()` — возвращает значение индикатора в виде вещественного числа;
- `setSegmentStyle(<Стиль>)` — задает стиль индикатора. В качестве параметра можно указать атрибуты `Outline`, `Filled` или `Flat` класса `QLCDNumber`;
- `setMode(<Режим>)` — задает режим отображения чисел. В качестве параметра можно указать следующие атрибуты класса `QLCDNumber`:

- `Hex` — `0` — шестнадцатеричное значение;
- `Dec` — `1` — десятичное значение;
- `Oct` — `2` — восьмеричное значение;
- `Bin` — `3` — двоичное значение.

Вместо метода `setMode()` удобнее воспользоваться методами-слотами `setHexMode()`, `setDecMode()`, `setOctMode()` и `setBinMode()`;

- ♦ `setSmallDecimalPoint(<Флаг>)` — если в качестве параметра указано значение `True`, десятичная точка будет отображаться как отдельный элемент (при этом значение выводится более компактно без пробелов до и после точки), а если значение `False` — то десятичная точка будет занимать позицию цифры (значение используется по умолчанию). Метод является слотом;
- ♦ `setDigitCount(<Число>)` — задает количество отображаемых цифр. Если в методе `setSmallDecimalPoint()` указано значение `False`, десятичная точка считается отдельной цифрой.

Класс `QLCDNumber` поддерживает сигнал `overflow`, генерируемый при попытке задать значение, которое не может быть отображено индикатором.

21.12. Индикатор хода процесса

Класс `QProgressBar` реализует индикатор хода процесса, с помощью которого можно информировать пользователя о текущем состоянии выполнения длительной операции. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QProgressBar
```

Формат конструктора класса `QProgressBar`:

```
<Объект> = QProgressBar([parent=<Родитель>])
```

Класс `QProgressBar` поддерживает следующий набор методов, которые могут быть нам полезны (полный их список смотрите на странице **https://doc.qt.io/qt-5/qprogressbar.html**):

- ♦ `setValue(<Значение>)` — задает новое целочисленное значение. Метод является слотом;
- ♦ `value()` — возвращает текущее значение индикатора в виде числа;
- ♦ `text()` — возвращает текст, отображаемый на индикаторе или рядом с ним;
- ♦ `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное значения в виде целых чисел. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы являются слотами;
- ♦ `reset()` — сбрасывает значение индикатора. Метод является слотом;
- ♦ `setOrientation(<Ориентация>)` — задает ориентацию индикатора. В качестве значения указываются атрибуты `Horizontal` или `Vertical` класса `QtCore.Qt`. Метод является слотом;
- ♦ `setTextVisible(<Флаг>)` — если в качестве параметра указано значение `False`, текст с текущим значением индикатора отображаться не будет;
- ♦ `setTextDirection(<Направление>)` — задает направление вывода текста при вертикальной ориентации индикатора. Обратите внимание, что при использовании стилей

"windows", "windowsxp" и "macintosh" при вертикальной ориентации текст вообще не отображается. В качестве значения указываются следующие атрибуты класса QProgressBar:

- TopToBottom — 0 — текст поворачивается на 90 градусов по часовой стрелке;
- BottomToTop — 1 — текст поворачивается на 90 градусов против часовой стрелки;

♦ setInvertedAppearance(<Флаг>) — если в качестве параметра указано значение True, направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево — при горизонтальной ориентации);

♦ setFormat(<Формат>) — задает формат вывода текстового представления значения. Параметром передается строка формата, в которой могут использоваться следующие специальные символы: %v — само текущее значение, %m — заданный методами setMaximun() или setRange() максимум, %p — текущее значение в процентах:

```
lcd.setFormat('Выполнено %v шагов из %m')
```

При изменении значения индикатора генерируется сигнал valueChanged(<Новое значение>). Внутри обработчика через параметр доступно новое значение, заданное целым числом.

21.13. Шкала с ползунком

Класс QSlider реализует шкалу с ползунком, который можно перемещать с помощью мыши или клавиатуры. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QSlider
```

Форматы конструктора класса QSlider:

```
<Объект> = QSlider([parent=<Родитель>])
<Объект> = QSlider(<Ориентация>[, parent=<Родитель>])
```

Параметр <Ориентация> позволяет задать ориентацию шкалы. В качестве значения указываются атрибуты Horizontal или Vertical (значение по умолчанию) класса QtCore.Qt.

Класс QSlider наследует следующие методы из класса QAbstractSlider (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qabstractslider.html**):

♦ setValue(<Значение>) — задает новое целочисленное значение. Метод является слотом;

♦ value() — возвращает текущее значение в виде числа;

♦ setSliderPosition(<Значение>) — задает текущее положение ползунка;

♦ sliderPosition() — возвращает текущее положение ползунка в виде числа. Если отслеживание перемещения ползунка включено (принято по умолчанию), то возвращаемое значение будет совпадать со значением, возвращаемым методом value(). Если отслеживание выключено, то при перемещении метод sliderPosition() вернет текущее положение, а метод value() — положение, которое имел ползунок до перемещения;

♦ setRange(<Минимум>, <Максимум>), setMinimum(<Минимум>) и setMaximum(<Максимум>) — задают минимальное и максимальное значения, представленные целыми числами. Метод setRange() является слотом;

♦ setOrientation(<Ориентация>) — задает ориентацию шкалы. В качестве значения указываются атрибуты Horizontal или Vertical класса QtCore.Qt. Метод является слотом;

- `setSingleStep(<Значение>)` — задает значение, на которое сдвинется ползунок при нажатии клавиш со стрелками;
- `setPageStep(<Значение>)` — задает значение, на которое сдвинется ползунок при нажатии клавиш <Page Up> и <Page Down>, повороте колесика мыши или щелчке мышью на шкале;
- `setInvertedAppearance(<Флаг>)` — если в качестве параметра указано значение `True`, направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево — при горизонтальной ориентации);
- `setInvertedControls(<Флаг>)` — если в качестве параметра указано значение `False`, то при изменении направления увеличения значения будет изменено и направление перемещения ползунка при нажатии клавиш <Page Up> и <Page Down>, повороте колесика мыши и нажатии клавиш со стрелками вверх и вниз;
- `setTracking(<Флаг>)` — если в качестве параметра указано значение `True`, отслеживание перемещения ползунка будет включено (принято по умолчанию). При этом сигнал `valueChanged` при перемещении ползунка станет генерироваться постоянно. Если в качестве параметра указано значение `False`, то сигнал `valueChanged` будет сгенерирован только при отпускании ползунка;
- `hasTracking()` — возвращает значение `True`, если отслеживание перемещения ползунка включено, и `False` — в противном случае.

Класс `QAbstractSlider` поддерживает сигналы:

- `actionTriggered(<Действие>)` — генерируется, когда производится взаимодействие с ползунком (например, при нажатии клавиши <Page Up>). Внутри обработчика через параметр доступно произведенное действие, которое описывается целым числом. Также можно использовать атрибуты `SliderNoAction` (0), `SliderSingleStepAdd` (1), `SliderSingleStepSub` (2), `SliderPageStepAdd` (3), `SliderPageStepSub` (4), `SliderToMinimum` (5), `SliderToMaximum` (6) и `SliderMove` (7) класса `QAbstractSlider`;
- `rangeChanged(<Минимум>, <Максимум>)` — генерируется при изменении диапазона значений. Внутри обработчика через параметры доступны новые минимальное и максимальное значения, заданные целыми числами;
- `sliderPressed` — генерируется при нажатии ползунка;
- `sliderMoved(<Положение>)` — генерируется постоянно при перемещении ползунка. Внутри обработчика через параметр доступно новое положение ползунка, выраженное целым числом;
- `sliderReleased` — генерируется при отпускании ранее нажатого ползунка;
- `valueChanged(<Значение>)` — генерируется при изменении значения. Внутри обработчика через параметр доступно новое значение в виде целого числа.

Класс `QSlider` дополнительно определяет следующие методы (здесь приведены только основные — полный их список смотрите на странице **https://doc.qt.io/qt-5/qslider.html**):

- `setTickPosition(<Позиция>)` — задает позицию рисок. В качестве параметра указываются следующие атрибуты класса `QSlider`:
  - `NoTicks` — без рисок;
  - `TicksBothSides` — риски по обе стороны;
  - `TicksAbove` — риски выводятся сверху;

- `TicksBelow` — риски выводятся снизу;
- `TicksLeft` — риски выводятся слева;
- `TicksRight` — риски выводятся справа;

♦ `setTickInterval(<Расстояние>)` — задает расстояние между рисками.

21.14. Круговая шкала с ползунком

Класс `QDial` реализует круглую шкалу с ползунком, который можно перемещать по кругу с помощью мыши или клавиатуры. Компонент, показанный на рис. 21.6, напоминает регулятор, используемый в различных устройствах для изменения или отображения каких-либо настроек. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QDial
```

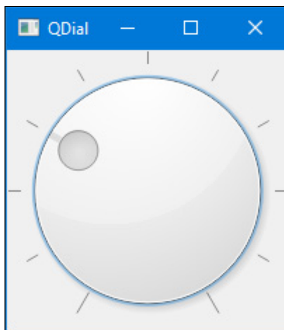

Рис. 21.6. Компонент `QDial`

Формат конструктора класса `QDial`:

```
<Объект> = QDial([parent=<Родитель>])
```

Класс `QDial` наследует все методы и сигналы класса `QAbstractSlider` (см. *разд. 21.13*) и определяет несколько дополнительных методов (здесь приведена только часть методов — полный их список смотрите на странице **https://doc.qt.io/qt-5/qdial.html**):

♦ `setNotchesVisible(<Флаг>)` — если в качестве параметра указано значение `True`, будут отображены риски. По умолчанию риски не выводятся. Метод является слотом;

♦ `setNotchTarget(<Значение>)` — задает рекомендуемое количество пикселов между рисками. В качестве параметра указывается вещественное число;

♦ `setWrapping(<Флаг>)` — если в качестве параметра указано значение `True`, то начало шкалы будет совпадать с ее концом. По умолчанию между началом шкалы и концом расположено пустое пространство. Метод является слотом.

21.15. Полоса прокрутки

Класс `QScrollBar` представляет горизонтальную или вертикальную полосу прокрутки. Изменить положение ползунка на полосе можно нажатием на кнопки, расположенные по краям полосы, щелчками мышью на полосе, собственно перемещением ползунка мышью,

нажатием клавиш на клавиатуре, а также выбрав соответствующий пункт из контекстного меню. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QScrollBar
```

Форматы конструктора класса `QScrollBar`:

```
<Объект> = QScrollBar([parent=<Родитель>])
<Объект> = QScrollBar(<Ориентация>[, parent=<Родитель>])
```

Параметр `<Ориентация>` позволяет задать ориентацию полосы прокрутки. В качестве значения указываются атрибуты `Horizontal` или `Vertical` (значение по умолчанию) класса `QtCore.Qt`.

Класс `QScrollBar` наследует все методы и сигналы класса `QAbstractSlider` (см. *разд. 21.13*) и не определяет дополнительных методов.

> **ПРИМЕЧАНИЕ**
>
> Полоса прокрутки редко используется отдельно. Гораздо удобнее воспользоваться областью с полосами прокрутки, которую реализует класс `QScrollArea` (см. *разд. 20.12*).

21.16. Веб-браузер

Класс `QWebEngineView`, определенный в модуле `QtWebEngineWidgets`, реализует полнофункциональный веб-браузер, поддерживающий HTML, CSS, JavaScript, вывод изображений и пр. (на рис. 21.7 в этом компоненте выведена главная страница Google). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QWebEngineView
```

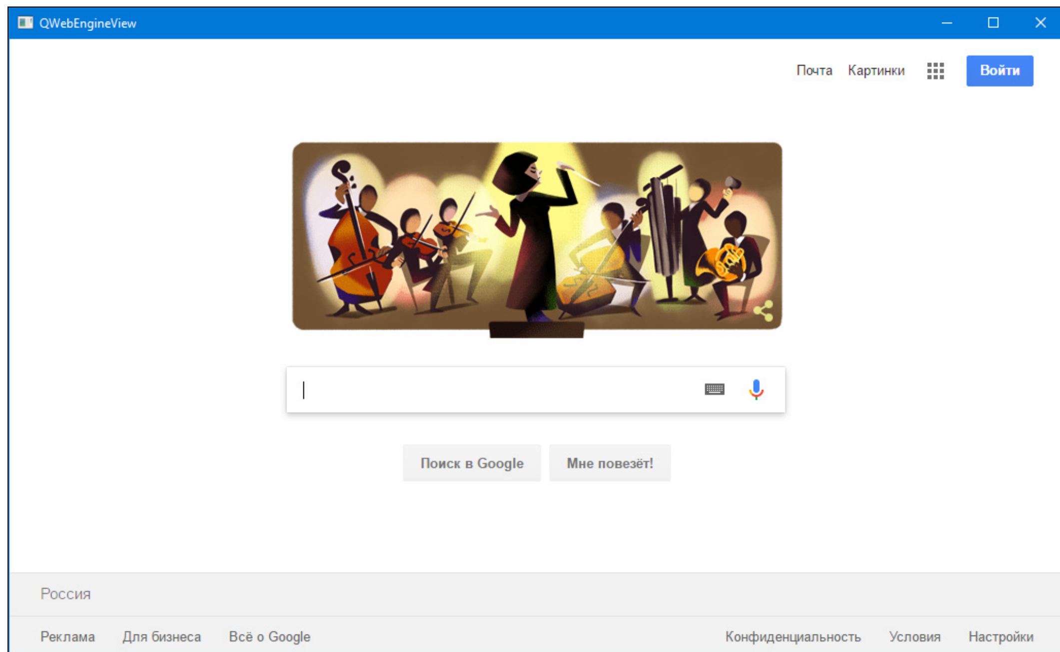

Рис. 21.7. Компонент `QWebEngineView`

Формат конструктора класса `QWebEngineView`:

```
<Объект> = QWebEngineView([parent=<Родитель>])
```

> ***Примечание***
>
> До версии 5.5 библиотеки PyQt вывод веб-страниц осуществлялся средствами класса `QWebView` из модуля `QtWebKitWidgets`. Однако в версии 5.5 этот модуль был объявлен нерекомендованным к использованию, а в версии 5.6 — удален.

Класс `QWebEngineView` поддерживает следующие полезные для нас методы (полный их список смотрите на странице **https://doc.qt.io/qt-5/qwebengineview.html**):

- `load(<QUrl>)` и `setUrl(<QUrl>)` — загружают и выводят страницу с указанным в параметре адресом, который задается в виде экземпляра класса `QUrl` из модуля `QtCore`:

  ```
  web.load(QtCore.QUrl('https://www.google.ru/'))
  ```

- `url()` — возвращает адрес текущей страницы в виде экземпляра класса `QUrl`;
- `title()` — возвращает заголовок (содержимое тега `<title>`) текущей страницы;
- `setHtml(<HTML-код>[, baseUrl=QUrl()])` — задает HTML-код страницы, которая будет отображена в компоненте.

  Необязательный параметр `baseUrl` указывает базовый адрес, относительно которого будут отсчитываться относительные адреса в гиперссылках, ссылках на файлы изображений, таблицы стилей, файлы сценариев и пр. Если этот параметр не указан, в качестве значения по умолчанию используется «пустой» экземпляр класса `QUrl`, и относительные адреса будут отсчитываться от каталога, где находится сам файл страницы:

  ```
  wv.setHtml("<h1>Заголовок</h1>")
  # Файл page2.html будет загружен с сайта http://www.somesite.ru/
  webview.setHtml("<a href='page2.html'>Вторая страница</a>", ↳
  QtCore.QUrl('http://www.somesite.ru/'))
  ```

- `selectedText()` — возвращает выделенный текст или пустую строку, если ничего не было выделено;
- `hasSelection()` — возвращает `True`, если фрагмент страницы был выделен, и `False` — в противном случае;
- `setZoomFactor(<Множитель>)` — задает масштаб самой страницы. Значение `1` указывает, что страница будет выведена в оригинальном масштабе, значение меньше единицы — в уменьшенном, значение больше единицы — увеличенном масштабе;
- `zoomFactor()` — возвращает масштаб страницы;
- `back()` — загружает предыдущий ресурс из списка истории. Метод является слотом;
- `forward()` — загружает следующий ресурс из списка истории. Метод является слотом;
- `reload()` — перезагружает страницу. Метод является слотом;
- `stop()` — останавливает загрузку страницы. Метод является слотом;
- `icon()` — возвращает в виде экземпляра класса `QIcon` значок, заданный для страницы;
- `findText(<Искомый текст>[, options=QWebEnginePage.FindFlags()][, resultCallback=0])` — позволяет найти на странице заданный фрагмент текста. Все найденные фрагменты будут выделены желтым фоном непосредственно в компоненте `QWebEngineView`.

Необязательный параметр `option` позволяет указать дополнительные параметры в виде экземпляра класса `QWebEnginePage.FindFlags` из того же модуля `QtWebEngineWidgets`. В конструкторе этого класса можно указать один из атрибутов класса `QWebEnginePage` из модуля `QtWebEngineWidgets` или их комбинации через оператор `|`:

- `FindBackward` — `1` — выполнять поиск в обратном, а не в прямом направлении;
- `FindCaseSensitively` — `2` — поиск с учетом регистра символов (по умолчанию выполняется поиск без учета регистра).

В необязательном параметре `resultCallback` можно указать функцию, которая будет вызвана по окончании поиска. Эта функция должна принимать единственный параметр, которым станет логическая величина `True`, если поиск увенчался успехом, или `False` — в противном случае.

Вот пример поиска с учетом регистра и выделением всех найденных совпадений:

```
web.findText('Python', options=QtWebEngineWidgets.QWebEnginePage.FindFlags(
    QtWebEngineWidgets.QWebEnginePage.FindBackward |
    QtWebEngineWidgets.QWebEnginePage.FindCaseSensitively))
```

♦ `triggerPageAction(<Действие>[, checked=False])` — выполняет над страницей указанное действие. В качестве действия задается один из атрибутов класса `QWebEnginePage` — этих атрибутов очень много, и все они приведены на странице **https://doc.qt.io/qt-5/qwebenginepage.html#WebAction-enum**. Необязательный параметр `checked` имеет смысл указывать лишь для действий, принимающих логический флаг:

```
# Выделение всей страницы
web.triggerPageAction(QtWebEngineWidgets.QWebEnginePage.SelectAll)
# Копирование выделенного фрагмента страницы
web.triggerPageAction(QtWebEngineWidgets.QWebEnginePage.Copy)
# Перезагрузка страницы, минуя кэш
web.triggerPageAction(QtWebEngineWidgets.QWebEnginePage.ReloadAndBypassCache)
```

♦ `page()` — возвращает экземпляр класса `QWebEnginePage` из модуля `QtWebEngineWidgets`, представляющий открытую веб-страницу.

Класс `QWebEngineView` поддерживает следующий набор полезных для нас сигналов (полный их список смотрите на странице **https://doc.qt.io/qt-5/qwebengineview.html**):

♦ `iconChanged` — генерируется после загрузки или изменения значка, заданного для страницы;

♦ `loadFinished(<Признак>)` — генерируется по окончании загрузки страницы. Значение `True` параметра указывает, что загрузка выполнена без проблем, `False` — что при загрузке произошли ошибки;

♦ `loadProgress(<Процент выполнения>)` — периодически генерируется в процессе загрузки страницы. В качестве параметра передается целое число от `0` до `100`, показывающее процент загрузки;

♦ `loadStarted` — генерируется после начала загрузки страницы;

♦ `selectionChanged` — генерируется при выделении нового фрагмента содержимого страницы;

♦ `titleChanged(<Текст>)` — генерируется при изменении текста заголовка страницы (содержимого тега `<title>`). В параметре, передаваемом обработчику, доступен этот текст в виде строки;

♦ `urlChanged(<QUrl>)` — генерируется при изменении адреса текущей страницы, что может быть вызвано, например, загрузкой новой страницы. Параметр — новый адрес.

Класс `QWebEnginePage`, представляющий открытую в веб-браузере страницу и получаемый вызовом метода `page()` класса `QWebEngineView`, поддерживает следующие полезные для нас методы (полный их список смотрите на странице **https://doc.qt.io/qt-5/qwebenginepage.html**):

♦ `print(<QPrinter>, <Функция>)` — печатает содержимое страницы на заданном принтере. Вторым параметром указывается функция, которая будет вызвана после окончания печати и должна принимать единственный параметр, которым будет значение `True`, если печать завершилась успешно, и `False` — в противном случае. Метод поддерживается PyQt, начиная с версии 5.8;

♦ `printToPdf()` — преобразует страницу в формат PDF. Форматы метода:

```
printToPdf(<Путь файла>,
           pageLayout=QPageLayout(QPageSize(QPageSize::A4),
           QPageLayout::Portrait, QMarginsF()))
printToPdf(<Функция>,
           pageLayout=QPageLayout(QPageSize(QPageSize::A4),
           QPageLayout::Portrait, QMarginsF()))
```

Первый формат сразу сохраняет преобразованную страницу в файле, чей путь указан первым параметром. Второй формат после преобразования вызывает указанную в первом параметре функцию, передавая ей в качестве единственного параметра массив типа `bytes`, хранящий преобразованную страницу.

Необязательный параметр `pageLayout` задает настройки страницы в виде экземпляра класса `QPageLayout` (он будет описан в *главе 29*). Если параметр не указан, будет выполнена печать на бумаге типоразмера A4, в портретной ориентации, без отступов.

Метод поддерживается PyQt, начиная с версии 5.7;

♦ `save(<Путь файла>, format=QWebEngineDownloadItem::MimeHtmlSaveFormat)` — сохраняет страницу в файле, чей путь указан в первом параметре. Необязательный параметр `format` задает формат файла — для него можно задать один из следующих атрибутов класса `QWebEngineDownloadItem`, определенного в модуле `QtWebEngineWidgets`:

- `SingleHtmlSaveFormat` — `0` — обычный HTML-файл. Связанные со страницей файлы (изображения, аудио- и видеоролики, таблицы стилей и пр.) не сохраняются;
- `CompleteHtmlSaveFormat` — `1` — то же самое, только связанные файлы будут сохранены в каталоге, находящемся там же, где и файл со страницей, и имеющем то же имя;
- `MimeHtmlSaveFormat` — `2` — страница и все связанные файлы сохраняются в одном файле.

Метод поддерживается PyQt, начиная с версии 5.8;

♦ `contentsSize()` — возвращает экземпляр класса `QSizeF`, хранящий размеры содержимого страницы. Метод поддерживается PyQt, начиная с версии 5.7;

♦ `scrollPosition()` — возвращает экземпляр класса `QPointF`, хранящий позицию прокрутки содержимого страницы. Метод поддерживается PyQt, начиная с версии 5.7;

♦ `setAudioMuted(<Флаг>)` — если с параметром передать значение `True`, все звуки, воспроизводящиеся на странице, будут приглушены. Чтобы снова сделать их слышимыми, нужно передать значение `False`. Метод поддерживается PyQt, начиная с версии 5.7;

- `isAudioMuted()` — возвращает `True`, если все звуки, воспроизводящиеся на странице, приглушены, и `False` — в противном случае. Метод поддерживается PyQt, начиная с версии 5.7;
- `setBackgroundColor(<Цвет>)` — задает для страницы фоновый цвет, указанный в виде экземпляра класса `QColor`. Метод поддерживается PyQt, начиная с версии 5.6;
- `backgroundColor()` — возвращает фоновый цвет страницы в виде экземпляра класса `QColor`. Метод поддерживается PyQt, начиная с версии 5.6.

ГЛАВА 22

Списки и таблицы

PyQt содержит широкий выбор компонентов, позволяющих отображать как простой список строк (в свернутом или развернутом состояниях), так и табличные данные. Кроме того, можно отобразить данные, которые имеют очень сложную структуру, — например, иерархическую. Благодаря поддержке концепции «модель-представление», позволяющей отделить данные от их отображения, одни и те же данные можно отображать сразу в нескольких компонентах без их дублирования.

22.1. Раскрывающийся список

Класс `QComboBox` реализует раскрывающийся список с возможностью выбора одного пункта. При щелчке мышью на поле появляется список возможных вариантов, а при выборе пункта список сворачивается. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QComboBox
```

Формат конструктора класса `QComboBox`:

```
<Объект> = QComboBox([parent=<Родитель>])
```

22.1.1. Добавление, изменение и удаление элементов

Для добавления, изменения, удаления и получения значения элементов предназначены следующие методы класса `QComboBox`:

- `addItem()` — добавляет один элемент в конец списка. Форматы метода:

  ```
  addItem(<Строка>[, <Данные>])
  addItem(<QIcon>, <Строка>[, <Данные>])
  ```

  В параметре `<Строка>` задается текст элемента списка, а в параметре `<QIcon>` — значок, который будет отображен перед текстом. Необязательный параметр `<Данные>` позволяет сохранить пользовательские данные — например, индекс в таблице базы данных;
- `addItems(<Список строк>)` — добавляет несколько элементов в конец списка;
- `insertItem()` — вставляет один элемент в указанную позицию списка. Все остальные элементы сдвигаются в конец списка. Форматы метода:

  ```
  insertItem(<Индекс>, <Строка>[, <Данные>])
  insertItem(<Индекс>, <QIcon>, <Строка>[, <Данные>])
  ```

- `insertItems(<Индекс>, <Список строк>)` — вставляет несколько элементов в указанную позицию списка. Все остальные элементы сдвигаются в конец списка;
- `insertSeparator(<Индекс>)` — вставляет разделительную линию в указанную позицию;
- `setItemText(<Индекс>, <Строка>)` — изменяет текст элемента с указанным индексом;
- `setItemIcon(<Индекс>, <QIcon>)` — изменяет значок элемента с указанным индексом;
- `setItemData(<Индекс>, <Данные>[, role=UserRole])` — изменяет данные для элемента с указанным индексом. Необязательный параметр `role` позволяет указать роль, для которой задаются данные. Например, если указать атрибут `ToolTipRole` класса `QtCore.Qt`, данные зададут текст всплывающей подсказки, которая будет отображена при наведении указателя мыши на элемент. По умолчанию изменяются пользовательские данные;
- `removeItem(<Индекс>)` — удаляет элемент с указанным индексом;
- `setCurrentIndex(<Индекс>)` — делает элемент с указанным индексом текущим. Метод является слотом;
- `currentIndex()` — возвращает индекс текущего элемента;
- `setCurrentText(<Строка>)` — делает элемент с текстом, совпадающим с указанной строкой, текущим. Метод является слотом;
- `currentText()` — возвращает текст текущего элемента;
- `itemText(<Индекс>)` — возвращает текст элемента с указанным индексом;
- `itemData(<Индекс>[, role=UserRole])` — возвращает данные, сохраненные в роли `role` элемента с индексом `<Индекс>`;
- `count()` — возвращает общее количество элементов списка. Получить количество элементов можно также с помощью функции `len()`;
- `clear()` — удаляет все элементы списка.

22.1.2. Изменение параметров списка

Управлять параметрами раскрывающегося списка позволяют следующие методы класса `QComboBox`:

- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь сможет вводить текст в раскрывающийся список и, возможно, добавлять таким образом в него новые элементы;
- `setInsertPolicy(<Режим>)` — задает режим добавления в список элементов, введенных пользователем. В качестве параметра указываются следующие атрибуты класса `QComboBox`:
  - `NoInsert` — `0` — элемент не будет добавлен;
  - `InsertAtTop` — `1` — элемент вставляется в начало списка;
  - `InsertAtCurrent` — `2` — будет изменен текст текущего элемента;
  - `InsertAtBottom` — `3` — элемент добавляется в конец списка;
  - `InsertAfterCurrent` — `4` — элемент вставляется после текущего элемента;
  - `InsertBeforeCurrent` — `5` — элемент вставляется перед текущим элементом;

   - `InsertAlphabetically` — 6 — при вставке учитывается алфавитный порядок следования элементов;
- `setEditText(<Текст>)` — вставляет текст в поле редактирования. Метод является слотом;
- `clearEditText()` — удаляет текст из поля редактирования. Метод является слотом;
- `setCompleter(<QCompleter>)` — позволяет предлагать возможные варианты значений, начинающиеся с введенных пользователем символов. В качестве параметра указывается экземпляр класса `QCompleter`;
- `setValidator(<QValidator>)` — устанавливает контроль ввода. В качестве значения указывается экземпляр класса, наследующего класс `QValidator` (см. *разд. 21.5.3*);
- `setDuplicatesEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может добавить элемент с повторяющимся текстом. По умолчанию повторы запрещены;
- `setMaxCount(<Количество>)` — задает максимальное количество элементов в списке. Если до вызова метода количество элементов превышало это количество, лишние элементы будут удалены;
- `setMaxVisibleItems(<Количество>)` — задает максимальное количество видимых элементов в раскрывающемся списке;
- `setMinimumContentsLength(<Количество>)` — задает минимальное количество символов, которое должно помещаться в раскрывающемся списке;
- `setSizeAdjustPolicy(<Режим>)` — задает режим установки ширины списка при изменении содержимого. В качестве параметра указываются следующие атрибуты класса `QComboBox`:
   - `AdjustToContents` — 0 — ширина списка подстраивается под ширину текущего содержимого;
   - `AdjustToContentsOnFirstShow` — 1 — ширина списка подстраивается под ширину содержимого, имевшегося в списке при первом его отображении;
   - `AdjustToMinimumContentsLength` — 2 — использовать вместо него `AdjustToContents` или `AdjustToContentsOnFirstShow`;
   - `AdjustToMinimumContentsLengthWithIcon` — 3 — используется значение минимальной ширины, которое установлено с помощью метода `setMinimumContentsLength()`, плюс ширина значка;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, список будет отображаться без рамки;
- `setIconSize(<QSize>)` — задает максимальный размер значков;
- `showPopup()` — разворачивает список;
- `hidePopup()` — сворачивает список.

22.1.3. Поиск элементов

Произвести поиск элемента в списке позволяют методы `findText()` (по тексту элемента) и `findData()` (по данным с указанной ролью). Методы возвращают индекс найденного элемента или значение `-1`, если таковой не был найден. Форматы методов:

```
findText(<Текст>[,  flags=MatchExactly | MatchCaseSensitive])
findData(<Данные>[, role=UserRole][, flags=MatchExactly | MatchCaseSensitive])
```

Параметр flags задает режим поиска. В качестве значения через оператор | можно указать комбинацию следующих атрибутов класса QtCore.Qt:

- ♦ MatchExactly — 0 — поиск полного соответствия;
- ♦ MatchContains — 1 — поиск совпадения с любой частью;
- ♦ MatchStartsWith — 2 — совпадение с началом;
- ♦ MatchEndsWith — 3 — совпадение с концом;
- ♦ MatchRegExp — 4 — поиск с помощью регулярного выражения;
- ♦ MatchWildcard — 5 — используются подстановочные знаки;
- ♦ MatchFixedString — 8 — поиск полного соответствия внутри строки, выполняемый по умолчанию без учета регистра символов;
- ♦ MatchCaseSensitive — 16 — поиск с учетом регистра символов;
- ♦ MatchWrap — 32 — если просмотрены все элементы, и подходящий элемент не найден, поиск начнется с начала списка;
- ♦ MatchRecursive — 64 — просмотр всей иерархии.

22.1.4. Сигналы

Класс QComboBox поддерживает следующие сигналы:

- ♦ activated(<Индекс>) и activated(<Текст>) — генерируются при выборе пользователем пункта в списке (даже если индекс не изменился). Внутри обработчика доступен целочисленный индекс или текст элемента;
- ♦ currentIndexChanged(<Индекс>) и currentIndexChanged(<Текст>) — генерируются при изменении текущего индекса. Внутри обработчика доступен целочисленный индекс (значение -1, если список пуст) или текст элемента;
- ♦ editTextChanged(<Текст>) — генерируется при изменении текста в поле. Внутри обработчика через параметр доступен новый текст;
- ♦ highlighted(<Индекс>) и highlighted(<Текст>) — генерируются при наведении указателя мыши на пункт в списке. Внутри обработчика доступен целочисленный индекс или текст элемента.

22.2. Список для выбора шрифта

Класс QFontComboBox реализует раскрывающийся список с названиями шрифтов. Шрифт можно выбрать из списка или ввести его название в поле — при этом станут отображаться названия, начинающиеся с введенных букв. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QComboBox — QFontComboBox
```

Формат конструктора класса QFontComboBox:

```
<Объект> = QFontComboBox([parent=<Родитель>])
```

Класс QFontComboBox наследует все методы и сигналы класса QComboBox (см. *разд. 22.1*) и определяет несколько дополнительных методов:

- `setCurrentFont(<QFont>)` — делает текущим элемент, соответствующий указанному шрифту. В качестве параметра указывается экземпляр класса `QFont`:

  ```
  comboBox.setCurrentFont(QtGui.QFont("Verdana"))
  ```

  Метод является слотом;
- `currentFont()` — возвращает экземпляр класса `QFont` с выбранным шрифтом. Вот пример вывода названия шрифта:

  ```
  print(comboBox.currentFont().family())
  ```

- `setFontFilters(<Фильтр>)` — ограничивает список указанными типами шрифтов. В качестве параметра указывается комбинация следующих атрибутов класса `QFontComboBox`:
  - `AllFonts` — `0` — все типы шрифтов;
  - `ScalableFonts` — `1` — масштабируемые шрифты;
  - `NonScalableFonts` — `2` — немасштабируемые шрифты;
  - `MonospacedFonts` — `4` — моноширинные шрифты;
  - `ProportionalFonts` — `8` — пропорциональные шрифты.

Класс `QFontComboBox` поддерживает сигнал `currentFontChanged(<QFont>)`, который генерируется при изменении текущего шрифта. Внутри обработчика доступен экземпляр класса `QFont` с текущим шрифтом.

22.3. Роли элементов

Каждый элемент списка хранит набор величин, каждая из которых относится к определенной роли: текст элемента, шрифт и цвет, которым отображается элемент, текст всплывающей подсказки и многое другое. Приведем роли элементов (атрибуты класса `QtCore.Qt`):

- `DisplayRole` — `0` — отображаемые данные (обычно текст);
- `DecorationRole` — `1` — изображение (обычно значок);
- `EditRole` — `2` — данные в виде, удобном для редактирования;
- `ToolTipRole` — `3` — текст всплывающей подсказки;
- `StatusTipRole` — `4` — текст для строки состояния;
- `WhatsThisRole` — `5` — текст для справки;
- `FontRole` — `6` — шрифт элемента. Указывается экземпляр класса `QFont`;
- `TextAlignmentRole` — `7` — выравнивание текста внутри элемента;
- `BackgroundRole` — `8` — фон элемента. Указывается экземпляр класса `QBrush`;
- `ForegroundRole` — `9` — цвет текста. Указывается экземпляр класса `QBrush`;
- `CheckStateRole` — `10` — статус флажка. Могут быть указаны следующие атрибуты класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок частично установлен;
  - `Checked` — `2` — флажок установлен;
- `AccessibleTextRole` — `11` — текст, выводящийся специализированными устройствами вывода — например, системами чтения с экрана;

- `AccessibleDescriptionRole` — 12 — описание элемента, выводящееся специализированными устройствами вывода — например, системами чтения с экрана;
- `SizeHintRole` — 13 — рекомендуемый размер элемента. Указывается экземпляр класса `QSize`;
- `UserRole` — 32 — любые пользовательские данные (например, индекс элемента в базе данных). Можно сохранить несколько данных, указав их в роли с индексом более `32`:

```
comboBox.setItemData(0, 50, role=QtCore.Qt.UserRole)
comboBox.setItemData(0, "Другие данные",
                     role=QtCore.Qt.UserRole + 1)
```

22.4. Модели

Для отображения данных в виде списков и таблиц применяется концепция «модель-представление», позволяющая отделить данные от их внешнего вида и избежать дублирования данных. В основе концепции лежат следующие составляющие:

- *модель* — является «оберткой» над данными. Позволяет считывать, добавлять, изменять, удалять данные и управлять ими;
- *представление* — предназначено для отображения элементов модели на экране. Сразу несколько представлений могут выводить одну и ту же модель;
- *модель выделения* — позволяет управлять выделением. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом;
- *промежуточная модель* — является прослойкой между моделью и представлением. Позволяет производить сортировку и фильтрацию данных без изменения порядка следования элементов в базовой модели;
- *делегат* — представляет компонент для вывода и редактирования данных. Существуют стандартные классы делегатов, кроме того, разработчик может создать свои классы.

22.4.1. Доступ к данным внутри модели

Доступ к данным внутри модели реализуется с помощью класса `QModelIndex` из модуля `QtCore`.

Чаще всего экземпляр класса `QModelIndex` создается с помощью метода `index()` какого-либо класса модели или метода `currentIndex()`, унаследованного моделями из класса `QAbstractItemView`. Такой экземпляр указывает на конкретные данные.

Если запрошенных данных в модели нет, возвращается пустой, невалидный экземпляр класса `QModelIndex`. Его также можно создать обычным вызовом конструктора:

```
<Объект> = QModelIndex()
```

Класс `QModelIndex` поддерживает следующие методы:

- `isValid()` — возвращает значение `True`, если объект является валидным, и `False` — в противном случае;
- `data([role=DisplayRole])` — возвращает данные, относящиеся к указанной в параметре `role` роли (по умолчанию — выводимый на экран текст элемента списка);

- `flags()` — возвращает свойства элемента в виде комбинации следующих атрибутов класса `QtCore.Qt`:
  - `NoItemFlags` — `0` — элемент не имеет свойств;
  - `ItemIsSelectable` — `1` — элемент можно выделить;
  - `ItemIsEditable` — `2` — элемент можно редактировать;
  - `ItemIsDragEnabled` — `4` — элемент можно перетаскивать;
  - `ItemIsDropEnabled` — `8` — в элемент можно сбрасывать перетаскиваемые данные;
  - `ItemIsUserCheckable` — `16` — элемент может быть установлен и сброшен;
  - `ItemIsEnabled` — `32` — пользователь может взаимодействовать с элементом;
  - `ItemIsTristate` — `64` — элемент имеет три состояния;
  - `ItemNeverHasChildren` — `128` — элемент не может иметь вложенные элементы;
- `row()` — возвращает индекс строки;
- `column()` — возвращает индекс столбца;
- `parent()` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на один уровень выше по иерархии. Если такого элемента нет, возвращается невалидный экземпляр класса `QModelIndex`;
- `child(<Строка>, <Столбец>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на один уровень ниже указанной позиции по иерархии. Если такого элемента нет, возвращается невалидный экземпляр класса `QModelIndex`;
- `sibling(<Строка>, <Столбец>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на том же уровне вложенности на указанной позиции. Если такого элемента нет, возвращается невалидный экземпляр класса `QModelIndex`;
- `model()` — возвращает ссылку на модель.

Также класс `QModelIndex` поддерживает операторы сравнения `==`, `<` и `!=`, позволяющие сравнивать экземпляры по их индексам.

Надо учитывать, что модель может измениться — тогда экземпляр класса `QModelIndex` будет ссылаться на несуществующий уже элемент. Если необходимо сохранить ссылку на элемент, следует воспользоваться классом `QPersistentModelIndex`, который содержит те же самые методы, но обеспечивает валидность ссылки.

22.4.2. Класс *QStringListModel*

Класс `QStringListModel` из модуля `QtCore` реализует одномерную модель, содержащую список строк. Ее содержимое можно отобразить с помощью классов `QListView`, `QComboBox` и др., передав в метод `setModel()` представления. Иерархия наследования:

```
QObject — QAbstractItemModel — QAbstractListModel — QStringListModel
```

Форматы конструктора класса `QStringListModel`:

```
<Объект> = QStringListModel([parent=None])
<Объект> = QStringListModel(<Список строк>[, parent=None])
```

Пример:

```
lst = ['Perl', 'PHP', 'Python', 'Ruby']
slm = QtCore.QStringListModel(lst, parent = window)
```

```
cbo = QtWidgets.QComboBox()
cbo.setModel(slm)
```

Класс `QStringListModel` наследует метод `index()` из класса `QAbstractListModel`, который возвращает индекс (экземпляр класса `QModelIndex`) элемента модели. Формат метода:

```
index(<Строка>[, column=0][, parent=QModelIndex()])
```

Первый параметр задает номер строки в модели, в которой хранится нужный элемент. Необязательный параметр `column` указывает номер столбца модели — для класса `QStringListModel`, позволяющего хранить простые списки строк, его следует задать равным `0`. Необязательный параметр `parent` позволяет задать элемент верхнего уровня для искомого элемента — если таковой не задан, будет выполнен поиск элемента на самом верхнем уровне иерархии.

Класс `QStringListModel` поддерживает также следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qstringlistmodel.html**):

- `setStringList(<Список строк>)` — устанавливает список строк в качестве содержимого модели;
- `stringList()` — возвращает список строк, хранящихся в модели;
- `insertRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — вставляет указанное количество пустых элементов в позицию, заданную первым параметром, остальные элементы сдвигаются в конец списка. Необязательный параметр `parent` позволяет указать элемент верхнего уровня, в который будут вложены добавляемые элементы, — если таковой не задан, элементы будут добавлены на самый верхний уровень иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество элементов, начиная с позиции, заданной первым параметром. Необязательный параметр `parent` позволяет указать элемент верхнего уровня, в который вложены удаляемые элементы, — если таковой не задан, элементы будут удалены из самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `setData(<QModelIndex>, <Значение>[, role=EditRole])` — задает значение для роли `role` элемента, на который указывает индекс `<QModelIndex>`. Метод возвращает значение `True`, если операция выполнена успешно:

  ```
  lst = QtWidgets.QComboBox()
  slm = QtCore.QStringListModel(parent = window)
  slm.insertRows(0, 4)
  slm.setData(slm.index(0), 'Perl')
  slm.setData(slm.index(1), 'PHP')
  slm.setData(slm.index(2), 'Python')
  slm.setData(slm.index(3), 'Ruby')
  lst.setModel(slm)
  ```

- `data(<QModelIndex>, <Роль>)` — возвращает данные, хранимые в указанной роли элемента, на который ссылается индекс `<QModelIndex>`;
- `rowCount([parent=QModelIndex()])` — возвращает количество элементов в модели. Необязательный параметр `parent` указывает элемент верхнего уровня, при этом будет возвращено количество вложенных в него элементов. Если параметр не задан, будет возвращено количество элементов верхнего уровня иерархии;

- `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` класса `QtCore.Qt`, сортировка производится в прямом порядке, а если `DescendingOrder` — в обратном.

22.4.3. Класс *QStandardItemModel*

Класс `QStandardItemModel` из модуля `QtGui` реализует двумерную (таблица) и иерархическую модели. Каждый элемент такой модели представлен классом `QStandardItem` из того же модуля. Вывести на экран ее содержимое можно с помощью классов `QTableView`, `QTreeView` и др., передав модель в метод `setModel()` представления. Иерархия наследования:

```
QObject — QAbstractItemModel — QStandardItemModel
```

Форматы конструктора класса `QStandardItemModel`:

```
<Объект> = QStandardItemModel([parent=None])
<Объект> = QStandardItemModel(<Количество строк>, <Количество столбцов>
                              [, parent=None])
```

Пример создания и вывода на экран таблицы из трех столбцов: значка, названия языка программирования и адреса веб-сайта приведен в листинге 22.1.

Листинг 22.1. Использование класса `QStandardItemModel`

```
from PyQt5 import QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QStandardItemModel")
tv = QtWidgets.QTableView(parent=window)
sti = QtGui.QStandardItemModel(parent=window)
lst1 = ['Perl', 'PHP', 'Python', 'Ruby']
lst2 = ['http://www.perl.org/', 'http://php.net/', 'https://www.python.org/',
'https://www.ruby-lang.org/']
for row in range(0, 4):
    if row == 2:
        iconfile = 'python.png'
    else:
        iconfile = 'icon.png'
    item1 = QtGui.QStandardItem(QtGui.QIcon(iconfile), '')
    item2 = QtGui.QStandardItem(lst1[row])
    item3 = QtGui.QStandardItem(lst2[row])
    sti.appendRow([item1, item2, item3])
sti.setHorizontalHeaderLabels(['Значок', 'Название', 'Сайт'])
tv.setModel(sti)
tv.setColumnWidth(0, 50)
tv.setColumnWidth(2, 180)
tv.resize(350, 150)
window.show()
sys.exit(app.exec_())
```

Класс `QStandardItemModel` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qstandarditemmodel.html**):

- `setRowCount(<Количество строк>)` — задает количество строк;
- `setColumnCount(<Количество столбцов>)` — задает количество столбцов;
- `rowCount([parent=QModelIndex()])` — возвращает количество строк. Необязательный параметр `parent` указывает элемент верхнего уровня, при этом будет возвращено количество вложенных в этот элемент строк, — если параметр не задан, будет возвращено количество строк верхнего уровня иерархии;
- `columnCount([parent=QModelIndex()])` — возвращает количество столбцов. Необязательный параметр `parent` в этом случае не используется;
- `setItem(<Строка>, <Столбец>, <QStandardItem>)` — устанавливает элемент в указанную ячейку;
- `appendRow(<Список>)` — добавляет одну строку в конец модели. В качестве параметра указывается список экземпляров класса `QStandardItem`, представляющих отдельные столбцы добавляемой строки;
- `appendRow(<QStandardItem>)` — добавляет строку из одной колонки в конец модели;
- `appendColumn(<Список>)` — добавляет один столбец в конец модели. В качестве параметра указывается список экземпляров класса `QStandardItem`, представляющих отдельные строки добавляемого столбца;
- `insertRow(<Индекс строки>, <Список>)` — добавляет одну строку в указанную позицию модели. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`, представляющих отдельные столбцы добавляемой строки;
- `insertRow(<Индекс>[, parent=QModelIndex()])` — добавляет одну пустую строку в указанную позицию модели. Необязательный параметр `parent` указывает элемент верхнего уровня, в который будет вложена добавляемая строка, — если параметр не задан, строка будет добавлена на самый верхний уровень иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertRow(<Индекс строки>, <QStandardItem>)` — добавляет строку из одного столбца в указанную позицию модели;
- `insertRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько пустых строк в указанную позицию модели. Необязательный параметр `parent` указывает элемент верхнего уровня, в который будут вложены добавляемые строки, — если параметр не задан, строки будут добавлены на самый верхний уровень иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertColumn(<Индекс столбца>, <Список>)` — добавляет один столбец в указанную позицию модели. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`, представляющих отдельные строки добавляемого столбца;
- `insertColumn(<Индекс>[, parent=QModelIndex()])` — добавляет один пустой столбец в указанную позицию. Необязательный параметр `parent` указывает элемент верхнего уровня — владелец элементов, в который будет добавлен столбец. Если этот параметр не задан, столбец будет добавлен в элементы самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция выполнена успешно;
- `insertColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько пустых столбцов в указанную позицию. Необязательный параметр `parent` ука-

зывает элемент верхнего уровня — владелец элементов, в который будут добавлены столбцы. Если этот параметр не задан, столбцы будут добавлены в элементы самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;

- `removeRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество строк, начиная со строки, имеющей индекс `<Индекс>`. Необязательный параметр `parent` указывает элемент верхнего уровня — владелец удаляемых строк. Если этот параметр не задан, будут удалены строки из самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс `<Индекс>`. Необязательный параметр `parent` указывает элемент верхнего уровня — владелец элементов, из которых будут удалены столбцы. Если этот параметр не задан, будут удалены столбцы из элементов самого верхнего уровня иерархии. Метод возвращает значение `True`, если операция успешно выполнена;
- `takeItem(<Строка>[, <Столбец>=0])` — удаляет указанный элемент из модели и возвращает его в виде экземпляра класса `QStandardItem`;
- `takeRow(<Индекс>)` — удаляет указанную строку из модели и возвращает ее в виде списка экземпляров класса `QStandardItem`;
- `takeColumn(<Индекс>)` — удаляет указанный столбец из модели и возвращает его в виде списка экземпляров класса `QStandardItem`;
- `clear()` — удаляет все элементы из модели;
- `item(<Строка>[, <Столбец>=0])` — возвращает ссылку на элемент (экземпляр класса `QStandardItem`), расположенный в указанной ячейке;
- `invisibleRootItem()` — возвращает ссылку на невидимый корневой элемент модели в виде экземпляра класса `QStandardItem`;
- `itemFromIndex(<QModelIndex>)` — возвращает ссылку на элемент (экземпляр класса `QStandardItem`), на который ссылается заданный индекс;
- `index(<Строка>, <Столбец>[, parent=QModelIndex()])` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного в указанной ячейке. Необязательный параметр `parent` задает элемент верхнего уровня для искомого элемента. Если таковой не задан, будет выполнен поиск элемента на самом верхнем уровне иерархии;
- `indexFromItem(<QStandardItem>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), ссылка на который передана в качестве параметра;
- `setData(<QModelIndex>, <Значение>[, role=EditRole])` — задает значение для роли `role` элемента, на который указывает индекс `<QModelIndex>`. Метод возвращает значение `True`, если операция успешно выполнена;
- `data(<QModelIndex>[, role=DisplayRole])` — возвращает данные, относящиеся к указанной роли элемента, на который ссылается индекс `<QModelIndex>`;
- `setHorizontalHeaderLabels(<Список строк>)` — задает заголовки столбцов. В качестве параметра указывается список строк;
- `setVerticalHeaderLabels(<Список строк>)` — задает заголовки строк. В качестве параметра указывается список строк;

- `setHorizontalHeaderItem(<Индекс>, <QStandardItem>)` — задает заголовок столбца. В первом параметре указывается индекс столбца, а во втором — экземпляр класса `QStandardItem`;
- `setVerticalHeaderItem(<Индекс>, <QStandardItem>)` — задает заголовок строки. В первом параметре указывается индекс строки, а во втором — экземпляр класса `QStandardItem`;
- `horizontalHeaderItem(<Индекс>)` — возвращает ссылку на указанный заголовок столбца (экземпляр класса `QStandardItem`);
- `verticalHeaderItem(<Индекс>)` — возвращает ссылку на указанный заголовок строки (экземпляр класса `QStandardItem`);
- `setHeaderData(<Индекс>, <Ориентация>, <Значение>[, role=EditRole])` — задает значение для указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором — ориентация (атрибут `Horizontal` или `Vertical` класса `QtCore.Qt`). Метод возвращает значение `True`, если операция успешно выполнена;
- `headerData(<Индекс>, <Ориентация>[, role=DisplayRole])` — возвращает значение, соответствующее указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором — ориентация;
- `findItems(<Текст>[, flags=MatchExactly][, column=0])` — производит поиск элемента внутри модели в указанном в параметре `column` столбце по заданному тексту. Допустимые значения параметра `flags` мы рассматривали в *разд. 22.1.3*. В качестве значения метод возвращает список экземпляров класса `QStandardItem` или пустой список;
- `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` класса `QtCore.Qt`, сортировка производится в прямом порядке, а если `DescendingOrder` — в обратном;
- `setSortRole(<Роль>)` — задает роль (см. *разд. 22.3*), по которой производится сортировка;
- `parent(<QModelIndex>)` — возвращает индекс (экземпляр класса `QModelIndex`) родительского элемента. В качестве параметра указывается индекс (экземпляр класса `QModelIndex`) элемента-потомка;
- `hasChildren([parent=QModelIndex()])` — возвращает `True`, если заданный элемент имеет хотя бы одного потомка, и `False` — в противном случае.

При изменении значения элемента генерируется сигнал `itemChanged(<QStandardItem>)`. Внутри обработчика через параметр доступна ссылка на элемент, представленный экземпляром класса `QStandardItem`.

22.4.4. Класс *QStandardItem*

Каждый элемент модели `QStandardItemModel` представлен классом `QStandardItem` из модуля `QtGui`. Этот класс не только описывает элемент, но и позволяет создавать вложенные структуры, в которых любой элемент может иметь произвольное количество вложенных в него дочерних элементов или элементов-потомков (что пригодится при выводе иерархического списка). Форматы конструктора класса:

```
<Объект> = QStandardItem()
<Объект> = QStandardItem(<Текст>)
```

```
<Объект> = QStandardItem(<QIcon>, <Текст>)
<Объект> = QStandardItem(<Количество строк>[, <Количество столбцов>=1])
```

Последний формат задает количество дочерних элементов и столбцов в них.

Наиболее часто используемые методы класса `QStandardItem` приведены далее (полный их список можно найти на странице **https://doc.qt.io/qt-5/qstandarditem.html**):

- `setRowCount(<Количество строк>)` — задает количество дочерних строк;
- `setColumnCount(<Количество столбцов>)` — задает количество столбцов в дочерних строках;
- `rowCount()` — возвращает количество дочерних строк;
- `columnCount()` — возвращает количество столбцов в дочерних строках;
- `row()` — возвращает индекс строки в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя (находится на самом верхнем уровне иерархии);
- `column()` — возвращает индекс столбца в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя;
- `setChild(<Строка>, <Столбец>, <QStandardItem>)` — устанавливает заданный третьим параметром элемент в указанную ячейку дочерней таблицы текущего элемента.

  Пример создания иерархии и вывода ее на экран с применением иерархического списка показан в листинге 22.2.

Листинг 22.2. Вывод иерархического списка

```
# -*- coding: utf-8 -*-
from PyQt5 import QtGui, QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QStandardItemModel")
tv = QtWidgets.QTreeView(parent=window)
sti = QtGui.QStandardItemModel(parent=window)
rootitem1 = QtGui.QStandardItem('QAbstractItemView')
rootitem2 = QtGui.QStandardItem('Базовый класс')
item1 = QtGui.QStandardItem('QListView')
item2 = QtGui.QStandardItem('Список')
rootitem1.appendRow([item1, item2])
item1 = QtGui.QStandardItem('QTableView')
item2 = QtGui.QStandardItem('Таблица')
rootitem1.appendRow([item1, item2])
item1 = QtGui.QStandardItem('QTreeView')
item2 = QtGui.QStandardItem('Иерархический список')
rootitem1.appendRow([item1, item2])
sti.appendRow([rootitem1, rootitem2])
sti.setHorizontalHeaderLabels(['Класс', 'Описание'])
tv.setModel(sti)
tv.setColumnWidth(0, 170)
```

```
tv.resize(400, 100)
window.show()
sys.exit(app.exec_())
```

- appendRow(<Список>) — добавляет одну строку в конец дочерней таблицы текущего элемента. В качестве параметра указывается список экземпляров класса QStandardItem, формирующих отдельные столбцы;
- appendRow(<QStandardItem>) — добавляет заданный элемент в конец дочерней таблицы текущего элемента, формируя строку с одним столбцом;
- appendRows(<Список>) — добавляет несколько строк, содержащих по одному столбцу, в конец дочерней таблицы текущего элемента. В качестве параметра указывается список экземпляров класса QStandardItem;
- appendColumn(<Список>) — добавляет один столбец в конец дочерней таблицы текущего элемента. В качестве параметра указывается список экземпляров класса QStandardItem, формирующих отдельные строки;
- insertRow(<Индекс строки>, <Список>) — вставляет одну строку в указанную позицию дочерней таблицы у текущего элемента. В качестве параметра <Список> указывается список экземпляров класса QStandardItem, формирующих отдельные столбцы;
- insertRow(<Индекс строки>, <QStandardItem>) — вставляет заданный элемент в указанную позицию дочерней таблицы у текущего элемента, формируя строку с одним столбцом;
- insertRows(<Индекс строки>, <Список>) — вставляет несколько строк, содержащих по одному столбцу, в указанную позицию дочерней таблицы у текущего элемента. В качестве параметра <Список> указывается список экземпляров класса QStandardItem;
- insertRows(<Индекс строки>, <Количество>) — вставляет несколько пустых строк в указанную позицию дочерней таблицы для текущего элемента;
- insertColumn(<Индекс столбца>, <Список>) — вставляет один столбец в указанную позицию дочерней таблицы у текущего элемента. В качестве параметра <Список> указывается список экземпляров класса QStandardItem, формирующих отдельные строки;
- insertColumns(<Индекс>, <Количество>) — вставляет несколько пустых столбцов в указанную позицию дочерней таблицы у текущего элемента;
- removeRow(<Индекс>) — удаляет строку с указанным индексом;
- removeRows(<Индекс>, <Количество>) — удаляет указанное количество строк, начиная со строки, имеющей индекс <Индекс>;
- removeColumn(<Индекс>) — удаляет столбец с указанным индексом;
- removeColumns(<Индекс>, <Количество>) — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс <Индекс>;
- takeChild(<Строка>[, <Столбец>=0]) — удаляет указанный дочерний элемент и возвращает его в виде экземпляра класса QStandardItem;
- takeRow(<Индекс>) — удаляет указанную строку из дочерней таблицы и возвращает ее в виде списка экземпляров класса QStandardItem;
- takeColumn(<Индекс>) — удаляет указанный столбец из дочерней таблицы и возвращает его в виде списка экземпляров класса QStandardItem;

- `parent()` — возвращает ссылку на родительский элемент (экземпляр класса `QStandardItem`) или значение `None`, если текущий элемент не имеет родителя;
- `child(<Строка>[, <Столбец>=0])` — возвращает ссылку на дочерний элемент (экземпляр класса `QStandardItem`) или значение `None`, если такового нет;
- `hasChildren()` — возвращает значение `True`, если существует хотя бы один дочерний элемент, и `False` — в противном случае;
- `setData(<Значение>[, role=UserRole+1])` — устанавливает значение для указанной роли;
- `data([UserRole+1])` — возвращает значение, которое соответствует роли, указанной в параметре;
- `setText(<Текст>)` — задает текст элемента;
- `text()` — возвращает текст элемента;
- `setTextAlignment(<Выравнивание>)` — задает выравнивание текста внутри элемента;
- `setIcon(<QIcon>)` — задает значок, который будет отображен перед текстом;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст для справки;
- `setFont(<QFont>)` — задает шрифт элемента;
- `setBackground(<QBrush>)` — задает цвет фона;
- `setForeground(<QBrush>)` — задает цвет текста;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, после текста элемента будет выведен флажок, который можно устанавливать и сбрасывать;
- `isCheckable()` — возвращает значение `True`, если после текста элемента выводится флажок, и `False` — в противном случае;
- `setCheckState(<Статус>)` — задает состояние флажка. Могут быть указаны следующие атрибуты класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок находится в неопределенном состоянии;
  - `Checked` — `2` — флажок установлен;
- `checkState()` — возвращает текущее состояние флажка;
- `setTristate(<Флаг>)` — если в качестве параметра указано значение `True`, флажок может иметь три состояния: установленное, сброшенное и неопределенное (промежуточное);
- `isTristate()` — возвращает значение `True`, если флажок может иметь три состояния, и `False` — в противном случае;
- `setFlags(<Флаги>)` — задает свойства элемента (см. *разд. 22.4.1*);
- `flags()` — возвращает значение установленных свойств элемента;
- `setSelectable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может выделить элемент;
- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может редактировать текст элемента;
- `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, перетаскивание элемента разрешено;

- setDropEnabled(<Флаг>) — если в качестве параметра указано значение True, сброс данных в элемент разрешен;
- setEnabled(<Флаг>) — если в качестве параметра указано значение True, пользователь может взаимодействовать с элементом. Значение False делает элемент недоступным;
- clone() — возвращает копию элемента в виде экземпляра класса QStandardItem;
- index() — возвращает индекс элемента (экземпляр класса QModelIndex);
- model() — возвращает ссылку на модель (экземпляр класса QStandardItemModel);
- sortChildren(<Индекс столбца>[, order=AscendingOrder]) — производит сортировку дочерней таблицы. Если во втором параметре указан атрибут AscendingOrder класса QtCore.Qt, сортировка производится в прямом порядке, а если DescendingOrder — в обратном.

22.5. Представления

Для отображения элементов модели предназначены следующие классы представлений:

- QListView — реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Пункты списка, помимо текстовой надписи, могут содержать значки;
- QTableView — реализует таблицу;
- QTreeView — реализует иерархический список.

Также можно воспользоваться классами QComboBox (раскрывающийся список — см. *разд. 22.1*), QListWidget (простой список), QTableWidget (таблица) и QTreeWidget (иерархический список). Последние три класса нарушают концепцию «модель-представление», хотя и отчасти базируются на ней. За подробной информацией по этим классам обращайтесь к документации.

22.5.1. Класс *QAbstractItemView*

Абстрактный класс QAbstractItemView является базовым классом для всех рассмотренных ранее представлений. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea — QAbstractItemView
```

Класс QAbstractItemView поддерживает такой набор полезных для нас методов (полный их список можно найти на странице **https://doc.qt.io/qt-5/qabstractitemview.html**):

- setModel(<QAbstractItemModel>) — задает для представления модель. В качестве параметра передается экземпляр одного из классов, порожденных от класса QAbstractItemModel;
- model() — возвращает заданную для представления модель;
- selectedIndexes() — возвращает выделенные в списке элементы, представленные списком экземпляров класса QModelIndex;
- setCurrentIndex(<QModelIndex>) — делает элемент с указанным индексом (экземпляр класса QModelIndex) текущим. Метод является слотом;
- currentIndex() — возвращает индекс (экземпляр класса QModelIndex) текущего элемента;
- setRootIndex(<QModelIndex>) — задает корневой элемент. В качестве параметра указывается экземпляр класса QModelIndex. Метод является слотом;

- `rootIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) корневого элемента;
- `setAlternatingRowColors(<Флаг>)` — если в качестве параметра указано значение `True`, то четные и нечетные строки будут иметь разный цвет фона;
- `setIndexWidget(<QModelIndex>, <QWidget>)` — устанавливает компонент в позицию, указанную индексом (экземпляр класса `QModelIndex`), и делает его потомком представления. Если в той позиции уже находится какой-либо компонент, он удаляется;
- `indexWidget(<QModelIndex>)` — возвращает ссылку на компонент, который был ранее установлен в позицию, указанную индексом (экземпляр класса `QModelIndex`);
- `setSelectionModel(<QItemSelectionModel>)` — устанавливает модель выделения;
- `selectionModel()` — возвращает модель выделения;
- `setSelectionMode(<Режим>)` — задает режим выделения элементов. В качестве параметра указываются следующие атрибуты класса `QAbstractItemView`:
  - `NoSelection` — `0` — элементы не могут быть выделены;
  - `SingleSelection` — `1` — можно выделить только один элемент;
  - `MultiSelection` — `2` — можно выделить несколько элементов. Повторный щелчок на элементе снимает выделение;
  - `ExtendedSelection` — `3` — можно выделить несколько элементов, щелкая на них мышью и удерживая при этом нажатой клавишу <Ctrl>. Можно также нажать на элементе левую кнопку мыши и перемещать мышь, не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;
  - `ContiguousSelection` — `4` — можно выделить несколько элементов, нажав на элементе левую кнопку мыши и перемещая мышь, не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;
- `setSelectionBehavior(<Режим>)` — задает режим представления выделенных элементов. В качестве параметра указываются следующие атрибуты класса `QAbstractItemView`:
  - `SelectItems` — `0` — выделяется отдельный элемент;
  - `SelectRows` — `1` — выделяется строка целиком;
  - `SelectColumns` — `2` — выделяется столбец целиком;
- `selectAll()` — выделяет все элементы. Метод является слотом;
- `clearSelection()` — снимает выделение. Метод является слотом;
- `edit(<QModelIndex>)` — переключает элемент с заданным индексом (экземпляр класса `QModelIndex`) в режим редактирования, не делая его выделенным. Метод является слотом;
- `setEditTriggers(<Режим>)` — задает действие, при котором элемент переключается в режим редактирования. В качестве параметра указывается комбинация следующих атрибутов класса `QAbstractItemView`:
  - `NoEditTriggers` — `0` — элемент не поддерживает редактирование;
  - `CurrentChanged` — `1` — при выделении элемента;
  - `DoubleClicked` — `2` — при двойном щелчке мышью;

  - SelectedClicked — 4 — при щелчке мышью на уже выделенном элементе;
  - EditKeyPressed — 8 — при нажатии клавиши <F2>;
  - AnyKeyPressed — 16 — при нажатии любой символьной клавиши;
  - AllEditTriggers — 31 — при любом упомянутом здесь действии;
- setIconSize(<QSize>) — задает размер значков;
- setTextElideMode(<Режим>) — задает режим обрезки текста, если он не помещается в отведенную область (в месте пропуска выводится троеточие). Могут быть указаны следующие атрибуты класса QtCore.Qt:
  - ElideLeft — 0 — текст обрезается слева;
  - ElideRight — 1 — текст обрезается справа;
  - ElideMiddle — 2 — текст вырезается посередине;
  - ElideNone — 3 — текст не обрезается;
- setTabKeyNavigation(<Флаг>) — если в качестве параметра указано значение True, между элементами можно перемещаться с помощью клавиш <Tab> и <Shift>+<Tab>;
- scrollTo(<QModelIndex>[, hint=EnsureVisible]) — прокручивает представление таким образом, чтобы элемент, на который ссылается индекс (экземпляр класса QModelIndex), был видим. В параметре hint указываются следующие атрибуты класса QAbstractItemView:
  - EnsureVisible — 0 — элемент должен находиться в области видимости;
  - PositionAtTop — 1 — элемент должен находиться в верхней части;
  - PositionAtBottom — 2 — элемент должен находиться в нижней части;
  - PositionAtCenter — 3 — элемент должен находиться в центре;
- scrollToTop() — прокручивает представление в самое начало. Метод является слотом;
- scrollToBottom() — прокручивает представление в самый конец. Метод является слотом;
- setDragEnabled(<Флаг>) — если в качестве параметра указано значение True, перетаскивание элементов разрешено;
- setDragDropMode(<Режим>) — задает режим работы drag & drop. В качестве параметра указываются следующие атрибуты класса QAbstractItemView:
  - NoDragDrop — 0 — drag & drop не поддерживается;
  - DragOnly — 1 — поддерживается только перетаскивание;
  - DropOnly — 2 — поддерживается только сбрасывание;
  - DragDrop — 3 — поддерживается перетаскивание и сбрасывание;
  - InternalMove — 4 — допускается лишь перетаскивание внутри компонента;
- setDropIndicatorShown(<Флаг>) — если в качестве параметра указано значение True, позиция возможного сброса элемента будет подсвечена;
- setAutoScroll(<Флаг>) — если в качестве параметра указано значение True, при перетаскивании пункта будет производиться автоматическая прокрутка;
- setAutoScrollMargin(<Отступ>) — задает расстояние от края области, при достижении которого будет производиться автоматическая прокрутка области;

- update(<QModelIndex>) — обновляет элемент с заданным индексом. Метод является слотом.

Класс QAbstractItemView поддерживает следующие сигналы:

- activated(<QModelIndex>) — генерируется при активизации элемента путем одинарного или двойного щелчка мышью или нажатия клавиши <Enter>. В обработчике через параметр доступен индекс активного элемента;
- pressed(<QModelIndex>) — генерируется при нажатии кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента;
- clicked(<QModelIndex>) — генерируется при щелчке мышью над элементом. Параметр хранит индекс элемента;
- doubleClicked(<QModelIndex>) — генерируется при двойном щелчке мышью над элементом. Параметр хранит индекс элемента;
- entered(<QModelIndex>) — генерируется при вхождении указателя мыши в область элемента. Чтобы сигнал сработал, необходимо включить обработку перемещения указателя вызовом метода setMouseTracking(), унаследованного от класса QWidget. Внутри обработчика через параметр доступен индекс элемента;
- viewportEntered — генерируется при вхождении указателя мыши в область компонента. Чтобы сигнал сработал, необходимо включить обработку перемещения указателя с помощью метода setMouseTracking(), унаследованного от класса QWidget.

22.5.2. Простой список

Класс QListView реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме текста, в любом пункте такого списка может присутствовать значок (рис. 22.1). Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QListView
```

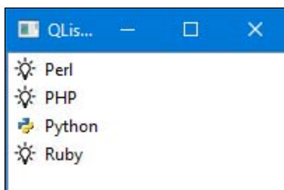

Рис. 22.1. Компонент QListView

Формат конструктора класса QListView:

```
<Объект> = QListView([parent=<Родитель>])
```

Типичный пример использования списка приведен в листинге 22.3.

Листинг 22.3. Простой список QListView

```
lv = QtWidgets.QListView()
sti = QtGui.QStandardItemModel(parent = window)
lst = ['Perl', 'PHP', 'Python', 'Ruby']
```

```
for row in range(0, 4):
    if row == 2:
        iconfile = 'python.png'
    else:
        iconfile = 'icon.png'
    item = QtGui.QStandardItem(QtGui.QIcon(iconfile), lst[row])
    sti.appendRow(item)
lv.setModel(sti)
```

Класс `QListView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 22.5.1*), включая методы `setModel()`, `model()` и `selectedIndexes()`. Помимо этого, он дополнительно определяет следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qlistview.html**):

- `setModelColumn(<Индекс>)` — задает индекс отображаемого столбца в табличной модели (по умолчанию отображается первый столбец с индексом `0`);
- `setViewMode(<Режим>)` — задает режим отображения элементов. В качестве параметра указываются следующие атрибуты класса `QListView`:
  - `ListMode` — `0` — элементы размещаются сверху вниз, а значки имеют маленькие размеры;
  - `IconMode` — `1` — элементы размещаются слева направо, а значки имеют большие размеры. Элементы можно свободно перемещать мышью;
- `setMovement(<Режим>)` — задает режим перемещения элементов. В качестве параметра указываются следующие атрибуты класса `QListView`:
  - `Static` — `0` — пользователь не может перемещать элементы;
  - `Free` — `1` — свободное перемещение;
  - `Snap` — `2` — перемещаемые элементы автоматически выравниваются по сетке, размеры которой задаются методом `setGridSize()`;
- `setGridSize(<QSize>)` — задает размеры сетки, по которой выравниваются перемещаемые элементы;
- `setResizeMode(<Режим>)` — задает режим расположения элементов при изменении размера списка. В качестве параметра указываются следующие атрибуты класса `QListView`:
  - `Fixed` — `0` — элементы остаются в том же положении;
  - `Adjust` — `1` — положение элементов изменяется при изменении размеров;
- `setFlow(<Режим>)` — задает порядок вывода элементов. В качестве параметра указываются следующие атрибуты класса `QListView`:
  - `LeftToRight` — `0` — слева направо;
  - `TopToBottom` — `1` — сверху вниз;
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `False`, перенос элементов на новую строку (если они не помещаются в ширину области) запрещен;
- `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, текст элементов при необходимости будет переноситься по строкам;
- `setLayoutMode(<Режим>)` — задает режим размещения элементов. В качестве параметра указываются следующие атрибуты класса `QListView`:

- `SinglePass` — `0` — элементы размещаются все сразу. Если список слишком большой, то окно останется заблокированным, пока все элементы не будут отображены;
- `Batched` — `1` — элементы размещаются блоками. Размер такого блока задается методом `setBatchSize(<Количество>)`;

♦ `setUniformItemSizes(<Флаг>)` — если в качестве параметра указано значение `True`, все элементы будут иметь одинаковый размер (по умолчанию они имеют разные размеры, зависящие от содержимого);

♦ `setSpacing(<Отступ>)` — задает отступ вокруг элемента;

♦ `setSelectionRectVisible(<Флаг>)` — если в качестве параметра указано значение `True`, будет отображаться вспомогательная рамка, показывающая область выделения. Метод доступен только при использовании режима множественного выделения;

♦ `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, строка с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает строку;

♦ `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае.

Класс `QListView` также поддерживает сигнал `indexesMoved(<Элементы>)`, генерируемый при перемещении элементов. Внутри обработчика через параметр доступен список перемещаемых элементов в виде экземпляров класса `QModelIndex`.

22.5.3. Таблица

Класс `QTableView` реализует таблицу (рис. 22.2). Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QTableView
```

QTableView

	Иконка	Название	Сайт
1		Perl	http://www.perl.org/
2		PHP	http://php.net/
3		Python	https://www.python.org/
4		Ruby	https://www.ruby-lang.org/

Рис. 22.2. Компонент `QTableView`

Формат конструктора класса `QTableView`:

```
<Объект> = QTableView([parent=<Родитель>])
```

Класс `QTableView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 22.5.1*) и дополнительно поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qtableview.html**):

♦ `selectRow(<Индекс>)` — выделяет строку с указанным индексом. Метод является слотом;

♦ `selectColumn(<Индекс>)` — выделяет столбец с указанным индексом. Метод является слотом;

- `horizontalHeader()` — возвращает ссылку на горизонтальный заголовок, представленный экземпляром класса `QHeaderView`;
- `verticalHeader()` — возвращает ссылку на вертикальный заголовок, представленный экземпляром класса `QHeaderView`. Например, вывести таблицу без заголовков можно следующим образом:

```
view.horizontalHeader().hide()
view.verticalHeader().hide()
```

- `setRowHeight(<Индекс>, <Высота>)` — задает высоту строки с указанным в первом параметре индексом;
- `rowHeight(<Индекс>)` — возвращает высоту строки;
- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `columnWidth(<Индекс>)` — возвращает ширину столбца;
- `resizeRowToContents(<Индекс строки>)` — изменяет размер указанной строки таким образом, чтобы в нее поместилось все содержимое. Метод является слотом;
- `resizeRowsToContents()` — изменяет размер всех строк таким образом, чтобы в них поместилось все содержимое. Метод является слотом;
- `resizeColumnToContents(<Индекс столбца>)` — изменяет размер указанного столбца таким образом, чтобы в него поместилось все содержимое. Метод является слотом;
- `resizeColumnsToContents()` — изменяет размер всех столбцов таким образом, чтобы в них поместилось содержимое. Метод является слотом;
- `setSpan(<Индекс строки>, <Индекс столбца>, <Количество строк>, <Количество столбцов>)` — растягивает элемент с указанными в первых двух параметрах индексами на заданное количество строк и столбцов, производя как бы объединение ячеек таблицы;
- `rowSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в строке, которое занимает элемент с указанными индексами;
- `columnSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в столбце, которое занимает элемент с указанными индексами;
- `clearSpans()` — отменяет все объединения ячеек;
- `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то строка с индексом, указанным в первом параметре, будет скрыта. Значение `False` отображает строку;
- `hideRow(<Индекс>)` — скрывает строку с указанным индексом. Метод является слотом;
- `showRow(<Индекс>)` — отображает строку с указанным индексом. Метод является слотом;
- `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `False` отображает столбец;
- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом;
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом;

- isColumnHidden(<Индекс>) — возвращает значение True, если столбец с указанным индексом скрыт, и False — в противном случае;
- isIndexHidden(<QModelIndex>) — возвращает значение True, если элемент с указанным индексом (экземпляр класса QModelIndex) скрыт, и False — в противном случае;
- setGridStyle(<Стиль>) — задает стиль линий сетки. В качестве параметра указываются следующие атрибуты класса QtCore.Qt:
  - NoPen — 0 — линии не выводятся;
  - SolidLine — 1 — сплошная линия;
  - DashLine — 2 — штриховая линия;
  - DotLine — 3 — точечная линия;
  - DashDotLine — 4 — штрих и точка, штрих и точка и т. д.;
  - DashDotDotLine — 5 — штрих и две точки, штрих и две точки и т. д.;
- setShowGrid(<Флаг>) — если в качестве параметра указано значение True, то сетка будет отображена, а если False — то скрыта. Метод является слотом;
- setSortingEnabled(<Флаг>) — если в качестве параметра указано значение True, столбцы можно сортировать с помощью щелчков мышью на их заголовках. При этом в заголовке показывается текущее направление сортировки;
- setCornerButtonEnabled(<Флаг>) — если в качестве параметра указано значение True, с помощью кнопки в левом верхнем углу заголовка можно выделить всю таблицу. Значение False отключает кнопку;
- setWordWrap(<Флаг>) — если в качестве параметра указано значение True, текст элементов при необходимости будет переноситься по строкам;
- sortByColumn(<Индекс столбца>, <Направление>) — производит сортировку. Если во втором параметре указан атрибут AscendingOrder класса QtCore.Qt, сортировка производится в прямом порядке, а если DescendingOrder — в обратном.

22.5.4. Иерархический список

Класс QTreeView реализует иерархический список (рис. 22.3). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QTreeView
```

Формат конструктора класса QTreeView:

```
<Объект> = QTreeView([parent=<Родитель>])
```

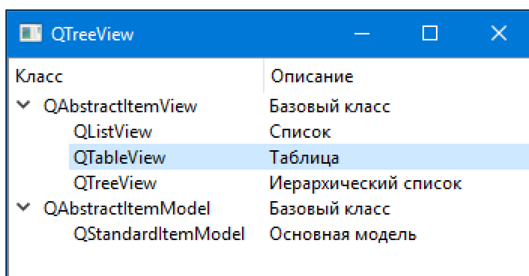

Рис. 22.3. Компонент QTreeView

Класс `QTreeView` наследует все методы и сигналы класса `QAbstractItemView` (см. *разд. 22.5.1*) и дополнительно поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qtreeview.html**):

- `header()` — возвращает ссылку на горизонтальный заголовок (экземпляр класса `QHeaderView`);
- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `columnWidth(<Индекс>)` — возвращает ширину столбца;
- `rowHeight(<QModelIndex>)` — возвращает высоту строки, в которой находится элемент с указанным индексом (экземпляр класса `QModelIndex`);
- `resizeColumnToContents(<Индекс столбца>)` — изменяет ширину указанного столбца таким образом, чтобы в нем поместилось все содержимое. Метод является слотом;
- `setUniformRowHeights(<Флаг>)` — если в качестве параметра указано значение `True`, все элементы будут иметь одинаковую высоту;
- `setHeaderHidden(<Флаг>)` — если в качестве параметра указано значение `True`, заголовок будет скрыт. Значение `False` отображает заголовок;
- `isHeaderHidden()` — возвращает значение `True`, если заголовок скрыт, и `False` — в противном случае;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `False` отображает столбец;
- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом;
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом;
- `isColumnHidden(<Индекс>)` — возвращает значение `True`, если столбец с указанным индексом скрыт, и `False` — в противном случае;
- `setRowHidden(<Индекс>, <QModelIndex>, <Флаг>)` — если в третьем параметре указано значение `True`, то строка с индексом `<Индекс>` и родителем `<QModelIndex>` будет скрыта. Значение `False` отображает строку;
- `isRowHidden(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с указанным индексом `<Индекс>` и родителем `<QModelIndex>` скрыта, и `False` — в противном случае;
- `isIndexHidden(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) скрыт, и `False` — в противном случае;
- `setExpanded(<QModelIndex>, <Флаг>)` — если во втором параметре указано значение `True`, то элементы, которые являются дочерними для элемента с указанным в первом параметре индексом, будут отображены, а если `False` — то скрыты. В первом параметре указывается экземпляр класса `QModelIndex`;
- `expand(<QModelIndex>)` — отображает элементы, которые являются дочерними для элемента с указанным индексом. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом;

- expandToDepth(<Уровень>) — отображает все дочерние элементы до указанного уровня. Метод является слотом;
- expandAll() — отображает все дочерние элементы. Метод является слотом;
- collapse(<QModelIndex>) — скрывает элементы, которые являются дочерними для элемента с указанным индексом. В качестве параметра указывается экземпляр класса QModelIndex. Метод является слотом;
- collapseAll() — скрывает все дочерние элементы. Метод является слотом;
- isExpanded(<QModelIndex>) — возвращает значение True, если элементы, которые являются дочерними для элемента с указанным индексом, отображены, и False — в противном случае. В качестве параметра указывается экземпляр класса QModelIndex;
- setItemsExpandable(<Флаг>) — если в качестве параметра указано значение False, пользователь не сможет отображать или скрывать дочерние элементы;
- setAnimated(<Флаг>) — если в качестве параметра указано значение True, отображение и сокрытие дочерних элементов будет производиться с анимацией;
- setIndentation(<Отступ>) — задает отступ для дочерних элементов;
- setRootIsDecorated(<Флаг>) — если в качестве параметра указано значение False, для элементов верхнего уровня не будут показываться элементы управления, с помощью которых производится отображение и сокрытие дочерних элементов;
- setFirstColumnSpanned(<Индекс строки>, <QModelIndex>, <Флаг>) — если третьим параметром передано значение True, содержимое первого столбца строки с указанным в первом параметре индексом и родителем, заданным во втором параметре (как экземпляр класса QModelIndex), займет всю ширину списка;
- setExpandsOnDoubleClick(<Флаг>) — если передать в параметре значение False, сворачивание и разворачивание пунктов списка будет выполняться по двойному щелчку мыши;
- setSortingEnabled(<Флаг>) — если в качестве параметра указано значение True, столбцы можно сортировать с помощью щелчка мышью на заголовке столбца. При этом в заголовке показывается текущее направление сортировки;
- sortByColumn(<Индекс столбца>[, AscendingOrder]) — производит сортировку. Если во втором параметре указан атрибут AscendingOrder класса QtCore.Qt, сортировка производится в прямом порядке, а если DescendingOrder — в обратном;
- setWordWrap(<Флаг>) — если в качестве параметра указано значение True, текст элементов при необходимости будет переноситься по строкам.

Класс QTreeView поддерживает сигналы:

- expanded(<QModelIndex>) — генерируется при отображении дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса QModelIndex) элемента;
- collapsed(<QModelIndex>) — генерируется при сокрытии дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса QModelIndex) элемента.

22.5.5. Управление заголовками строк и столбцов

Класс QHeaderView представляет заголовки строк и столбцов в компонентах QTableView и QTreeView. Получить ссылки на заголовки в классе QTableView позволяют методы horizontalHeader() и verticalHeader(), а для установки заголовков предназначены методы

setHorizontalHeader(<QHeaderView>) и setVerticalHeader(<QHeaderView>). Получить ссылку на заголовок в классе QTreeView позволяет метод header(), а для установки заголовка предназначен метод setHeader(<QHeaderView>). Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                          QAbstractItemView — QHeaderView
```

Формат конструктора класса QHeaderView:

```
<Объект> = QHeaderView(<Ориентация>[, parent=<Родитель>])
```

Параметр <Ориентация> позволяет задать ориентацию заголовка. В качестве значения указываются атрибуты Horizontal или Vertical класса QtCore.Qt.

Класс QHeaderView наследует все методы и сигналы класса QAbstractItemView (см. *разд. 22.5.1*) и дополнительно определяет следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qheaderview.html**):

- count() — возвращает количество секций в заголовке. Получить количество секций можно также с помощью функции len();
- setDefaultSectionSize(<Размер>) — задает размер секций по умолчанию;
- defaultSectionSize() — возвращает размер секций по умолчанию;
- setMinimumSectionSize(<Размер>) — задает минимальный размер секций;
- minimumSectionSize() — возвращает минимальный размер секций;
- setMaximumSectionSize(<Размер>) — задает максимальный размер секций;
- maximumSectionSize() — возвращает максимальный размер секций;
- resizeSection(<Индекс>, <Размер>) — изменяет размер секции с указанным индексом;
- sectionSize(<Индекс>) — возвращает размер секции с указанным индексом;
- setSectionResizeMode(<Режим>) — задает режим изменения размеров для всех секций. В качестве параметра могут быть указаны следующие атрибуты класса QHeaderView:
  - Interactive — 0 — размер может быть изменен пользователем или программно;
  - Stretch — 1 — секции автоматически равномерно распределяют свободное пространство между собой. Размер не может быть изменен ни пользователем, ни программно;
  - Fixed — 2 — размер может быть изменен только программно;
  - ResizeToContents — 3 — размер определяется автоматически по содержимому секции. Размер не может быть изменен ни пользователем, ни программно;
- setSectionResizeMode(<Индекс>, <Режим>) — задает режим изменения размеров для секции с указанным индексом;
- setStretchLastSection(<Флаг>) — если в качестве параметра указано значение True, последняя секция будет занимать все свободное пространство;
- setCascadingSectionResizes(<Флаг>) — если в качестве параметра указано значение True, изменение размеров одной секции может привести к изменению размеров других секций;
- setSectionHidden(<Индекс>, <Флаг>) — если во втором параметре указано значение True, секция с индексом, указанным в первом параметре, будет скрыта. Значение False отображает секцию;

- hideSection(<Индекс>) — скрывает секцию с указанным индексом;
- showSection(<Индекс>) — отображает секцию с указанным индексом;
- isSectionHidden(<Индекс>) — возвращает значение True, если секция с указанным индексом скрыта, и False — в противном случае;
- sectionsHidden() — возвращает значение True, если существует хотя бы одна скрытая секция, и False — в противном случае;
- hiddenSectionCount() — возвращает количество скрытых секций;
- setDefaultAlignment(<Выравнивание>) — задает выравнивание текста внутри заголовков;
- setHighlightSections(<Флаг>) — если в качестве параметра указано значение True, то текст заголовка текущей секции будет выделен;
- setSectionsClickable(<Флаг>) — если в качестве параметра указано значение True, заголовок будет реагировать на щелчок мышью, при этом выделяя все элементы секции;
- setSectionsMovable(<Флаг>) — если в качестве параметра указано значение True, пользователь может перемещать секции с помощью мыши;
- sectionsMovable() — возвращает значение True, если пользователь может перемещать секции с помощью мыши, и False — в противном случае;
- moveSection(<Откуда>, <Куда>) — позволяет переместить секцию. В параметрах указываются визуальные индексы;
- swapSections(<Секция1>, <Секция2>) — меняет две секции местами. В параметрах указываются визуальные индексы;
- visualIndex(<Логический индекс>) — преобразует логический индекс (первоначальный порядок следования) в визуальный (отображаемый в настоящее время порядок следования). Если преобразование прошло неудачно, возвращается значение -1;
- logicalIndex(<Визуальный индекс>) — преобразует визуальный индекс (отображаемый в настоящее время порядок следования) в логический (первоначальный порядок следования. Если преобразование прошло неудачно, возвращается значение -1;
- saveState() — возвращает экземпляр класса QByteArray с текущими размерами и положением секций;
- restoreState(<QByteArray>) — восстанавливает размеры и положение секций на основе экземпляра класса QByteArray, возвращаемого методом saveState().

Класс QHeaderView поддерживает следующие сигналы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qheaderview.html**):

- sectionPressed(<Логический индекс>) — генерируется при нажатии левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен целочисленный логический индекс секции;
- sectionClicked(<Логический индекс>) — генерируется при нажатии и отпускании левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен целочисленный логический индекс секции;
- sectionDoubleClicked(<Логический индекс>) — генерируется при двойном щелчке мышью на заголовке секции. Внутри обработчика через параметр доступен целочисленный логический индекс секции;

- `sectionMoved(<Логический индекс>, <Старый визуальный индекс>, <Новый визуальный индекс>)` — генерируется при изменении положения секции. Все параметры целочисленные;
- `sectionResized(<Логический индекс>, <Старый размер>, <Новый размер>)` — генерируется непрерывно при изменении размера секции. Все параметры целочисленные.

22.6. Управление выделением элементов

Класс `QItemSelectionModel`, объявленный в модуле `QtCore`, реализует модель, позволяющую централизованно управлять выделением сразу в нескольких представлениях. Установить модель выделения позволяет метод `setSelectionModel(<QItemSelectionModel>)` класса `QAbstractItemView`, а получить ссылку на модель можно с помощью метода `selectionModel()`. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении. Иерархия наследования выглядит так:

```
QObject — QItemSelectionModel
```

Форматы конструктора класса `QItemSelectionModel`:

```
<Объект> = QItemSelectionModel(<Модель>)
<Объект> = QItemSelectionModel(<Модель>, <Родитель>)
```

Класс `QItemSelectionModel` поддерживает следующие полезные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qitemselectionmodel.html**):

- `hasSelection()` — возвращает значение `True`, если существует выделенный элемент, и `False` — в противном случае;
- `isSelected(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) выделен, и `False` — в противном случае;
- `isRowSelected(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с индексом `<Индекс>` и родителем `<QModelIndex>` выделена, и `False` — в противном случае;
- `isColumnSelected(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если столбец с индексом `<Индекс>` и родителем `<QModelIndex>` выделен, и `False` — в противном случае;
- `rowIntersectsSelection(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с индексом `<Индекс>` и родителем `<QModelIndex>` содержит выделенный элемент, и `False` — в противном случае;
- `columnIntersectsSelection(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если столбец с индексом `<Индекс>` и родителем `<QModelIndex>` содержит выделенный элемент, и `False` — в противном случае;
- `selectedIndexes()` — возвращает список индексов (экземпляров класса `QModelIndex`) выделенных элементов или пустой список, если выделенных элементов нет;
- `selectedRows([<Индекс столбца>=0])` — возвращает список индексов (экземпляров класса `QModelIndex`) выделенных элементов из указанного столбца. Элемент попадет в список только в том случае, если строка выделена полностью;
- `selectedColumns([<Индекс строки>=0])` — возвращает список индексов (экземпляров класса `QModelIndex`) выделенных элементов из указанной строки. Элемент попадет в список только в том случае, если столбец выделен полностью;

- `selection()` — возвращает ссылку на экземпляр класса `QItemSelection`;
- `select(<QModelIndex>, <Режим>)` — изменяет выделение элемента с указанным индексом. Во втором параметре указываются следующие атрибуты (или их комбинация через оператор `|`) класса `QItemSelectionModel`:
  - `NoUpdate` — без изменений;
  - `Clear` — снимает выделение всех элементов;
  - `Select` — выделяет элемент;
  - `Deselect` — снимает выделение с элемента;
  - `Toggle` — выделяет элемент, если он не выделен, или снимает выделение, если элемент был выделен;
  - `Current` — обновляет выделение текущего элемента;
  - `Rows` — индекс будет расширен так, чтобы охватить всю строку;
  - `Columns` — индекс будет расширен так, чтобы охватить весь столбец;
  - `SelectCurrent` — комбинация `Select | Current`;
  - `ToggleCurrent` — комбинация `Toggle | Current`;
  - `ClearAndSelect` — комбинация `Clear | Select`.

  Метод является слотом;
- `select(<QItemSelection>, <Режим>)` — изменяет выделение элементов. Метод является слотом;
- `setCurrentIndex(<QModelIndex>, <Режим>)` — делает элемент текущим и изменяет режим выделения. Метод является слотом;
- `currentIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) текущего элемента;
- `clearSelection()` — снимает все выделения. Метод является слотом.

Класс `QItemSelectionModel` поддерживает следующие сигналы:

- `currentChanged(<QModelIndex>, <QModelIndex>)` — генерируется при изменении индекса текущего элемента. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй — индекс нового элемента;
- `currentRowChanged(<QModelIndex>, <QModelIndex>)` — генерируется при выделении элемента из другой строки. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй — индекс нового элемента;
- `currentColumnChanged(<QModelIndex>, <QModelIndex>)` — генерируется при выделении элемента из другого столбца. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй — индекс нового элемента;
- `selectionChanged(<QItemSelection>, <QItemSelection>)` — генерируется при изменении выделения. Внутри обработчика через первый параметр доступно предыдущее выделение, а через второй — новое выделение.

22.7. Промежуточные модели

Как вы уже знаете, одну модель можно установить в нескольких представлениях. При этом изменение порядка следования элементов в одном представлении повлечет за собой изменение порядка следования элементов в другом. Чтобы предотвратить изменение порядка

следования элементов в базовой модели, следует создать промежуточную модель с помощью класса `QSortFilterProxyModel`, объявленного в модуле `QtCore`, и установить ее в представлении. Иерархия наследования для класса `QSortFilterProxyModel` выглядит так:

```
QObject — QAbstractItemModel — QAbstractProxyModel —
                               QSortFilterProxyModel
```

Формат конструктора класса `QSortFilterProxyModel`:

```
<Объект> = QSortFilterProxyModel([parent=<Родитель>])
```

Класс `QSortFilterProxyModel` наследует следующие методы из класса `QAbstractProxyModel` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qabstractproxymodel.html**):

- `setSourceModel(<Модель>)` — устанавливает базовую модель;
- `sourceModel()` — возвращает ссылку на базовую модель.

Класс `QSortFilterProxyModel` поддерживает основные методы обычных моделей и дополнительно определяет следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qsortfilterproxymodel.html**):

- `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` класса `QtCore.Qt`, сортировка производится в прямом порядке, а если `DescendingOrder` — в обратном. Если в параметре `<Индекс столбца>` указать значение `-1`, будет использован порядок следования элементов из базовой модели.

  > **Примечание**
  >
  > Чтобы включить сортировку столбцов пользователем, следует передать значение `True` в метод `setSortingEnabled()` объекта представления.

- `setSortRole(<Роль>)` — задает роль (см. *разд. 22.3*), по которой производится сортировка. По умолчанию сортировка производится по роли `DisplayRole`;
- `setSortCaseSensitivity(<Режим>)` — если в качестве параметра указать атрибут `CaseInsensitive` класса `QtCore.Qt`, при сортировке регистр символов учитываться не будет, а если `CaseSensitive` — то будет;
- `setSortLocaleAware(<Флаг>)` — если в качестве параметра указать значение `True`, при сортировке будут учитываться настройки локали;
- `setFilterFixedString(<Фрагмент>)` — выбор из модели элементов, которые содержат заданный фрагмент. Если указать пустую строку, в результат попадут все строки из базовой модели. Метод является слотом;
- `setFilterRegExp()` — выбор из модели элементов, соответствующих указанному регулярному выражению. Если указать пустую строку, в результат попадут все строки из базовой модели. Форматы метода:

  ```
  setFilterRegExp(<QRegExp>)
  setFilterRegExp(<Строка с шаблоном>)
  ```

  В первом формате указывается экземпляр класса `QRegExp`, а во втором — строка с шаблоном регулярного выражения. Второй формат метода является слотом;
- `setFilterWildcard(<Шаблон>)` — выбор из модели элементов, соответствующих указанной строке, которая может содержать подстановочные знаки:

- `?` — один любой символ;
- `*` — нуль или более любых символов;
- `[...]` — диапазон значений.

Остальные символы трактуются как есть. Если в качестве параметра указать пустую строку, в результат попадут все элементы из базовой модели. Метод является слотом;

♦ `setFilterKeyColumn(<Индекс>)` — задает индекс столбца, по которому будет производиться фильтрация. Если в качестве параметра указать значение `-1`, будут просматриваться элементы во всех столбцах. По умолчанию фильтрация производится по первому столбцу;

♦ `setFilterRole(<Роль>)` — задает роль (см. *разд. 22.3*), по которой производится фильтрация. По умолчанию сортировка производится по роли `DisplayRole`;

♦ `setFilterCaseSensitivity(<Режим>)` — если в качестве параметра указать атрибут `CaseInsensitive` класса `QtCore.Qt`, при фильтрации регистр символов учитываться не будет, а если `CaseSensitive` — то будет;

♦ `setDynamicSortFilter(<Флаг>)` — если в качестве параметра указано значение `False`, при изменении базовой модели не будет производиться повторная сортировка или фильтрация.

22.8. Использование делегатов

Все три представления, рассмотренные в *разд. 22.5*, дают возможность редактирования текста их элементов. Например, в таблице (класс `QTableView`) мы можем дважды щелкнуть мышью на любом элементе, после чего в нем появится поле ввода. Введем в это поле новый текст и нажмем клавишу <Enter> для подтверждения ввода или <Esc> — для отмены.

За редактирование данных в представлении отвечает особый класс, называемый *делегатом*. Он создает компонент, в котором будет выполняться редактирование значения (*редактор*), задает его параметры, заносит в него само редактируемое значение, а по окончании редактирования переносит его назад, в модель.

По умолчанию в качестве делегата используется класс `QItemDelegate` из модуля `QtWidgets`. А в качестве компонента-редактора применяется однострочное поле ввода (класс `QLineEdit`, рассмотренный нами в *главе 21*).

Если мы хотим использовать для редактирования значения в каком-либо столбце или строке другой редактор — например, многострочное поле ввода, поле ввода даты или целого числа, мы создадим другой делегат и назначим его представлению. Класс, представляющий делегат, должен быть унаследован от класса `QStyledItemDelegate`.

Иерархия наследования классов `QItemDelegate` и `QStyledItemDelegate`:

```
QObject – QAbstractItemDelegate - QItemDelegate
QObject – QAbstractItemDelegate - QStyledItemDelegate
```

В новом классе-делегате нам следует переопределить следующие методы:

♦ `createEditor()` — создает компонент, который будет использоваться для редактирования данных, и задает его параметры. Формат метода:

```
createEditor(self, <Родитель>, <Настройки>, <Индекс>)
```

Вторым параметром передается ссылка на компонент-представление, который станет родителем создаваемого редактора (список, таблица или иерархический список). Третьим параметром передается экземпляр класса `QStyleOptionViewItem`, хранящий дополнительные настройки делегата. Четвертым параметром можно получить индекс текущего элемента модели, представленный экземпляром класса `QModelIndex`.

Метод `createEditor()` должен создать компонент-редактор, задать для него в качестве родителя компонент-представление (он передается вторым параметром) и вернуть созданный компонент в качестве результата.

Чтобы отказаться от использования собственного делегата и указать представлению использовать делегат по умолчанию, в методе `createEditor()` следует вернуть значение `None`;

- `setEditorData()` — заносит в компонент-редактор, созданный в методе `createEditor()`, данные из текущего элемента модели, тем самым подготавливая редактор для редактирования этих данных. Формат метода:

  ```
  setEditorData(self, <Редактор>, <Индекс>)
  ```

  Вторым параметром передается компонент-редактор, а третьим — индекс текущего элемента модели в виде экземпляра класса `QModelIndex`;

- `updateEditorGeometry()` — задает размеры редактора соответственно размерам области, отведенной под него в компоненте-представлении. Формат:

  ```
  updateEditorGeometry(self, <Редактор>, <Настройки>, <Индекс>)
  ```

  Вторым параметром передается ссылка на компонент-редактор, третьим — ссылка на экземпляр класса `QStyleOptionViewItem`, хранящий настройки делегата, четвертым — индекс текущего элемента модели, представленный экземпляром класса `QModelIndex`.

  Размеры отведенной под редактор области мы можем получить из атрибута `rect` экземпляра класса `QStyleOptionViewItem`, переданного третьим параметром (полное описание класса `QStyleOptionViewItem` приведено на странице **https://doc.qt.io/qt-5/qstyleoptionviewitem.html**, а описание класса `QStyleOption`, от которого он порожден, — на странице **https://doc.qt.io/qt-5/qstyleoption.html**);

- `setModelData()` — по окончании редактирования переносит значение из редактора в текущий элемент модели. Формат:

  ```
  setModelData(self, <Редактор>, <Модель>, <Индекс>)
  ```

  Вторым параметром передается ссылка на компонент-редактор, третьим — ссылка на модель, четвертым — индекс текущего элемента модели, представленный экземпляром класса `QModelIndex`.

Полное описание класса `QAbstractItemDelegate` можно найти на странице **https://doc.qt.io/qt-5/qabstractitemdelegate.html**, класса `QitemDelegate` — на странице **https://doc.qt.io/qt-5/qitemdelegate.html**, а класса `QStyledItemDelegate` — на странице **https://doc.qt.io/qt-5/qstyleditemdelegate.html**.

Для назначения делегатов представлению следует применять следующие методы, унаследованные от класса `QAbstractItemView`:

- `setItemDelegate(<QAbstractItemDelegate>)` — назначает делегат для всего представления. В параметре передается класс делегата, унаследованный от класса `QAbstractItemDelegate`;

- `setItemDelegateForColumn(<Индекс столбца>, <QAbstractItemDelegate>)` — назначает делегат для столбца представления с указанным индексом. В параметре передается класс делегата, унаследованный от класса `QAbstractItemDelegate`;
- `setItemDelegateForRow(<Индекс строки>, <QAbstractItemDelegate>)` — назначает делегат для строки представления с указанным индексом. В параметре передается класс делегата, унаследованный от класса `QAbstractItemDelegate`.

  Если в какой-либо ячейке представления действуют одновременно два делегата, заданные для столбца и для строки, будет использоваться делегат, заданный для строки.

В качестве примера рассмотрим небольшое складское приложение (листинг 22.4), позволяющее править количество каких-либо имеющихся на складе позиций с применением поля ввода целочисленных значений (класс `QSpinBox`).

Листинг 22.4. Использование делегата

```
from PyQt5 import QtCore, QtWidgets, QtGui
import sys

# Создаем класс делегата
class SpinBoxDelegate(QtWidgets.QStyledItemDelegate):
    def createEditor(self, parent, options, index):
        # Создаем компонент-редактор, используемый для правки значений
        # количества позиций
        editor = QtWidgets.QSpinBox(parent)
        editor.setFrame(False)
        editor.setMinimum(0)
        editor.setSingleStep(1)
        return editor
    def setEditorData(self, editor, index):
        # Заносим в компонент-редактор значение количества
        value = int(index.model().data(index, QtCore.Qt.EditRole))
        editor.setValue(value)
    def updateEditorGeometry(self, editor, options, index):
        # Указываем размеры компонента-редактора
        editor.setGeometry(options.rect)
    def setModelData(self, editor, model, index):
        # Заносим исправленное значение количества в модель
        value = str(editor.value())
        model.setData(index, value, QtCore.Qt.EditRole);

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QTableView()
window.setWindowTitle("Использование делегата")
sti = QtGui.QStandardItemModel(parent = window)
lst1 = ['Дискета', 'Бумага для принтера', 'Барабан для принтера']
lst2 = ["10", "3", "8"]
for row in range(0, 3):
    item1 = QtGui.QStandardItem(lst1[row])
    item2 = QtGui.QStandardItem(lst2[row])
    sti.appendRow([item1, item2])
```

```
sti.setHorizontalHeaderLabels(['Товар', 'Кол-во'])
window.setModel(sti)
# Назначаем делегат второму столбцу таблицы
window.setItemDelegateForColumn(1, SpinBoxDelegate())
window.setColumnWidth(0, 150)
window.resize(300, 150)
window.show()
sys.exit(app.exec_())
```

Результат выполнения кода из листинга 22.4 показан на рис. 22.4.

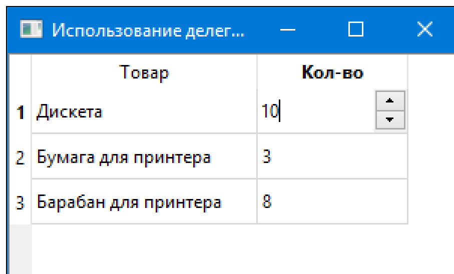

Рис. 22.4. Использование делегата

ГЛАВА 23

Работа с базами данных

PyQt 5 включает в свой состав средства для работы с базами данных формата SQLite, MySQL, Oracle, PostgreSQL и др., не требующие установки никаких дополнительных Python-библиотек. С помощью этих средств мы можем выполнять любые SQL-запросы и обрабатывать их результаты, получать доступ к отдельным таблицам базы, работать с транзакциями, а также использовать особые модели для вывода содержимого таблиц или запросов в любом из компонентов-представлений, рассмотренных в *главе 22*.

> ***Внимание!***
> Для успешного доступа к базам данных всех форматов, кроме SQLite и ODBC, требуется установить соответствующий клиент, поскольку в комплект поставки PyQt он не входит.

Все классы, обеспечивающие работу с базами данных и рассмотренные в этой главе, определены в модуле `QtSql`.

23.1. Соединение с базой данных

За соединение с базой данных и обработку транзакций отвечает класс `QSqlDatabase`.

Чтобы установить соединение с базой, следует вызвать статический метод `addDatabase()` этого класса. Формат вызова:

```
addDatabase(<Формат базы данных>[, connectionName=""])
```

Первым параметром указывается строка, обозначающая формат открываемой базы данных. Поддерживаются следующие форматы: `QMYSQL` (MySQL), `QODBC` (ODBC), `QPSQL` (PostgreSQL), `QSQLITE2` (SQLite версии 2) и `QSQLITE` (SQLite версии 3).

Вторым параметром можно задать имя соединения, что может оказаться полезным, если приложение работает сразу с несколькими базами. Если имя соединения не указано, устанавливаемое соединение будет помечено как используемое по умолчанию.

Метод `addDatabase()` возвращает экземпляр класса `QSqlDatabase`, представляющий базу данных, с которой установлено соединение. Теперь мы можем задать параметры базы, воспользовавшись одним из приведенных далее методов `QSqlDatabase`:

- `setHostName(<Хост>)` — задает хост, на котором расположена база данных. Используется только для серверов данных наподобие MySQL;
- `setPort(<Номер порта>)` — задает номер порта, через который будет выполнено подключение к хосту. Используется только для серверов данных и лишь в том случае, если сервер настроен на использование порта, отличного от порта по умолчанию;

- setDatabaseName(<Имя или путь к базе данных>) — задает имя базы данных (для серверов данных), путь к ней (для «настольных» баз данных, таких как SQLite) или полный набор параметров подключения (если используется ODBC);
- setUserName(<Имя>) — задает имя для подключения к базе. Используется только для серверов данных;
- setPassword(<Пароль>) — задает пароль для подключения к базе. Используется только для серверов данных;
- setConnectOptions(<Параметры>) — задает набор дополнительных параметров для подключения к базе в виде строки. Набор поддерживаемых дополнительных параметров различен в зависимости от выбранного формата и приведен в документации по классу QSqlDatabase.

Для работы с базой предназначены следующие методы класса QSqlDatabase:

- open() — открывает базу данных. Возвращает True, если база была успешно открыта, и False — в противном случае;

> ***Внимание!***
>
> Перед созданием соединения с базой данных обязательно следует создать объект приложения (экземпляр класса QApplication). Если этого не сделать, PyQt не сможет загрузить драйвер указанного формата баз данных, и соединение не будет создано.
>
> Открываемая база данных уже должна существовать на диске или сервере. Единственное исключение — база формата SQLite, которая, в случае ее отсутствия, будет создана автоматически.

- open(<Имя>, <Пароль>) — открывает базу данных с указанными именем и паролем. Возвращает True, если база была успешно открыта, и False — в противном случае;
- isOpen() — возвращает True, если база данных в настоящее время открыта, и False — в противном случае;
- isOpenError() — возвращает True, если при попытке открытия базы данных возникли ошибки, и False — в противном случае;
- transaction() — запускает транзакцию, если формат базы поддерживает таковые. Если же формат базы не поддерживает транзакции, то не делает ничего. Возвращает True, если транзакция была успешно запущена, и False — в противном случае;
- commit() — завершает транзакцию, если формат базы поддерживает таковые. Если же формат базы не поддерживает транзакции, то не делает ничего. Возвращает True, если транзакция была успешно завершена, и False — в противном случае;
- rollback() — отменяет транзакцию, если формат базы поддерживает таковые. Если же формат базы не поддерживает транзакции, то не делает ничего. Возвращает True, если транзакция была успешно отменена, и False — в противном случае;
- lastError() — возвращает сведения о последней возникшей при работе с базой ошибке в виде экземпляра класса QSqlError;
- connectionName() — возвращает строку с именем соединения с базой или пустую строку для соединения по умолчанию;
- tables([type=Tables]) — возвращает список таблиц, хранящихся в базе. В параметре type можно указать тип таблиц в виде одного из атрибутов класса QSql или их комбинации через оператор |:

- `Tables` — `1` — обычные таблицы;
- `SystemTables` — `2` — служебные таблицы;
- `Views` — `4` — представления;
- `AllTables` — `255` — все здесь указанное;

- `record(<Имя таблицы>)` — возвращает сведения о структуре таблицы с переданным именем, представленные экземпляром класса `QSqlRecord`, или пустой экземпляр этого класса, если таблицы с таким именем нет;
- `primaryIndex(<Имя таблицы>)` — возвращает сведения о ключевом индексе таблицы с переданным именем, представленные экземпляром класса `QSqlIndex`, или пустой экземпляр этого класса, если таблицы с таким именем нет;
- `close()` — закрывает базу данных.

Также нам могут пригодиться следующие статические методы класса `QSqlDatabase`:

- `contains([connectionName=""])` — возвращает `True`, если имеется соединение с базой данных с указанным именем, и `False` — в противном случае;
- `connectionNames()` — возвращает список имен всех созданных соединений с базами данных. Соединение по умолчанию обозначается пустой строкой;
- `database([connectionName=""][, ][open=True])` — возвращает сведения о соединении с базой данных, имеющем указанное имя, в виде экземпляра класса `QSqlDatabase`. Если в параметре `open` указано значение `True`, база данных будет открыта. Если такового соединения нет, возвращается некорректно сформированный экземпляр класса `QSqlDatabase`;
- `cloneDatabase(<QSqlDatabase>, <Имя соединения>)` — создает копию указанного в первом параметре соединения с базой и дает ему имя, заданное во втором параметре. Возвращаемый результат — экземпляр класса `QSqlDatabase`, представляющий созданную копию соединения;
- `removeDatabase(<Имя соединения>)` — удаляет соединение с указанным именем. Соединение по умолчанию обозначается пустой строкой;
- `isDriverAvailable(<Формат>)` — возвращает `True`, если указанный в виде строки формат баз данных поддерживается PyQt, и `False` — в противном случае;
- `drivers()` — возвращает список всех поддерживаемых PyQt форматов баз данных.

В листинге 23.1 показан код, выполняющий соединение с базами данных различных форматов и их открытие.

Листинг 23.1. Соединение с базами данных различных форматов

```
from PyQt5 import QtWidgets, QtSql
import sys
# Создаем объект приложения, иначе поддержка баз данных не будет работать
app = QtWidgets.QApplication(sys.argv)

# Открываем базу данных SQLite, находящуюся в той же папке, что и файл
# с этой программой
con1 = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con1.setDatabaseName('data.sqlite')
```

```
con1.open()
con1.close()

# Открываем базу данных MySQL
con2 = QtSql.QSqlDatabase.addDatabase('QMYSQL')
con2.setHostName("somehost");
con2.setDatabaseName("somedb");
con2.setUserName("someuser");
con2.setPassword("password");
con2.open();
con2.close()

# Открываем базу данных Microsoft Access через ODBC
con3 = QtSql.QSqlDatabase.addDatabase("QODBC");
con3.setDatabaseName("DRIVER={Microsoft Access Driver (*.mdb)};↳
FIL={MS Access};DBQ=c:/work/data.mdb");
con3.open()
con3.close()
```

Полное описание класса `QSqlDatabase` приведено на странице **https://doc.qt.io/qt-5/qsqldatabase.html**.

23.2. Получение сведений о структуре таблицы

PyQt позволяет получить некоторые сведения о структуре таблиц, хранящихся в базе: списки полей таблицы, параметры отдельного поля, индекса и ошибки, возникшей при работе с базой.

23.2.1. Получение сведений о таблице

Сведения о структуре таблицы можно получить вызовом метода `record()` класса `QSqlDatabase`. Эти сведения представляются экземпляром класса `QSqlRecord`.

Для получения сведений о полях таблицы используются следующие методы этого класса:

- `count()` — возвращает количество полей в таблице;
- `fieldName(<Индекс поля>)` — возвращает имя поля, имеющее заданный индекс, или пустую строку, если индекс некорректен;
- `field(<Индекс поля>)` — возвращает сведения о поле (экземпляр класса `QSqlField`), чей индекс задан в качестве параметра;
- `field(<Имя поля>)` — возвращает сведения о поле (экземпляр класса `QSqlField`), чье имя задано в качестве параметра;
- `indexOf(<Имя поля>)` — возвращает индекс поля с указанным именем или `-1`, если такого поля нет. При поиске поля не учитывается регистр символов;
- `contains(<Имя поля>)` — возвращает `True`, если поле с указанным именем существует, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если в таблице нет полей, и `False` — в противном случае.

Полное описание класса QSqlRecord приведено на странице **https://doc.qt.io/qt-5/qsqlrecord.html**.

23.2.2. Получение сведений об отдельном поле

Сведения об отдельном поле таблицы возвращаются методом field() класса QSqlRecord. Их представляет экземпляр класса QSqlField, поддерживающий следующие методы:

- name() — возвращает имя поля;
- type() — возвращает тип поля в виде одного из следующих атрибутов класса QVariant, объявленного в модуле QtCore (здесь приведен список лишь наиболее часто употребляемых типов — полный их список можно найти по адресу **https://doc.qt.io/qt-5/qvariant-obsolete.html#Type-enum**):
  - Invalid — неизвестный тип;
  - Bool — логический (bool);
  - ByteArray — массив байтов (QByteArray, bytes);
  - Char — строка из одного символа (str);
  - Date — значение даты (QDate или datetime.date);
  - DateTime — значение даты и времени (QDateTime или datetime.datetime);
  - Double — вещественное число (float);
  - Int и LongLong — целое число (int);
  - String — строка (str);
  - Time — значение времени (QTime или datetime.time);
  - UInt и ULongLong — положительное целое число (int);
- length() — возвращает длину поля;
- precision() — возвращает количество знаков после запятой для полей, хранящих вещественные числа;
- defaultValue() — возвращает значение поля по умолчанию;
- requiredStatus() — возвращает признак, является ли поле обязательным к заполнению, в виде одного из атрибутов класса QSqlField:
  - Required — 1 — поле является обязательным к заполнению;
  - Optional — 0 — поле не является обязательным к заполнению;
  - Unknown — -1 — определить признак обязательности заполнения поля не представляется возможным;
- isAutoValue() — возвращает True, если значение в поле заносится автоматически (что может быть, например, у поля автоинкремента), и False — в противном случае;
- isReadOnly() — возвращает True, если поле доступно только для чтения, и False — в противном случае.

Полное описание класса QSqlField приведено на странице **https://doc.qt.io/qt-5/qsqlfield.html**.

23.2.3. Получение сведений об индексе

Сведения о ключевом индексе, возвращаемые методом `primaryIndex()` класса `QSqlDatabase`, представлены экземпляром класса `QSqlIndex`. Он наследует все методы класса `QSqlRecord`, тем самым позволяя узнать, в частности, список полей, на основе которых создан индекс.

Также он определяет следующие методы:

- ♦ `name()` — возвращает имя индекса или пустую строку для ключевого индекса;
- ♦ `isDescending(<Номер поля>)` — возвращает `True`, если поле с указанным номером в индексе отсортировано по убыванию, и `False` — в противном случае.

Полное описание класса `QSqlIndex` приведено на странице **https://doc.qt.io/qt-5/qsqlindex.html**.

23.2.4. Получение сведений об ошибке

Сведения об ошибке, возникшей при работе с базой данных, представляются экземпляром класса `QSqlError`. Выяснить, что за ошибка произошла и каковы ее причины, позволят следующие методы вышеупомянутого класса:

- ♦ `type()` — возвращает код ошибки в виде одного из следующих атрибутов класса `QSqlError`:
  - `NoError` — `0` — никакой ошибки не возникло;
  - `ConnectionError` — `1` — ошибка соединения с базой данных;
  - `StatementError` — `2` — ошибка в коде SQL-запроса;
  - `TransactionError` — `3` — ошибка в обработке транзакции;
  - `UnknownError` — `4` — ошибка неустановленной природы.

  Если код ошибки не удается определить, возвращается `-1`;
- ♦ `text()` — возвращает полное текстовое описание ошибки (фактически — значения, возвращаемые методами `databaseText()` и `driverText()`, объединенные в одну строку);
- ♦ `databaseText()` — возвращает текстовое описание ошибки, сгенерированное базой данных;
- ♦ `driverText()` — возвращает текстовое описание ошибки, сгенерированное драйвером базы данных, который входит в состав PyQt;
- ♦ `nativeErrorCode()` — возвращает строковый код ошибки, специфический для выбранного формата баз данных.

Пример:

```
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
if con.open():
  # Работаем с базой данных
else:
  # Выводим текст описания ошибки
  print(con.lastError().text())
```

Полное описание класса `QSqlError` можно найти на странице **https://doc.qt.io/qt-5/qsqlerror.html**.

23.3. Выполнение SQL-запросов и получение их результатов

Класс `QSqlQuery` позволяет выполнять SQL-запросы любого назначения: создания необходимых таблиц и индексов, добавления, изменения, удаления и, разумеется, выборки записей. Это один из наиболее развитых механизмов работы с данными, предоставляемых PyQt.

> ***Примечание***
>
> Далее будут рассмотрены лишь наиболее часто используемые возможности класса `QSqlQuery`. Полное его описание приведено на странице **https://doc.qt.io/qt-5/qsqlquery.html**.

23.3.1. Выполнение запросов

Чтобы выполнить запрос к базе, сначала следует создать экземпляр класса `QSqlQuery`. Для этого используется один из следующих форматов вызова его конструктора:

```
QSqlQuery([<SQL-код>][, db=QSqlDatabase()])
QSqlQuery(<QSqlDatabase>)
QSqlQuery(<QSqlQuery>)
```

Первый формат позволяет сразу задать SQL-код, который следует выполнить, и немедленно запустить его на исполнение. Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию.

Второй формат создает пустой запрос, не содержащий ни SQL-кода, ни каких-либо прочих параметров, но позволяющий указать соединение к нужной базе данных. Третий запрос создает копию запроса, переданного в параметре.

Для выполнения запросов используются следующие методы класса `QSqlQuery`:

- `exec(<SQL-код>)` — немедленно выполняет переданный в параметре SQL-код. Если последний был успешно выполнен, возвращает `True` и переводит запрос в активное состояние, в противном случае возвращает `False`. Пример использования этого метода показан в листинге 23.2.

Листинг 23.2. Использование метода `exec()`

```
from PyQt5 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
# Проверяем, есть ли в базе данных таблица good, и, если таковой нет,
# создаем ее SQL-командой CREATE TABLE
if 'good' not in con.tables():
    query = QtSql.QSqlQuery()
    query.exec("create table good(id integer primary key autoincrement, ↳
    goodname text, goodcount integer) ")
con.close()
```

Совет

Метод `exec()` следует использовать в тех случаях, если SQL-запрос не принимает параметров. В противном случае рекомендуется применять методы, рассмотренные далее.

- `prepare(<SQL-код>)` — подготавливает SQL-запрос к выполнению. Применяется, если SQL-запрос содержит параметры. Параметры в коде запроса могут быть заданы либо в стиле ODBC (вопросительными знаками), либо в стиле Oracle (символьными обозначениями, предваренными знаком двоеточия). Метод возвращает `True`, если SQL-запрос был успешно подготовлен, и `False` — в противном случае;
- `exec_()` — выполняет подготовленный ранее запрос. Возвращает `True`, если запрос был успешно выполнен, и `False` — в противном случае;
- `addBindValue(<Значение параметра>[, paramType=In])` — задает значение очередного по счету параметра: так, первый вызов этого метода задает значение для первого параметра, второй вызов — для второго и т. д. Необязательный параметр `paramType` указывает тип параметра — здесь практически всегда используется атрибут `In` класса `QSql`, означающий, что этот параметр служит для занесения значения в базу.

В листинге 23.3 приведен пример использования методов `prepare()`, `addBindValue()` и `exec_()`.

Листинг 23.3. Использование методов `prepare()`, `addBindValue()` и `exec_()`

```
from PyQt5 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
# Добавляем в только что созданную таблицу good запись,
# используя SQL-команду INSERT
query.prepare("insert into good values(null, ?, ?)")
query.addBindValue('Дискета')
query.addBindValue(10)
query.exec_()
con.close()
```

- `bindValue(<Номер параметра>, <Значение параметра>[, paramType=In])` — задает значение для параметра с указанным порядковым номером (листинг 23.4).

Листинг 23.4. Использование метода `bindValue()` для задания параметров по их порядковым номерам

```
from PyQt5 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
```

```
query.prepare("insert into good values(null, ?, ?)")
query.bindValue(0, 'Компакт-диск')
query.bindValue(1, 5)
query.exec_()
con.close()
```

♦ `bindValue(<Обозначение параметра>, <Значение параметра>[, paramType=In])` — задает значение для параметра с указанным символьным обозначением (листинг 23.5).

Листинг 23.5. Использование метода `bindValue()` для задания параметров по их символьным обозначениям

```
from PyQt5 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
query.prepare("insert into good values(null, :name, :count)")
query.bindValue(':name', 'Флеш-накопитель')
query.bindValue(':count', 20)
query.exec_()
con.close()
```

♦ `execBatch([mode=ValuesAsRows])` — если в вызове метода `addBindValue()` или `bindValue()` в качестве значения параметра был указан список, выполнит подготовленный запрос.

  Необязательный параметр `mode` позволяет указать, как будут интерпретироваться отдельные элементы списка. В настоящее время в качестве его значения для всех форматов баз данных поддерживается лишь атрибут `ValuesAsRows` класса `QSqlQuery`, говорящий, что подготовленный запрос должен быть выполнен столько раз, сколько элементов присутствует в списке, при этом на каждом выполнении запроса в его код подставляется очередной элемент списка.

  Метод возвращает `True`, если запрос был успешно выполнен, и `False` — в противном случае.

  Листинг 23.6 представляет пример добавления в таблицу сразу нескольких записей с применением метода `execBatch()`.

Листинг 23.6. Использование метода `execBatch()` для добавления в таблицу сразу нескольких записей

```
from PyQt5 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
query.prepare("insert into good values(null, :name, :count)")
```

```
lst1 = ['Бумага офисная', 'Фотобумага', 'Картридж']
lst2 = [15, 8, 3]
query.bindValue(':name', lst1)
query.bindValue(':count', lst2)
query.execBatch()
con.close()
```

♦ `setForwardOnly(<Флаг>)` — если передано значение `True`, по результату запроса можно будет перемещаться только «вперед», т. е. от начала к концу. Такой режим выполнения запроса существенно сокращает потребление системных ресурсов. Этот метод должен быть вызван перед выполнением запроса, который возвращает результат:

```
query.prepare("select * from good order by goodname")
query.setForwardOnly(True)
query.exec_()
```

23.3.2. Обработка результатов выполнения запросов

Если был выполнен запрос на выборку данных (SQL-команда `SELECT`), следует получить результат его выполнения. Для этого мы используем методы класса `QSqlQuery`, описанные в этом разделе.

Запрос на выборку данных поддерживает особый внутренний указатель, указывающий на запись результата, содержимое которой в настоящее время доступно для получения. Однако сразу после выполнения запроса этот указатель хранит неопределенное значение, не идентифицирующее никакую реальную запись. Поэтому перед собственно выборкой данных необходимо позиционировать этот указатель на нужную запись:

♦ `first()` — позиционирует указатель запроса на первую запись результата. Возвращает `True`, если позиционирование прошло успешно, и `False` — в противном случае;

♦ `next()` — позиционирует указатель запроса на следующую запись результата или на первую запись, если этот метод был вызван сразу после выполнения запроса. Возвращает `True`, если позиционирование прошло успешно, и `False` — в противном случае;

♦ `previous()` — позиционирует указатель запроса на предыдущую запись результата или на последнюю запись, если указатель в текущий момент находится за последней записью. Возвращает `True`, если позиционирование прошло успешно, и `False` — в противном случае;

♦ `last()` — позиционирует указатель запроса на последнюю запись результата. Возвращает `True`, если позиционирование прошло успешно, и `False` — в противном случае;

♦ `seek(<Номер записи>[, relative=False])` — позиционирует указатель на запись с указанным номером (нумерация записей начинается с нуля). Если необязательным параметром `relative` передано значение `True`, то позиционирование выполняется относительно текущей записи: положительные значения вызывают смещение указателя «вперед» (к концу), а отрицательные — «назад» (к началу). Возвращает `True`, если позиционирование прошло успешно, и `False` — в противном случае;

♦ `isValid()` — возвращает `True`, если внутренний указатель указывает на какую-либо запись, и `False`, если он имеет неопределенное значение;

♦ `at()` — возвращает номер записи, на которую указывает внутренний указатель запроса;

- `size()` — возвращает количество записей, возвращенных в результате выполнения запроса, или `-1`, если этот запрос не выполнял выборку данных.

Для собственно выборки данных следует применять описанные далее методы:

- `value(<Индекс поля>)` — возвращает значение поля текущей записи с заданным индексом. Поля нумеруются в том порядке, в котором они присутствуют в таблице базы или в SQL-коде запроса;
- `value(<Имя поля>)` — возвращает значение поля текущей записи с заданным именем;
- `isNull(<Индекс поля>)` — возвращает `True`, если в поле с указанным индексом нет значения, и `False` — в противном случае;
- `isNull(<Имя поля>)` — возвращает `True`, если в поле с указанным именем нет значения, и `False` — в противном случае;
- `record()` — если внутренний указатель установлен на какую-либо запись, возвращает сведения об этой записи, в противном случае возвращаются сведения о самой таблице. Возвращаемым результатом является экземпляр класса `QSqlRecord`;
- `isSelect()` — возвращает `True`, если был выполнен запрос на выборку данных, и `False`, если исполнялся запрос иного рода.

В листинге 23.7 приведен код, извлекающий данные из таблицы `good` созданной ранее базы данных и выводящий их на экран.

Листинг 23.7. Выборка данных из базы

```
from PyQt5 import QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
query = QtSql.QSqlQuery()
query.exec("select * from good order by goodname")
lst = []
if query.isActive():
    query.first()
    while query.isValid():
        lst.append(query.value('goodname') + ': ' + ↳
        str(query.value('goodcount')) + ' шт.')
        query.next()
    for p in lst: print(p)
con.close()
```

Результат выполнения этого кода:

```
Бумага офисная: 15 шт.
Дискета: 10 шт.
Картридж: 3 шт.
Компакт-диск: 5 шт.
Флеш-накопитель: 20 шт.
Фотобумага: 8 шт.
```

23.3.3. Очистка запроса

После первого выполнения метода `exec()`, `exec_()` или `execBatch()`) запрос переходит в активное состояние. Это значит, что выполнить любую другую SQL-команду с его помощью невозможно.

Метод `isActive()` класса `QSqlQuery` возвращает `True`, если запрос находится в активном состоянии. Если же запрос неактивен, метод возвращает `False`.

Если один и тот же экземпляр класса `QSqlQuery` планируется использовать для выполнения нескольких SQL-команд, перед выполнением новой команды следует сбросить его, переведя тем самым в неактивное состояние и освободив занимаемые им системные ресурсы. Это выполняется вызовом метода `clear()`:

```
query = QtSql.QSqlQuery()
query.exec("select * from good order by goodname")
# Обрабатываем результат запроса
query.clear()
query.exec("select count(*) as cnt from good")
# Работаем с новым запросом
```

23.3.4. Получение служебных сведений о запросе

Класс `QSqlQuery` позволяет также получить всевозможные служебные сведения о запросе. Для этого применяются следующие методы:

- `numRowsAffected()` — возвращает количество записей, обработанных в процессе выполнения запроса, или `-1`, если это количество не удается определить. Для запросов выборки данных возвращает `None` — в этом случае следует вызывать метод `size()`;
- `lastInsertId()` — возвращает идентификатор последней добавленной записи. Если запрос не добавлял записи, или если формат базы данных не позволяет определить идентификатор последней добавленной записи, возвращает `None`;
- `lastError()` — возвращает экземпляр объекта `QSqlError`, описывающий последнюю возникшую в базе данных ошибку;
- `executedQuery()` — возвращает SQL-код последнего выполненного запроса или пустую строку, если никакой запрос еще не был выполнен;
- `lastQuery()` — возвращает код последнего выполненного запроса или пустую строку, если никакой запрос еще не был выполнен. Отличается от метода `executedQuery()` тем, что все именованные параметры (заданные символьными обозначениями) в возвращаемом SQL-коде заменяются вопросительными знаками;
- `boundValue(<Номер параметра>)` — возвращает значение параметра запроса с указанным номером;
- `boundValue(<Обозначение параметра>)` — возвращает значение параметра запроса с указанным символьным обозначением;
- `boundValues()` — возвращает словарь, ключами элементов которого служат символьные обозначения параметров, а значениями элементов — значения этих параметров. Если параметры обозначены вопросительными знаками, в качестве ключей используются произвольные строки вида `:a` для первого параметра, `:bb` для второго и т. д.

23.4. Модели, связанные с данными

Очень часто данные, хранящиеся в базе, выводятся на экран с применением таких компонентов, как списки или таблицы (подробно списки и таблицы описаны в *главе 22*). Для этих случаев PyQt предоставляет два класса-модели, извлекающие данные напрямую из базы.

23.4.1. Модель, связанная с SQL-запросом

Если требуется вывести на экран данные, извлеченные в результате выполнения SQL-запроса, и эти данные не требуется редактировать, имеет смысл использовать класс `QSqlQueryModel`. Он представляет модель, связанную с SQL-запросом. Иерархия наследования этого класса:

```
QObject - QAbstractItemModel - QAbstractTableModel - QSqlQueryModel
```

Конструктор класса:

```
<Объект> = QSqlQueryModel([parent=None])
```

Класс `QSqlQueryModel` поддерживает следующие методы (здесь приведен их сокращенный список, а полный список методов этого класса доступен на страницах **https://doc.qt.io/qt-5/qsqlquerymodel.html** и **https://doc.qt.io/qt-5/qabstractitemmodel.html**):

- `setQuery(<Код запроса>[, db=QSqlDatabase()])` — задает код запроса для модели. Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию;
- `query()` — возвращает код запроса, заданного для модели;
- `record()` — возвращает экземпляр класса `QSqlRecord`, представляющий сведения о структуре результата запроса;
- `record(<Индекс строки>)` — возвращает экземпляр класса `QSqlRecord`, представляющий сведения о записи, которая соответствует строке модели с указанным индексом;
- `lastError()` — возвращает экземпляр объекта `QSqlError`, описывающий последнюю возникшую в базе данных ошибку;
- `index(<Строка>, <Столбец>[, parent=QModelIndex()])` — возвращает индекс (экземпляр класса `QModelIndex`) элемента модели, находящегося на пересечении строки и столбца с указанными индексами. Необязательный параметр `parent` позволяет задать элемент верхнего уровня для искомого элемента — если таковой не задан, будет выполнен поиск элемента на самом верхнем уровне иерархии;
- `data(<QModelIndex>[, role=DisplayRole])` — возвращает данные, хранимые в указанной в параметре `role` роли элемента, на который ссылается индекс `<QModelIndex>`;
- `rowCount([parent=QModelIndex()])` — возвращает количество элементов в модели. Необязательный параметр `parent` указывает элемент верхнего уровня, при этом будет возвращено количество вложенных в него элементов. Если параметр не задан, возвращается количество элементов верхнего уровня иерархии;
- `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` класса `QtCore.Qt`, сортировка производится в прямом порядке, а если `DescendingOrder` — в обратном;
- `setHeaderData(<Индекс>, <Ориентация>, <Значение>[, role=EditRole])` — задает значение для указанной роли заголовка. В первом параметре указывается индекс строки или

столбца, а во втором — ориентация (атрибут `Horizontal` или `Vertical` класса `QtCore.Qt`). Метод возвращает значение `True`, если операция успешно выполнена;

- `headerData(<Индекс>, <Ориентация>[, role=DisplayRole])` — возвращает значение, соответствующее указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором — ориентация.

Рассмотрим пример, выводящий данные из созданной нами ранее базы с помощью компонента таблицы (листинг 23.8).

Листинг 23.8. Использование модели, привязанной к SQL-запросу

```
from PyQt5 import QtCore, QtWidgets, QtSql
import sys
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QTableView()
window.setWindowTitle("QSqlQueryModel")
# Устанавливаем соединение с базой данных
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
# Создаем модель
sqm = QtSql.QSqlQueryModel(parent=window)
sqm.setQuery('select * from good order by goodname')
# Задаем заголовки для столбцов модели
sqm.setHeaderData(1, QtCore.Qt.Horizontal, 'Название')
sqm.setHeaderData(2, QtCore.Qt.Horizontal, 'Кол-во')
# Задаем для таблицы только что созданную модель
window.setModel(sqm)
# Скрываем первый столбец, в котором выводится идентификатор
window.hideColumn(0)
window.setColumnWidth(1, 150)
window.setColumnWidth(2, 60)
window.resize(260, 160)
window.show()
sys.exit(app.exec_())
```

23.4.2. Модель, связанная с таблицей

Если необходимо дать пользователю возможность редактировать данные, хранящиеся в базе, следует использовать класс `QSqlTableModel`. Он представляет модель, связанную непосредственно с указанной таблицей базы данных. Иерархия наследования:

```
QObject - QAbstractItemModel - QAbstractTableModel - QSqlQueryModel -
                               QSqlTableModel
```

Конструктор класса:

```
<Объект> = QSqlTableModel([parent=None][, db=QSqlDatabase()])
```

Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию.

Класс `QSqlTableModel` наследует все методы из класса `QSqlQueryModel` (см. *разд. 23.4.1*) и в дополнение к ним определяет следующие наиболее полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-5/qsqltablemodel.html**):

- `setTable(<Имя таблицы>)` — задает таблицу, данные из которой будут представлены в модели. Отметим, что этот метод лишь выполняет получение из базы данных структуры указанной таблицы, но не загружает сами эти данные;
- `tableName()` — возвращает имя таблицы, заданной для модели;
- `setSort(<Индекс столбца>, <Порядок сортировки>)` — задает сортировку данных. Если во втором параметре указан атрибут `AscendingOrder` класса `QtCore.Qt`, сортировка производится в прямом порядке, а если `DescendingOrder` — в обратном;
- `setFilter(<Условие фильтрации>)` — задает условие для фильтрации данных в виде строки в том формате, который применяется в SQL-команде `WHERE`;
- `filter()` — возвращает строку с фильтром, заданным для модели;
- `select()` — считывает в модель данные из заданной ранее таблицы с учетом указанных параметров сортировки и фильтрации. Возвращает `True`, если считывание данных прошло успешно, и `False` — в противном случае:

  ```
  stm = QtSql.QSqlTableModel(parent=window)
  stm.setTable('good')
  stm.setSort(1, QtCore.Qt.DescendingOrder)
  stm.setFilter('goodcount > 2')
  stm.select()
  ```

  Метод является слотом;
- `setEditStrategy(<Режим редактирования>)` — указывает режим редактирования данных в модели. В качестве параметра используется один из атрибутов класса `QSqlTableModel`:
  - `OnFieldChange` — `0` — все изменения переносятся в базу данных немедленно;
  - `OnRowChange` — `1` — изменения переносятся в базу лишь после того, как пользователь перейдет на другую строку;
  - `OnManualSubmit` — `2` — изменения переносятся в базу только после вызова метода `submit()` или `submitAll()`;
- `insertRow(<Индекс>[, parent=QModelIndex()])` — вставляет пустую запись в позицию, заданную первым параметром. Возвращает значение `True`, если запись была успешно добавлена, и `False` — в противном случае;
- `insertRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — вставляет указанное количество пустых записей в позицию, заданную первым параметром. Возвращает значение `True`, если записи были успешно добавлены, и `False` — в противном случае;
- `setData(<QModelIndex>, <Значение>[, role=EditRole])` — задает значение для роли `role` поля записи, на которое указывает индекс `<QModelIndex>`. Возвращает значение `True`, если данные были успешно занесены в запись, и `False` — в противном случае;
- `removeRow(<Индекс>[, parent=QModelIndex()])` — удаляет запись с указанным индексом. Возвращает значение `True`, если запись была успешно удалена, и `False` — в противном случае;
- `removeRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество записей, начиная с записи с указанным индексом. Возвращает значение `True`, если записи были успешно удалены, и `False` — в противном случае;

Примечание

Нужно отметить, что после удаления записи вызовом метода `removeRow()` или `removeRows()` в модели останется пустая запись, реально не представляющая никакой записи из таблицы. Чтобы убрать ее, достаточно выполнить повторное считывание данных в модель вызовом метода `select()`.

- `insertRecord(<Индекс>, <QSqlRecord>)` — добавляет в модель новую запись в позицию, указанную первым параметром. Если значение первого параметра отрицательное, запись добавляется в конец модели. Добавляемая запись представляется экземпляром объекта `QSqlRecord`, уже заполненным необходимыми данными. Возвращает `True`, если запись была успешно добавлена, и `False` — в противном случае;
- `setRecord(<Индекс>, <QSqlRecord>)` — заменяет запись в позиции, указанной первым параметром, новой записью, которая передается вторым параметром в виде экземпляра объекта `QSqlRecord`, уже заполненного необходимыми данными. Возвращает `True`, если запись была успешно изменена, и `False` — в противном случае;
- `submit()` — переносит в базу данных изменения, сделанные в текущей записи, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно перенесены, и `False` — в противном случае. Метод является слотом;
- `submitAll()` — переносит в базу данных изменения, сделанные во всех записях, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно перенесены, и `False` — в противном случае. Метод является слотом;
- `revert()` — отменяет изменения, сделанные в текущей записи, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно отменены, и `False` — в противном случае. Метод является слотом;
- `revertRow(<Индекс записи>)` — отменяет изменения, сделанные в записи с заданным индексом, если был задан режим редактирования `OnManualSubmit`;
- `revertAll()` — отменяет изменения, сделанные во всех записях, если был задан режим редактирования `OnManualSubmit`. Возвращает `True`, если изменения были успешно отменены, и `False` — в противном случае. Метод является слотом;
- `selectRow(<Индекс строки>)` — обновляет содержимое строки с указанным индексом. Возвращает `True`, если запись была успешно обновлена, и `False` — в противном случае. Метод является слотом;
- `isDirty(<QModelIndex>)` — возвращает `True`, если запись с указанным индексом (экземпляр класса `QModelIndex`) была изменена, но эти изменения еще не были перенесены в базу данных, и `False` — в противном случае;
- `isDirty()` — возвращает `True`, если хотя бы одна запись в модели была изменена, но эти изменения еще не были перенесены в базу данных, и `False` — в противном случае;
- `fieldIndex(<Имя поля>)` — возвращает индекс поля с указанным именем или `-1`, если такого поля нет;
- `primaryKey()` — возвращает сведения о ключевом индексе таблицы, представленные экземпляром класса `QSqlIndex`, или пустой экземпляр этого класса, если таблица не содержит ключевого индекса.

Методы `insertRecord()` и `setRecord()`, предназначенные, соответственно, для добавления и изменения записи, принимают в качестве второго параметра экземпляр класса `QSqlRecord`.

Чтобы создать этот экземпляр, нам следует знать формат вызова конструктора класса. Вот он:

```
<Объект> = QSqlRecord([<QSqlRecord>])
```

Если в параметре указать экземпляр класса `QSqlRecord`, будет создана его копия. Обычно при создании новой записи здесь указывают значение, возвращенное методом `record()` класса `QSqlDatabase` (оно хранит сведения о структуре таблицы и, следовательно, представляет пустую запись), а при правке существующей записи — значение, возвращенное методом `record()`, который унаследован классом `QSqlTableModel` от класса `QSqlQueryModel` (оно представляет запись, которую нужно отредактировать).

Класс `QSqlRecord`, в дополнение к методам, рассмотренным нами в *разд. 23.2.1*, поддерживает следующие методы:

- `value(<Индекс поля>)` — возвращает значение поля текущей записи с заданным индексом;
- `value(<Имя поля>)` — возвращает значение поля текущей записи с заданным именем;
- `setValue(<Индекс поля>, <Значение>)` — заносит в поле с указанным индексом новое значение;
- `setValue(<Имя поля>, <Значение>)` — заносит в поле с указанным именем новое значение;
- `isNull(<Индекс поля>)` — возвращает `True`, если в поле с указанным индексом нет значения, и `False` — в противном случае;
- `isNull(<Имя поля>)` — возвращает `True`, если в поле с указанным именем нет значения, и `False` — в противном случае;
- `setNull(<Индекс поля>)` — удаляет значение из поля с указанным индексом;
- `setNull(<Имя поля>)` — удаляет значение из поля с указанным именем;
- `clearValues()` — удаляет значения из всех полей записи;
- `setGenerated(<Индекс поля>, <Флаг>)` — если вторым параметром передано `False`, поле с указанным индексом помечается как неактуальное, и хранящееся в нем значение не будет перенесено в таблицу;
- `setGenerated(<Имя поля>, <Флаг>)` — если вторым параметром передано `False`, поле с указанным именем помечается как неактуальное, и хранящееся в нем значение не будет перенесено в таблицу;
- `isGenerated(<Индекс поля>)` — возвращает `False`, если поле с указанным индексом помечено как неактуальное, и `True` — в противном случае;
- `isGenerated(<Имя поля>)` — возвращает `False`, если поле с указанным именем помечено как неактуальное, и `True` — в противном случае.

Вот пример кода, добавляющего новую запись в модель:

```
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
stm = QtSql.QSqlTableModel()
stm.setTable('good')
stm.select()
rec = con.record('good')
```

```
rec.setValue('goodname', 'Коврик для мыши')
rec.setValue('goodcount', 2)
stm.insertRecord(-1, rec)
```

А вот пример кода, редактирующего существующую запись с индексом `3`:

```
rec = stm.record(3)
rec.setValue('goodcount', 5)
stm.setRecord(3, rec)
```

Класс `QSqlTableModel` поддерживает такие сигналы:

- `primeInsert(<Индекс записи>, <QSqlRecord>)` — генерируется перед добавлением записи в модель. В первом параметре доступен целочисленный индекс добавляемой записи, а во втором — сама добавляемая запись, обычно пустая, в которую можно занести какие-либо изначальные данные;
- `beforeInsert(<QSqlRecord>)` — генерируется перед добавлением новой записи в таблицу. В параметре доступна добавляемая запись;
- `beforeUpdate(<Индекс записи>, <QSqlRecord>)` — генерируется перед изменением записи в таблице. В параметрах доступны целочисленный индекс изменяемой записи и сама изменяемая запись;
- `beforeDelete(<Индекс записи>)` — генерируется перед удалением записи из таблицы. В параметре доступен целочисленный индекс удаляемой записи;
- `dataChanged(<QModelIndex>, <QModelIndex>, roles=[])` — генерируется при изменении данных в модели пользователем. Первым параметром передается индекс верхней левой из набора измененных записей, вторым — индекс правой нижней. Необязательный параметр `roles` хранит список ролей, данные которых изменились. Если указан пустой список, значит, изменились данные во всех ролях.

  Сигнал `dataChanged` — идеальное место для вызова методов `submit()` или `submitAll()` в случае, если для модели был задан режим редактирования `OnManualSubmit`. Как мы знаем, эти методы выполняют сохранение отредактированных данных в базе.

В листинге 23.9 представлен код тестового складского приложения, позволяющего не только править, но и добавлять и удалять записи нажатием соответствующих кнопок. А на рис. 23.1 можно увидеть само это приложение в работе.

Листинг 23.9. Использование модели, привязанной к таблице

```
from PyQt5 import QtCore, QtWidgets, QtSql
import sys

def addRecord():
    # Вставляем пустую запись, в которую пользователь сможет
    # ввести нужные данные
    stm.insertRow(stm.rowCount())

def delRecord():
    # Удаляем запись из модели
    stm.removeRow(tv.currentIndex().row())
    # Выполняем повторное считывание данных в модель,
    # чтобы убрать пустую "мусорную" запись
    stm.select()
```

```
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QSqlTableModel")
# Устанавливаем соединение с базой данных
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
# Создаем модель
stm = QtSql.QSqlTableModel(parent=window)
stm.setTable('good')
stm.setSort(1, QtCore.Qt.AscendingOrder)
stm.select()
# Задаем заголовки для столбцов модели
stm.setHeaderData(1, QtCore.Qt.Horizontal, 'Название')
stm.setHeaderData(2, QtCore.Qt.Horizontal, 'Кол-во')
# Задаем для таблицы только что созданную модель
vbox = QtWidgets.QVBoxLayout()
tv = QtWidgets.QTableView()
tv.setModel(stm)
# Скрываем первый столбец, в котором выводится идентификатор
tv.hideColumn(0)
tv.setColumnWidth(1, 150)
tv.setColumnWidth(2, 60)
vbox.addWidget(tv)
btnAdd = QtWidgets.QPushButton("&Добавить запись")
btnAdd.clicked.connect(addRecord)
vbox.addWidget(btnAdd)
btnDel = QtWidgets.QPushButton("&Удалить запись")
btnDel.clicked.connect(delRecord)
vbox.addWidget(btnDel)
window.setLayout(vbox)
window.resize(300, 250)
window.show()
sys.exit(app.exec_())
```

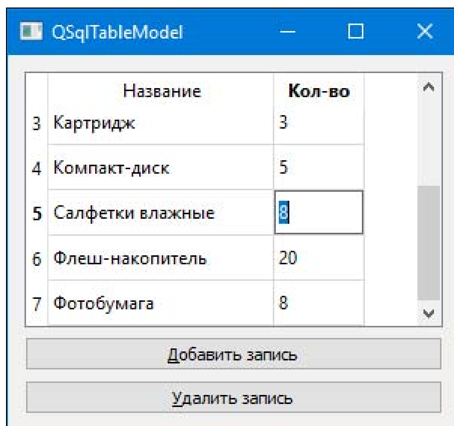

Рис. 23.1. Пример складского приложения, использующего модель `QSqlTableModel`

23.4.3. Модель, поддерживающая межтабличные связи

Предположим, что мы решили расширить наше простенькое складское приложение, введя разбиение товаров на категории. В базе данных `data.sqlite` мы создали таблицу `category` с полями `id` и `catname`, а в таблицу `good` добавили поле `category`, где будут храниться идентификаторы категорий.

Теперь попытаемся вывести содержимое таблицы `good` на экран с помощью модели `QSqlTableModel` и компонента таблицы `QTableView`. И сразу увидим, что в колонке, где показывается содержимое поля `category`, выводятся числовые идентификаторы категорий (рис. 23.2). А нам хотелось бы видеть там наименования категорий вместо непонятной цифири.

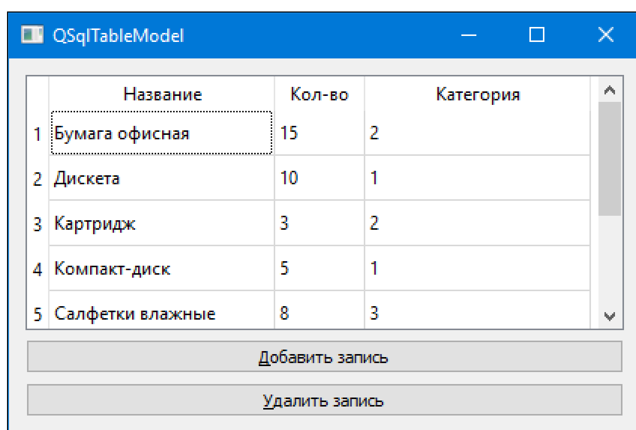

Рис. 23.2. Пример складского приложения после доработки: вместо названий категорий выводятся их числовые идентификаторы

Сделать это поможет класс `QSqlRelationalTableModel`, добавляющий уже известной нам модели `QSqlTableModel` возможность связывать таблицы. Мы указываем поле внешнего ключа, первичную таблицу и в ней — поле первичного ключа и поле, откуда будет взято значение для вывода на экран.

Иерархия наследования класса `QSqlRelationalTableModel`:

```
QObject - QAbstractItemModel - QAbstractTableModel - QSqlQueryModel -
                                QSqlTableModel - QSqlRelationalTableModel
```

Конструктор класса:

```
<Объект> = QSqlRelationalQueryModel([parent=None][, db=QSqlDatabase()])
```

Необязательный параметр `db` задает соединение с базой данных, запрос к которой следует выполнить, — если он не указан, будет использоваться соединение по умолчанию.

Класс `QSqlRelationalTableModel` наследует все методы класса `QSqlTableModel` (см. *разд. 23.4.2*) и в дополнение к ним определяет следующие полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-5/qsqlrelationaltablemodel.html**):

- `setRelation(<Индекс столбца>, <QSqlRelation>)` — задает связь для поля с указанным индексом. Сведения об устанавливаемой связи представляются экземпляром класса `QSqlRelation`, о котором мы поговорим чуть позже;

- `setJoinMode(<Режим связывания>)` — задает режим связывания для всей модели. В качестве параметра указывается один из атрибутов класса `QSqlRelationalTableModel`:
  - `InnerJoin` — `0` — каждой записи вторичной таблицы должна соответствовать связанная с ней запись первичной таблицы. Используется по умолчанию;
  - `LeftJoin` — `1` — записи вторичной таблицы не обязательно должна соответствовать связанная запись первичной таблицы.

Теперь о классе `QSqlRelation`. Он представляет связь, устанавливаемую между полями таблиц. Конструктор этого класса имеет такой формат:

```
<Объект> = QSqlRelation(<Имя первичной таблицы>,
                        <Имя поля первичного ключа>,
                        <Имя поля, выводящегося на экран>)
```

Поля, чьи имена указываются во втором и третьем параметрах, относятся к первичной таблице.

Класс `QSqlRelation` поддерживает несколько методов, но они не очень нам интересны (полное описание этого класса доступно на странице **https://doc.qt.io/qt-5/qsqlrelation.html**).

В листинге 23.10 приведен код исправленного складского приложения, а на рис. 23.3 показан его интерфейс.

Листинг 23.10. Использование модели `QSqlRelationalTableModel`

```
from PyQt5 import QtCore, QtWidgets, QtSql
import sys

def addRecord():
    stm.insertRow(stm.rowCount())

def delRecord():
    stm.removeRow(tv.currentIndex().row())
    stm.select()

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("QRelationalSqlTableModel")
con = QtSql.QSqlDatabase.addDatabase('QSQLITE')
con.setDatabaseName('data.sqlite')
con.open()
stm = QtSql.QSqlRelationalTableModel(parent=window)
stm.setTable('good')
stm.setSort(1, QtCore.Qt.AscendingOrder)
# Задаем для поля категории связь с таблицей списка категорий
stm.setRelation(3, QtSql.QSqlRelation('category', 'id', 'catname'))
stm.select()
stm.setHeaderData(1, QtCore.Qt.Horizontal, 'Название')
stm.setHeaderData(2, QtCore.Qt.Horizontal, 'Кол-во')
stm.setHeaderData(3, QtCore.Qt.Horizontal, 'Категория')
vbox = QtWidgets.QVBoxLayout()
tv = QtWidgets.QTableView()
```

```
tv.setModel(stm)
tv.hideColumn(0)
tv.setColumnWidth(1, 150)
tv.setColumnWidth(2, 60)
tv.setColumnWidth(3, 150)
vbox.addWidget(tv)
btnAdd = QtWidgets.QPushButton("&Добавить запись")
btnAdd.clicked.connect(addRecord)
vbox.addWidget(btnAdd)
btnDel = QtWidgets.QPushButton("&Удалить запись")
btnDel.clicked.connect(delRecord)
vbox.addWidget(btnDel)
window.setLayout(vbox)
window.resize(420, 250)
window.show()
sys.exit(app.exec_())
```

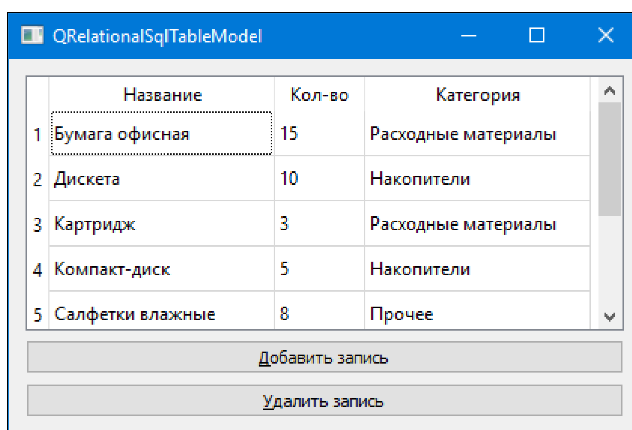

Рис. 23.3. Пример складского приложения, использующего модель `QSqlRelationalTableModel`: на экран выводятся названия категорий

23.4.4. Использование связанных делегатов

К сожалению, наше новое приложение имеет один существеннейший недостаток — как только мы решим добавить новую запись или даже исправить уже существующую, то столкнемся с тем, что все сделанные нами изменения не сохраняются. Почему?

Дело в том, что модель `QSqlRelationalTableModel` «не знает», как перевести введенное нами название категории в ее идентификатор, который и хранится в поле `category` таблицы `good`. Она лишь выполняет попытку занести строковое название категории в поле целочисленного типа, что вполне ожидаемо вызывает ошибку, и запись в таблице не сохраняется.

Исправить такое положение дел нам позволит особый делегат, называемый *связанным* (о делегатах рассказывалось в *разд. 22.8*). Он способен выполнить поиск в первичной таблице нужной записи, извлечь ее идентификатор и сохранить его в поле вторичной таблицы. А, кроме того, он представляет все доступные для занесения в поле значения, взятые из первичной таблицы, в виде раскрывающегося списка — очень удобно!

Функциональность связанного делегата реализует класс `QSqlRelationalDelegate`. Иерархия наследования:

```
QObject – QAbstractItemDelegate – QItemDelegate – QSqlRelationalDelegate
```

Использовать связанный делегат очень просто — нужно лишь создать его экземпляр, передав конструктору класса ссылку на компонент-представление (в нашем случае — таблицу), и вызвать у представления метод `setItemDelegate()`, `setItemDelegateForColumn()` или `setItemDelegateForRow()`, указав в нем только что созданный делегат.

Исходя из этого, давайте, наконец, доделаем до конца наше складское приложение, дав пользователю возможность выбирать категории товаров из списка. Для этого нам потребуется лишь вставить в код листинга 23.10 всего одно новое выражение (в приведенном далее листинге 23.11 оно выделено полужирным шрифтом):

Листинг 23.11. Использование связанного делегата (фрагмент исправленного кода из листинга 23.10)

```
. . .
tv = QtWidgets.QTableView()
tv.setModel(stm)
tv.setItemDelegateForColumn(3, QtSql.QSqlRelationalDelegate(tv))
tv.hideColumn(0)
. . .
```

Интерфейс законченного приложения показан на рис. 23.4.

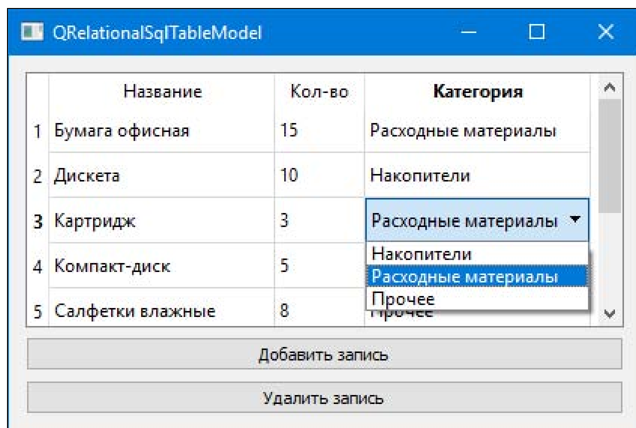

Рис. 23.4. Окончательный вариант складского приложения, использующего связанный делегат

ГЛАВА 24

Работа с графикой

Все компоненты, которые мы рассматривали в предыдущих главах, на самом деле нарисованы. То есть, каждый раз, когда компонент становится видимым (в первый раз, при отображении части компонента, ранее перекрытой другим окном, или после изменения его параметров), вызывается метод `paintEvent()` (см. *разд. 19.7.3*). Вызвать событие перерисовки компонента можно и искусственно — с помощью методов `repaint()` и `update()` класса `QWidget`. Внутри метода `paintEvent()` выполняется рисование компонента с помощью методов класса `QPainter`.

Класс `QPainter` поддерживает все необходимые средства, позволяющие выполнять рисование геометрических фигур и вывод текста на поверхности, которая реализуется классом `QPaintDevice`. Класс `QWidget` наследует класс `QPaintDevice`. В свою очередь класс `QWidget` наследуют все компоненты, поэтому мы можем рисовать на поверхности любого компонента. Класс `QPaintDevice` наследуют также классы `QPicture`, `QPixmap`, `QImage`, `QPagedPaintDevice` и некоторые другие.

Класс `QPicture` позволяет сохранить команды рисования в метафайл, а затем считать их из файла и воспроизвести на какой-либо поверхности. Классы `QPixmap` и `QImage` позволяют обрабатывать изображения. Основные методы этих классов мы рассмотрим далее в этой главе.

Класс `QPagedPaintDevice` является базовым для классов `QPrinter` (позволяет выводить документы на печать) и `QPdfWriter` (используется для экспорта документов в PDF-файлы). Мы рассмотрим их в *главе 29*.

Все описанные в этой главе классы объявлены в модуле `QtGui`, если не указано обратное.

Библиотека PyQt 5 также позволяет работать с SVG-графикой и включает в свой состав поддержку технологии OpenGL, предназначенной для обработки двумерной и трехмерной графики. Рассмотрение этих возможностей выходит за рамки нашей книги, поэтому за подробной информацией о них вам следует обратиться к соответствующей документации.

24.1. Вспомогательные классы

Прежде чем изучать работу с графикой, необходимо рассмотреть несколько вспомогательных классов, с помощью которых производится настройка различных параметров: цвета, характеристик шрифта, стиля пера и кисти. Кроме того, мы рассмотрим классы, описывающие геометрические фигуры (например, линию и многоугольник).

24.1.1. Класс *QColor*: цвет

Класс `QColor` описывает цвет в цветовых моделях RGB, CMYK, HSV или HSL. Форматы конструктора класса `QColor`:

```
<Объект> = QColor()
<Объект> = QColor(<Красный>, <Зеленый>, <Синий>[, alpha=255])
<Объект> = QColor(<Строка>)
<Объект> = QColor(<Атрибут цвета>)
<Объект> = QColor(<Число>)
<Объект> = QColor(<QColor>)
```

Первый конструктор создает невалидный объект. Проверить объект на валидность можно с помощью метода `isValid()`. Метод возвращает значение `True`, если объект является валидным, и `False` — в противном случае.

Второй конструктор позволяет указать целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `alpha` задает степень прозрачности цвета. Значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному.

Вот пример указания красного цвета:

```
red = QtGui.QColor(255, 0, 0)
```

В третьем конструкторе цвет указывается в виде строки в форматах `"#RGB"`, `"#RRGGBB"`, `"#AARRGGBB"` (здесь `AA` обозначает степень прозрачности цвета), `"#RRRGGGBBB"`, `"#RRRRGGGGBBBB"`, `"Название цвета"` или `"transparent"` (для прозрачного цвета):

```
red = QtGui.QColor("#f00")
darkBlue = QtGui.QColor("#000080")
semiTransparentDarkBlue = QtGui.QColor("#7F000080")
white = QtGui.QColor("white")
```

Получить список всех поддерживаемых названий цветов позволяет статический метод `colorNames()`. Проверить правильность строки с названием цвета можно с помощью статического метода `isValidColor(<Строка>)`, который возвращает значение `True`, если строка является правильным наименованием цвета, и `False` — в противном случае:

```
print(QtGui.QColor.colorNames()) # ['aliceblue', 'antiquewhite', ...]
print(QtGui.QColor.isValidColor("lightcyan")) # True
```

В четвертом конструкторе указываются следующие атрибуты из класса `QtCore.Qt`: `white`, `black`, `red`, `darkRed`, `green`, `darkGreen`, `blue`, `darkBlue`, `cyan`, `darkCyan`, `magenta`, `darkMagenta`, `yellow`, `darkYellow`, `gray`, `darkGray`, `lightGray`, `color0`, `color1` или `transparent` (прозрачный цвет). Атрибуты `color0` (прозрачный цвет) и `color1` (непрозрачный цвет) используются в двухцветных изображениях:

```
black = QtCore.Qt.black
```

В пятом конструкторе указывается целочисленное значение цвета, а шестой конструктор создает новый объект на основе указанного в параметре.

Задать или получить значения в цветовой модели RGB (Red, Green, Blue — красный, зеленый, синий) позволяют следующие методы:

- `setNamedColor(<Строка>)` — задает название цвета в виде строки в форматах `"#RGB"`, `"#RRGGBB"`, `"#AARRGGBB"`, `"#RRRGGGBBB"`, `"#RRRRGGGGBBBB"`, `"Название цвета"` или `"transparent"` (для прозрачного цвета);

- `name()` — возвращает строковое представление цвета в формате `"#RRGGBB"`;
- `name(<Формат>)` — возвращает строковое представление цвета в заданном формате. В качестве формата указывается один из атрибутов класса `QColor`: `HexRgb` (`0`, формат `"#RRGGBB"`) или `HexArgb` (`1`, формат `"#AARRGGBB"`);
- `setRgb(<Красный>, <Зеленый>, <Синий>[, alpha=255])` — задает целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `alpha` задает степень прозрачности цвета: значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному;
- `setRgb(<Число>)` и `setRgba(<Число>)` — задают целочисленное значение цвета, второй метод — со степенью прозрачности;
- `setRed(<Красный>)`, `setGreen(<Зеленый>)`, `setBlue(<Синий>)` и `setAlpha(<Прозрачность>)` — задают значения отдельных составляющих цвета. В качестве параметров указываются числа от `0` до `255`;
- `fromRgb(<Красный>, <Зеленый>, <Синий>[, alpha=255])` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются числа от `0` до `255`:

  ```
  white = QtGui.QColor.fromRgb(255, 255, 255, 255)
  ```

  Метод является статическим;
- `fromRgb(<Число>)` и `fromRgba(<Число>)` — возвращают экземпляр класса `QColor` со значениями, соответствующими целым числам, которые указаны в параметрах:

  ```
  white = QtGui.QColor.fromRgba(4294967295)
  ```

  Метод является статическим;
- `getRgb()` — возвращает кортеж из четырех целочисленных значений `(<Красный>, <Зеленый>, <Синий>, <Прозрачность>)`;
- `red()`, `green()`, `blue()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета;
- `rgb()` и `rgba()` — возвращают целочисленное значение цвета;
- `setRgbF(<Красный>, <Зеленый>, <Синий>[, alpha=1.0])` — задает значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Необязательный параметр `alpha` задает степень прозрачности цвета: значение `0.0` соответствует прозрачному цвету, а значение `1.0` — полностью непрозрачному;
- `setRedF(<Красный>)`, `setGreenF(<Зеленый>)`, `setBlueF(<Синий>)` и `setAlphaF(<Прозрачность>)` — задают значения отдельных составляющих цвета. В качестве параметров указываются вещественные числа от `0.0` до `1.0`;
- `fromRgbF(<Красный>, <Зеленый>, <Синий>[, alpha=1.0])` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются вещественные числа от `0.0` до `1.0`:

  ```
  white = QtGui.QColor.fromRgbF(1.0, 1.0, 1.0, 1.0)
  ```

  Метод является статическим;
- `getRgbF()` — возвращает кортеж из четырех вещественных значений `(<Красный>, <Зеленый>, <Синий>, <Прозрачность>)`;

- `redF()`, `greenF()`, `blueF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета;
- `lighter([factor=150])` — если параметр имеет значение больше `100`, то возвращает новый объект с более светлым цветом, а если меньше `100` — то с более темным;
- `darker([factor=200])` — если параметр имеет значение больше `100`, то возвращает новый объект с более темным цветом, а если меньше `100` — то с более светлым.

Задать или получить значения в цветовой модели CMYK (Cyan, Magenta, Yellow, Key — голубой, пурпурный, желтый, «ключевой», он же черный) позволяют следующие методы:

- `setCmyk(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=255])` — задает целочисленные значения составляющих цвета модели CMYK. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `alpha` задает степень прозрачности цвета: значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному;
- `fromCmyk(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=255])` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются числа от `0` до `255`:

  ```
  white = QtGui.QColor.fromCmyk(0, 0, 0, 0, 255)
  ```

  Метод является статическим;
- `getCmyk()` — возвращает кортеж из пяти целочисленных значений `(<Голубой>, <Пурпурный>, <Желтый>, <Черный>, <Прозрачность>)`;
- `cyan()`, `magenta()`, `yellow()`, `black()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета;
- `setCmykF(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=1.0])` — задает значения составляющих цвета модели CMYK. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Необязательный параметр `alpha` задает степень прозрачности цвета: значение `0.0` соответствует прозрачному цвету, а значение `1.0` — полностью непрозрачному;
- `fromCmykF(<Голубой>, <Пурпурный>, <Желтый>, <Черный>[, alpha=1.0])` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются вещественные числа от `0.0` до `1.0`:

  ```
  white = QtGui.QColor.fromCmykF(0.0, 0.0, 0.0, 0.0, 1.0)
  ```

  Метод является статическим;
- `getCmykF()` — возвращает кортеж из пяти вещественных значений `(<Голубой>, <Пурпурный>, <Желтый>, <Черный>, <Прозрачность>)`;
- `cyanF()`, `magentaF()`, `yellowF()`, `blackF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета.

Задать или получить значения в цветовой модели HSV (Hue, Saturation, Value — оттенок, насыщенность, значение, она же яркость) позволяют следующие методы:

- `setHsv(<Оттенок>, <Насыщенность>, <Значение>[, alpha=255])` — задает целочисленные значения составляющих цвета модели HSV. В первом параметре указывается число от `0` до `359`, а в остальных параметрах — числа от `0` до `255`;

- fromHsv(<Оттенок>, <Насыщенность>, <Значение>[, alpha=255]) — возвращает экземпляр класса QColor с указанными значениями:

  ```
  white = QtGui.QColor.fromHsv(0, 0, 255, 255)
  ```

  Метод является статическим;
- getHsv() — возвращает кортеж из четырех целочисленных значений (<Оттенок>, <Насыщенность>, <Значение>, <Прозрачность>);
- hsvHue(), hsvSaturation(), value() и alpha() — возвращают целочисленные значения отдельных составляющих цвета;
- setHsvF(<Оттенок>, <Насыщенность>, <Значение>[, alpha=1.0]) — задает значения составляющих цвета модели HSV. В качестве параметров указываются вещественные числа от 0.0 до 1.0;
- fromHsvF(<Оттенок>, <Насыщенность>, <Значение>[, alpha=1.0]) — возвращает экземпляр класса QColor с указанными значениями. В качестве параметров указываются вещественные числа от 0.0 до 1.0:

  ```
  white = QtGui.QColor.fromHsvF(0.0, 0.0, 1.0, 1.0)
  ```

  Метод является статическим;
- getHsvF() — возвращает кортеж из четырех вещественных значений (<Оттенок>, <Насыщенность>, <Значение>, <Прозрачность>);
- hsvHueF(), hsvSaturationF(), valueF() и alphaF() — возвращают вещественные значения отдельных составляющих цвета.

Цветовая модель HSL (Hue, Saturation, Lightness — оттенок, насыщенность, яркость) отличается от модели HSV только последней составляющей. Описание этой модели и полный перечень методов для установки и получения значений вы найдете в соответствующей документации.

Для получения типа используемой модели и преобразования между моделями предназначены следующие методы:

- spec() — позволяет узнать тип используемой модели. Возвращает значение одного из следующих атрибутов, определенных в классе QColor: Invalid (0), Rgb (1), Hsv (2), Cmyk (3) или Hsl (4);
- convertTo(<Тип модели>) — преобразует тип модели. В качестве параметра указываются атрибуты, которые приведены в описании метода spec(). Метод возвращает новый объект. Пример преобразования:

  ```
  whiteHSV = QtGui.QColor.fromHsv(0, 0, 255)
  whiteRGB = whiteHSV.convertTo(QtGui.QColor.Rgb)
  ```

  Вместо метода convertTo() удобнее воспользоваться методами toRgb(), toCmyk(), toHsv() или toHsl(), которые возвращают новый объект:

  ```
  whiteHSV = QtGui.QColor.fromHsv(0, 0, 255)
  whiteRGB = whiteHSV.toRgb()
  ```

24.1.2. Класс *QPen*: перо

Класс QPen описывает виртуальное перо, с помощью которого производится рисование точек, линий и контуров фигур. Форматы конструктора класса:

```
<Объект> = QPen()
<Объект> = QPen(<QColor>)
<Объект> = QPen(<Стиль>)
<Объект> = QPen(<QBrush>, <Ширина>[, style=SolidLine][, cap=SquareCap][,
                join=BevelJoin])
<Объект> = QPen(<QPen>)
```

Первый конструктор создает перо черного цвета с настройками по умолчанию. Второй конструктор задает только цвет пера с помощью экземпляра класса QColor. Третий конструктор позволяет указать стиль линии — в качестве значения указываются следующие атрибуты класса QtCore.Qt:

- NoPen — 0 — линия не выводится;
- SolidLine — 1 — сплошная линия;
- DashLine — 2 — штриховая линия;
- DotLine — 3 — точечная линия;
- DashDotLine — 4 — штрих и точка, штрих и точка и т. д.;
- DashDotDotLine — 5 — штрих и две точки, штрих и две точки и т. д.;
- CustomDashLine — 6 — пользовательский стиль.

Четвертый конструктор позволяет задать все характеристики пера за один раз: в первом параметре указывается экземпляр класса QBrush или QColor, ширина линии передается во втором параметре, стиль линии — в необязательном параметре style, а необязательный параметр cap задает стиль концов линии, где в качестве значения указываются следующие атрибуты класса QtCore.Qt:

- FlatCap — 0 — квадратный конец линии. Длина линии не превышает указанных граничных точек;
- SquareCap — 16 — квадратный конец линии. Длина линии увеличивается с обоих концов на половину ширины линии;
- RoundCap — 32 — скругленные концы. Длина линии увеличивается с обоих концов на половину ширины линии.

Необязательный параметр join задает стиль перехода одной линии в другую — в качестве значения указываются следующие атрибуты класса QtCore.Qt:

- MiterJoin — 0 — линии соединяются под острым углом;
- BevelJoin — 64 — пространство между концами линий заполняется цветом линии;
- RoundJoin — 128 — скругленные углы;
- SvgMiterJoin — 256 — линии соединяются под острым углом, как определено в спецификации SVG 1.2 Tiny.

Последний конструктор создает новый объект на основе указанного в параметре.

Класс QPen поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qpen.html**):

- setColor(<QColor>) — задает цвет линии;
- setBrush(<QBrush>) — задает кисть;
- setWidth(<Ширина типа int>) и setWidthF(<Ширина типа float>) — задают ширину линии целым числом или числом с плавающей точкой соответственно;

- setStyle(<Стиль>) — задает стиль линии (см. значения параметра style в четвертом формате конструктора класса QPen);
- setCapStyle(<Стиль>) — задает стиль концов линии (см. значения параметра cap в четвертом формате конструктора класса QPen);
- setJoinStyle(<Стиль>) — задает стиль перехода одной линии в другую (см. значения параметра join в четвертом формате конструктора класса QPen).

24.1.3. Класс *QBrush*: кисть

Класс QBrush описывает виртуальную кисть, с помощью которой производится заливка фигур. Форматы конструктора класса:

```
<Объект> = QBrush()
<Объект> = QBrush(<QColor>[, style=SolidPattern])
<Объект> = QBrush(<Атрибут цвета>[, style=SolidPattern])
<Объект> = QBrush(<Стиль кисти>)
<Объект> = QBrush(<QGradient>)
<Объект> = QBrush(<QColor>, <QPixmap>)
<Объект> = QBrush(<Атрибут цвета>, <QPixmap>)
<Объект> = QBrush(<QPixmap>)
<Объект> = QBrush(<QImage>)
<Объект> = QBrush(<QBrush>)
```

Параметр <QColor> задает цвет кисти в виде экземпляра класса QColor, а параметр <Атрибут цвета> — в виде атрибута класса QtCore.Qt (например, black).

В параметрах <Стиль кисти> и style указываются атрибуты класса QtCore.Qt, задающие стиль кисти: NoBrush, SolidPattern, Dense1Pattern, Dense2Pattern, Dense3Pattern, Dense4Pattern, Dense5Pattern, Dense6Pattern, Dense7Pattern, CrossPattern и др. С помощью этого параметра можно сделать цвет сплошным (SolidPattern) или имеющим текстуру (например, атрибут CrossPattern задает текстуру в виде сетки).

Параметр <QGradient> позволяет установить градиентную заливку. В качестве значения указываются экземпляры классов, порожденных от класса QGradient: QLinearGradient (линейный градиент), QConicalGradient (конический градиент) или QRadialGradient (радиальный градиент). За подробной информацией по этим классам обращайтесь к соответствующей документации.

Параметры <QPixmap> и <QImage> предназначены для установки изображения в качестве текстуры, которой будут заливаться рисуемые фигуры.

Класс QBrush поддерживает следующие полезные для нас методы (полный их список приведен на странице **https://doc.qt.io/qt-5/qbrush.html**):

- setColor(<QColor>) и setColor(<Атрибут цвета>) — задают цвет кисти;
- setStyle(<Стиль>) — задает стиль кисти (см. значения параметра style в конструкторе класса QBrush);
- setTexture(<QPixmap>) — устанавливает растровое изображение в качестве текстуры. Можно указать экземпляр класса QPixmap или QBitmap;
- setTextureImage(<QImage>) — устанавливает изображение в качестве текстуры.

24.1.4. Класс *QLine*: линия

Класс `QLine` из модуля `QtCore` описывает координаты линии. Форматы конструктора класса:

```
<Объект> = QLine()
<Объект> = QLine(<QPoint>, <QPoint>)
<Объект> = QLine(<X1>, <Y1>, <X2>, <Y2>)
```

Первый конструктор создает линию, имеющую неустановленные местоположение и размеры. Во втором и третьем конструкторах указываются координаты начальной и конечной точек в виде экземпляров класса `QPoint` или целочисленных значений через запятую.

Класс `QLine` поддерживает следующие основные методы (полный их список приведен на странице **https://doc.qt.io/qt-5/qline.html**):

- `isNull()` — возвращает значение `True`, если начальная или конечная точка не установлены, и `False` — в противном случае;
- `setPoints(<QPoint>, <QPoint>)` — задает координаты начальной и конечной точек в виде экземпляров класса `QPoint`;
- `setLine(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты начальной и конечной точек в виде целочисленных значений через запятую;
- `setP1(<QPoint>)` — задает координаты начальной точки;
- `setP2(<QPoint>)` — задает координаты конечной точки;
- `p1()` — возвращает координаты (экземпляр класса `QPoint`) начальной точки;
- `p2()` — возвращает координаты (экземпляр класса `QPoint`) конечной точки;
- `center()` — возвращает координаты (экземпляр класса `QPoint`) центральной точки. Поддержка этого метода появилась в PyQt 5.8;
- `x1()`, `y1()`, `x2()` и `y2()` — возвращают значения отдельных составляющих координат начальной и конечной точек в виде целых чисел;
- `dx()` — возвращает горизонтальную составляющую вектора линии;
- `dy()` — возвращает вертикальную составляющую вектора линии.

> ***Примечание***
>
> Класс `QLine` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QLineF`.

24.1.5. Класс *QPolygon*: многоугольник

Класс `QPolygon` описывает координаты вершин многоугольника. Форматы конструктора класса:

```
<Объект> = QPolygon()
<Объект> = QPolygon(<Список с экземплярами класса QPoint>)
<Объект> = QPolygon(<QRect>[, closed=False])
<Объект> = QPolygon(<Количество вершин>)
<Объект> = QPolygon(<QPolygon>)
```

Первый конструктор создает пустой объект. Заполнить объект координатами вершин можно с помощью оператора <<. Вот пример добавления координат вершин треугольника:

```
polygon = QtGui.QPolygon()
polygon << QtCore.QPoint(20, 50) << QtCore.QPoint(280, 50)
polygon << QtCore.QPoint(150, 280)
```

Во втором конструкторе указывается список с экземплярами класса `QPoint`, которые задают координаты отдельных вершин:

```
polygon = QtGui.QPolygon([QtCore.QPoint(20, 50), QtCore.QPoint(280, 50),
                          QtCore.QPoint(150, 280)])
```

Третий конструктор создает многоугольник на основе экземпляра класса `QRect`. Если параметр `closed` имеет значение `False`, то будут созданы четыре вершины, а если значение `True` — то пять вершин.

В четвертом конструкторе можно указать количество вершин, а затем задать координаты путем присваивания значения по индексу:

```
polygon = QtGui.QPolygon(3)
polygon[0] = QtCore.QPoint(20, 50)
polygon[1] = QtCore.QPoint(280, 50)
polygon[2] = QtCore.QPoint(150, 280)
```

Пятый конструктор создает новый объект на основе другого объекта.

Класс `QPolygon` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qpolygon.html**):

- `setPoints()` — устанавливает координаты вершин. Ранее установленные значения удаляются. Форматы метода:

  ```
  setPoints(<Список с координатами>)
  setPoints(<X1>, <Y1>[, ..., <Xn>, <Yn>])
  ```

  Пример указания значений:

  ```
  polygon = QtGui.QPolygon()
  polygon.setPoints([20,50, 280,50, 150,280])
  ```

- `prepend(<QPoint>)` — добавляет новую вершину в начало объекта;
- `append(<QPoint>)` — добавляет новую вершину в конец объекта. Добавить вершину можно также с помощью операторов `<<` и `+=`;
- `insert(<Индекс>, <QPoint>)` — добавляет новую вершину в указанную позицию;
- `setPoint()` — задает координаты для вершины с указанным индексом. Форматы метода:

  ```
  setPoint(<Индекс>, <QPoint>)
  setPoint(<Индекс>, <X>, <Y>)
  ```

  Можно также задать координаты путем присваивания значения по индексу:

  ```
  polygon = QtGui.QPolygon(3)
  polygon.setPoint(0, QtCore.QPoint(20, 50))
  polygon.setPoint(1, 280, 50)
  polygon[2] = QtCore.QPoint(150, 280)
  ```

- `point(<Индекс>)` — возвращает экземпляр класса `QPoint` с координатами вершины, индекс которой указан в параметре. Получить значение можно также с помощью операции доступа по индексу:

```
polygon = QtGui.QPolygon([20,50, 280,50, 150,280])
print(polygon.point(0)) # PyQt5.QtCore.QPoint(20, 50)
print(polygon[1])       # PyQt5.QtCore.QPoint(280, 50)
```

- `remove(<Индекс>[, <Количество>])` — удаляет указанное количество вершин, начиная с индекса `<Индекс>`. Если второй параметр не указан, удаляется одна вершина. Удалить вершину можно также с помощью оператора `del` по индексу или срезу;
- `clear()` — удаляет все вершины;
- `size()` и `count([<QPoint>])` — возвращают количество вершин. Если в методе `count()` указан параметр, возвращается только количество вершин с указанными координатами. Получить количество вершин можно также с помощью функции `len()`;
- `isEmpty()` — возвращает значение `True`, если объект пустой (многоугольник не содержит ни одной вершины), и `False` — в противном случае;
- `boundingRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которую вписан многоугольник.

> ***Примечание***
>
> Класс `QPolygon` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QPolygonF`.

24.1.6. Класс *QFont*: шрифт

Класс `QFont` описывает характеристики шрифта. Форматы конструктора класса:

```
<Объект> = QFont()
<Объект> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1][, italic=False])
<Объект> = QFont(<QFont>)
```

Первый конструктор создает объект шрифта с настройками, используемыми приложением по умолчанию. Установить шрифт приложения по умолчанию позволяет статический метод `setFont()` класса `QApplication`.

Второй конструктор позволяет указать основные характеристики шрифта. В первом параметре указывается название шрифта или семейства в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: число от `0` до `99` или значение атрибута `Light` (`25`), `Normal` (`50`), `DemiBold` (`63`), `Bold` (`75`) или `Black` (`87`) класса `QFont`. Если в параметре `italic` указано значение `True`, шрифт будет курсивным.

Третий конструктор создает новый объект на основе другого объекта.

Класс `QFont` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти по адресу **https://doc.qt.io/qt-5/qfont.html**):

- `setFamily(<Название шрифта>)` — задает название шрифта или семейства шрифтов;
- `family()` — возвращает название шрифта;
- `setPointSize(<Размер типа int>)` и `setPointSizeF(<Размер типа float>)` — задают размер шрифта в пунктах;
- `pointSize()` — возвращает размер шрифта в пунктах в виде целого числа или значение `-1`, если размер шрифта был установлен в пикселах;

- `pointSizeF()` — возвращает размер шрифта в пунктах в виде вещественного числа или значение `-1`, если размер шрифта был установлен в пикселах;
- `setPixelSize(<Размер>)` — задает размер шрифта в пикселах;
- `pixelSize()` — возвращает размер шрифта в пикселах или `-1`, если размер шрифта был установлен в пунктах;
- `setWeight(<Жирность>)` — задает степень жирности шрифта (см. описание параметра `weight` во втором конструкторе класса `QFont`);
- `weight()` — возвращает степень жирности шрифта;
- `setBold(<Флаг>)` — если в качестве параметра указано значение `True`, то жирность шрифта устанавливается равной значению атрибута `Bold`, а если `False` — то равной значению атрибута `Normal` класса `QFont`;
- `bold()` — возвращает значение `True`, если степень жирности шрифта больше значения атрибута `Normal` класса `QFont`, и `False` — в противном случае;
- `setItalic(<Флаг>)` — если в качестве параметра указано значение `True`, шрифт будет курсивным, а если `False` — обычного начертания;
- `italic()` — возвращает значение `True`, если шрифт курсивный, и `False` — в противном случае;
- `setUnderline(<Флаг>)` — если в качестве параметра указано значение `True`, текст будет подчеркнутым, а если `False` — не подчеркнутым;
- `underline()` — возвращает значение `True`, если текст подчеркнут, и `False` — в противном случае;
- `setOverline(<Флаг>)` — если в качестве параметра указано значение `True`, над текстом будет выводиться черта;
- `overline()` — возвращает значение `True`, если над текстом будет выводиться черта, и `False` — в противном случае;
- `setStrikeOut(<Флаг>)` — если в качестве параметра указано значение `True`, текст будет зачеркнутым;
- `strikeOut()` — возвращает значение `True`, если текст будет зачеркнутым, и `False` — в противном случае.

Получить список всех доступных шрифтов позволяет метод `families()` класса `QFontDatabase`. Метод возвращает список строк. Отметим, что перед его вызовом следует создать экземпляр класса `QApplication`, в противном случае мы получим ошибку исполнения:

```
from PyQt5 import QtGui, QtWidgets
app = QtWidgets.QApplication(list())
fdb = QtGui.QFontDatabase()
print(fdb.families())
```

Чтобы получить список доступных стилей для указанного шрифта, следует воспользоваться методом `styles(<Название шрифта>)` класса `QFontDatabase`:

```
print(fdb.styles("Arial"))
# ['Обычный', 'Полужирный', 'Полужирный Курсив', 'Курсив']
```

Получить допустимые размеры для указанного стиля можно с помощью метода `smoothSizes(<Название шрифта>, <Стиль>)` класса `QFontDatabase`:

```
print(fdb.smoothSizes("Arial", "Обычный"))
# [6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72]
```

Очень часто необходимо произвести выравнивание выводимого текста внутри некоторой области. Чтобы это сделать, нужно знать размеры области, в которую вписан текст. Получить эти значения позволяют следующие методы класса `QFontMetrics`:

- `width(<Текст>[, length=-1])` — возвращает расстояние от начала текста `<Текст>` до позиции, в которой должен начаться другой текст. Параметр `length` позволяет ограничить количество символов;
- `height()` — возвращает высоту шрифта;
- `boundingRect(<Текст>)` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которую вписан текст.

Вот пример получения размеров области:

```
font = QtGui.QFont("Tahoma", 16)
fm = QtGui.QFontMetrics(font)
print(fm.width("Строка"))        # 67
print(fm.height())               # 25
print(fm.boundingRect("Строка")) # PyQt5.QtCore.QRect(0, -21, 65, 25)
```

Обратите внимание, что значения, возвращаемые методами `width()` и `QRect.width()`, различаются.

> **ПРИМЕЧАНИЕ**
>
> Класс `QFontMetrics` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QFontMetricsF`.

24.2. Класс *QPainter*

Класс `QPainter` содержит все необходимые средства, позволяющие выполнять рисование геометрических фигур и вывод текста на поверхности, которая реализуется классом `QPaintDevice`. Класс `QPaintDevice` наследуют классы `QWidget`, `QPicture`, `QPixmap`, `QImage`, `QPagedPaintDevice` и некоторые другие. Таким образом, мы можем рисовать на поверхности любого компонента, на изображении или на печатаемой странице. Форматы конструктора класса:

```
<Объект> = QPainter()
<Объект> = QPainter(<QPaintDevice>)
```

Первый конструктор создает объект, который не подключен ни к одному устройству. Чтобы подключиться к устройству и захватить контекст рисования, необходимо вызвать метод `begin(<QPaintDevice>)` и передать ему ссылку на экземпляр класса, являющегося наследником класса `QPaintDevice`. Метод возвращает значение `True`, если контекст успешно захвачен, и `False` — в противном случае. В один момент времени только один объект может рисовать на устройстве, поэтому после окончания рисования необходимо освободить контекст рисования с помощью метода `end()`. С учетом сказанного код, позволяющий рисовать на компоненте, будет выглядеть так:

```
def paintEvent(self, e):
    # Компонент, на котором выполняется рисование, передается в параметре self
    painter = QtGui.QPainter()
    painter.begin(self)
```

```
    # Здесь производим рисование на компоненте
    painter.end()
```

Второй конструктор принимает ссылку на экземпляр класса, являющегося наследником класса `QPaintDevice`, подключается к этому устройству и сразу захватывает контекст рисования. Контекст рисования автоматически освобождается внутри деструктора класса `QPainter` при уничтожении объекта. Так как объект автоматически уничтожается при выходе из метода `paintEvent()`, то метод `end()` можно и не вызывать.

Вот пример рисования на компоненте:

```
def paintEvent(self, e):
    painter = QtGui.QPainter(self)
    # Здесь производим рисование на компоненте
```

Проверить успешность захвата контекста рисования можно с помощью метода `isActive()`: он возвращает значение `True`, если контекст захвачен, и `False` — в противном случае.

24.2.1. Рисование линий и фигур

После захвата контекста рисования следует установить перо и кисть. С помощью пера производится рисование точек, линий и контуров фигур, а с помощью кисти — заполнение фона фигур. Установить перо позволяет метод `setPen()` класса `QPainter`. Форматы метода:

```
setPen(<QPen>)
setPen(<QColor>)
setPen(<Стиль пера>)
```

Для установки кисти предназначен метод `setBrush()`. Форматы метода:

```
setBrush(<QBrush>)
setBrush(<Стиль кисти>)
```

Устанавливать перо или кисть необходимо перед каждой операцией рисования, требующей изменения цвета или стиля. Если перо или кисть не установлены, будут использоваться объекты с настройками по умолчанию. После установки пера и кисти можно приступать к рисованию точек, линий, фигур, текста и др.

Для рисования точек, линий и фигур класс `QPainter` предоставляет следующие наиболее часто употребляемые методы (полный их список приведен на странице **https://doc.qt.io/qt-5/qpainter.html**):

- `drawPoint()` — рисует точку. Форматы метода:

  ```
  drawPoint(<X>, <Y>)
  drawPoint(<QPoint>)
  drawPoint(<QPointF>)
  ```

- `drawPoints()` — рисует несколько точек. Форматы метода:

  ```
  drawPoints(<QPoint 1>[, ..., <QPoint N>])
  drawPoints(<QPointF 1>[, ..., <QPointF N>])
  drawPoints(<QPolygon>)
  drawPoints(<QPolygonF>)
  ```

- `drawLine()` — рисует линию. Форматы метода:

  ```
  drawLine(<QLine>)
  drawLine(<QLineF>)
  ```

```
drawLine(<QPoint>, <QPoint>)
drawLine(<QPointF>, <QPointF>)
drawLine(<X1>, <Y1>, <X2>, <Y2>)
```

- drawLines() — рисует несколько отдельных линий. Форматы метода:

```
drawLines(<QLine 1>[, ..., <QLine N>])
drawLines(<QLineF 1>[, ..., <QLineF N>])
drawLines(<Список с экземплярами класса QLineF>)
drawLines(<QPoint 1>[, ..., <QPoint N>])
drawLines(<QPointF 1>[, ..., <QPointF N>])
```

- drawPolyline() — рисует несколько линий, которые соединяют указанные точки. Первая и последняя точки не соединяются. Форматы метода:

```
drawPolyline(<QPoint 1>[, ..., <QPoint N>])
drawPolyline(<QPointF 1>[, ..., <QPointF N>])
drawPolyline(<QPolygon>)
drawPolyline(<QPolygonF>)
```

- drawRect() — рисует прямоугольник с границей и заливкой. Чтобы убрать границу, следует использовать перо со стилем NoPen, а чтобы убрать заливку — кисть со стилем NoBrush. Форматы метода:

```
drawRect(<X>, <Y>, <Ширина>, <Высота>)
drawRect(<QRect>)
drawRect(<QRectF>)
```

- fillRect() — рисует прямоугольник с заливкой без границы. Форматы метода:

```
fillRect(<X>, <Y>, <Ширина>, <Высота>, <Заливка>)
fillRect(<QRect>, <Заливка>)
fillRect(<QRectF>, <Заливка>)
```

  <Заливка> может быть задана экземплярами классов <QColor>, <QBrush> в виде стиля кисти или атрибута цвета;

- drawRoundedRect() — рисует прямоугольник с границей, заливкой и скругленными краями. Форматы метода:

```
drawRoundedRect(<X>, <Y>, <Ширина>, <Высота>,
                <Скругление по горизонтали>, <Скругление по вертикали>[,
                mode = Qt::AbsoluteSize])
drawRoundedRect(<QRect>, <Скругление по горизонтали>,
                <Скругление по вертикали>[, mode = Qt::AbsoluteSize])
drawRoundedRect(<QRectF>, <Скругление по горизонтали>,
                <Скругление по вертикали>[, mode = Qt::AbsoluteSize])
```

  Параметры <Скругление по горизонтали> и <Скругление по вертикали> задают радиусы скругления углов по горизонтали и вертикали. Необязательный параметр mode указывает, в каких единицах измеряются радиусы скругления углов, и задается одним из следующих атрибутов класса QtCore.Qt:

  - AbsoluteSize — 0 — радиусы указываются в пикселах;
  - RelativeSize — 1 — радиусы указываются в процентах от соответствующего размера рисуемого прямоугольника;

♦ `drawPolygon()` — рисует многоугольник с границей и заливкой. Форматы метода:

```
drawPolygon(<QPoint 1>[, ..., <QPoint N>])
drawPolygon(<QPointF 1>[, ..., <QPointF N>])
drawPolygon(<QPolygon>[, fillRule=OddEvenFill])
drawPolygon(<QPolygonF>[, fillRule=OddEvenFill])
```

Необязательный параметр `fillRule` задает алгоритм определения, находится ли какая-либо точка внутри нарисованного прямоугольника или вне его. В качестве его значения указывается атрибут `OddEvenFill` (`0`) или `WindingFill` (`1`) класса `QtCore.Qt`;

♦ `drawEllipse()` — рисует эллипс с границей и заливкой. Форматы метода:

```
drawEllipse(<X>, <Y>, <Ширина>, <Высота>)
drawEllipse(<QRect>)
drawEllipse(<QRectF>)
drawEllipse(<QPoint>, <int rX>, <int rY>)
drawEllipse(<QPointF>, <float rX>, <float rY>)
```

В первых трех форматах указываются координаты и размеры прямоугольника, в который необходимо вписать эллипс. В двух последних форматах первый параметр задает координаты центра, параметр `rX` — радиус по оси `X`, а параметр `rY` — радиус по оси `Y`;

♦ `drawArc()` — рисует дугу. Форматы метода:

```
drawArc(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>, <Угол>)
drawArc(<QRect>, <Начальный угол>, <Угол>)
drawArc(<QRectF>, <Начальный угол>, <Угол>)
```

Следует учитывать, что значения углов задаются в значениях $^1/_{16}$°. Полный круг эквивалентен значению $5760 = 16 \times 360$. Нулевой угол находится в позиции «трех часов». Положительные значения углов отсчитываются против часовой стрелки, а отрицательные — по часовой стрелке;

♦ `drawChord()` — рисует замкнутую дугу. Аналогичен методу `drawArc()`, но соединяет крайние точки дуги прямой линией. Форматы метода:

```
drawChord(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>, <Угол>)
drawChord(<QRect>, <Начальный угол>, <Угол>)
drawChord(<QRectF>, <Начальный угол>, <Угол>)
```

♦ `drawPie()` — рисует замкнутый сектор. Аналогичен методу `drawArc()`, но соединяет крайние точки дуги с центром окружности. Форматы метода:

```
drawPie(<X>, <Y>, <Ширина>, <Высота>, <Начальный угол>,  <Угол>)
drawPie(<QRect>, <Начальный угол>, <Угол>)
drawPie(<QRectF>, <Начальный угол>, <Угол>)
```

При выводе некоторых фигур (например, эллипса) контур может отображаться в виде «лесенки». Чтобы сгладить контуры фигур, следует вызвать метод `setRenderHint()` и передать ему в качестве единственного параметра атрибут `Antialiasing` класса `QPainter`:

```
painter.setRenderHint(QtGui.QPainter.Antialiasing)
```

Если требуется отключить сглаживание, следует вызвать тот же метод, но передать ему вторым параметром значение `False`:

```
painter.setRenderHint(QtGui.QPainter.Antialiasing, False)
```

24.2.2. Вывод текста

Вывести текст позволяет метод `drawText()` класса `QPainter`. Форматы метода:

```
drawText(<X>, <Y>, <Текст>)
drawText(<QPoint>, <Текст>)
drawText(<QPointF>, <Текст>)
drawText(<X>, <Y>, <Ширина>, <Высота>, <Флаги>, <Текст>)
drawText(<QRect>, <Флаги>, <Текст>)
drawText(<QRectF>, <Флаги>, <Текст>)
drawText(<QRectF>, <Текст>[, option=QTextOption()])
```

Первые три формата метода выводят текст, начиная с указанных координат.

Следующие три формата выводят текст в указанную прямоугольную область. При этом текст, который не помещается в эту область, будет обрезан, если не указан флаг `TextDontClip`. Методы возвращают экземпляр класса `QRect` (`QRectF` для шестого формата) с координатами и размерами прямоугольника, в который вписан текст. В параметре `<Флаги>` через оператор `|` указываются атрибуты `AlignLeft`, `AlignRight`, `AlignHCenter`, `AlignTop`, `AlignBottom`, `AlignVCenter` или `AlignCenter` класса `QtCore.Qt`, задающие выравнивание текста внутри прямоугольной области, а также следующие атрибуты:

- `TextSingleLine` — все пробельные символы (табуляция, возвраты каретки и переводы строки) трактуются как пробелы, и текст выводится в одну строку;
- `TextDontClip` — часть текста, вышедшая за пределы указанной прямоугольной области, не будет обрезаться;
- `TextExpandTabs` — символы табуляции будут обрабатываться;
- `TextShowMnemonic` — символ, перед которым указан знак `&`, будет подчеркнут. Чтобы вывести символ `&`, его необходимо удвоить;
- `TextWordWrap` — если текст не помещается на одной строке, будет произведен перенос слова без его разрыва;
- `TextWrapAnywhere` — перенос строки может быть выполнен внутри слова;
- `TextHideMnemonic` — то же самое, что и `TextShowMnemonic`, но символ не подчеркивается;
- `TextDontPrint` — текст не будет напечатан;
- `TextIncludeTrailingSpaces` — размеры текста будут возвращаться с учетом начальных и конечных пробелов, если таковые есть в тексте;
- `TextJustificationForced` — задает выравнивание по ширине для последней строки текста.

Седьмой формат метода `drawText()` также выводит текст в указанную прямоугольную область, но выравнивание текста и другие опции задаются с помощью экземпляра класса `QTextOption`. Например, с помощью этого класса можно отобразить непечатаемые символы (символ пробела, табуляцию и др.).

Получить координаты и размеры прямоугольника, в который вписывается текст, позволяет метод `boundingRect()`. Форматы метода:

```
boundingRect(<X>, <Y>, <Ширина>, <Высота>, <Флаги>, <Текст>)
boundingRect(<QRect>, <Флаги>, <Текст>)
boundingRect(<QRectF>, <Флаги>, <Текст>)
boundingRect(<QRectF>, <Текст>[, option=QTextOption()])
```

Первые два формата метода `boundingRect()` возвращают экземпляр класса `QRect`, а последние два — экземпляр класса `QRectF`.

При выводе текста линии букв могут отображаться в виде «лесенки». Чтобы сгладить контуры, следует вызвать метод `setRenderHint()` и передать ему атрибут `TextAntialiasing` класса `QPainter`:

```
painter.setRenderHint(QtGui.QPainter.TextAntialiasing)
```

Если требуется отключить сглаживание, следует вызвать тот же метод, но передать ему вторым параметром значение `False`:

```
painter.setRenderHint(QtGui.QPainter.TextAntialiasing, False)
```

24.2.3. Вывод изображения

Для вывода растровых изображений предназначены методы `drawPixmap()` и `drawImage()` класса `QPainter`. Метод `drawPixmap()` предназначен для вывода изображений, хранимых в экземпляре класса `QPixmap`. Форматы метода:

```
drawPixmap(<X>, <Y>, <QPixmap>)
drawPixmap(<QPoint>, <QPixmap>)
drawPixmap(<QPointF>, <QPixmap>)
drawPixmap(<X>, <Y>, <Ширина>, <Высота>, <QPixmap>)
drawPixmap(<QRect>, <QPixmap>)
drawPixmap(<X1>, <Y1>, <QPixmap>, <X2>, <Y2>, <Ширина2>, <Высота2>)
drawPixmap(<QPoint>, <QPixmap>, <QRect>)
drawPixmap(<QPointF>, <QPixmap>, <QRectF>)
drawPixmap(<X1>, <Y1>, <Ширина1>, <Высота1>, <QPixmap>,
           <X2>, <Y2>, <Ширина2>, <Высота2>)
drawPixmap(<QRect>, <QPixmap>, <QRect>)
drawPixmap(<QRectF>, <QPixmap>, <QRectF>)
```

Первые три формата задают координаты, в которые будет установлен левый верхний угол выводимого изображения, и экземпляр класса `QPixmap`:

```
pixmap = QtGui.QPixmap("foto.jpg")
painter.drawPixmap(3, 3, pixmap)
```

Четвертый и пятый форматы позволяют ограничить вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Шестой, седьмой и восьмой форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QPixmap` в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`.

Последние три формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QPixmap` в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`. Если размеры области не совпадают с размерами фрагмента изображения, производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Метод `drawImage()` предназначен для вывода изображений, хранимых в экземплярах класса `QImage`. Форматы метода:

```
drawImage(<QPoint>, <QImage>)
drawImage(<QPointF>, <QImage>)
drawImage(<QRect>, <QImage>)
drawImage(<QRectF>, <QImage>)
drawImage(<X1>, <Y1>, <QImage>[, sx=0][, sy=0][, sw=-1][, sh=-1][,
          flags=AutoColor])
drawImage(<QPoint>, <QImage>, <QRect>[, flags=AutoColor])
drawImage(<QPointF>, <QImage>, <QRectF>[, flags=AutoColor])
drawImage(<QRect>, <QImage>, <QRect>[, flags=AutoColor])
drawImage(<QRectF>, <QImage>, <QRectF>[, flags=AutoColor])
```

Первые два формата, а также пятый формат со значениями по умолчанию задают координаты, по которым будет находиться левый верхний угол выводимого изображения, и экземпляр класса `QImage`:

```
img = QtGui.QImage("foto.jpg")
painter.drawImage(3, 3, img)
```

Третий и четвертый форматы позволяют ограничить вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Пятый, шестой и седьмой форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QImage` в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`.

Последние два формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QImage` в виде экземпляров классов `QRect` или `QRectF`. Если размеры области не совпадают с размерами фрагмента изображения, производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Необязательный параметр `flags` задает цветовые преобразования, которые будут выполнены при выводе изображения (фактически — при неявном преобразовании экземпляра класса `QImage` в экземпляр класса `QPixmap`, которое обязательно выполняется перед выводом). Они указываются в виде атрибутов класса `QtCore.Qt`, приведенных на странице **https://doc.qt.io/qt-5/qt.html#ImageConversionFlag-enum**. В большинстве случаев имеет смысл использовать заданное по умолчанию значение `AutoColor` этого параметра.

24.2.4. Преобразование систем координат

Существуют две системы координат: физическая (viewport, система координат устройства) и логическая (window). При рисовании координаты из логической системы координат преобразуются в систему координат устройства. По умолчанию эти две системы координат совпадают.

В некоторых случаях возникает необходимость изменить координаты. Выполнить изменение физической системы координат позволяет метод `setViewport(<X>, <Y>, <Ширина>,`

<Высота>) или setViewport(<QRect>), а получить текущие значения — метод viewport(). Выполнить изменение логической системы координат позволяет метод setWindow(<X>, <Y>, <Ширина>, <Высота>) или setWindow(<QRect>), а получить текущие значения — метод window() класса QPainter.

Произвести дополнительную трансформацию системы координат позволяют следующие методы того же класса QPainter:

- translate() — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу. Положительная ось X направлена вправо, а положительная ось Y — вниз. Форматы метода:

```
translate(<X>, <Y>)
translate(<QPoint>)
translate(<QPointF>)
```

- rotate(<Угол>) — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;
- scale(<По оси X>, <По оси Y>) — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- shear(<По горизонтали>, <По вертикали>) — сдвигает систему координат. В качестве значений указываются вещественные числа.

Все указанные трансформации влияют на последующие операции рисования и не изменяют ранее нарисованные фигуры. Чтобы после трансформации восстановить систему координат, следует предварительно сохранить состояние в стеке с помощью метода save(), а после окончания рисования вызвать метод restore():

```
painter.save()      # Сохраняем состояние
# Трансформируем и рисуем
painter.restore()   # Восстанавливаем состояние
```

Несколько трансформаций можно произвести последовательно друг за другом. При этом надо учитывать, что порядок следования трансформаций имеет значение.

Если одна и та же последовательность трансформаций выполняется несколько раз, то ее можно сохранить в экземпляре класса QTransform, а затем установить с помощью метода setTransform():

```
transform = QtGui.QTransform()
transform.translate(105, 105)
transform.rotate(45.0)
painter.setTransform(transform)
painter.fillRect(-25, -25, 50, 50, QtCore.Qt.green)
```

24.2.5. Сохранение команд рисования в файл

Класс QPicture выполняет роль устройства для рисования с возможностью сохранения команд рисования в файл специального формата и последующего вывода его на экран. Иерархия наследования:

```
QPaintDevice — QPicture
```

Форматы конструктора класса:

```
<Объект> = QPicture([formatVersion=-1])
<Объект> = QPicture(<QPicture>)
```

Первый конструктор создает пустой рисунок. Необязательный параметр `formatVersion` задает версию формата. Если параметр не указан, то используется формат, принятый в текущей версии PyQt. Второй конструктор создает копию рисунка.

Для сохранения и загрузки рисунка предназначены следующие методы:

- `save(<Путь к файлу>)` — сохраняет рисунок в файл. Возвращает значение `True`, если рисунок успешно сохранен, и `False` — в противном случае;
- `load(<Путь к файлу>)` — загружает рисунок из файла. Возвращает значение `True`, если рисунок успешно загружен, и `False` — в противном случае.

Для вывода загруженного рисунка на устройство рисования предназначен метод `drawPicture()` класса `QPainter`. Форматы метода:

```
drawPicture(<X>, <Y>, <QPicture>)
drawPicture(<QPoint>, <QPicture>)
drawPicture(<QPointF>, <QPicture>)
```

Пример сохранения рисунка:

```
painter = QtGui.QPainter()
pic = QtGui.QPicture()
painter.begin(pic)
# Здесь что-то рисуем
painter.end()
pic.save("pic.dat")
```

Пример вывода загруженного рисунка на поверхность компонента:

```
def paintEvent(self, e):
    painter = QtGui.QPainter(self)
    pic = QtGui.QPicture()
    pic.load("pic.dat")
    painter.drawPicture(0, 0, pic)
```

24.3. Работа с изображениями

Библиотека PyQt включает несколько классов, позволяющих работать с растровыми изображениями в контекстно-зависимом (классы `QPixmap` и `QBitmap`) и контекстно-независимом (класс `QImage`) представлениях.

Получить список форматов, которые можно загрузить, позволяет статический метод `supportedImageFormats()` класса `QImageReader`, возвращающий список с экземплярами класса `QByteArray`. Получим список поддерживаемых форматов для чтения:

```
for i in QtGui.QImageReader.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Результат выполнения:

```
BMP CUR GIF ICNS ICO JPEG JPG PBM PGM PNG PPM SVG SVGZ TGA TIF TIFF WBMP WEBP XBM
XPM
```

Получить список форматов, в которых можно сохранить изображение, позволяет статический метод `supportedImageFormats()` класса `QImageWriter`, возвращающий список с экземплярами класса `QByteArray`. Получим список поддерживаемых форматов для записи:

```
for i in QtGui.QImageWriter.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Результат выполнения:

```
BMP CUR ICNS ICO JPEG JPG PBM PGM PNG PPM TIF TIFF WBMP WEBP XBM XPM
```

Обратите внимание, что мы можем загрузить изображение в формате `GIF`, но не имеем возможности сохранить изображение в этом формате, поскольку алгоритм сжатия, используемый в нем, защищен патентом.

24.3.1. Класс *QPixmap*

Класс `QPixmap` предназначен для работы с изображениями в контекстно-зависимом представлении. Данные хранятся в виде, позволяющем отображать изображение на экране наиболее эффективным способом, поэтому класс `QPixmap` часто используется в качестве буфера для предварительного рисования графики перед выводом ее на экран. Иерархия наследования:

```
QPaintDevice — QPixmap
```

Поскольку класс `QPixmap` наследует класс `QPaintDevice`, мы можем использовать его как поверхность для рисования. Вывести изображение позволяет метод `drawPixmap()` класса `QPainter` (см. *разд. 24.2.3*).

Форматы конструктора класса:

```
<Объект> = QPixmap()
<Объект> = QPixmap(<Ширина>, <Высота>)
<Объект> = QPixmap(<QSize>)
<Объект> = QPixmap(<Путь к файлу>[, format=None][, flags=AutoColor])
<Объект> = QPixmap(<QPixmap>)
```

Первый конструктор создает пустой объект изображения. Второй и третий конструкторы позволяют указать размеры изображения: если размеры равны нулю, то будет создан пустой объект. Четвертый конструктор предназначен для загрузки изображения из файла. Во втором параметре указывается тип изображения в виде строки (например, `"PNG"`) — если он не указан, то формат будет определен по расширению загружаемого файла. Пятый конструктор создает копию изображения.

Класс `QPixmap` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qpixmap.html**):

- `isNull()` — возвращает значение `True`, если объект является пустым, и `False` — в противном случае;
- `load(<Путь к файлу>[, format=None][, flags=AutoColor])` — загружает изображение из файла. В первом параметре указывается абсолютный или относительный путь к файлу. Во втором параметре можно задать формат файла в виде строки — если он не указан, формат определяется по расширению файла. Необязательный параметр `flags` задает тип преобразования цветов. Метод возвращает значение `True`, если изображение успешно загружено, и `False` — в противном случае;

- `loadFromData(<QByteArray>[, format=None][, flags=AutoColor])` — загружает изображение из экземпляра класса `QByteArray`. В первом параметре можно указать данные, имеющие тип `bytes`. Метод возвращает значение `True`, если изображение успешно загружено, и `False` — в противном случае;
- `save(<Путь к файлу>[, format=None][, quality=-1])` — сохраняет изображение в файл. В первом параметре указывается абсолютный или относительный путь к файлу. Во втором параметре можно задать формат файла в виде строки — если он не указан, формат будет определен по расширению файла. Необязательный параметр `quality` позволяет задать качество изображения. Можно передать значение в диапазоне от `0` до `100`, значение `-1` указывает качество по умолчанию. Метод возвращает значение `True`, если изображение успешно сохранено, и `False` — в противном случае;
- `convertFromImage(<QImage>[, flags=AutoColor])` — преобразует экземпляр класса `QImage` в экземпляр класса `QPixmap`. Метод возвращает значение `True`, если изображение успешно преобразовано, и `False` — в противном случае;
- `fromImage(<QImage>[, flags=AutoColor])` — преобразует экземпляр класса `QImage` в экземпляр класса `QPixmap`, который и возвращает. Метод является статическим;
- `toImage()` — преобразует экземпляр класса `QPixmap` в экземпляр класса `QImage` и возвращает его;
- `fill([color=white])` — производит заливку изображения указанным цветом;
- `width()` — возвращает ширину изображения;
- `height()` — возвращает высоту изображения;
- `size()` — возвращает экземпляр класса `QSize` с размерами изображения;
- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, ограничивающей изображение;
- `depth()` — возвращает глубину цвета;
- `isQBitmap()` — возвращает значение `True`, если глубина цвета равна одному биту (т. е. это монохромное изображение), и `False` — в противном случае;
- `createMaskFromColor(<QColor[, mode=MaskInColor]>)` — создает на основе изображения маску в виде экземпляра класса `QBitmap` и возвращает ее. Первый параметр задает цвет — области, закрашенные этим цветом, будут на маске либо прозрачными, либо непрозрачными. Необязательный параметр `mode` задает режим создания маски в виде следующих атрибутов класса `QtCore.Qt`:
  - `MaskInColor` — `0` — области, закрашенные указанным цветом, будут прозрачными;
  - `MaskOutColor` — `1` — области, закрашенные указанным цветом, будут непрозрачными;
- `setMask(<QBitmap>)` — устанавливает маску;
- `mask()` — возвращает экземпляр класса `QBitmap` с маской изображения;
- `copy()` — возвращает экземпляр класса `QPixmap` с фрагментом изображения. Если параметр `rect` не указан, изображение копируется полностью. Форматы метода:

  ```
  copy([rect=QRect()])
  copy(<X>, <Y>, <Ширина>, <Высота>)
  ```
- `scaled()` — изменяет размер изображения и возвращает результат в виде экземпляра класса `QPixmap`. Исходное изображение не изменяется. Форматы метода:

```
scaled(<Ширина>, <Высота>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
scaled(<QSize>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
```

В необязательном параметре `aspectRatioMode` могут быть указаны следующие атрибуты класса `QtCore.Qt`:

- `IgnoreAspectRatio` — `0` — изменяет размеры без сохранения пропорций сторон;
- `KeepAspectRatio` — `1` — изменяет размеры с сохранением пропорций сторон. При этом часть области нового изображения может оказаться незаполненной;
- `KeepAspectRatioByExpanding` — `2` — изменяет размеры с сохранением пропорций сторон. При этом часть нового изображения может выйти за пределы его области.

В необязательном параметре `transformMode` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `FastTransformation` — `0` — сглаживание выключено;
- `SmoothTransformation` — `1` — сглаживание включено;

♦ `scaledToWidth(<Ширина>[, mode=FastTransformation])` — изменяет ширину изображения и возвращает результат в виде экземпляра класса `QPixmap`. Высота изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;

♦ `scaledToHeight(<Высота>[, mode=FastTransformation])` — изменяет высоту изображения и возвращает результат в виде экземпляра класса `QPixmap`. Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;

♦ `transformed(<QTransform>[, mode=FastTransformation])` — производит трансформацию изображения (например, поворот) и возвращает результат в виде экземпляра класса `QPixmap`. Исходное изображение не изменяется. Трансформация задается с помощью экземпляра класса `QTransform`. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;

♦ `swap(<QPixmap>)` — заменяет текущее изображение указанным в параметре;

♦ `hasAlpha()` — возвращает `True`, если изображение имеет прозрачные области, и `False` — в противном случае;

♦ `hasAlphaChannel()` — возвращает `True`, если формат изображения поддерживает прозрачность, и `False` — в противном случае.

24.3.2. Класс *QBitmap*

Класс `QBitmap` предназначен для работы в контекстно-зависимом представлении с монохромными изображениями, имеющими глубину цвета, равную одному биту. Наиболее часто класс `QBitmap` используется для создания масок изображений. Иерархия наследования:

```
QPaintDevice — QPixmap — QBitmap
```

Поскольку класс `QBitmap` наследует класс `QPaintDevice`, мы можем использовать его как поверхность для рисования. Цвет пера и кисти задается атрибутами `color0` (прозрачный цвет) и `color1` (непрозрачный цвет) класса `QtCore.Qt`. Вывести изображение позволяет метод `drawPixmap()` класса `QPainter` (см. *разд. 24.2.3*).

Форматы конструктора класса:

```
<Объект> = QBitmap()
<Объект> = QBitmap(<Ширина>, <Высота>)
<Объект> = QBitmap(<QSize>)
<Объект> = QBitmap(<Путь к файлу>[, format=None])
<Объект> = QBitmap(<QPixmap>)
<Объект> = QBitmap(<QBitmap>)
```

Класс QBitmap наследует все методы класса QPixmap и определяет следующие дополнительные методы (здесь приведены только нас интересующие — полный их список можно найти на странице **https://doc.qt.io/qt-5/qbitmap.html**):

- fromImage(<QImage>[, flags=AutoColor]) — преобразует экземпляр класса QImage в экземпляр класса QBitmap и возвращает его. Метод является статическим;
- transformed(<QTransform>) — производит трансформацию изображения (например, поворот) и возвращает экземпляр класса QBitmap. Исходное изображение не изменяется. Трансформация задается экземпляром класса QTransform;
- clear() — очищает изображение, устанавливая все биты изображения в значение color0.

24.3.3. Класс *QImage*

Класс QImage предназначен для работы с изображениями в контекстно-независимом представлении. Иерархия наследования:

```
QPaintDevice — QImage
```

Поскольку класс QImage наследует класс QPaintDevice, мы можем использовать его как поверхность для рисования. Однако следует учитывать, что не на всех форматах изображения можно рисовать, — для рисования лучше использовать изображение формата Format_ARGB32_Premultiplied. Вывести изображение позволяет метод drawImage() класса QPainter (см. *разд. 24.2.3*).

Форматы конструктора класса:

```
<Объект> = QImage()
<Объект> = QImage(<Ширина>, <Высота>, <Формат>)
<Объект> = QImage(<QSize>, <Формат>)
<Объект> = QImage(<Путь к файлу>[, <Тип изображения>])
<Объект> = QImage(<QImage>)
```

Первый конструктор создает пустой объект изображения. Второй и третий конструкторы позволяют указать размеры изображения — если таковые равны нулю, будет создан пустой объект. Четвертый конструктор предназначен для загрузки изображения из файла. Во втором параметре указывается тип изображения в виде строки — если он не указан, формат будет определен по расширению загружаемого файла. Пятый конструктор создает копию изображения.

В параметре <Формат> можно указать следующие атрибуты класса QImage (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qimage.html#Format-enum**):

- Format_Invalid — 0 — неверный формат;
- Format_Mono — 1 — глубина цвета 1 бит;

- ♦ Format_MonoLSB — 2 — глубина цвета 1 бит;
- ♦ Format_Indexed8 — 3 — глубина цвета 8 битов;
- ♦ Format_RGB32 — 4 — RGB без альфа-канала, глубина цвета 32 бита;
- ♦ Format_ARGB32 — 5 — RGB с альфа-каналом, глубина цвета 32 бита;
- ♦ Format_ARGB32_Premultiplied — 6 — RGB с альфа-каналом, глубина цвета 32 бита. Этот формат лучше использовать для рисования.

Класс QImage поддерживает большое количество методов, из которых мы рассмотрим лишь основные (полный их список приведен на странице **https://doc.qt.io/qt-5/qimage.html**):

- ♦ isNull() — возвращает значение True, если объект является пустым, и False — в противном случае;
- ♦ load(<Путь к файлу>[, format=None]) — загружает изображение из файла. В первом параметре задается абсолютный или относительный путь к файлу. Во втором параметре указывается формат файла в виде строки — если он не указан, формат определяется по расширению файла. Метод возвращает значение True, если изображение успешно загружено, и False — в противном случае;
- ♦ loadFromData(<QByteArray>[, format=None]) — загружает изображение из экземпляра класса QByteArray. В первом параметре можно указать данные, имеющие тип bytes. Во втором параметре указывается тип изображения в виде строки (например, "PNG"). Метод возвращает значение True, если изображение успешно загружено, и False — в противном случае;
- ♦ fromData(<QByteArray>[, format=None]) — загружает изображение из экземпляра класса QByteArray. В первом параметре можно указать данные, имеющие тип bytes. Во втором параметре указывается тип изображения в виде строки (например, "PNG"). Метод возвращает экземпляр класса QImage. Метод является статическим;
- ♦ save(<Путь к файлу>[, format=None][, quality=-1]) — сохраняет изображение в файл. В первом параметре указывается абсолютный или относительный путь к файлу. Во втором параметре можно задать формат файла в виде строки — если он не указан, формат определяется по расширению файла. Необязательный параметр quality позволяет задать качество изображения. Можно передать значение в диапазоне от 0 до 100; значение -1 указывает качество по умолчанию. Метод возвращает значение True, если изображение успешно сохранено, и False — в противном случае;
- ♦ fill(<Цвет>) — производит заливку изображения определенным цветом. В качестве параметра указывается атрибут цвета (например, QtCore.Qt.white), экземпляр класса QColor или целочисленное значение цвета, получить которое позволяют методы rgb() и rgba() класса QColor:

```
img.fill(QtCore.Qt.red)
img.fill(QtGui.QColor("#ff0000"))
img.fill(QtGui.QColor("#ff0000").rgb())
```

- ♦ width() — возвращает ширину изображения;
- ♦ height() — возвращает высоту изображения;
- ♦ size() — возвращает экземпляр класса QSize с размерами изображения;
- ♦ rect() — возвращает экземпляр класса QRect с координатами и размерами прямоугольной области, ограничивающей изображение;

♦ `depth()` — возвращает глубину цвета;

♦ `format()` — возвращает формат изображения (см. значения параметра `<Формат>` в конструкторе класса `QImage`);

♦ `setPixel()` — задает цвет указанного пиксела. Форматы метода:

```
setPixel(<X>, <Y>, <Индекс или цвет>)
setPixel(<QPoint>, <Индекс или цвет>)
```

В параметре `<Индекс или цвет>` для 8-битных изображений задается индекс цвета в палитре, а для 32-битных — целочисленное значение цвета, получить которое позволяют методы `rgb()` и `rgba()` класса `QColor`;

♦ `pixel()` — возвращает целочисленное значение цвета указанного пиксела. Это значение можно передать конструктору класса `QColor`, а затем получить значения различных составляющих цвета. Форматы метода:

```
pixel(<X>, <Y>)
pixel(<QPoint>)
```

♦ `convertToFormat()` — преобразует формат изображения (см. значения параметра `<Формат>` в конструкторе класса `QImage`) и возвращает новый экземпляр класса `QImage`. Исходное изображение не изменяется. Форматы метода:

```
convertToFormat(<Формат>[, flags=AutoColor])
convertToFormat(<Формат>, <Таблица цветов>[, flags=AutoColor])
```

♦ `copy()` — возвращает экземпляр класса `QImage` с фрагментом изображения. Если параметр `rect` не указан, изображение копируется полностью. Форматы метода:

```
copy([rect=QRect()])
copy(<X>, <Y>, <Ширина>, <Высота>)
```

♦ `scaled()` — изменяет размер изображения и возвращает результат в виде экземпляра класса `QImage`. Исходное изображение не изменяется. Форматы метода:

```
scaled(<Ширина>, <Высота>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
scaled(<QSize>[, aspectRatioMode=IgnoreAspectRatio][,
       transformMode=FastTransformation])
```

В необязательном параметре `aspectRatioMode` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `IgnoreAspectRatio` — `0` — изменяет размеры без сохранения пропорций сторон;
- `KeepAspectRatio` — `1` — изменяет размеры с сохранением пропорций сторон. При этом часть области нового изображения может оказаться незаполненной;
- `KeepAspectRatioByExpanding` — `2` — изменяет размеры с сохранением пропорций сторон. При этом часть нового изображения может выйти за пределы его области.

В необязательном параметре `transformMode` могут быть указаны следующие атрибуты класса `QtCore.Qt`:

- `FastTransformation` — `0` — сглаживание выключено;
- `SmoothTransformation` — `1` — сглаживание включено;

♦ `scaledToWidth(<Ширина>[, mode=FastTransformation])` — изменяет ширину изображения и возвращает результат в виде экземпляра класса `QImage`. Высота изображения изме-

няется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;

- `scaledToHeight(<Высота>[, mode=FastTransformation])` — изменяет высоту изображения и возвращает результат в виде экземпляра класса `QImage`. Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;
- `transformed(<QTransform>[, mode=FastTransformation])` — производит трансформацию изображения (например, поворот) и возвращает результат в виде экземпляра класса `QImage`. Исходное изображение не изменяется. Трансформация задается с помощью экземпляра класса `QTransform`. Параметр `mode` аналогичен параметру `transformMode` в методе `scaled()`;
- `invertPixels([mode=InvertRgb])` — инвертирует значения всех пикселов в изображении. В необязательном параметре `mode` может быть указан атрибут `InvertRgb` или `InvertRgba` класса `QImage`;
- `mirrored([horizontal=False][, vertical=True])` — создает зеркальный образ изображения. Метод возвращает экземпляр класса `QImage`. Исходное изображение не изменяется.

24.3.4. Класс *QIcon*

Класс `QIcon` представляет значки в различных размерах, режимах и состояниях. Обратите внимание, что он не наследует класс `QPaintDevice`, — следовательно, мы не можем использовать его как поверхность для рисования. Форматы конструктора:

```
<Объект> = QIcon()
<Объект> = QIcon(<Путь к файлу>)
<Объект> = QIcon(<QPixmap>)
<Объект> = QIcon(<QIcon>)
```

Первый конструктор создает пустой объект значка. Второй конструктор выполняет загрузку значка из файла, причем файл загружается при первой попытке использования, а не сразу. Третий конструктор создает значок на основе экземпляра класса `QPixmap`, а четвертый — создает копию значка.

Класс `QIcon` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qicon.html**):

- `isNull()` — возвращает значение `True`, если объект является пустым, и `False` — в противном случае;
- `addFile(<Путь к файлу>[, size=QSize()][, mode=Normal][, state=Off])` — добавляет значок для указанного размера, режима и состояния. Можно добавить несколько значков, вызывая метод с разными значениями параметров. Параметр `size` задает размер значка в виде экземпляра класса `QSize` (т. к. загрузка значка производится при первой попытке использования, заранее размер значка неизвестен, и нам придется задать его явно). В параметре `mode` можно указать следующие атрибуты класса `QIcon`: `Normal` (обычное состояние компонента), `Disabled` (компонент недоступен), `Active` (компонент активен) или `Selected` (компонент выделен — обычно используется для элементов представлений). В параметре `state` указываются атрибуты `Off` (отключенное состояние) или `On` (включенное состояние) класса `QIcon`;

- `addPixmap(<QPixmap>[, mode=Normal][, state=Off])` — добавляет значок для указанного режима и состояния. Значок создается на основе экземпляра класса `QPixmap`;
- `availableSizes([mode=Normal][, state=Off])` — возвращает доступные размеры (список с экземплярами класса `QSize`) значков для указанного режима и состояния;
- `actualSize(<QSize>[, mode=Normal][, state=Off])` — возвращает фактический размер (экземпляр класса `QSize`) для указанного размера, режима и состояния. Фактический размер может быть меньше размера, указанного в первом параметре, но не больше;
- `pixmap()` — возвращает значок (экземпляр класса `QPixmap`), который примерно соответствует указанному размеру, режиму и состоянию. Форматы метода:

  ```
  pixmap(<Ширина>, <Высота>[, mode=Normal][, state=Off])
  pixmap(<Ширина и высота>[, mode=Normal][, state=Off])
  pixmap(<QSize>[, mode=Normal][, state=Off])
  ```

  Во втором формате первый параметр задает и ширину, и высоту значка (предполагается, что значок имеет квадратную форму).

Вместо загрузки значка из файла можно воспользоваться одним из встроенных в PyQt 5 стандартных значков. Загрузить стандартный значок позволяет следующий код:

```
ico = window.style().standardIcon(QtWidgets.QStyle.SP_MessageBoxCritical)
```

Найти список всех встроенных значков можно в документации к классу `QStyle` на странице **https://doc.qt.io/qt-5/qstyle.html#StandardPixmap-enum**.

ГЛАВА 25

Графическая сцена

Графическая сцена позволяет отображать объекты (линии, прямоугольники и др.) и производить с ними различные манипуляции (перемещать с помощью мыши, преобразовывать систему координат и др.). Для отображения графических объектов применяется концепция «модель-представление», позволяющая отделить данные от их отображения и избежать дублирования данных. Благодаря этому одну и ту же сцену можно отобразить сразу в нескольких представлениях без дублирования. В основе концепции лежат следующие классы:

- `QGraphicsScene` — исполняет роль сцены, на которой расположены графические объекты (модель). Этот класс также поддерживает ряд методов для управления такими объектами;
- `QGraphicsItem` — является базовым классом для графических объектов (элементов модели). Можно наследовать этот класс и реализовать свой графический объект или воспользоваться готовыми классами, например: `QGraphicsRectItem` (прямоугольник), `QGraphicsEllipseItem` (эллипс) и др.;
- `QGraphicsView` — предназначен для отображения сцены (представление). Одну сцену можно отображать с помощью нескольких представлений.

Все описанные в этой главе классы объявлены в модуле `QtWidgets`, если не указано иное.

25.1. Класс *QGraphicsScene*: сцена

Класс `QGraphicsScene` исполняет роль сцены, на которой расположены графические объекты, и поддерживает множество методов для управления этими объектами. Иерархия наследования выглядит так:

```
QObject — QGraphicsScene
```

Форматы конструктора:

```
<Объект> = QGraphicsScene([parent=None])
<Объект> = QGraphicsScene(<X>, <Y>, <Ширина>, <Высота>[, parent=None])
<Объект> = QGraphicsScene(<QRectF>[, parent=None])
```

Первый конструктор создает сцену, не имеющую определенного размера. Второй и третий конструкторы позволяют указать размеры сцены в виде вещественных чисел или экземпляра класса `QRectF`. В качестве параметра `parent` можно указать ссылку на родительский компонент.

25.1.1. Настройка сцены

Для настройки различных параметров сцены предназначены следующие методы класса `QGraphicsScene`:

- `setSceneRect()` — задает координаты и размеры сцены. Форматы метода:

  ```
  setSceneRect(<X>, <Y>, <Ширина>, <Высота>)
  setSceneRect(<QRectF>)
  ```

- `sceneRect()` — возвращает экземпляр класса `QRectF` с координатами и размерами сцены;
- `width()` и `height()` — возвращают ширину и высоту сцены соответственно в виде вещественного числа;
- `itemsBoundingRect()` — возвращает экземпляр класса `QRectF` с координатами и размерами прямоугольника, в который вписываются все объекты, расположенные на сцене;
- `setBackgroundBrush(<QBrush>)` — задает кисть для заднего плана (фона), расположенного под графическими объектами. Чтобы изменить задний план, также можно переопределить у сцены метод `drawBackground()` и выполнять перерисовку заднего плана при каждом вызове внутри него;
- `setForegroundBrush(<QBrush>)` — задает кисть для переднего плана, расположенного над графическими объектами. Чтобы изменить передний план, также можно переопределить у сцены метод `drawForeground()` и внутри него выполнять перерисовку переднего плана при каждом вызове;
- `setFont(<QFont>)` — задает шрифт сцены по умолчанию;
- `setItemIndexMethod(<Режим>)` — задает режим индексации объектов сцены. В качестве параметра указываются следующие атрибуты класса `QGraphicsScene`:
  - `BspTreeIndex` — `0` — для поиска объектов используется индекс в виде бинарного дерева. Этот режим следует применять для сцен, большинство объектов которых являются статическими;
  - `NoIndex` — `-1` — индекс не используется. Этот режим следует применять для сцен, содержимое которых часто меняется;
- `setBspTreeDepth(<Число>)` — задает глубину дерева при использовании режима `BspTreeIndex`. По умолчанию установлено значение `0`, которое говорит, что глубина выбирается автоматически;
- `bspTreeDepth()` — возвращает текущее значение глубины дерева при использовании режима `BspTreeIndex`.

25.1.2. Добавление и удаление графических объектов

Для добавления графических объектов на сцену и удаления их оттуда предназначены следующие методы класса `QGraphicsScene`:

- `addItem(<QGraphicsItem>)` — добавляет графический объект на сцену. В качестве значения указывается экземпляр класса, наследующего класс `QGraphicsItem`, — например, `QGraphicsEllipseItem` (эллипс);
- `addLine()` — создает линию, добавляет ее на сцену и возвращает ссылку на представляющий ее экземпляр класса `QGraphicsLineItem`. Форматы метода:

```
addLine(<X1>, <Y1>, <X2>, <Y2>[, pen=QPen()])
addLine(<QLineF>[, pen=QPen()])
```

- `addRect()` — создает прямоугольник, добавляет его на сцену и возвращает ссылку на представляющий его экземпляр класса `QGraphicsRectItem`. Форматы метода:

```
addRect(<X>, <Y>, <Ширина>, <Высота>[, pen=QPen()][, brush=QBrush()])
addRect(<QRectF>[, pen=QPen()][, brush=QBrush()])
```

- `addPolygon(<QPolygonF>[, pen=QPen()][, brush=QBrush()])` — создает многоугольник, добавляет его на сцену и возвращает ссылку на представляющий его экземпляр класса `QGraphicsPolygonItem`;
- `addEllipse()` — создает эллипс, добавляет его на сцену и возвращает ссылку на представляющий его экземпляр класса `QGraphicsEllipseItem`. Форматы метода:

```
addEllipse(<X>, <Y>, <Ширина>, <Высота>[, pen=QPen()][, brush=QBrush()])
addEllipse(<QRectF>[, pen=QPen()][, brush=QBrush()])
```

- `addPixmap(<QPixmap>)` — создает изображение, добавляет его на сцену и возвращает ссылку на представляющий его экземпляр класса `QGraphicsPixmapItem`;
- `addSimpleText(<Текст>[, font=QFont()])` — создает фрагмент простого текста, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает ссылку на представляющий его экземпляр класса `QGraphicsSimpleTextItem`;
- `addText(<Текст>[, font=QFont()])` — создает фрагмент форматированного текста, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает ссылку на представляющий его экземпляр класса `QGraphicsTextItem`;
- `addPath(<QPainterPath>[, pen=QPen()][, brush=QBrush()])` — создает сложную фигуру («путь»), добавляет ее на сцену и возвращает ссылку на представляющий ее экземпляр класса `QGraphicsPathItem`;
- `removeItem(<QGraphicsItem>)` — убирает графический объект и всех его потомков со сцены. Графический объект при этом не удаляется и, например, может быть добавлен на другую сцену. В качестве значения указывается экземпляр класса, который наследует класс `QGraphicsItem`, например, `QGraphicsEllipseItem` (эллипс);
- `clear()` — удаляет все элементы со сцены. Метод является слотом;
- `createItemGroup(<Список с объектами>)` — группирует объекты, добавляет группу на сцену и возвращает представляющий созданную группу экземпляр класса `QGraphicsItemGroup`;
- `destroyItemGroup(<QGraphicsItemGroup>)` — удаляет группу со сцены, при этом сохраняя все содержащиеся в группе элементы.

25.1.3. Добавление компонентов на сцену

Помимо графических объектов, на сцену можно добавить компоненты, которые будут функционировать как обычно. Добавить компонент на сцену позволяет метод `addWidget()` класса `QGraphicsScene`. Формат метода:

```
addWidget(<QWidget>[, flags=0])
```

В первом параметре указывается экземпляр класса, который наследует класс `QWidget`. Во втором параметре задается тип окна (см. *разд. 18.2*). Метод возвращает ссылку на экземпляр класса `QGraphicsProxyWidget`, представляющий добавленный компонент.

25.1.4. Поиск объектов

Для поиска графических объектов, находящихся на сцене, предназначены следующие методы класса `QGraphicsScene`:

- `itemAt()` — возвращает ссылку на верхний видимый объект, который расположен по указанным координатам, или значение `None`, если никакого объекта там нет. Форматы метода:

```
itemAt(<X>, <Y>, <QTransform>)
itemAt(<QPointF>, <QTransform>)
```

  Третий параметр задает примененные к сцене преобразования системы координат (см. *разд. 24.2.4*). Его необходимо указывать, если на сцене присутствуют объекты, игнорирующие преобразования. В противном случае следует указать пустой экземпляр класса `QTransform`;

- `collidingItems(<QGraphicsItem>[, mode=IntersectsItemShape])` — возвращает список со ссылками на объекты, которые находятся в указанном в первом параметре объекте или пересекаются с ним. Если таких объектов нет, метод возвращает пустой список. Необязательный параметр `mode` указывает режим поиска графических объектов и должен задаваться следующими атрибутами класса `QtCore.Qt`:
  - `ContainsItemShape` — `0` — ищутся объекты, чьи границы полностью находятся внутри заданного объекта;
  - `IntersectsItemShape` — `1` — ищутся объекты, чьи границы полностью находятся внутри заданного объекта или пересекаются с его границей;
  - `ContainsItemBoundingRect` — `2` — ищутся объекты, чьи охватывающие прямоугольники (т. е. прямоугольники минимальных размеров, в которые искомые объекты помещаются полностью) полностью находятся внутри охватывающего прямоугольника заданного объекта;
  - `IntersectsItemBoundingRect` — `3` — ищутся объекты, чьи охватывающие прямоугольники (т. е. прямоугольники минимальных размеров, в которые искомые объекты помещаются полностью) полностью находятся внутри охватывающего прямоугольника заданного объекта или пересекаются с его границами;

- `items()` — возвращает список со ссылками на все объекты или на объекты, расположенные по указанным координатам, или на объекты, попадающие в указанную область. Если объектов нет, то возвращается пустой список. Форматы метода:

```
items([order=DescendingOrder])
items(<QPointF>[, mode=IntersectsItemShape][, order=DescendingOrder][,
      deviceTransform=QTransform()])
items(<X>, <Y>, <Ширина>, <Высота>[, mode=IntersectsItemShape][,
      order=DescendingOrder][, deviceTransform=QTransform()])
items(<QRectF>[, mode=IntersectsItemShape][, order=DescendingOrder][,
      deviceTransform=QTransform()])
items(<QPolygonF>[, mode=IntersectsItemShape][, order=DescendingOrder][,
      deviceTransform=QTransform()])
items(<QPainterPath>[, mode=IntersectsItemShape][, order=DescendingOrder][,
      deviceTransform=QTransform()])
```

В необязательном параметре `order`, задающем порядок сортировки объектов, указываются атрибуты `AscendingOrder` (в алфавитном порядке) или `DescendingOrder` (в обратном порядке) класса `QtCore.Qt`.

В необязательном параметре `mode` указываются следующие атрибуты класса `QtCore.Qt`:

* `ContainsItemShape` — `0` — объект попадет в список, если все точки объекта находятся внутри заданной области;
* `IntersectsItemShape` — `1` — объект попадет в список, если любая точка объекта попадает в заданную область;
* `ContainsItemBoundingRect` — `2` — объект попадет в список, если охватывающий прямоугольник полностью находится внутри заданной области;
* `IntersectsItemBoundingRect` — `3` — объект попадет в список, если любая точка охватывающего прямоугольника попадает в заданную область.

Необязательный параметр `deviceTransform` задает примененные к сцене преобразования системы координат (см. *разд. 24.2.4*). Его необходимо указывать, если на сцене присутствуют объекты, игнорирующие преобразования.

25.1.5. Управление фокусом ввода

Получить фокус ввода с клавиатуры может как сцена в целом, так и отдельный объект на ней. Если фокус установлен на отдельный объект, все события клавиатуры перенаправляются этому объекту. Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable`, например, с помощью метода `setFlag()` класса `QGraphicsItem`. Для управления фокусом ввода предназначены следующие методы этого класса:

♦ `setFocus([focusReason=OtherFocusReason])` — устанавливает фокус ввода на сцену. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 19.8.1*);

♦ `setFocusItem(<QGraphicsItem>[, focusReason=OtherFocusReason])` — устанавливает фокус ввода на указанный графический объект на сцене. Если сцена была вне фокуса ввода, она автоматически получит фокус. Если в первом параметре указано значение `None`, или объект не может принимать фокус, метод просто убирает фокус с объекта, обладающего им в текущий момент. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 19.8.1*);

♦ `clearFocus()` — убирает фокус ввода со сцены. Объект, который обладает фокусом ввода в текущий момент, потеряет его, но получит его снова, когда фокус будет опять установлен на сцену;

♦ `hasFocus()` — возвращает значение `True`, если сцена имеет фокус ввода, и `False` — в противном случае;

♦ `focusItem()` — возвращает ссылку на объект, который обладает фокусом ввода, или значение `None`;

♦ `setStickyFocus(<Флаг>)` — если в качестве параметра указано значение `True`, то при щелчке мышью на фоне сцены или на объекте, который не может принимать фокус, объект, владеющий фокусом, не потеряет его. По умолчанию фокус убирается;

♦ `stickyFocus()` — возвращает значение `True`, если фокус ввода не будет убран с объекта при щелчке мышью на фоне или на объекте, который не может принимать фокус.

25.1.6. Управление выделением объектов

Чтобы объект можно было выделить (с помощью мыши или программно), необходимо установить флаг `ItemIsSelectable`, например, с помощью метода `setFlag()` класса `QGraphicsItem`. Для управления выделением объектов предназначены следующие методы этого класса:

- ♦ `setSelectionArea()` — выделяет объекты внутри указанной области. Чтобы выделить только один объект, следует воспользоваться методом `setSelected()` класса `QGraphicsItem`. Форматы метода `setSelectionArea()`:

```
setSelectionArea(<QPainterPath>, <QTransform>)
setSelectionArea(<QPainterPath>[, mode=IntersectsItemShape][,
                deviceTransform=QTransform()])
setSelectionArea(<QPainterPath>, <Операция>[, mode=IntersectsItemShape][,
                deviceTransform=QTransform()])
```

  В необязательном параметре `mode` могут быть указаны следующие атрибуты класса `QtCore.Qt`:

  - `ContainsItemShape` — `0` — объект будет выделен, если все точки объекта находятся внутри области выделения;
  - `IntersectsItemShape` — `1` — объект будет выделен, если любая точка объекта попадет в область выделения;
  - `ContainsItemBoundingRect` — `2` — объект будет выделен, если охватывающий прямоугольник полностью находится внутри области выделения;
  - `IntersectsItemBoundingRect` — `3` — объект будет выделен, если любая точка охватывающего прямоугольника попадет в область выделения.

  Второй параметр первого формата и параметр `deviceTransform` второго формата задают примененные к сцене преобразования системы координат (см. *разд. 24.2.4*).

  Третий формат позволяет дополнительно указать, какое преобразование (параметр `<Операция>`) следует выполнить с ранее выделенными объектами сцены. В качестве параметра `<Операция>` задается один из следующих атрибутов класса `QtCore.Qt`:

  - `ReplaceSelection` — `0` — ранее выделенные объекты перестанут быть выделенными;
  - `AddToSelection` — `1` — ранее выделенные объекты останутся выделенными.

  Этот формат поддерживается, начиная с PyQt 5.5;
- ♦ `selectionArea()` — возвращает область выделения в виде экземпляра класса `QPainterPath`;
- ♦ `selectedItems()` — возвращает список со ссылками на выделенные объекты или пустой список, если выделенных объектов нет;
- ♦ `clearSelection()` — снимает выделение. Метод является слотом.

25.1.7. Прочие методы и сигналы

Помимо рассмотренных ранее, класс `QGraphicsScene` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsscene.html**):

- ♦ `isActive()` — возвращает значение `True`, если сцена отображается на экране с помощью какого-либо представления, и `False` — в противном случае;

- `views()` — возвращает список с представлениями (экземпляры класса `QGraphicsView`), в которых выводится сцена. Если сцена вообще не выводится на экран, возвращается пустой список;
- `mouseGrabberItem()` — возвращает ссылку на объект, который владеет мышью, или `None`, если такого объекта нет;
- `render()` — позволяет вывести содержимое сцены на устройство рисования или принтер. Формат метода:

  ```
  render(<QPainter>[, target=QRectF()][, source=QRectF()][, mode=KeepAspectRatio])
  ```

  Параметр `target` задает координаты и размеры устройства рисования, а параметр `source` — координаты и размеры прямоугольной области на сцене. Если параметры не указаны, используются размеры устройства рисования и сцены. В параметре `mode`, задающем режим изменения размеров графики при выводе, могут быть указаны следующие атрибуты класса `QtCore.Qt`:

  - `IgnoreAspectRatio` — `0` — изменяет размеры без сохранения пропорций сторон;
  - `KeepAspectRatio` — `1` — изменяет размеры с сохранением пропорций сторон. При этом часть области на устройстве рисования может оказаться незаполненной;
  - `KeepAspectRatioByExpanding` — `2` — изменяет размеры с сохранением пропорций сторон. При этом часть выводимой графики может выйти за пределы области на устройстве рисования;
- `invalidate()` — вызывает перерисовку указанных слоев внутри прямоугольной области на сцене. Форматы метода:

  ```
  invalidate(<X>, <Y>, <Ширина>, <Высота>[, layers=AllLayers])
  invalidate([rect=QRectF()][, layers=AllLayers])
  ```

  В параметре `layers`, задающем слои, которые требуется перерисовать, могут быть указаны следующие атрибуты класса `QGraphicsScene`:

  - `ItemLayer` — `1` — слой объекта;
  - `BackgroundLayer` — `2` — слой заднего плана;
  - `ForegroundLayer` — `4` — слой переднего плана;
  - `AllLayers` — `65535` — все слои. Вначале отрисовывается слой заднего плана, затем — слой объекта и в конце — слой переднего плана.

  Второй формат метода является слотом;
- `update()` — вызывает перерисовку указанной прямоугольной области сцены. Форматы метода:

  ```
  update(<X>, <Y>, <Ширина>, <Высота>)
  update([rect=QRectF()])
  ```

  Второй формат метода является слотом.

Класс `QGraphicsScene` поддерживает следующие сигналы:

- `changed(<Список областей>)` — генерируется при изменении сцены. Внутри обработчика через параметр доступен список с экземплярами класса `QRectF`, представляющими изменившиеся области сцены;
- `sceneRectChanged(<QRectF>)` — генерируется при изменении размеров сцены. Внутри обработчика через параметр доступен экземпляр класса `QRectF` с новыми координатами и размерами сцены;

- ♦ `selectionChanged()` — генерируется при изменении выделения объектов;
- ♦ `focusItemChanged(<QGraphicItem>, <QGraphicItem>, <Причина изменения фокуса>)` — генерируется при изменении фокуса клавиатурного ввода. Через параметры доступны объект, получивший фокус ввода, объект, потерявший фокус ввода, и причина изменения фокуса (см. *разд. 19.8.1*).

25.2. Класс *QGraphicsView*: представление

Класс `QGraphicsView` предназначен для отображения сцены. Одну сцену можно отображать с помощью нескольких представлений. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea — QGraphicsView
```

Форматы конструктора класса:

```
<Объект> = QGraphicsView([parent=None])
<Объект> = QGraphicsView(<QGraphicsScene>[, parent=None])
```

Второй формат сразу же позволяет установить в представлении сцену, которая будет выводиться на экран.

25.2.1. Настройка представления

Для настройки различных параметров представления применяются следующие методы класса `QGraphicsView` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsview.html**):

- ♦ `setScene(<QGraphicsScene>)` — устанавливает выводимую на экран сцену;
- ♦ `scene()` — возвращает ссылку на установленную сцену в виде экземпляра класса `QGraphicsScene`;
- ♦ `setSceneRect()` — задает координаты и размеры сцены. Форматы метода:

  ```
  setSceneRect(<X>, <Y>, <Ширина>, <Высота>)
  setSceneRect(<QRectF>)
  ```

- ♦ `sceneRect()` — возвращает экземпляр класса `QRectF` с координатами и размерами сцены;
- ♦ `setBackgroundBrush(<QBrush>)` — задает кисть для заднего плана (фона) сцены (расположен под графическими объектами);
- ♦ `setForegroundBrush(<QBrush>)` — задает кисть для переднего плана сцены (расположен над графическими объектами);
- ♦ `setCacheMode(<Режим>)` — задает режим кэширования выводимых объектов. В качестве параметра могут быть указаны следующие атрибуты класса `QGraphicsView`:
  - `CacheNone` — `0` — без кэширования;
  - `CacheBackground` — `1` — кэшируется только задний фон;
- ♦ `resetCachedContent()` — сбрасывает кэш;
- ♦ `setAlignment(<Выравнивание>)` — задает выравнивание сцены в случае, если содержимое сцены полностью помещается в представлении. По умолчанию сцена центрируется по горизонтали и вертикали. Вот пример установки сцены в левом верхнем углу представления:

  ```
  view.setAlignment(QtCore.Qt.AlignLeft | QtCore.Qt.AlignTop)
  ```

- `setInteractive(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь может взаимодействовать с объектами на сцене (интерактивный режим, используется по умолчанию). Значение `False` разрешает только просмотр сцены;
- `isInteractive()` — возвращает значение `True`, если задан интерактивный режим, и `False` — в противном случае;
- `setDragMode(<Режим>)` — задает действие, которое производится при щелчке левой кнопкой мыши на фоне и перемещении мыши. Получить текущее действие позволяет метод `dragMode()`. В качестве параметра могут быть указаны следующие атрибуты класса `QGraphicsView`:
  - `NoDrag` — `0` — ничего не происходит;
  - `ScrollHandDrag` — `1` — перемещение мыши при нажатой левой кнопке будет приводить к прокрутке сцены. При этом указатель мыши примет вид сжатой или разжатой руки;
  - `RubberBandDrag` — `2` — создается область выделения. Объекты, частично или полностью (задается с помощью метода `setRubberBandSelectionMode()`) попавшие в эту область, будут выделены (при условии, что установлен флаг `ItemIsSelectable`). Действие выполняется только в интерактивном режиме;
- `setRubberBandSelectionMode(<Режим>)` — задает режим выделения объектов при установленном флаге `RubberBandDrag`. В параметре `<Режим>` могут быть указаны следующие атрибуты класса `QtCore.Qt`:
  - `ContainsItemShape` — `0` — объект будет выделен, если все точки объекта находятся внутри области выделения;
  - `IntersectsItemShape` — `1` — объект будет выделен, если любая точка объекта попадет в область выделения;
  - `ContainsItemBoundingRect` — `2` — объект будет выделен, если охватывающий прямоугольник полностью находится внутри области выделения;
  - `IntersectsItemBoundingRect` — `3` — объект будет выделен, если любая точка охватывающего прямоугольника попадет в область выделения;
- `setRenderHint(<Опция вывода>[, <Флаг>])` — управляет опциями, влияющими на качество отображения сцены. Если вторым параметром передано значение `True`, или если второй параметр вообще не указан, `<Опция вывода>` активизируется, если же передано `False`, опция деактивируется. Сама `<Опция вывода>` указывается в виде одного из следующих атрибутов класса `QPainter`:
  - `Antialiasing` — `1` — выполнять сглаживание краев у графических объектов;
  - `TextAntialiasing` — `2` — выполнять сглаживание текста. Активизирован по умолчанию;
  - `SmoothPixmapTransform` — `4` — использовать более качественное сглаживание при трансформациях растровых изображений;
  - `Qt4CompatiblePainting` — `32` — использовать тот же способ сглаживания, что применялся в PyQt 4. Присутствует для совместимости.

  Вот пример активизирования для представления всех доступных опций вывода:

  ```
  view.setRenderHint(QtGui.QPainter.Antialiasing)
  view.setRenderHint(QtGui.QPainter.TestAntialiasing)
  view.setRenderHint(QtGui.QPainter.SmoothPixmapTransform)
  ```

♦ `setRenderHints(<Опции вывода>)` — активизирует сразу все указанные `<Опции вывода>` (те, что не были указаны, деактивируются). `<Опции вывода>` задаются в виде комбинации описанных ранее атрибутов класса `QPainter`, перечисленных через оператор `|`.

Вот пример активизирования для представления всех доступных опций:

```
view.setRenderHints(QtGui.QPainter.Antialiasing |
                    QtGui.QPainter.TestAntialiasing |
                    QtGui.QPainter.SmoothPixmapTransform)
```

25.2.2. Преобразования между координатами представления и сцены

Для преобразования между координатами представления и сцены предназначены следующие методы класса `QGraphicsView`:

♦ `mapFromScene()` — преобразует координаты точки из системы координат сцены в систему координат представления. Форматы метода (справа указан тип возвращаемого значения):

```
mapFromScene(<X>, <Y>)                             -> QPoint
mapFromScene(<QPointF>)                            -> QPoint
mapFromScene(<X>, <Y>, <Ширина>, <Высота>) -> QPolygon
mapFromScene(<QRectF>)                             -> QPolygon
mapFromScene(<QPolygonF>)                          -> QPolygon
mapFromScene(<QPainterPath>)                       -> QPainterPath
```

♦ `mapToScene()` — преобразует координаты точки из системы координат представления в систему координат сцены. Форматы метода (справа указан тип возвращаемого значения):

```
mapToScene(<X>, <Y>)                               -> QPointF
mapToScene(<QPoint>)                               -> QPointF
mapToScene(<X>, <Y>, <Ширина>, <Высота>)   -> QPolygonF
mapToScene(<QRect>)                                -> QPolygonF
mapToScene(<QPolygon>)                             -> QPolygonF
mapToScene(<QPainterPath>)                         -> QPainterPath
```

25.2.3. Поиск объектов

Для поиска объектов на сцене предназначены следующие методы:

♦ `itemAt()` — возвращает ссылку на верхний видимый объект, который расположен по указанным координатам, или значение `None`, если никакого объекта там нет. В качестве значений указываются координаты в системе координат представления, а не сцены. Форматы метода:

```
itemAt(<X>, <Y>)
itemAt(<QPoint>)
```

♦ `items()` — возвращает список со ссылками на все объекты, или на объекты, расположенные по указанным координатам, или на объекты, попадающие в указанную область. Если объектов нет, возвращается пустой список. В качестве значений указываются координаты в системе координат представления, а не сцены. Форматы метода:

```
items()
items(<X>, <Y>)
items(<QPoint>)
items(<X>, <Y>, <Ширина>, <Высота>[, mode=IntersectsItemShape])
items(<QRect>[, mode=IntersectsItemShape])
items(<QPolygon>[, mode=IntersectsItemShape])
items(<QPainterPath>[, mode=IntersectsItemShape])
```

Допустимые значения параметра `mode` мы уже рассматривали в *разд. 25.1.4*.

25.2.4. Преобразование системы координат

Выполнить преобразование системы координат позволяют следующие методы класса `QGraphicsView`:

- `translate(<X>, <Y>)` — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу, ось `X` направлена вправо, а ось `Y` — вниз;
- `rotate(<Угол>)` — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки;
- `scale(<По оси X>, <По оси Y>)` — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы — то увеличение;
- `shear(<По горизонтали>, <По вертикали>)` — сдвигает систему координат. В качестве значений указываются вещественные числа;
- `resetTransform()` — отменяет все преобразования.

Несколько преобразований можно произвести последовательно друг за другом. При этом надо учитывать, что порядок следования преобразований имеет значение.

Если одна и та же последовательность выполняется несколько раз, то ее можно сохранить в экземпляре класса `QTransform`, а затем установить с помощью метода `setTransform()`. Получить ссылку на установленную матрицу позволяет метод `transform()`.

25.2.5. Прочие методы

Помимо рассмотренных ранее, класс `QGraphicsView` поддерживает следующие методы (Здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsview.html**):

- `centerOn()` — прокручивает область таким образом, чтобы указанная точка или объект находились в центре видимой области представления. Форматы метода:

  ```
  centerOn(<X>, <Y>)
  centerOn(<QPointF>)
  centerOn(<QGraphicsItem>)
  ```

- `ensureVisible()` — прокручивает область таким образом, чтобы указанный прямоугольник или объект находились в видимой области представления. Форматы метода:

  ```
  ensureVisible(<X>, <Y>, <Ширина>, <Высота>[, xMargin=50][, yMargin=50])
  ensureVisible(<QRectF>[, xMargin=50][, yMargin=50])
  ensureVisible(<QGraphicsItem>[, xMargin=50][, yMargin=50])
  ```

Необязательные параметры `xMargin` и `yMargin` задают минимальные значения просветов между границами графического объекта и краями представления по горизонтали и вертикали соответственно;

- `fitInView()` — прокручивает и масштабирует область таким образом, чтобы указанный прямоугольник или объект занимали всю видимую область представления. Форматы метода:

```
fitInView(<X>, <Y>, <Ширина>, <Высота>[, mode=IgnoreAspectRatio])
fitInView(<QRectF>[, mode=IgnoreAspectRatio])
fitInView(<QGraphicsItem>[, mode=IgnoreAspectRatio])
```

  Необязательный параметр `mode` задает режим изменения размеров. Его значения рассматривались в *разд. 25.1.7* (см. описание метода `render()` класса `QGraphicsScene`);

- `render()` — позволяет вывести содержимое представления на устройство рисования или принтер. Формат метода:

```
render(<QPainter>[, target=QRectF()][, source=QRectF()][, mode=KeepAspectRatio])
```

  Назначение параметров этого метода сходно с таковым у метода `render()` класса `QGraphicsScene` (см. *разд. 25.1.7*);

- `invalidateScene([rect=QRectF()][, layers=AllLayers])` — вызывает перерисовку указанных слоев внутри прямоугольной области на сцене. Назначение его параметров сходно с таковым у одноименного метода класса `QGraphicsScene` (см. *разд. 25.1.7*). Метод является слотом;

- `updateSceneRect(<QRectF>)` — вызывает перерисовку указанной прямоугольной области сцены. Метод является слотом;

- `updateScene(<Список с экземплярами класса QRectF>)` — вызывает перерисовку указанных прямоугольных областей. Метод является слотом.

25.3. Класс *QGraphicsItem*: базовый класс для графических объектов

Абстрактный класс `QGraphicsItem` является базовым классом для графических объектов. Формат конструктора класса:

```
QGraphicsItem([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский объект (экземпляр класса, наследующего класс `QGraphicsItem`).

Поскольку класс `QGraphicsItem` является абстрактным, создать его экземпляр нельзя. Чтобы создать новый графический объект, следует наследовать этот класс и переопределить, как минимум, методы `boundingRect()` и `paint()`. Метод `boundingRect()` должен возвращать экземпляр класса `QRectF` с координатами и размерами прямоугольной области, ограничивающей объект. Внутри метода `paint()` необходимо выполнить рисование объекта. Формат метода `paint()`:

```
paint(self, <QPainter>, <QStyleOptionGraphicsItem>, widget=None)
```

Для обработки столкновений следует также переопределить метод `shape()`. Метод должен возвращать экземпляр класса `QPainterPath`.

25.3.1. Настройка объекта

Для настройки различных параметров объекта предназначены следующие методы класса `QGraphicsItem` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsitem.html**):

- `setPos()` — задает позицию объекта относительно родителя или сцены (при отсутствии родителя). Форматы метода:

  ```
  setPos(<X>, <Y>)
  setPos(<QPointF>)
  ```

- `pos()` — возвращает экземпляр класса `QPointF` с текущими координатами относительно родителя или сцены (при отсутствии родителя);
- `scenePos()` — возвращает экземпляр класса `QPointF` с текущими координатами относительно сцены;
- `sceneBoundingRect()` — возвращает экземпляр класса `QRectF`, который содержит координаты (относительно сцены) и размеры прямоугольника, ограничивающего объект;
- `setX(<X>)` и `setY(<Y>)` — задают позицию объекта по отдельным осям;
- `x()` и `y()` — возвращают позицию объекта по отдельным осям;
- `setZValue(<Z>)` — задает позицию объекта по оси `Z`. Объект с бóльшим значением этого параметра рисуется выше объекта с меньшим значением. По умолчанию для всех объектов значение позиции по оси `Z` равно `0.0`;
- `zValue()` — возвращает позицию объекта по оси `Z`;
- `moveBy(<По оси X>, <По оси Y>)` — сдвигает объект на указанное смещение относительно текущей позиции;
- `prepareGeometryChange()` — этот метод следует вызвать перед изменением размеров объекта, чтобы поддержать индекс сцены в актуальном состоянии;
- `scene()` — возвращает ссылку на сцену (экземпляр класса `QGraphicsScene`) или значение `None`, если объект не помещен на сцену;
- `setFlag(<Флаг>[, enabled=True])` — устанавливает (если второй параметр имеет значение `True`) или сбрасывает (если второй параметр имеет значение `False`) указанный флаг. В первом параметре могут быть указаны следующие атрибуты класса `QGraphicsItem` (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsitem.html#GraphicsItemFlag-enum**):
  - `ItemIsMovable` — `0x1` — объект можно перемещать с помощью мыши;
  - `ItemIsSelectable` — `0x2` — объект можно выделять;
  - `ItemIsFocusable` — `0x4` — объект может получить фокус ввода;
  - `ItemIgnoresTransformations` — `0x20` — объект игнорирует наследуемые преобразования;
  - `ItemIgnoresParentOpacity` — `0x40` — объект игнорирует прозрачность родителя;
  - `ItemDoesntPropagateOpacityToChildren` — `0x80` — прозрачность объекта не распространяется на потомков;
  - `ItemStacksBehindParent` — `0x100` — объект располагается позади родителя;
  - `ItemIsPanel` — `0x4000` — объект является панелью;

- `setFlags(<Флаги>)` — устанавливает сразу несколько флагов. Атрибуты (см. описание метода `setFlag()`) указываются через оператор `|`;
- `flags()` — возвращает комбинацию установленных флагов (см. описание метода `setFlag()`);
- `setOpacity(<Число>)` — задает степень прозрачности объекта. В качестве значения указывается вещественное число от `0.0` (полностью прозрачный) до `1.0` (полностью непрозрачный);
- `opacity()` — возвращает степень прозрачности объекта;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `setCursor(<Курсор>)` — задает внешний вид указателя мыши при наведении указателя на объект (см. *разд. 19.9.5*);
- `unsetCursor()` — отменяет изменение указателя мыши;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то объект будет видим. Значение `False` скрывает объект;
- `show()` — делает объект видимым;
- `hide()` — скрывает объект;
- `isVisible()` — возвращает значение `True`, если объект видим, и `False` — если скрыт;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, объект будет доступен. Значение `False` делает объект недоступным. Недоступный объект не получает никаких событий, и его нельзя выделить;
- `isEnabled()` — возвращает значение `True`, если объект доступен, и `False` — если недоступен;
- `setSelected(<Флаг>)` — если в качестве параметра указано значение `True`, объект будет выделен. Значение `False` снимает выделение. Чтобы объект можно было выделить, необходимо установить флаг `ItemIsSelectable`, например, с помощью метода `setFlag()` класса `QGraphicsItem`;
- `isSelected()` — возвращает значение `True`, если объект выделен, и `False` — в противном случае;
- `setFocus([focusReason=OtherFocusReason])` — устанавливает фокус ввода на объект. В параметре `focusReason` можно указать причину изменения фокуса ввода (см. *разд. 19.8.1*). Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable`, например, с помощью метода `setFlag()` класса `QGraphicsItem`;
- `clearFocus()` — убирает фокус ввода с объекта;
- `hasFocus()` — возвращает значение `True`, если объект находится в фокусе ввода, и `False` — в противном случае;
- `grabKeyboard()` — захватывает ввод с клавиатуры;
- `ungrabKeyboard()` — освобождает ввод с клавиатуры;
- `grabMouse()` — захватывает мышь;
- `ungrabMouse()` — освобождает мышь.

25.3.2. Выполнение преобразований

Задать преобразования для графического объекта можно, воспользовавшись классом `QTransform`. Для работы с преобразованиями, заданными экземплярами этого класса, класс `QGraphicsItem` поддерживает следующие методы:

- `setTransform(<QTransform>[, combine=False])` — устанавливает преобразования, заданные в первом параметре. Если вторым параметром передать значение `True`, новые преобразования будут объединены с уже установленными, — в противном случае они заменят установленные ранее преобразования;
- `transform()` — возвращает экземпляр класса `QTransform`, представляющий заданные для объекта преобразования;
- `sceneTransform()` — возвращает экземпляр класса `QTransform`, который представляет преобразования, заданные для самой графической сцены.

Есть и более простой способ задания преобразований — использование следующих методов класса `QGraphicsItem`:

- `setTransformOriginPoint()` — перемещает начало координат в указанную точку. Форматы метода:

```
setTransformOriginPoint(<X>, <Y>)
setTransformOriginPoint(<QPointF>)
```

- `setRotation(<Угол>)` — поворачивает систему координат на указанное количество градусов (указывается вещественное число): положительное значение вызывает поворот по часовой стрелке, а отрицательное — против часовой стрелки;
- `rotation()` — возвращает текущий угол поворота;
- `setScale(<Значение>)` — масштабирует систему координат. В качестве значений указываются вещественные числа: если значение меньше единицы, выполняется уменьшение, а если больше — увеличение;
- `scale()` — возвращает текущий масштаб;
- `resetTransform()` — отменяет все преобразования.

25.3.3. Прочие методы

Помимо рассмотренных ранее, класс `QGraphicsItem` поддерживает следующие полезные для нас методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsitem.html**):

- `setParentItem(<QGraphicsItem>)` — задает родителя для текущего объекта. Местоположение дочернего объекта задается в координатах родительского объекта;
- `parentItem()` — возвращает ссылку на родительский объект;
- `topLevelItem()` — возвращает ссылку на родительский объект верхнего уровня;
- `childItems()` — возвращает список с дочерними объектами;
- `contains(<QPointF>)` — возвращает `True`, если точка с указанными координатами (экземпляр класса `QPointF`) входит в состав графического объекта, и `False` — в противном случае;
- `collidingItems([mode=IntersectsItemShape])` — возвращает список со ссылками на объекты, которые находятся в текущем объекте или пересекаются с ним. Если таких объектов нет, возвращается пустой список. Возможные значения параметра `mode` см. в *разд. 25.1.4*;

- `collidesWithItem(<QGraphicsItem>[, mode=IntersectsItemShape])` — возвращает значение `True`, если объект находится в объекте, указанном в первом параметре, или пересекается с ним. Возможные значения параметра `mode` см. в *разд. 25.1.4*;
- `collidesWithPath(<QGraphicsPath>[, mode=IntersectsItemShape])` — возвращает значение `True`, если объект находится внутри пути, указанного в первом параметре, или пересекается с ним. Возможные значения параметра `mode` см. в *разд. 25.1.4*;
- `ensureVisible()` — прокручивает область таким образом, чтобы указанный прямоугольник находился в видимой области представления. Форматы метода:

  ```
  ensureVisible(<X>, <Y>, <Ширина>, <Высота>[, xMargin=50][, yMargin=50])
  ensureVisible([rect=QRectF()][, xMargin=50][, yMargin=50])
  ```

  Необязательные параметры `xMargin` и `yMargin` задают минимальные значения просветов между границами графического объекта и краями представления по горизонтали и вертикали соответственно;
- `update(<QRectF>)` — вызывает перерисовку указанной прямоугольной области. Форматы метода:

  ```
  update(<X>, <Y>, <Ширина>, <Высота>)
  update([rect=QRectF()])
  ```

25.4. Графические объекты

Вместо создания собственного объекта путем наследования класса `QGraphicsItem` можно воспользоваться следующими стандартными классами:

- `QGraphicsLineItem` — линия;
- `QGraphicsRectItem` — прямоугольник;
- `QGraphicsPolygonItem` — многоугольник;
- `QGraphicsEllipseItem` — эллипс;
- `QGraphicsPixmapItem` — изображение;
- `QGraphicsSimpleTextItem` — простой текст;
- `QGraphicsTextItem` — форматированный текст;
- `QGraphicsPathItem` — путь;
- `QGraphicsSvgItem` — SVG-графика.

25.4.1. Линия

Класс `QGraphicsLineItem` описывает линию. Иерархия наследования:

```
QGraphicsItem — QGraphicsLineItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsLineItem([parent=None])
<Объект> = QGraphicsLineItem(<X1>, <Y1>, <X2>, <Y2>[, parent=None])
<Объект> = QGraphicsLineItem(<QLineF>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsLineItem` наследует все методы класса `QGraphicsItem` и включает поддержку следующих методов (полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicslineitem.html**):

- `setLine()` — задает параметры линии. Форматы метода:

```
setLine(<X1>, <Y1>, <X2>, <Y2>)
setLine(<QLineF>)
```

- `line()` — возвращает параметры линии в виде экземпляра класса `QLineF`;
- `setPen(<QPen>)` — устанавливает перо.

25.4.2. Класс *QAbstractGraphicsShapeItem*

Класс `QAbstractGraphicsShapeItem` является базовым классом всех графических фигур, более сложных, чем линия. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem
```

Класс `QAbstractGraphicsShapeItem` поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qabstractgraphicsshapeitem.html**):

- `setPen(<QPen>)` — устанавливает перо;
- `setBrush(<QBrush>)` — устанавливает кисть.

25.4.3. Прямоугольник

Класс `QGraphicsRectItem` описывает прямоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsRectItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsRectItem([parent=None])
<Объект> = QGraphicsRectItem(<X>, <Y>, <Ширина>, <Высота>[, parent=None])
<Объект> = QGraphicsRectItem(<QRectF>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsRectItem` наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsrectitem.html**):

- `setRect()` — задает параметры прямоугольника. Форматы метода:

```
setRect(<X>, <Y>, <Ширина>, <Высота>)
setRect(<QRectF>)
```

- `rect()` — возвращает параметры прямоугольника в виде экземпляра класса `QRectF`.

25.4.4. Многоугольник

Класс `QGraphicsPolygonItem` описывает многоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsPolygonItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsPolygonItem([parent=None])
<Объект> = QGraphicsPolygonItem(<QPolygonF>[, parent=None])
```

В параметре parent можно указать ссылку на родительский объект.

Класс QGraphicsPolygonItem наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicspolygonitem.html**):

- setPolygon(<QPolygonF>) — задает параметры многоугольника;
- polygon() — возвращает параметры многоугольника в виде экземпляра класса QPolygonF.

25.4.5. Эллипс

Класс QGraphicsEllipseItem описывает эллипс. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsEllipseItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsEllipseItem([parent=None])
<Объект> = QGraphicsEllipseItem(<X>, <Y>, <Ширина>, <Высота>[, parent=None])
<Объект> = QGraphicsEllipseItem(<QRectF>[, parent=None])
```

В параметре parent можно указать ссылку на родительский объект.

Класс QGraphicsEllipseItem наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsellipseitem.html**):

- setRect() — задает параметры прямоугольника, в который необходимо вписать эллипс. Форматы метода:

  ```
  setRect(<X>, <Y>, <Ширина>, <Высота>)
  setRect(<QRectF>)
  ```

- rect() — возвращает параметры прямоугольника, в который вписан эллипс, в виде экземпляра класса QRectF;
- setStartAngle(<Угол>) и setSpanAngle(<Угол>) — задают начальный и конечный углы сектора соответственно. Следует учитывать, что значения углов задаются в значениях $^1/_{16}°$. Полный круг эквивалентен значению $16 \times 360 = 5760$. Нулевой угол находится в позиции «трех часов». Положительные значения углов отсчитываются против часовой стрелки, а отрицательные — по часовой стрелке;
- startAngle() и spanAngle() — возвращают значения начального и конечного углов сектора соответственно.

25.4.6. Изображение

Класс QGraphicsPixmapItem описывает изображение. Иерархия наследования:

```
QGraphicsItem — QGraphicsPixmapItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsPixmapItem([parent=None])
<Объект> = QGraphicsPixmapItem(<QPixmap>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsPixmapItem` наследует все методы из класса `QGraphicsItem` и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicspixmapitem.html**):

- `setPixmap(<QPixmap>)` — задает изображение;
- `pixmap()` — возвращает представляющий изображение экземпляр класса `QPixmap`;
- `setOffset()` — задает местоположение изображения. Форматы метода:

  ```
  setOffset(<X>, <Y>)
  setOffset(<QPointF>)
  ```
- `offset()` — возвращает местоположение изображения в виде экземпляра класса `QPointF`;
- `setShapeMode(<Режим>)` — задает режим определения формы изображения. В качестве параметра могут быть указаны следующие атрибуты класса `QGraphicsPixmapItem`:
  - `MaskShape` — `0` — используется результат выполнения метода `mask()` класса `QPixmap` (значение по умолчанию);
  - `BoundingRectShape` — `1` — форма определяется по контуру изображения;
  - `HeuristicMaskShape` — `2` — используется результат выполнения метода `createHeuristicMask()` класса `QPixmap`;
- `setTransformationMode(<Режим>)` — задает режим сглаживания. В качестве параметра могут быть указаны следующие атрибуты класса `QtCore.Qt`:
  - `FastTransformation` — `0` — сглаживание выключено (по умолчанию);
  - `SmoothTransformation` — `1` — сглаживание включено.

25.4.7. Простой текст

Класс `QGraphicsSimpleTextItem` описывает простой текст. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsSimpleTextItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsSimpleTextItem([parent=None])
<Объект> = QGraphicsSimpleTextItem(<Текст>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsSimpleTextItem` наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicssimpletextitem.html**):

- `setText(<Текст>)` — задает текст;
- `text()` — возвращает текст;
- `setFont(<QFont>)` — задает шрифт;
- `font()` — возвращает описывающий заданный шрифт экземпляр класса `QFont`.

25.4.8. Форматированный текст

Класс `QGraphicsTextItem` описывает форматированный текст. Иерархия наследования:

```
(QObject, QGraphicsItem) — QGraphicsObject — QGraphicsTextItem
```

Форматы конструктора класса:

```
<Объект> = QGraphicsTextItem([parent=None])
<Объект> = QGraphicsTextItem(<Текст>[, parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsTextItem` наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicstextitem.html**):

- `setPlainText(<Простой текст>)` — задает простой текст;
- `toPlainText()` — возвращает простой текст;
- `setHtml(<HTML-код>)` — задает форматированный текст в виде HTML-кода;
- `toHtml()` — возвращает форматированный текст в виде HTML-кода;
- `setFont(<QFont>)` — устанавливает шрифт;
- `font()` — возвращает описывающий установленный шрифт экземпляр класса `QFont`;
- `setDefaultTextColor(<QColor>)` — задает цвет текста по умолчанию;
- `defaultTextColor()` — возвращает цвет текста по умолчанию;
- `setTextWidth(<Ширина>)` — задает предпочитаемую ширину строки. Если текст не помещается в установленную ширину, он будет перенесен на новую строку;
- `textWidth()` — возвращает предпочитаемую ширину текста;
- `setDocument(<QTextDocument>)` — задает объект документа в виде экземпляра класса `QTextDocument` (см. *разд. 21.6.4*);
- `document()` — возвращает ссылку на объект документа (экземпляр класса `QTextDocument`; см. *разд. 21.6.4*);
- `setTextCursor(<QTextCursor>)` — устанавливает объект курсора в виде экземпляра класса `QTextCursor` (см. *разд. 21.6.5*);
- `textCursor()` — возвращает объект курсора (экземпляр класса `QtextCursor` — см. *разд. 21.6.5*);
- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом. По умолчанию используется режим `NoTextInteraction`, при котором пользователь не может взаимодействовать с текстом. Допустимые режимы приведены в *разд. 21.1* (см. описание метода `setTextInteractionFlags()`);
- `setTabChangesFocus(<Флаг>)` — если в качестве параметра указано значение `False`, то нажатием клавиши <Tab> можно будет вставить в текст символ табуляции. Если указано значение `True`, клавиша <Tab> станет использоваться для передачи фокуса;
- `setOpenExternalLinks(<Флаг>)` — если в качестве параметра указано значение `True`, щелчок на гиперссылке приведет к открытию браузера, используемого в системе по умолчанию, и загрузке страницы с указанным в гиперссылке URL-адресом. Метод работает только при использовании режима `TextBrowserInteraction`.

Класс `QGraphicsTextItem` поддерживает следующие сигналы:

- ♦ `linkActivated(<URL>)` — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен URL-адрес гиперссылки в виде строки;
- ♦ `linkHovered(<URL>)` — генерируется при наведении указателя мыши на гиперссылку. Через параметр внутри обработчика доступен URL-адрес гиперссылки в виде строки.

25.5. Группировка объектов

Объединить несколько объектов в группу позволяет класс `QGraphicsItemGroup`. Над сгруппированными объектами можно выполнять различные преобразования, например перемещать или поворачивать их одновременно. Иерархия наследования для класса `QGraphicsItemGroup` выглядит так:

```
QGraphicsItem — QGraphicsItemGroup
```

Формат конструктора класса:

```
<Объект> = QGraphicsItemGroup([parent=None])
```

В параметре `parent` можно указать ссылку на родительский объект.

Класс `QGraphicsItemGroup` наследует все методы класса `QGraphicsItem` и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsitemgroup.html**):

- ♦ `addToGroup(<QGraphicsItem>)` — добавляет объект в группу;
- ♦ `removeFromGroup(<QGraphicsItem>)` — удаляет объект из группы.

Создать группу и добавить ее на сцену можно и с помощью метода `createItemGroup(<Список с объектами>)` класса `QGraphicsScene`. Метод возвращает ссылку на группу (экземпляр класса `QGraphicsItemGroup`). Удалить группу со сцены позволяет метод `destroyItemGroup(<QGraphicsItemGroup>)` класса `QGraphicsScene`.

Добавить объект в группу позволяет также метод `setGroup(<QGraphicsItemGroup>)` класса `QGraphicsItem`. Получить ссылку на группу, в которой находится объект, можно вызовом метода `group()` класса `QGraphicsItem`. Если объект не находится ни в какой группе, метод возвращает `None`.

25.6. Эффекты

К графическим объектам можно применить различные эффекты, например изменение прозрачности или цвета, отображение тени или размытие. Наследуя класс `QGraphicsEffect` и переопределяя в нем метод `draw()`, можно создать свой эффект.

Для установки эффекта и получения ссылки на него предназначены следующие методы класса `QGraphicsItem`:

- ♦ `setGraphicsEffect(<QGraphicsEffect>)` — задает эффект;
- ♦ `graphicsEffect()` — возвращает ссылку на эффект или значение `None`, если эффект не был установлен.

25.6.1. Класс *QGraphicsEffect*

Класс `QGraphicsEffect` является базовым классом для всех эффектов. Иерархия наследования выглядит так:

```
QObject — QGraphicsEffect
```

Формат конструктора класса:

```
QGraphicsEffect([parent=None])
```

Класс `QGraphicsEffect` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicseffect.html**):

- `draw(self, <QPainter>)` — собственно рисует эффект. Этот абстрактный метод должен быть переопределен в производных классах;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `False`, эффект отключается. Значение `True` разрешает использование эффекта. Метод является слотом;
- `isEnabled()` — возвращает значение `True`, если эффект включен, и `False` — в противном случае;
- `update()` — вызывает перерисовку эффекта. Метод является слотом.

Класс `QGraphicsEffect` поддерживает сигнал `enabledChanged(<Флаг>)`, который генерируется при изменении статуса эффекта. Внутри обработчика через параметр доступно значение `True`, если эффект включен, и `False` — в противном случае.

25.6.2. Тень

Класс `QGraphicsDropShadowEffect` реализует вывод тени у объекта. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsDropShadowEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsDropShadowEffect([parent=None])
```

Класс `QGraphicsDropShadowEffect` наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsdropshadoweffect.html**):

- `setColor(<QColor>)` — задает цвет тени. По умолчанию используется полупрозрачный темно-серый цвет (`QColor(63, 63, 63, 180)`). Метод является слотом;
- `color()` — возвращает цвет тени (экземпляр класса `QColor`);
- `setBlurRadius(<Значение>)` — задает радиус размытия тени в виде вещественного числа. Метод является слотом;
- `blurRadius()` — возвращает радиус размытия тени;
- `setOffset()` — задает смещение тени. Форматы метода:

  ```
  setOffset(<По оси X>, <По оси Y>)
  setOffset(<Смещение по обеим осям>)
  setOffset(<QPointF>)
  ```

  Второй конструктор задает смещение сразу по обеим осям координат. В первом и втором конструкторах параметры задаются вещественными числами. Метод является слотом;

- `offset()` — возвращает смещение тени в виде экземпляра класса `QPointF`;
- `setXOffset(<Смещение>)` — задает смещение по оси `X` в виде вещественного числа. Метод является слотом;
- `xOffset()` — возвращает смещение по оси `X`;
- `setYOffset(<Смещение>)` — задает смещение по оси `Y` в виде вещественного числа. Метод является слотом;
- `yOffset()` — возвращает смещение по оси `Y`.

Класс `QGraphicsDropShadowEffect` поддерживает сигналы:

- `colorChanged(<QColor>)` — генерируется при изменении цвета тени. Внутри обработчика через параметр доступен новый цвет;
- `blurRadiusChanged(<Радиус размытия>)` — генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение в виде вещественного числа;
- `offsetChanged(<QPointF>)` — генерируется при изменении смещения. Внутри обработчика через параметр доступно новое значение.

25.6.3. Размытие

Класс `QGraphicsBlurEffect` реализует эффект размытия. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsBlurEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsBlurEffect([parent=None])
```

Класс `QGraphicsBlurEffect` наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsblureffect.html**):

- `setBlurRadius(<Значение>)` — задает радиус размытия в виде вещественного числа. Метод является слотом;
- `blurRadius()` — возвращает радиус размытия.

Класс `QGraphicsBlurEffect` поддерживает сигнал `blurRadiusChanged(<Радиус размытия>)`, который генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение в виде вещественного числа.

25.6.4. Изменение цвета

Класс `QGraphicsColorizeEffect` реализует эффект изменения цвета. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsColorizeEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsColorizeEffect([parent=None])
```

Класс `QGraphicsColorizeEffect` наследует все методы из базовых классов и поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicscolorizeeffect.html**):

- `setColor(<QColor>)` — задает цвет. По умолчанию используется светло-синий цвет (`QColor(0, 0, 192, 255)`). Метод является слотом;

- `color()` — возвращает текущий цвет в виде экземпляра класса `QColor`;
- `setStrength(<Значение>)` — задает интенсивность цвета. В качестве значения указывается вещественное число от `0.0` до `1.0` (значение по умолчанию). Метод является слотом;
- `strength()` — возвращает интенсивность цвета.

Класс `QGraphicsColorizeEffect` поддерживает сигналы:

- `colorChanged(<QColor>)` — генерируется при изменении цвета. Внутри обработчика через параметр доступен новый цвет;
- `strengthChanged(<Интенсивность>)` — генерируется при изменении интенсивности цвета. Внутри обработчика через параметр доступно новое значение в виде вещественного числа.

25.6.5. Изменение прозрачности

Класс `QGraphicsOpacityEffect` реализует эффект прозрачности. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsOpacityEffect
```

Формат конструктора класса:

```
<Объект> = QGraphicsOpacityEffect([parent=None])
```

Класс `QGraphicsOpacityEffect` наследует все методы базовых классов и поддерживает следующие методы (здесь приведены только самые полезные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsopacityeffect.html**):

- `setOpacity(<Значение>)` — задает степень прозрачности. В качестве значения указывается вещественное число от `0.0` до `1.0`. По умолчанию используется значение `0.7`. Метод является слотом;
- `opacity()` — возвращает степень прозрачности;
- `setOpacityMask(<QBrush>)` — задает маску. Метод является слотом;
- `opacityMask()` — возвращает маску в виде экземпляра класса `QBrush`.

Класс `QGraphicsOpacityEffect` поддерживает следующие сигналы:

- `opacityChanged(<Прозрачность>)` — генерируется при изменении степени прозрачности. Внутри обработчика через параметр доступно новое значение в виде вещественного числа;
- `opacityMaskChanged(<QBrush>)` — генерируется при изменении маски. Через параметр доступна новая маска.

25.7. Обработка событий

Все происходящие в графических объектах события изначально возникают в компоненте-представлении. Компонент-представление преобразует их и передает объекту сцены, который, в свою очередь, перенаправляет их объекту, способному обработать возникшее событие (так, щелчок мыши передается объекту, который расположен по координатам щелчка).

Обработка событий в представлении ничем не отличается от обычной обработки событий, рассмотренной в *главе 19*. Однако обработка событий в графическом объекте имеет свои особенности, которые мы и рассмотрим в этом разделе.

25.7.1. События клавиатуры

При обработке событий клавиатуры следует учитывать, что:

- графический объект должен иметь возможность принимать фокус ввода. Для этого необходимо установить флаг `ItemIsFocusable`, например, с помощью метода `setFlag()` класса `QGraphicsItem`;
- объект должен быть в фокусе ввода. Методы, позволяющие управлять фокусом ввода, мы рассматривали в *разд. 25.1.5* и *25.3.1*;
- чтобы захватить эксклюзивный ввод с клавиатуры, следует вызвать у графического объекта метод `grabKeyboard()`, а чтобы освободить ввод — метод `ungrabKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме <Tab> и <Shift>+<Tab>, используемых для передачи фокуса;
- если событие обработано, нужно вызвать метод `accept()` у объекта события, в противном случае — метод `ignore()`.

Для обработки событий клавиатуры следует наследовать класс, реализующий графический объект, и переопределить в нем методы:

- `focusInEvent(self, <event>)` — вызывается при получении фокуса ввода. Через параметр `<event>` доступен экземпляр класса `QFocusEvent` (см. *разд. 19.8.1*);
- `focusOutEvent(self, <event>)` — вызывается при потере фокуса ввода. Через параметр `<event>` доступен экземпляр класса `QFocusEvent` (см. *разд. 19.8.1*);
- `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши на клавиатуре. Если клавишу удерживать нажатой, метод будет вызываться постоянно, пока ее не отпустят. Через параметр `<event>` доступен экземпляр класса `QKeyEvent` (см. *разд. 19.8.3*);
- `keyReleaseEvent(self, <event>)` — вызывается при отпускании нажатой ранее клавиши. Через параметр `<event>` доступен экземпляр класса `QKeyEvent` (см. *разд. 19.8.3*).

С помощью метода `setFocusProxy(<QGraphicsItem>)` класса `QGraphicsItem` можно указать объект, который будет обрабатывать события клавиатуры вместо текущего объекта. Получить ссылку на назначенный ранее объект-обработчик событий клавиатуры позволяет метод `focusProxy()`.

25.7.2. События мыши

Для обработки нажатия кнопки мыши и перемещения мыши следует наследовать класс, реализующий графический объект, и переопределить в нем такие методы:

- `mousePressEvent(self, <event>)` — вызывается при нажатии кнопки мыши над объектом. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneMouseEvent`. Если событие принято, необходимо вызвать метод `accept()` объекта события, в противном случае — метод `ignore()`. Если был вызван метод `ignore()`, методы `mouseReleaseEvent()` и `mouseMoveEvent()` выполнены не будут.

  С помощью метода `setAcceptedMouseButtons(<Кнопки>)` класса `QGraphicsItem` можно указать кнопки, события от которых объект будет принимать. По умолчанию объект принимает события от всех кнопок мыши. Если в параметре `<Кнопки>` указать атрибут `NoButton` класса `QtCore.Qt`, объект вообще не станет принимать события от кнопок мыши;
- `mouseReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой кнопки мыши. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneMouseEvent`;

- `mouseDoubleClickEvent(self, <event>)` — вызывается при двойном щелчке мышью в области объекта. Через параметр `<event>` доступен экземпляр класса `QgraphicsSceneMouseEvent`;
- `mouseMoveEvent(self, <event>)` — вызывается при перемещении мыши. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneMouseEvent`.

Класс `QGraphicsSceneMouseEvent` наследует все методы классов `QGraphicsSceneEvent` и `QEvent` и добавляет поддержку следующих методов:

- `pos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах области объекта;
- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах сцены;
- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши в пределах экрана;
- `lastPos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции мыши в пределах области объекта;
- `lastScenePos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции мыши в пределах сцены;
- `lastScreenPos()` — возвращает экземпляр класса `QPoint` с координатами последней запомненной представлением позиции мыши в пределах экрана;
- `buttonDownPos(<Кнопка>)` — возвращает экземпляр класса `QPointF` с координатами щелчка указанной кнопки мыши в пределах области объекта;
- `buttonDownScenePos(<Кнопка>)` — возвращает экземпляр класса `QPointF` с координатами щелчка указанной кнопки мыши в пределах сцены;
- `buttonDownScreenPos(<Кнопка>)` — возвращает экземпляр класса `QPoint` с координатами щелчка указанной кнопки мыши в пределах экрана;
- `button()` — возвращает обозначение кнопки мыши, нажатие которой вызвало событие;
- `buttons()` — возвращает комбинацию обозначений всех кнопок мыши, одновременное нажатие которых вызвало событие;
- `modifiers()` — возвращает комбинацию обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты вместе с кнопкой мыши.

По умолчанию событие мыши перехватывает объект, на котором был произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне объекта, следует захватить мышь с помощью метода `grabMouse()` класса `QGraphicsItem`. Освободить захваченную ранее мышь позволяет метод `ungrabMouse()`. Получить ссылку на объект, захвативший мышь, можно с помощью метода `mouseGrabberItem()` класса `QGraphicsScene`.

Для обработки прочих событий мыши нужно наследовать класс, реализующий графический объект, и переопределить следующие методы:

- `hoverEnterEvent(self, <event>)` — вызывается при наведении указателя мыши на область объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneHoverEvent`;
- `hoverLeaveEvent(self, <event>)` — вызывается, когда указатель мыши покидает область объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneHoverEvent`;

- hoverMoveEvent(self, <event>) — вызывается при перемещении указателя мыши внутри области объекта. Через параметр <event> доступен экземпляр класса QGraphicsSceneHoverEvent;
- wheelEvent(self, <event>) — вызывается при повороте колесика мыши при нахождении указателя мыши над объектом. Чтобы обрабатывать событие, в любом случае следует захватить мышь. Через параметр <event> доступен экземпляр класса QGraphicsSceneWheelEvent.

Следует учитывать, что методы hoverEnterEvent(), hoverLeaveEvent() и hoverMoveEvent() будут вызваны только в том случае, если обработка этих событий разрешена. Чтобы разрешить обработку событий перемещения мыши, следует вызвать метод setAcceptHoverEvents(<Флаг>) класса QGraphicsItem и передать ему значение True. Значение False запрещает обработку событий перемещения указателя. Получить текущее состояние позволяет метод acceptHoverEvents().

Класс QGraphicsSceneHoverEvent наследует все методы классов QGraphicsSceneEvent и QEvent и добавляет поддержку своих методов:

- pos() — возвращает экземпляр класса QPointF с координатами указателя мыши в пределах области объекта;
- scenePos() — возвращает экземпляр класса QPointF с координатами указателя мыши в пределах сцены;
- screenPos() — возвращает экземпляр класса QPoint с координатами указателя мыши в пределах экрана;
- lastPos() — возвращает экземпляр класса QPointF с координатами последней запомненной представлением позиции мыши в пределах области объекта;
- lastScenePos() — возвращает экземпляр класса QPointF с координатами последней запомненной представлением позиции мыши в пределах сцены;
- lastScreenPos() — возвращает экземпляр класса QPoint с координатами последней запомненной представлением позиции мыши в пределах экрана;
- modifiers() — возвращает комбинацию обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты одновременно с перемещением мыши.

Класс QGraphicsSceneWheelEvent наследует все методы из классов QGraphicsSceneEvent и QEvent и добавляет поддержку следующих методов:

- delta() — возвращает расстояние поворота колесика, измеряемое в $^1/_8$°. Положительное значение означает, что колесико поворачивалось в направлении от пользователя, отрицательное — к пользователю;
- orientation() — возвращает направление вращения колесика в виде значения одного из следующих атрибутов класса QtCore.Qt:
  - Horizontal — 1 — по горизонтали;
  - Vertical — 2 — по вертикали;
- pos() — возвращает экземпляр класса QPointF с координатами указателя мыши в пределах области объекта;
- scenePos() — возвращает экземпляр класса QPointF с координатами указателя мыши в пределах сцены;

- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши в пределах экрана;
- `buttons()` — возвращает комбинацию обозначений всех кнопок мыши, нажатых одновременно с вращением колесика;
- `modifiers()` — возвращает комбинацию обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты одновременно с вращением колесика.

25.7.3. Обработка перетаскивания и сброса

Прежде чем обрабатывать перетаскивание и сброс, необходимо сообщить системе, что графический объект может их обработать. Для этого следует вызвать метод `setAcceptDrops()` класса `QGraphicsItem` и передать ему значение `True`.

Обработка перетаскивания и сброса в графическом объекте выполняется следующим образом:

- внутри метода `dragEnterEvent()` проверяется MIME-тип перетаскиваемых данных и действие. Если графический объект способен обработать сброс этих данных и соглашается с предложенным действием, необходимо вызвать метод `acceptProposedAction()` объекта события. Если нужно изменить действие, методу `setDropAction()` объекта события передается новое действие, а затем у того же объекта вызывается метод `accept()` вместо метода `acceptProposedAction()`;
- если необходимо ограничить область сброса некоторым участком графического объекта, можно дополнительно определить в нем метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области графического объекта. При согласии со сбрасыванием следует вызвать у объекта события метод `accept()`;
- внутри метода `dropEvent()` производится обработка сброса.

Обработать события, возникающие при перетаскивании и сбрасывании объектов, позволяют следующие методы:

- `dragEnterEvent(self, <event>)` — вызывается, когда перетаскиваемый объект входит в область графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`;
- `dragLeaveEvent(self, <event>)` — вызывается, когда перетаскиваемый объект покидает область графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`;
- `dragMoveEvent(self, <event>)` — вызывается при перетаскивании объекта внутри области графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`;
- `dropEvent(self, <event>)` — вызывается при сбрасывании объекта в области графического объекта. Через параметр `<event>` доступен экземпляр класса `QGraphicsSceneDragDropEvent`.

Класс `QGraphicsSceneDragDropEvent` наследует все методы классов `QGraphicsSceneEvent` и `QEvent` и добавляет поддержку методов:

- `mimeData()` — возвращает экземпляр класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе;

- `pos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах области объекта;
- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах сцены;
- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши в пределах экрана;
- `possibleActions()` — возвращает комбинацию возможных действий при сбрасывании;
- `proposedAction()` — возвращает действие по умолчанию при сбрасывании;
- `acceptProposedAction()` — подтверждает готовность принять перемещаемые данные и согласие с действием по умолчанию, возвращаемым методом `proposedAction()`;
- `setDropAction(<Действие>)` — указывает другое действие при сбрасывании. После изменения действия следует вызвать метод `accept()`, а не `acceptProposedAction()`;
- `dropAction()` — возвращает действие, которое должно быть выполнено при сбрасывании;
- `buttons()` — возвращает комбинацию обозначений всех кнопок мыши, нажатых в процессе перетаскивания;
- `modifiers()` — возвращает комбинацию обозначений всех клавиш-модификаторов (<Shift>, <Ctrl>, <Alt> и др.), что были нажаты в процессе перетаскивания;
- `source()` — возвращает ссылку на источник события или значение `None`.

25.7.4. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы специализированному методу. Для этого в классе графического объекта необходимо переопределить метод `sceneEvent(self, <event>)`. Через параметр `<event>` здесь будет доступен объект с дополнительной информацией о событии. Тип этого объекта различен для разных типов событий. Внутри метода следует вернуть значение `True`, если событие обработано, и `False` — в противном случае. Если вернуть значение `True`, специализированный метод (например, `mousePressEvent()`) выполняться не будет.

Чтобы произвести фильтрацию событий какого-либо объекта, в классе графического объекта необходимо переопределить метод `sceneEventFilter(self, <QGraphicsItem>, <event>)`. Через параметр `<QGraphicsItem>` здесь будет доступна ссылка на объект, в котором возникло событие, а через параметр `<event>` — объект с информацией о самом событии. Тип этого объекта различен для разных типов событий. Внутри метода следует вернуть значение `True`, если событие обработано, и `False` — в противном случае. Если вернуть значение `True`, объект, в котором возникло событие, не получит его.

Указать, события какого объекта фильтруются, позволяют следующие методы класса `QGraphicsItem`:

- `installSceneEventFilter(<QGraphicsItem>)` — задает объект, который будет производить фильтрацию событий текущего объекта;
- `removeSceneEventFilter(<QGraphicsItem>)` — удаляет объект-фильтр событий;
- `setFiltersChildEvents(<Флаг>)` — если в качестве параметра указано значение `True`, текущий объект будет производить фильтрацию событий всех своих дочерних объектов.

25.7.5. Обработка изменения состояния объекта

Чтобы обработать изменение состояния объекта, следует переопределить метод `itemChange(self, <Состояние>, <Значение>)` в классе графического объекта. Метод должен возвращать новое значение. Через параметр `<Состояние>` доступно состояние, которое было изменено, в виде значения одного из следующих атрибутов класса `QGraphicsItem` (здесь перечислены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qgraphicsitem.html#GraphicsItemChange-enum**):

- `ItemEnabledChange` — 3 — изменилось состояние доступности;
- `ItemEnabledHasChanged` — 13 — изменилось состояние доступности. Возвращаемое значение игнорируется;
- `ItemPositionChange` — 0 — изменилась позиция объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `ItemPositionHasChanged` — 9 — изменилась позиция объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;
- `ItemScenePositionHasChanged` — 27 — изменилась позиция объекта на сцене с учетом преобразований, примененных к самому объекту и его родителям. Метод будет вызван, только если установлен флаг `ItemSendsScenePositionChanges`. Возвращаемое значение игнорируется;
- `ItemTransformChange` — 8 — изменилась матрица преобразований. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `ItemTransformHasChanged` — 10 — изменилась матрица преобразований. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;
- `ItemSelectedChange` — 4 — изменилось выделение объекта;
- `ItemSelectedHasChanged` — 14 — изменилось выделение объекта. Возвращаемое значение игнорируется;
- `ItemVisibleChange` — 2 — изменилось состояние видимости объекта;
- `ItemVisibleHasChanged` — 12 — изменилось состояние видимости объекта. Возвращаемое значение игнорируется;
- `ItemCursorChange` — 17 — изменился курсор;
- `ItemCursorHasChanged` — 18 — изменился курсор. Возвращаемое значение игнорируется;
- `ItemToolTipChange` — 19 — изменилась всплывающая подсказка;
- `ItemToolTipHasChanged` — 20 — изменилась всплывающая подсказка. Возвращаемое значение игнорируется;
- `ItemFlagsChange` — 21 — изменились флаги;
- `ItemFlagsHaveChanged` — 22 — изменились флаги. Возвращаемое значение игнорируется;
- `ItemZValueChange` — 23 — изменилось положение по оси `Z`;
- `ItemZValueHasChanged` — 24 — изменилось положение по оси `Z`. Возвращаемое значение игнорируется;
- `ItemOpacityChange` — 25 — изменилась прозрачность объекта;

- `ItemOpacityHasChanged` — `26` — изменилась прозрачность объекта. Возвращаемое значение игнорируется;
- `ItemParentChange` — `5` — изменился родитель объекта;
- `ItemParentHasChanged` — `15` — изменился родитель объекта. Возвращаемое значение игнорируется;
- `ItemChildAddedChange` — `6` — в объект был добавлен потомок;
- `ItemChildRemovedChange` — `7` — из объекта был удален потомок;
- `ItemSceneChange` — `11` — объект был перемещен на другую сцену;
- `ItemSceneHasChanged` — `16` — объект был перемещен на другую сцену. Возвращаемое значение игнорируется.

Внимание!

Вызов некоторых методов из метода `itemChange()` может привести к рекурсии. За подробной информацией обращайтесь к документации по классу `QGraphicsItem`.

Примечание

PyQt 5 также поддерживает помещение на сцену видеозаписей в качестве отдельных графических объектов и создание анимации. За подробным описанием обращайтесь к документации по этой библиотеке.

ГЛАВА 26

Диалоговые окна

Диалоговые окна предназначены для информирования пользователя и получения от него требуемых данных. В большинстве случаев окна подобного рода являются модальными (т. е. блокирующими все окна приложения или только родительское окно) и отображаются на непродолжительный промежуток времени. Для работы с диалоговыми окнами в PyQt предназначен класс `QDialog`, который предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус его завершения и выполнить прочие задачи. Класс `QDialog` наследуют другие классы, которые реализуют готовые диалоговые окна. Например, класс `QMessageBox` предоставляет диалоговые окна для вывода сообщений, класс `QInputDialog` — для ввода данных, класс `QFileDialog` — для выбора каталога или файла и т. д.

Все рассмотренные в этой главе классы определены в модуле `QtWidgets`, если не указано иное.

26.1. Пользовательские диалоговые окна

Класс `QDialog` реализует диалоговое окно. По умолчанию окно выводится с рамкой и заголовком, в котором расположены кнопки **Справка** и **Закрыть**. Размеры окна можно изменять с помощью мыши. Иерархия наследования для класса `QDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog
```

Конструктор класса `QDialog` имеет следующий формат:

```
<Объект> = QDialog([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительское окно. Если родитель не указан или имеет значение `None`, диалоговое окно будет центрироваться относительно экрана, если указана — относительно родительского окна (это также позволяет создать модальное диалоговое окно, которое будет блокировать только окно родителя, а не все окна приложения). Какие именно значения можно указать в параметре `flags`, мы уже рассматривали в *разд. 18.2*. Тип окна по умолчанию — `Dialog`.

Класс `QDialog` наследует все методы базовых классов и дополнительно реализует следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qdialog.html**):

- ♦ `exec()` — отображает модальное диалоговое окно, дожидается закрытия окна и возвращает код возврата в виде значения следующих атрибутов класса `QDialog`:

- `Accepted` — `1` — нажата кнопка **OK**;
- `Rejected` — `0` — нажата кнопка **Cancel**, кнопка **Закрыть** в заголовке окна или клавиша <Esc>.

Метод является слотом. Вместо него можно использовать метод `exec_()`, предусмотренный для совместимости с предыдущими версиями PyQt.

Вот пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки (класс `MyDialog` является наследником класса `QDialog`, а `window` — ссылка на главное окно):

```
def on_clicked():
    dialog = MyDialog(window)
    result = dialog.exec_()
    if result == QtWidgets.QDialog.Accepted:
        print("Нажата кнопка OK")
        # Здесь получаем данные из диалогового окна
    else:
        print("Нажата кнопка Cancel")
```

- `accept()` — закрывает модальное диалоговое окно и устанавливает код возврата равным значению атрибута `Accepted` класса `QDialog`. Метод является слотом. Обычно его соединяют с сигналом нажатия кнопки **OK**:

```
self.btnOK.clicked.connect(self.accept)
```

- `reject()` — закрывает модальное диалоговое окно и устанавливает код возврата равным значению атрибута `Rejected` класса `QDialog`. Метод является слотом. Обычно его соединяют с сигналом нажатия кнопки **Cancel**:

```
self.btnCancel.clicked.connect(self.reject)
```

- `done(<Код возврата>)` — закрывает модальное диалоговое окно и устанавливает код возврата равным значению параметра. Метод является слотом;
- `setResult(<Код возврата>)` — устанавливает код возврата;
- `result()` — возвращает код завершения;
- `setSizeGripEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то в правом нижнем углу диалогового окна будет отображен значок изменения размера, а если `False` — то скрыт (значение по умолчанию);
- `isSizeGripEnabled()` — возвращает значение `True`, если значок изменения размера отображается в правом нижнем углу диалогового окна, и `False` — в противном случае;
- `setVisible(<Флаг>)` — если в параметре указано значение `True`, диалоговое окно будет отображено, а если значение `False` — то скрыто. Вместо этого метода можно пользоваться более удобными методами `show()` и `hide()`;
- `open()` — отображает диалоговое окно в модальном режиме, но не дожидается его закрытия. Блокируется только родительское окно, а не все окна приложения. Метод является слотом;
- `setModal(<Флаг>)` — если в качестве параметра указано значение `True`, окно будет модальным, а если `False` — обычным. Обратите внимание, что окно, открываемое с помощью метода `exec()`, всегда будет модальным, — независимо от значения, заданного вызовом метода `setModal()`. Чтобы диалоговое окно было не модальным, нужно отображать его с помощью метода `show()` или `setVisible()`. После вызова этих методов

следует вызвать методы `raise_()` (чтобы поместить окно поверх всех окон) и `activateWindow()` (чтобы сделать окно активным, т. е. имеющим фокус ввода).

Указать, что окно является модальным, позволяет также метод `setWindowModality(<Флаг>)` класса `QWidget`. В качестве параметра могут быть указаны следующие атрибуты класса `QtCore.Qt`:

- `NonModal` — `0` — окно не является модальным;
- `WindowModal` — `1` — окно блокирует только родительские окна в пределах иерархии;
- `ApplicationModal` — `2` — окно блокирует все окна в приложении.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение позволяет метод `windowModality()` класса `QWidget`. Проверить, является ли окно модальным, можно с помощью метода `isModal()` того же класса, который возвращает `True`, если окно является модальным, и `False` — в противном случае.

Класс `QDialog` поддерживает следующие сигналы:

- `accepted()` — генерируется при установке флага `Accepted` (нажата кнопка **OK**). Не генерируется при закрытии окна с помощью метода `hide()` или `setVisible()`;
- `rejected()` — генерируется при установке флага `Rejected` (нажата кнопка **Cancel**, кнопка **Закрыть** в заголовке окна или клавиша <Esc>). Не генерируется при закрытии окна с помощью метода `hide()` или `setVisible()`;
- `finished(<Код завершения>)` — генерируется при установке кода завершения пользователем или вызовом методов `accept()`, `reject()` и `done()`. Внутри обработчика через параметр доступен целочисленный код завершения. Сигнал не генерируется при закрытии окна с помощью метода `hide()` или `setVisible()`.

Для всех кнопок, добавляемых в диалоговое окно, автоматически вызывается метод `setAutoDefault()` со значением `True` в качестве параметра. В этом случае кнопка может быть нажата с помощью клавиши <Enter> при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>.

С помощью метода `setDefault()` можно указать кнопку по умолчанию. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент — например, на текстовое поле.

26.2. Класс *QDialogButtonBox*

Класс `QDialogButtonBox` представляет контейнер, в который можно добавить различные кнопки: как стандартные, так и пользовательские. Внешний вид контейнера и расположение кнопок в нем зависят от используемой операционной системы. Иерархия наследования для класса `QDialogButtonBox`:

```
(QObject, QPaintDevice) — QWidget — QDialogButtonBox
```

Форматы конструктора класса `QDialogButtonBox`:

```
<Объект> = QDialogButtonBox([parent=None])
<Объект> = QDialogButtonBox(<Ориентация>[, parent=None])
<Объект> = QDialogButtonBox(<Стандартные кнопки>[, parent=None])
<Объект> = QDialogButtonBox(<Стандартные кнопки>, <Ориентация>[, parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент. Параметр `<Ориентация>` задает порядок расположения кнопок внутри контейнера. В качестве значения указываются атрибуты `Horizontal` (по горизонтали — значение по умолчанию) или `Vertical` (по вертикали) класса `QtCore.Qt`. В параметре `<Стандартные кнопки>` указываются следующие атрибуты (или их комбинация через оператор `|`) класса `QDialogButtonBox`:

- `NoButton` — кнопки не установлены;
- `Ok` — кнопка **OK** с ролью `AcceptRole`;
- `Cancel` — кнопка **Cancel** с ролью `RejectRole`;
- `Yes` — кнопка **Yes** с ролью `YesRole`;
- `YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `No` — кнопка **No** с ролью `NoRole`;
- `NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `Open` — кнопка **Open** с ролью `AcceptRole`;
- `Close` — кнопка **Close** с ролью `RejectRole`;
- `Save` — кнопка **Save** с ролью `AcceptRole`;
- `SaveAll` — кнопка **Save All** с ролью `AcceptRole`;
- `Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `Reset` — кнопка **Reset** с ролью `ResetRole`;
- `RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `Help` — кнопка **Help** с ролью `HelpRole`;
- `Abort` — кнопка **Abort** с ролью `RejectRole`;
- `Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

Класс `QDialogButtonBox` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-5/qdialogbuttonbox.html**):

- `setOrientation(<Ориентация>)` — задает порядок расположения кнопок внутри контейнера. В качестве значения указываются атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) класса `QtCore.Qt`;
- `setStandardButtons(<Стандартные кнопки>)` — добавляет в контейнер стандартные кнопки:

```
self.box.setStandardButtons(QtWidgets.QDialogButtonBox.Ok |
                            QtWidgets.QDialogButtonBox.Cancel)
```

- `addButton()` — добавляет кнопку в контейнер. Форматы метода:

```
addButton(<Стандартная кнопка>)
addButton(<Текст>, <Роль>)
addButton(<QAbstractButton>, <Роль>)
```

Первый формат добавляет стандартную кнопку или кнопки (значение параметра должно представлять собой один из рассмотренных ранее атрибутов класса `QDialogButtonBox` или их комбинацию через оператор `|`). Второй формат принимает в качестве первого параметра надпись для добавляемой кнопки, а в качестве второго — ее роль. Третий формат принимает добавляемую кнопку в виде экземпляра одного из подклассов класса `QabstractButton` — как правило, класса `QPushButton`, представляющего обычную кнопку.

В качестве роли указывается один из следующих атрибутов класса `QDialogButtonBox`:

- `InvalidRole` — `-1` — ошибочная роль;
- `AcceptRole` — `0` — нажатие кнопки устанавливает код возврата равным значению атрибута `Accepted`;
- `RejectRole` — `1` — нажатие кнопки устанавливает код возврата равным значению атрибута `Rejected`;
- `DestructiveRole` — `2` — кнопка для отказа от изменений;
- `ActionRole` — `3` — нажатие кнопки приводит к выполнению операции, которая не связана с закрытием окна;
- `HelpRole` — `4` — кнопка для отображения справки;
- `YesRole` — `5` — кнопка **Yes**;
- `NoRole` — `6` — кнопка **No**;
- `ResetRole` — `7` — кнопка для установки значений по умолчанию;
- `ApplyRole` — `8` — кнопка для принятия изменений.

Если роль недействительна, кнопка добавлена не будет.

Первый и второй форматы возвращают ссылку на сгенерированную и добавленную в контейнер кнопку, третий не возвращает ничего:

```
self.btnYes = QtWidget.QPushButton("&Да")
self.box.addButton(self.btnYes,
                   QtWidget.QDialogButtonBox.AcceptRole)
self.btnNo = self.box.addButton(QtWidget.QDialogButtonBox.No)
self.btnCancel = self.box.addButton("&Cancel",
                   QtWidget.QDialogButtonBox.RejectRole)
```

- `button(<Стандартная кнопка>)` — возвращает ссылку на стандартную кнопку, соответствующую указанному обозначению, или `None`, если такой стандартной кнопки в контейнере нет;
- `standardButton(<QAbstractButton>)` — возвращает обозначение стандартной кнопки, переданной в качестве параметра, или значение атрибута `NoButton`, если такой кнопки в контейнере нет;
- `standardButtons()` — возвращает комбинацию обозначений всех стандартных кнопок, добавленных в контейнер;
- `buttonRole(<QAbstractButton>)` — возвращает роль указанной в параметре кнопки. Если такая кнопка отсутствует, возвращается значение атрибута `InvalidRole`;
- `buttons()` — возвращает список со ссылками на все кнопки, которые были добавлены в контейнер;

- removeButton(<QAbstractButton>) — удаляет кнопку из контейнера, при этом не удаляя объект кнопки;
- clear() — очищает контейнер и удаляет все кнопки;
- setCenterButtons(<Флаг>) — если в качестве параметра указано значение True, кнопки будут выравниваться по центру контейнера.

Класс QDialogButtonBox поддерживает следующие сигналы:

- accepted() — генерируется при нажатии кнопки с ролью AcceptRole или YesRole. Этот сигнал можно соединить со слотом accept() объекта диалогового окна:

```
self.box.accepted.connect(self.accept)
```

- rejected() — генерируется при нажатии кнопки с ролью RejectRole или NoRole. Этот сигнал можно соединить со слотом reject() объекта диалогового окна:

```
self.box.rejected.connect(self.reject)
```

- helpRequested() — генерируется при нажатии кнопки с ролью HelpRole;
- clicked(<QAbstractButton>) — генерируется при нажатии любой кнопки внутри контейнера. Внутри обработчика через параметр доступна ссылка на кнопку.

26.3. Класс *QMessageBox*

Класс QMessageBox реализует стандартные окна-предупреждения для вывода сообщений. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QDialog — QMessageBox
```

Форматы конструктора класса QMessageBox:

```
<Объект> = QMessageBox([parent=None])
<Объект> = QMessageBox(<Значок>, <Текст заголовка>, <Текст сообщения>[,
                       buttons=NoButton][, parent=None][,
                       flags=Dialog | MSWindowsFixedSizeDialogHint])
```

Если в параметре parent указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр flags задает тип окна (см. *разд. 18.2*). В параметре <Значок> могут быть указаны следующие атрибуты класса QMessageBox:

- NoIcon — 0 — нет значка;
- Question — 4 — значок со знаком вопроса;
- Information — 1 — значок информационного сообщения;
- Warning — 2 — значок предупреждающего сообщения;
- Critical — 3 — значок критического сообщения.

В параметре buttons указываются следующие атрибуты (или их комбинация через оператор |) класса QMessageBox:

- NoButton — кнопки не установлены;
- Ok — кнопка **OK** с ролью AcceptRole;
- Cancel — кнопка **Cancel** с ролью RejectRole;
- Yes — кнопка **Yes** с ролью YesRole;

- `YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `No` — кнопка **No** с ролью `NoRole`;
- `NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `Open` — кнопка **Open** с ролью `AcceptRole`;
- `Close` — кнопка **Close** с ролью `RejectRole`;
- `Save` — кнопка **Save** с ролью `AcceptRole`;
- `SaveAll` — кнопка **Save All** с ролью `AcceptRole`;
- `Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `Reset` — кнопка **Reset** с ролью `ResetRole`;
- `RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `Help` — кнопка **Help** с ролью `HelpRole`;
- `Abort` — кнопка **Abort** с ролью `RejectRole`;
- `Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

После создания экземпляра класса следует вызвать метод `exec()` (или оставленный для совместимости со старыми версиями PyQt метод `exec_()`), чтобы вывести окно на экран. Метод возвращает числовое обозначение нажатой кнопки:

```
dialog = QtWidgets.QMessageBox(QtWidgets.QMessageBox.Critical,
                               "Текст заголовка", "Текст сообщения",
                               buttons = QtWidgets.QMessageBox.Ok |
                               QtWidgets.QMessageBox.Cancel,
                               parent=window)
result = dialog.exec()
```

26.3.1. Основные методы и сигналы

Класс `QMessageBox` наследует все методы базовых классов и поддерживает следующие дополнительные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qmessagebox.html**):

- `setIcon(<Значок>)` — устанавливает стандартный значок;
- `setIconPixmap(<QPixmap>)` — устанавливает пользовательский значок. В качестве параметра указывается экземпляр класса `QPixmap`;
- `setWindowTitle(<Текст заголовка>)` — задает текст заголовка окна;
- `setText(<Текст сообщения>)` — задает текст сообщения. Можно указать как обычный текст, так и текст в формате HTML. Перенос строки в обычной строке осуществляется с помощью символа `\n`, а в строке в формате HTML — с помощью тега `<br>`;
- `setInformativeText(<Текст>)` — задает дополнительный текст сообщения, который отображается под обычным текстом сообщения. Можно указать как обычный текст, так и HTML-код;

- `setDetailedText(<Текст>)` — задает текст подробного описания сообщения. Если он задан, в окно будет добавлена кнопка **Show Details**, с помощью которой можно отобразить скрытую панель с подробным описанием;
- `setTextFormat(<Режим>)` — задает режим отображения текста сообщения. Могут быть указаны следующие атрибуты класса `QtCore.Qt`:
  - `PlainText` — `0` — простой текст;
  - `RichText` — `1` — форматированный текст;
  - `AutoText` — `2` — автоматическое определение (режим по умолчанию). Если текст содержит HTML-теги, используется режим `RichText`, в противном случае — режим `PlainText`;
- `setStandardButtons(<Стандартные кнопки>)` — добавляет стандартные кнопки:

```
dialog.setStandardButtons(QtWidgets.QMessageBox.Ok |
                          QtWidgets.QMessageBox.Cancel)
```

- `addButton()` — добавляет кнопку в окно. Форматы метода:

```
addButton(<Стандартная кнопка>)
addButton(<Текст>, <Роль>)
addButton(<QAbstractButton>, <Роль>)
```

  Первый формат добавляет стандартную кнопку или кнопки (значение параметра должно быть одним из рассмотренных ранее атрибутов класса `QMessageBox` или их комбинацией через оператор `|`). Второй формат принимает в качестве первого параметра надпись для добавляемой кнопки, а в качестве второго — ее роль. Третий формат принимает добавляемую кнопку в виде экземпляра одного из подклассов класса `QAbstractButton` — как правило, класса `QPushButton`, представляющего обычную кнопку.

  В качестве роли указывается один из следующих атрибутов класса `QMessageBox`:
  - `InvalidRole` — `-1` — ошибочная роль;
  - `AcceptRole` — `0` — нажатие кнопки устанавливает код возврата равным значению атрибута `Accepted`;
  - `RejectRole` — `1` — нажатие кнопки устанавливает код возврата равным значению атрибута `Rejected`;
  - `DestructiveRole` — `2` — кнопка для отказа от изменений;
  - `ActionRole` — `3` — нажатие кнопки приводит к выполнению операции, которая не связана с закрытием окна;
  - `HelpRole` — `4` — кнопка для отображения справки;
  - `YesRole` — `5` — кнопка **Yes**;
  - `NoRole` — `6` — кнопка **No**;
  - `ResetRole` — `7` — кнопка для установки значений по умолчанию;
  - `ApplyRole` — `8` — кнопка для принятия изменений.

  Если роль недействительна, кнопка добавлена не будет.

  Первый и второй форматы возвращают ссылку на сгенерированную и добавленную в окно кнопку, третий не возвращает ничего:

```
btnYes = QtWidgets.QPushButton("&Да")
dialog.addButton(btnYes, QtWidgets.QMessageBox.AcceptRole)
btnNo = dialog.addButton("&Нет", QtWidgets.QMessageBox.RejectRole)
btnCancel = dialog.addButton(QtWidgets.QMessageBox.Cancel)
```

- `setDefaultButton()` — задает кнопку по умолчанию, которая сработает при нажатии клавиши <Enter>. Форматы метода:

  ```
  setDefaultButton(<Стандартная кнопка>)
  setDefaultButton(<QPushButton>)
  ```

- `setEscapeButton()` — задает кнопку, которая сработает при нажатии клавиши <Esc>. Форматы метода:

  ```
  setEscapeButton(<Стандартная кнопка>)
  setEscapeButton(<QAbstractButton>)
  ```

- `setCheckBox(<QCheckBox>)` — задает флажок, который будет выводиться в окне. Чтобы убрать заданный ранее флажок, нужно передать значение `None`;
- `checkBox()` — возвращает заданный в методе `setCheckBox()` флажок или `None`, если такового нет;
- `clickedButton()` — возвращает ссылку на кнопку, которая была нажата, или значение `None`;
- `button(<Стандартная кнопка>)` — возвращает ссылку на стандартную кнопку, соответствующую указанному обозначению, или `None`, если такой стандартной кнопки в контейнере нет;
- `standardButton(<QAbstractButton>)` — возвращает обозначение стандартной кнопки, переданной в качестве параметра, или значение атрибута `NoButton`, если такой кнопки в контейнере нет;
- `standardButtons()` — возвращает комбинацию обозначений всех стандартных кнопок, добавленных в окно;
- `buttonRole(<QAbstractButton>)` — возвращает роль указанной в параметре кнопки. Если такая кнопка отсутствует, метод возвращает значение атрибута `InvalidRole`;
- `buttons()` — возвращает список со ссылками на все кнопки, которые были добавлены в окно;
- `removeButton(<QAbstractButton>)` — удаляет кнопку из окна, при этом не удаляя объект кнопки.

Класс `QMessageBox` поддерживает сигнал `buttonClicked(<QAbstractButton>)`, генерируемый при нажатии кнопки в окне. Внутри обработчика через параметр доступна ссылка на нажатую кнопку.

26.3.2. Окно информационного сообщения

Помимо рассмотренных ранее, класс `QMessageBox` предлагает несколько статических методов, выводящих типовые окна-предупреждения.

Для вывода информационного сообщения предназначен статический метод `information()`. Формат метода:

```
information(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
            buttons=Ok][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию выводится кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `information()` возвращает числовое обозначение нажатой кнопки:

```
QtWidgets.QMessageBox.information(window, "Текст заголовка",
                                  "Текст сообщения",
                                  buttons=QtWidgets.QMessageBox.Close,
                                  defaultButton=QtWidgets.QMessageBox.Close)
```

Результат выполнения этого кода показан на рис. 26.1.

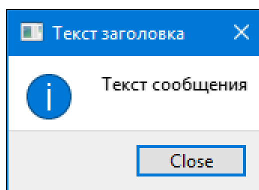

Рис. 26.1. Информационное окно-предупреждение

26.3.3. Окно подтверждения

Для вывода окна с запросом подтверждения каких-либо действий предназначен статический метод `question()`. Формат метода:

```
question(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
         buttons=Yes | No][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать выводимые стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображаются кнопки **Yes и No**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `question()` возвращает числовое обозначение нажатой кнопки:

```
result = QtWidgets.QMessageBox.question(window, "Текст заголовка",
               "Вы действительно хотите выполнить действие?",
               buttons=QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No |
               QtWidgets.QMessageBox.Cancel,
               defaultButton=QtWidgets.QMessageBox.Cancel)
```

Результат выполнения этого кода показан на рис. 26.2.

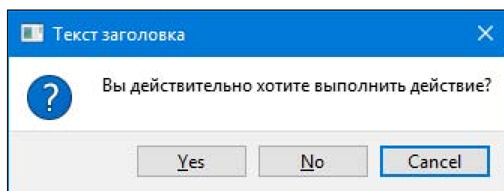

Рис. 26.2. Окно запроса подтверждения

26.3.4. Окно предупреждающего сообщения

Для вывода окна с предупреждающим сообщением предназначен статический метод `warning()`. Формат метода:

```
warning(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
        buttons=Ok][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать выводимые стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `warning()` возвращает числовое обозначение нажатой кнопки:

```
result = QtWidgets.QMessageBox.warning(window, "Текст заголовка",
               "Действие может быть опасным. Продолжить?",
               buttons=QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No |
               QtWidgets.QMessageBox.Cancel,
               defaultButton=QtWidgets.QMessageBox.Cancel)
```

Результат выполнения этого кода показан на рис. 26.3.

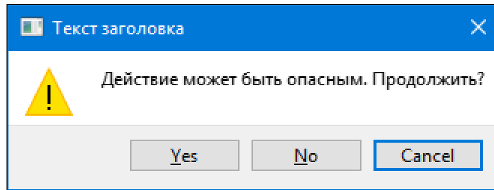

Рис. 26.3. Окно предупреждающего сообщения

26.3.5. Окно критического сообщения

Для вывода окна с критическим сообщением предназначен статический метод `critical()`. Формат метода:

```
critical(<Родитель>, <Текст заголовка>, <Текст сообщения>[,
        buttons=Ok][, defaultButton=NoButton])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать выводящиеся на экран стандартные кнопки (атрибуты, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `critical()` возвращает числовое обозначение нажатой кнопки:

```
QtWidgets.QMessageBox.critical(window, "Текст заголовка",
      "Программа выполнила недопустимую ошибку и будет закрыта",
      defaultButton=QtWidgets.QMessageBox.Ok)
```

Результат выполнения этого кода показан на рис. 26.4.

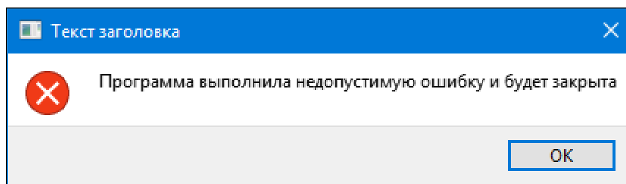

Рис. 26.4. Окно критического сообщения

26.3.6. Окно сведений о программе

Для вывода окна со сведениями о программе и авторских правах ее разработчиков предназначен статический метод `about()`. Формат метода:

```
about(<Родитель>, <Текст заголовка>, <Текст сообщения>)
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Слева от текста сообщения отображается значок приложения (см. *разд. 18.9*), если он был установлен:

```
QtWidgets.QMessageBox.about(window, "Текст заголовка", "Описание программы")
```

26.3.7. Окно сведений о библиотеке Qt

Для вывода окна с описанием используемой версии библиотеки Qt предназначен статический метод `aboutQt()`. Формат метода:

```
aboutQt(<Родитель>[, title=""])
```

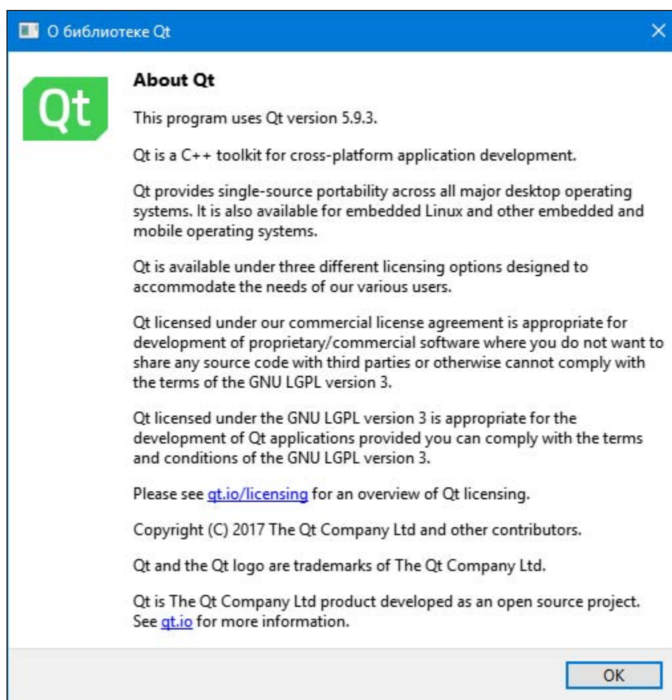

Рис. 26.5. Окно сведений о библиотеке Qt

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. В параметре `title` можно указать текст, выводимый в заголовке окна. Если параметр не указан, то выводится заголовок **About Qt**:

```
QtWidgets.QMessageBox.aboutQt(window, title="О библиотеке Qt")
```

Результат выполнения этого кода показан на рис. 26.5.

26.4. Класс *QInputDialog*

Класс `QInputDialog` представляет модальные диалоговые окна для ввода различных данных. Иерархия наследования для этого класса выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QInputDialog
```

Формат конструктора класса `QInputDialog`:

```
<Объект> = QInputDialog([parent=None][, flags=0])
```

Если в параметре `parent` указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `flags` задает тип окна (см. *разд. 18.2*).

После создания экземпляра класса следует вызвать метод `exec()` (или `exec_()`, оставленный для совместимости с предыдущими версиями PyQt), чтобы отобразить окно. Метод возвращает код возврата в виде значения следующих атрибутов класса `QDialog`: `Accepted` или `Rejected`.

Вот пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки:

```
def on_clicked():
    dialog = QtWidgets.QInputDialog(window)
    result = dialog.exec_()
    if result == QtWidgets.QDialog.Accepted:
        print("Нажата кнопка OK")
        # Здесь получаем данные из диалогового окна
    else:
        print("Нажата кнопка Cancel")
```

26.4.1. Основные методы и сигналы

Класс `QInputDialog` наследует все методы базовых классов и добавляет к ним следующие собственные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qinputdialog.html**):

- `setLabelText(<Текст>)` — задает текст, отображаемый над полем ввода;
- `setOkButtonText(<Текст>)` — задает текст надписи для кнопки **OK**:

  ```
  dialog.setOkButtonText("&Ввод")
  ```

- `setCancelButtonText(<Текст>)` — задает текст надписи для кнопки **Cancel**:

  ```
  dialog.setCancelButtonText("&Отмена")
  ```

- `setInputMode(<Режим>)` — задает режим ввода данных. В качестве параметра указываются следующие атрибуты класса `QInputDialog`:
  - `TextInput` — `0` — ввод текста;
  - `IntInput` — `1` — ввод целого числа;
  - `DoubleInput` — `2` — ввод вещественного числа;
- `setTextEchoMode(<Режим>)` — задает режим отображения текста в поле. Могут быть указаны следующие атрибуты класса `QLineEdit`:
  - `Normal` — `0` — показывать символы как они были введены;
  - `NoEcho` — `1` — не показывать вводимые символы;
  - `Password` — `2` — вместо любого символа выводится звездочка (`*`);
  - `PasswordEchoOnEdit` — `3` — показывать символы при вводе, а при потере фокуса вместо них отображать звездочки;
- `setTextValue(<Текст>)` — задает текст по умолчанию, отображаемый в текстовом поле;
- `textValue()` — возвращает текст, введенный в текстовое поле;
- `setIntValue(<Значение>)` — задает целочисленное значение при использовании режима `IntInput`;
- `intValue()` — возвращает целочисленное значение, введенное в поле, при использовании режима `IntInput`;
- `setIntRange(<Минимум>, <Максимум>)`, `setIntMinimum(<Минимум>)` и `setIntMaximum(<Максимум>)` — задают диапазон допустимых целочисленных значений при использовании режима `IntInput`;
- `setIntStep(<Шаг>)` — задает шаг приращения значения при нажатии кнопок со стрелками в правой части поля при использовании режима `IntInput`;
- `setDoubleValue(<Значение>)` — задает вещественное значение при использовании режима `DoubleInput`;
- `doubleValue()` — возвращает вещественное значение, введенное в поле, при использовании режима `DoubleInput`;
- `setDoubleRange(<Минимум>, <Максимум>)`, `setDoubleMinimum(<Минимум>)` и `setDoubleMaximum(<Максимум>)` — задают диапазон допустимых вещественных значений при использовании режима `DoubleInput`;
- `setDoubleDecimals(<Значение>)` — задает количество цифр после десятичной точки при использовании режима `DoubleInput`;
- `setComboBoxItems(<Список строк>)` — задает пункты, которые будут присутствовать в раскрывающемся списке, выводящемся в этом случае вместо обычного поля ввода. Набор пунктов указывается в виде списка строк;
- `setComboBoxEditable(<Флаг>)` — если в качестве параметра указано значение `True`, пользователь сможет ввести произвольное значение в раскрывающийся список, как если бы он был обычным полем ввода;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, заданная в первом параметре опция будет установлена, а если `False` — сброшена. Опции указываются в виде следующих атрибутов класса `QInputDialog`:

   - `NoButtons` — `1` — не выводить кнопки **OK** и **Cancel**;
   - `UseListViewForComboBoxItems` — `2` — для отображения списка строк будет использоваться класс `QListView` (обычный список), а не `QComboBox` (раскрывающийся список);
   - `UsePlainTextEditForTextInput` — `4` — вместо класса `QLineEdit` (однострочное текстовое поле) будет использован класс `QPlainTextEdit` (многострочное поле для ввода обычного текста);
- `setOptions(<Опции>)` — устанавливает несколько опций (см. описание метода `setOption()`) сразу.

Класс `QInputDialog` поддерживает сигналы:

- `textValueChanged(<Значение>)` — генерируется при изменении значения в текстовом поле. Внутри обработчика через параметр доступно новое значение в виде строки. Сигнал генерируется при использовании режима `TextInput`;
- `textValueSelected(<Значение>)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение в виде строки. Сигнал генерируется при использовании режима `TextInput`;
- `intValueChanged(<Значение>)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение в виде целого числа. Сигнал генерируется при использовании режима `IntInput`;
- `intValueSelected(<Значение>)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение в виде целого числа. Сигнал генерируется при использовании режима `IntInput`;
- `doubleValueChanged(<Значение>)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение в виде вещественного числа. Сигнал генерируется при использовании режима `DoubleInput`;
- `doubleValueSelected(<Значение>)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение в виде вещественного числа. Сигнал генерируется при использовании режима `DoubleInput`.

26.4.2. Окно для ввода строки

Класс `QInputDialog` также поддерживает несколько статических методов, реализующих типовые диалоговые окна.

Окно для ввода обычной строки или пароля выводится вызовом статического метода `getText()`. Формат метода:

```
getText(<Родитель>, <Текст заголовка>, <Текст подсказки>[, echo=Normal][,
        text=""][, flags=0][, inputMethodHints=QtCore.Qt.ImhNone])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не экрана. Необязательный параметр `echo` задает режим отображения текста в поле (см. описание метода `setTextEchoMode()` в *разд. 26.4.1*). Параметр `text` устанавливает значение поля по умолчанию, а в параметре `flags` можно указать тип окна.

Параметр `inputMethodHints` указывает дополнительные опции текстового поля, которое будет присутствовать в выведенном окне. В качестве его значения можно указать один из

многочисленных атрибутов класса `QtCore.Qt`, список которых приведен на странице **https://doc.qt.io/qt-5/qt.html#InputMethodHint-enum**, или их комбинацию через оператор `|`.

Метод `getText()` возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй — значение `True`, если была нажата кнопка **OK**, или значение `False`, если были нажаты кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна:

```
s, ok = QtWidgets.QInputDialog.getText(window, "Это заголовок окна",
                                       "Это текст подсказки",
                                       text="Значение по умолчанию")
if ok:
    print("Введено значение:", s)
```

Результат выполнения этого кода показан на рис. 26.6.

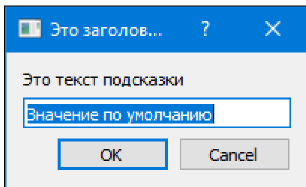

Рис. 26.6. Окно для ввода строки

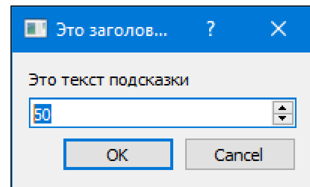

Рис. 26.7. Окно для ввода целого числа

26.4.3. Окно для ввода целого числа

Окно для ввода целого числа реализуется с помощью статического метода `getInt()`. Формат метода:

```
getInt(<Родитель>, <Текст заголовка>, <Текст подсказки>[, value=0][,
       min=-2147483647][, max=2147483647][, step=1][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минимальное значение, параметр `max` — максимальное значение, а параметр `step` — шаг приращения. Параметр `flags` позволяет указать тип окна. Методы возвращают кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй — значение `True`, если была нажата кнопка **OK**, или значение `False`, если были нажаты кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна:

```
n, ok = QtWidgets.QInputDialog.getInt(window, "Это заголовок окна",
                                      "Это текст подсказки",
                                      value=50, min=0, max=100, step=2)
if ok:
    print("Введено значение:", n)
```

Результат выполнения этого кода показан на рис. 26.7.

26.4.4. Окно для ввода вещественного числа

Окно для ввода вещественного числа реализуется с помощью статического метода `getDouble()`. Формат метода:

```
getDouble(<Родитель>, <Текст заголовка>, <Текст подсказки>[, value=0][,
          min=-2147483647][, max=2147483647][, decimals=1][, flags=0])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минимальное значение, параметр `max` — максимальное значение, а параметр `decimals` — количество цифр после десятичной точки. Параметр `flags` позволяет указать тип окна. Метод возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй элемент — значение `True`, если была нажата кнопка **OK**, или значение `False`, если были нажаты кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна:

```
n, ok = QtWidgets.QInputDialog.getDouble(window, "Это заголовок окна",
                                          "Это текст подсказки",
                                          value=50.0, min=0.0, max=100.0,
                                          decimals=2)
if ok:
    print("Введено значение:", n)
```

26.4.5. Окно для выбора пункта из списка

Окно для выбора пункта из списка выводится статическим методом `getItem()`. Формат метода:

```
getItem(<Родитель>, <Текст заголовка>, <Текст подсказки>, <Список строк>[,
        current=0][, editable=True][, flags=0][,
        inputMethodHints=QtCore.Qt.ImhNone])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `current` устанавливает индекс пункта, выбранного по умолчанию. Если в параметре `editable` указано значение `True`, пользователь может ввести произвольное значение в список вручную. Параметр `flags` позволяет указать тип окна.

Параметр `inputMethodHints` указывает дополнительные опции списка, который будет присутствовать в выведенном окне, — этот параметр имеет смысл указывать только в том случае, если для параметра `editable` задано значение `True`. В качестве его значения можно указать один из многочисленных атрибутов класса `QtCore.Qt`, список которых приведен на странице **https://doc.qt.io/qt-5/qt.html#InputMethodHint-enum**, или их комбинацию через оператор `|`.

Метод возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступен текст выбранного пункта, а через второй — значение `True`, если была нажата кнопка **OK**, или значение `False`, если были нажаты кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна:

```
s, ok = QtWidgets.QInputDialog.getItem(window, "Это заголовок окна",
        "Это текст подсказки", ["Пункт 1", "Пункт 2", "Пункт 3"],
        current=1, editable=False)
if ok:
    print("Текст выбранного пункта:", s)
```

Результат выполнения этого кода показан на рис. 26.8.

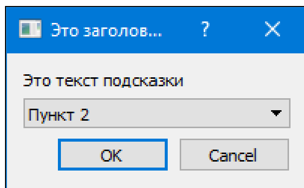

Рис. 26.8. Окно для выбора пункта из списка

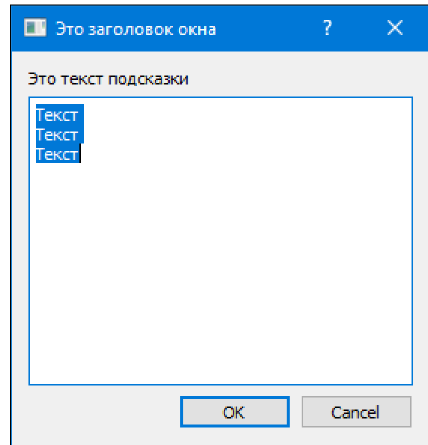

Рис. 26.9. Окно для ввода большого фрагмента текста

26.4.6. Окно для ввода большого текста

Окно для ввода большого фрагмента обычного текста реализуется статическим методом `getMultiLineText()`. Формат метода:

```
getMultiLineText(<Родитель>, <Текст заголовка>, <Текст подсказки>[,
                 text=""][, flags=0][, inputMethodHints=QtCore.Qt.ImhNone])
```

В параметре `<Родитель>` указывается ссылка на родительское окно или значение `None`. Если указана ссылка, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `text` устанавливает значение поля по умолчанию, а в параметре `flags` можно указать тип окна.

Параметр `inputMethodHints` указывает дополнительные опции текстового поля, которое будет присутствовать в выведенном окне. В качестве его значения можно указать один из многочисленных атрибутов класса `QtCore.Qt`, список которых приведен на странице **https://doc.qt.io/qt-5/qt.html#InputMethodHint-enum**, или их комбинацию через оператор `|`.

Метод `getMultiLineText()` возвращает кортеж из двух элементов: `(<Значение>, <Статус>)`. Через первый элемент доступно введенное значение, а через второй — значение `True`, если была нажата кнопка **OK**, или значение `False`, если были нажаты кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна:

```
s, ok = QtWidgets.QInputDialog.getMultiLineText(window, "Это заголовок окна",
        "Это текст подсказки", text = "Текст\nТекст\nТекст")
if ok:
    print("Текст выбранного пункта:", s)
```

Результат выполнения этого кода показан на рис. 26.9.

26.5. Класс *QFileDialog*

Класс `QFileDialog` реализует модальные диалоговые окна для выбора файла или каталога. Иерархия наследования для него выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QFileDialog
```

Форматы конструктора класса `QFileDialog`:

```
<Объект> = QFileDialog(<Родитель>, <Тип окна>)
<Объект> = QFileDialog([parent=None][, caption=""][, directory=""][, filter=""])
```

Если в параметрах `<Родитель>` и `parent` указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `<Тип окна>` задает тип окна (см. *разд. 18.2*). Необязательный параметр `caption` позволяет указать заголовок окна, параметр `directory` — начальный каталог, а параметр `filter` — фильтр для отбора файлов, которые будут выведены в диалоговом окне (например, фильтр `"Images (*.png *.jpg)"` задаст вывод только файлов с расширениями png и jpg).

После создания экземпляра класса следует вызвать метод `exec()` (или оставленный для совместимости с предыдущими версиями PyQt метод `exec_()`), чтобы вывести диалоговое окно на экран. Метод возвращает код возврата в виде значения атрибутов `Accepted` или `Rejected` класса `QDialog`.

26.5.1. Основные методы и сигналы

Класс `QFileDialog` наследует все методы базовых классов и определяет следующие собственные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qfiledialog.html**):

- `setAcceptMode(<Тип>)` — задает тип окна. В качестве параметра указываются следующие атрибуты класса `QFileDialog`:
  - `AcceptOpen` — `0` — окно для открытия файла (по умолчанию);
  - `AcceptSave` — `1` — окно для сохранения файла;
- `setViewMode(<Режим>)` — задает режим вывода списка файлов. В качестве параметра указываются следующие атрибуты класса `QFileDialog`:
  - `Detail` — `0` — отображается подробная информация о файлах;
  - `List` — `1` — отображается только значок и название файла;
- `setFileMode(<Тип>)` — задает тип возвращаемого значения. В качестве параметра указываются следующие атрибуты класса `QFileDialog`:
  - `AnyFile` — `0` — любой файл независимо от того, существует он или нет;
  - `ExistingFile` — `1` — существующий файл;
  - `Directory` — `2` — каталог;
  - `ExistingFiles` — `3` — список из нескольких существующих файлов. Несколько файлов можно выбрать, удерживая нажатой клавишу <Ctrl>;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, указанная в первом параметре опция будет установлена, а если `False` — сброшена. В первом параметре можно указать следующие атрибуты класса `QFileDialog` (здесь

приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qfiledialog.html#Option-enum**):

- ShowDirsOnly — отображать только названия каталогов. Опция работает только при использовании типа возвращаемого значения Directory;
- DontConfirmOverwrite — не спрашивать разрешения на перезапись существующего файла;
- ReadOnly — режим только для чтения;
- HideNameFilterDetails — скрывает детали фильтра;

♦ setOptions(<Опции>) — позволяет установить сразу несколько опций;

♦ setDirectory() — задает отображаемый каталог. Форматы метода:

```
setDirectory(<Путь>)
setDirectory(<QDir>)
```

♦ directory() — возвращает экземпляр класса QDir с путем к отображаемому каталогу;

♦ setDirectoryUrl(<QUrl>) — задает отображаемый каталог в виде экземпляра класса QUrl из модуля QtCore:

```
dialog.setDirectoryUrl(QtCore.QUrl.fromLocalFile("C:\\book"))
```

♦ directoryUrl() — возвращает экземпляр класса QUrl с путем к отображаемому каталогу;

♦ setNameFilter(<Фильтр>) — устанавливает фильтр. Чтобы установить несколько фильтров, необходимо указать их через две точки с запятой:

```
dialog.setNameFilter("All (*);;Images (*.png *.jpg)")
```

♦ setNameFilters(<Список фильтров>) — устанавливает сразу несколько фильтров:

```
dialog.setNameFilters(["All (*)", "Images (*.png *.jpg)"])
```

♦ selectFile(<Название файла>) — выбирает указанный файл;

♦ selectUrl(<QUrl>) — выбирает файл, указанный в виде экземпляра класса QUrl;

♦ selectedFiles() — возвращает список с выбранными файлами;

♦ selectedUrls() — возвращает список экземпляров класса QUrl, представляющих выбранные файлы;

♦ setDefaultSuffix(<Расширение>) — задает расширение, которое добавляется к файлу при отсутствии указанного расширения;

♦ setHistory(<Список>) — задает список истории;

♦ setSidebarUrls(<Список с QUrl>) — задает список папок, отображаемый на боковой панели:

```
dialog.setSidebarUrls([QtCore.QUrl.fromLocalFile("C:\\book"),
              QtCore.QUrl.fromLocalFile("C:\\book\\eclipse")])
```

♦ setLabelText(<Тип надписи>, <Текст>) — позволяет изменить текст указанной надписи. В первом параметре указываются следующие атрибуты класса QFileDialog:

- LookIn — 0 — надпись слева от списка с каталогами;
- FileName — 1 — надпись слева от поля с названием файла;
- FileType — 2 — надпись слева от поля с типами файлов;

* `Accept` — `3` — надпись на кнопке, нажатие которой приведет к подтверждению действия (по умолчанию **Open** или **Save**);
* `Reject` — `4` — надпись на кнопке, нажатие которой приведет к отказу от действия (по умолчанию **Cancel**);

♦ `saveState()` — возвращает экземпляр класса `QByteArray` с текущими параметрами диалогового окна;

♦ `restoreState(<QByteArray>)` — восстанавливает параметры диалогового окна и возвращает `True`, если восстановление прошло успешно, и `False` — в противном случае.

Класс `QFileDialog` поддерживает следующие сигналы:

♦ `currentChanged(<Путь>)` — генерируется при изменении текущего файла. Внутри обработчика через параметр доступен новый путь в виде строки;

♦ `currentUrlChanged(<QUrl>)` — генерируется при изменении текущего файла. Внутри обработчика через параметр доступен новый путь;

♦ `directoryEntered(<Путь>)` — генерируется при изменении каталога. Внутри обработчика через параметр доступен новый путь в виде строки;

♦ `directoryUrlEntered(<QUrl>)` — генерируется при изменении каталога. Внутри обработчика через параметр доступен новый путь;

♦ `fileSelected(<Путь>)` — генерируется при выборе файла и подтверждении выбора. Внутри обработчика через параметр доступен путь в виде строки;

♦ `filesSelected(<Список путей>)` — генерируется при выборе нескольких файлов и подтверждении выбора. Внутри обработчика через параметр доступен список с путями, заданными в виде строк;

♦ `urlSelected(<QUrl>)` — генерируется при выборе файла и подтверждении выбора. Внутри обработчика через параметр доступен путь;

♦ `urlsSelected(<Список QUrl>)` — генерируется при выборе нескольких файлов и подтверждении выбора. Внутри обработчика через параметр доступен список с путями;

♦ `filterSelected(<Фильтр>)` — генерируется при изменении фильтра. Внутри обработчика через параметр доступен новый фильтр в виде строки.

26.5.2. Окно для выбора каталога

Помимо рассмотренных методов, класс `QFileDialog` поддерживает несколько статических методов, реализующих типовые диалоговые окна.

Окно для выбора каталога реализуется с помощью статических методов `getExistingDirectory()` и `getExistingDirectoryUrl()`. Форматы методов:

```
getExistingDirectory([parent=None][, caption=""][,
                     directory=""][, options=ShowDirsOnly])
getExistingDirectoryUrl([parent=None][, caption=""][,
                        directory=QUrl()][, options=ShowDirsOnly])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `directory` задает текущий каталог, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 26.5.1*). Метод `getExistingDirectory()` воз-

вращает выбранный каталог или пустую строку, а метод getExistingDirectoryUrl() — экземпляр класса QUrl с выбранным путем или пустой экземпляр:

```
dirName = QtWidgets.QFileDialog.getExistingDirectory(parent=window,
                                directory=QtCore.QDir.currentPath())
dir = QtWidgets.QFileDialog.getExistingDirectoryUrl(parent=window,
                      directory=QtCore.QUrl.fromLocalFile(QtCore.QDir.currentPath()))
dirName = dir.toLocalFile()
```

Результат выполнения этого кода показан на рис. 26.10.

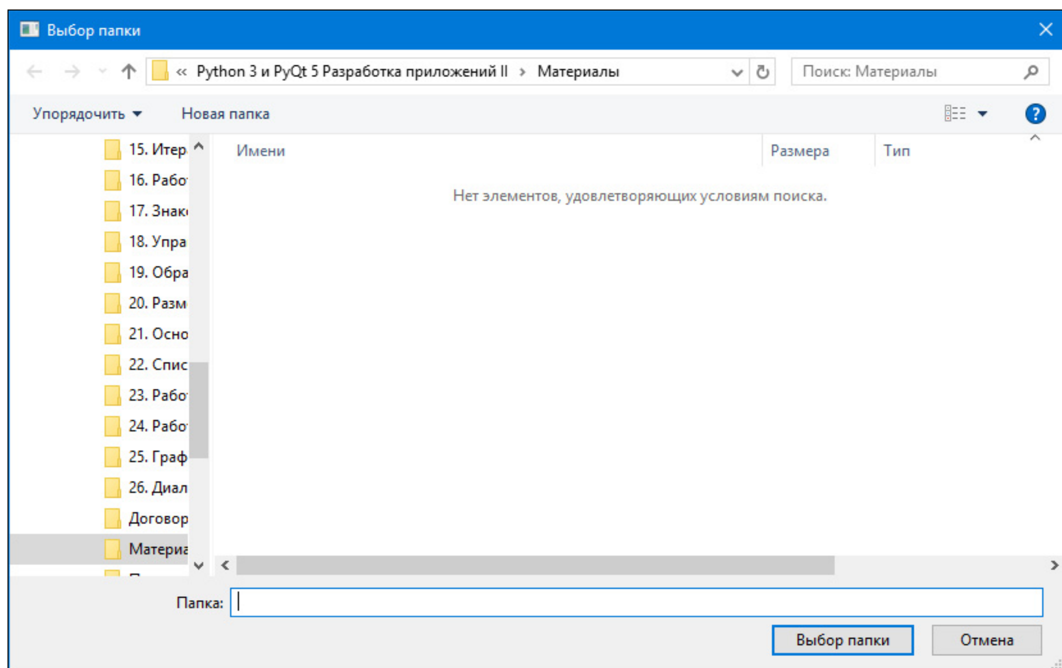

Рис. 26.10. Окно для выбора каталога

26.5.3. Окно для открытия файлов

Окно для открытия одного файла реализуется с помощью статических методов getOpenFileName() и getOpenFileUrl(). Форматы методов:

```
getOpenFileName([parent=None][, caption=""][,directory=""][,
                 filter=""][, initialFilter=''][, options=0])
getOpenFileUrl([parent=None][, caption=""][,directory=""][,
                 filter=""][, initialFilter=''][, options=0])
```

В параметре parent указывается ссылка на родительское окно или значение None. Необязательный параметр caption задает текст заголовка окна, параметр directory — текущий каталог, параметр filter — фильтр, параметр initialFilter — фильтр, который будет выбран изначально, а параметр options устанавливает опции (см. описание метода setOption() в *разд. 26.5.1*). Метод getOpenFileName() возвращает кортеж из двух элементов: первым элементом будет выбранный файл или пустая строка, вторым — выбранный фильтр. Метод getOpenFileUrl() также возвращает кортеж из двух элементов: первый —

экземпляр класса QUrl с путем выбранного файла или пустой экземпляр, второй — выбранный фильтр:

```
file = QtWidgets.QFileDialog.getOpenFileName(parent=window,
           caption="Заголовок окна", directory="c:\\python36",
           filter="All (*);;Exes (*.exe *.dll)",
           initialFilter="Exes (*.exe *.dll)")
fileName = file[0]
file = QtWidgets.QFileDialog.getOpenFileUrl(parent=window,
       caption="Заголовок окна", directory="file:///c:\\python36",
       filter="All (*);;Exes (*.exe *.dll)", initialFilter="Exes (*.exe *.dll)")
fileName = file[0].toLocalFile()
```

Результат выполнения кода из последнего примера показан на рис. 26.11.

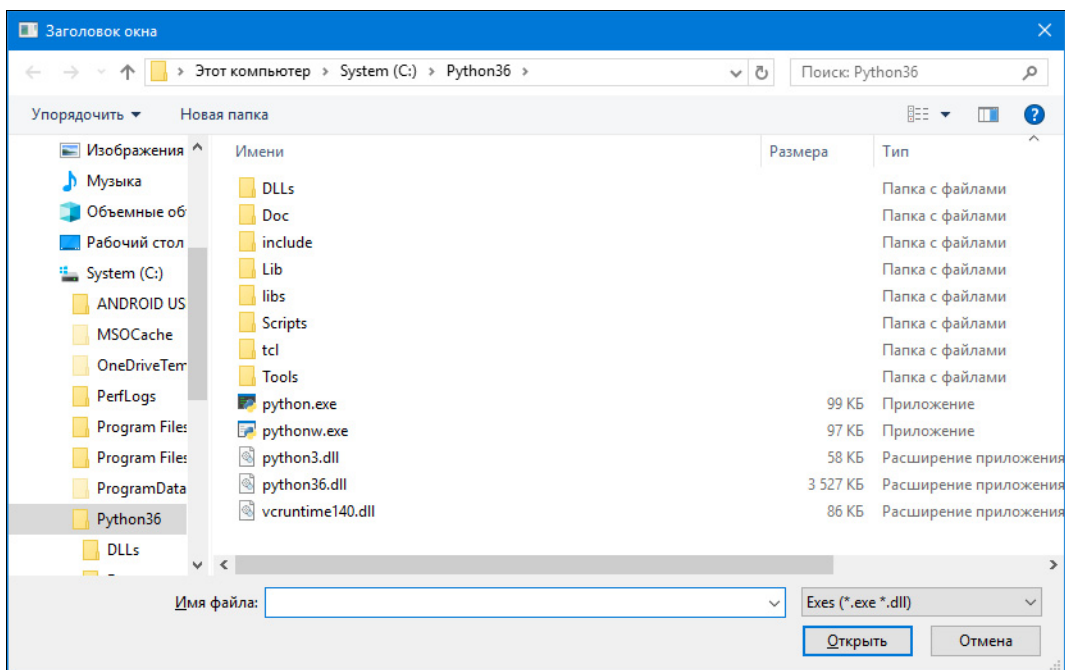

Рис. 26.11. Окно для открытия файла

Окно для открытия сразу нескольких файлов реализуется с помощью статических методов getOpenFileNames() и getOpenFileUrls(). Форматы методов:

```
getOpenFileNames([parent=None][, caption=""][, directory=""][,
                 filter=""][, initialFilter=''][, options=0])
getOpenFileUrls([parent=None][, caption=""][, directory=""][,
                filter=""][, initialFilter=''][, options=0])
```

Метод getOpenFileNames() возвращает кортеж из двух элементов: первый — список с путями к выбранным файлам или пустой список, второй — выбранный фильтр. Метод getOpenFileUrls() также возвращает кортеж из двух элементов: первый — список экземпляров класса QUrl с путями к выбранным файлам или пустой список, второй — выбранный фильтр:

```
arr = QtWidgets.QFileDialog.getOpenFileNames(parent=window,
      caption="Заголовок окна", directory="c:\\python36",
      filter="All (*);;Exes (*.exe *.dll)",
      initialFilter="Exes (*.exe *.dll)")
files = arr[0]
arr = QtWidgets.QFileDialog.getOpenFileUrls(parent=window,
      caption="Заголовок окна", directory="file:///c:\\python36",
      filter="All (*);;Exes (*.exe *.dll)",
      initialFilter="Exes (*.exe *.dll)")
files = list(a.toLocalFile() for a in arr[0])
```

26.5.4. Окно для сохранения файла

Окно для сохранения файла реализуется статическими методами `getSaveFileName()` и `getSaveFileUrl()`. Форматы методов:

```
getSaveFileName([parent=None][, caption=""][, directory=""][,
                filter=""][, initialFilter=''][, options=0])
getSaveFileUrl([parent=None][, caption=""][, directory=""][,
               filter=""][, initialFilter=''][, options=0])
```

В параметре `parent` указывается ссылка на родительское окно или значение `None`. Необязательный параметр `caption` задает текст заголовка окна, параметр `directory` — текущий каталог, параметр `filter` — фильтр, параметр `initialFilter` — фильтр, который будет выбран изначально, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 26.5.1*). Метод `getSaveFileName()` возвращает кортеж из двух элементов:

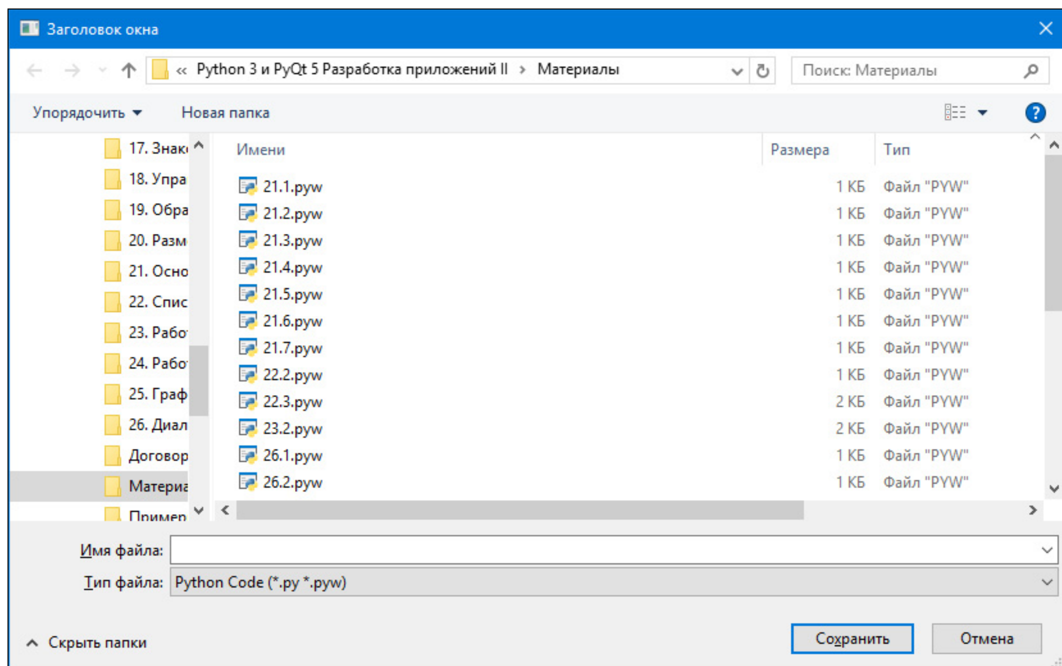

Рис. 26.12. Окно для сохранения файла

первым элементом будет выбранный файл или пустая строка, вторым — выбранный фильтр. Метод `getSaveFileUrl()` также возвращает кортеж из двух элементов: первый — экземпляр класса `QUrl` с путем выбранного файла или пустой экземпляр, второй — выбранный фильтр:

```
f = QtWidgets.QFileDialog.getSaveFileName(parent=window,
    caption="Заголовок окна", directory=QtCore.QDir.currentPath(),
    filter="All (*);;Python Code (*.py *.pyw)")
fileName = f[0]
f = QtWidgets.QFileDialog.getSaveFileUrl(parent=window,
    caption="Заголовок окна", directory="file:///" + QtCore.QDir.currentPath(),
    filter="All (*);;Python Code (*.py *.pyw)")
fileName = f[0].toLocalFile()
```

Результат выполнения кода из последнего примера показан на рис. 26.12.

26.6. Окно для выбора цвета

Окно для выбора цвета (рис. 26.13) реализуется с помощью статического метода `getColor()` класса `QColorDialog`. Формат метода:

```
getColor([initial=white][, parent=None][, title=""][, options=0])
```

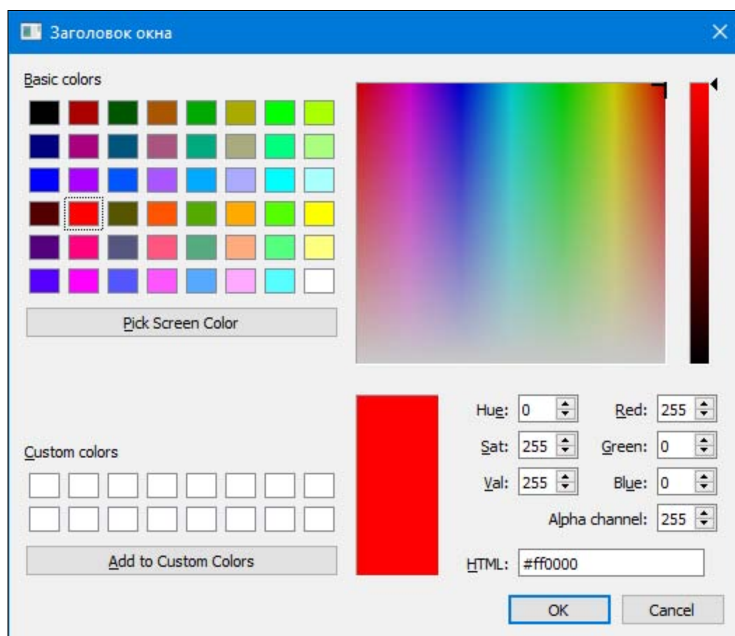

Рис. 26.13. Окно для выбора цвета

Параметр `initial` задает начальный цвет. В параметре `parent` указывается ссылка на родительское окно или значение `None`. Параметр `title` позволяет указать заголовок окна. В параметре `options` могут быть указаны следующие атрибуты (или их комбинация) класса `QColorDialog`:

- `ShowAlphaChannel` — пользователь может выбрать значение прозрачности;
- `NoButtons` — кнопки **OK** и **Cancel** не отображаются;
- `DontUseNativeDialog` — использовать встроенное в библиотеку Qt диалоговое окно выбора цвета вместо системного.

Метод возвращает экземпляр класса `QColor`, представляющий выбранный цвет. Если пользователь нажмет кнопку **Cancel**, возвращенный экземпляр будет невалидным.

Пример:

```
color = QtWidgets.QColorDialog.getColor(initial=QtGui.QColor("#ff0000"),
        parent=window, title="Заголовок окна",
        options=QtWidgets.QColorDialog.ShowAlphaChannel)
if color.isValid():
    print(color.red(), color.green(), color.blue(), color.alpha())
```

26.7. Окно для выбора шрифта

Окно для выбора шрифта реализуется с помощью статического метода `getFont()` класса `QFontDialog`. Форматы метода:

```
getFont([parent=None])
getFont(<QFont>[, parent=None][, caption=""][, options=0])
```

Параметр `<QFont>` во втором формате задает начальный шрифт. В параметре `parent` указывается ссылка на родительское окно или значение `None`. Параметр `caption` позволяет указать заголовок окна. В параметре `options` могут быть указаны следующие атрибуты класса `QFontDialog` или их комбинация:

- `NoButtons` — кнопки **OK** и **Cancel** не отображаются;
- `DontUseNativeDialog` — использовать встроенное в библиотеку Qt диалоговое окно выбора шрифта вместо системного;
- `ScalableFonts` — масштабируемые шрифты;
- `NonScalableFonts` — немасштабируемые шрифты;
- `MonospacedFonts` — моноширинные шрифты;
- `ProportionalFonts` — пропорциональные шрифты.

Метод возвращает кортеж из двух элементов: `(<QFont>, <Статус>)`. Если второй элемент содержит значение `True`, первый элемент хранит экземпляр класса `QFont` с выбранным шрифтом. Пример:

```
(font, ok) = QtWidgets.QFontDialog.getFont(QtGui.QFont("Tahoma", 16),
                              parent=window, caption="Заголовок окна")
if ok:
    print(font.family(), font.pointSize(), font.weight(),
          font.italic(), font.underline())
```

Результат выполнения этого кода показан на рис. 26.14.

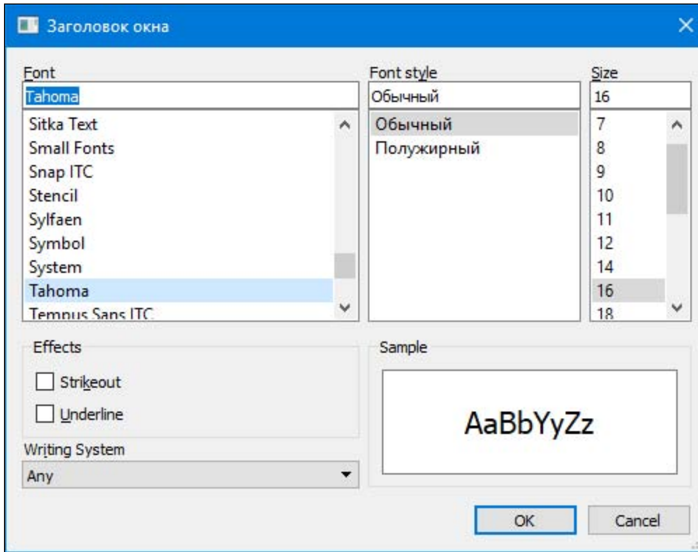

Рис. 26.14. Окно для выбора шрифта

26.8. Окно для вывода сообщения об ошибке

Класс `QErrorMessage` реализует немодальное диалоговое окно с сообщением об ошибке (рис. 26.15). Окно содержит надпись с собственно сообщением и флажок. Если пользователь снимает флажок, окно больше отображаться не будет. Иерархия наследования для класса `QErrorMessage` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QErrorMessage
```

Формат конструктора класса `QErrorMessage`:

```
<Объект> = QErrorMessage([parent=None])
```

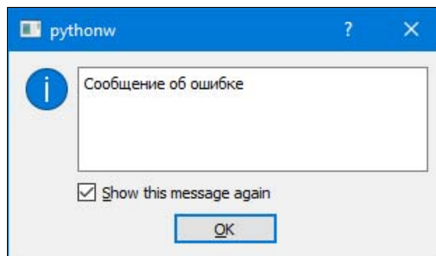

Рис. 26.15. Окно для вывода сообщения об ошибке

Для отображения окна предназначен метод `showMessage()`. Форматы его вызова:

```
showMessage(<Текст сообщения>)
showMessage(<Текст сообщения>, <Тип>)
```

Если пользователь установил флажок, и тип сообщения не был задан, все последующие сообщения об ошибках не будут выводиться на экран. Если же был указан тип сообщения, то не будут выводиться лишь сообщения того же типа. Оба формата метода являются слотами:

```
ems = QtWidgets.QErrorMessage(parent=window)
ems.showMessage("Сообщение об ошибке")
ems.showMessage("Сообщение об ошибке типа warning", "warning")
```

26.9. Окно с индикатором хода процесса

Класс `QProgressDialog` реализует диалоговое окно с индикатором хода процесса и кнопкой **Cancel** (рис. 26.16). Иерархия наследования для класса `QProgressDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QProgressDialog
```

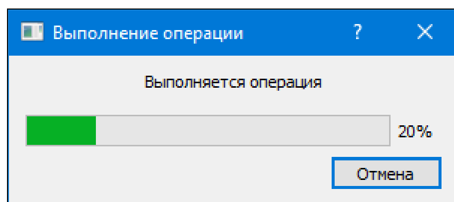

Рис. 26.16. Окно с индикатором хода процесса

Форматы конструктора класса `QProgressDialog`:

```
<Объект> = QProgressDialog([parent=None][, flags=0])
<Объект> = QProgressDialog(<Текст над индикатором>, <Текст на кнопке Cancel>,
                           <Минимум>, <Максимум>[, parent=None][, flags=0])
```

Значения минимума и максимума должны быть заданы в виде целых чисел. Если в параметре `parent` указана ссылка на родительское окно, диалоговое окно будет центрироваться относительно родительского окна, а не экрана. Параметр `flags` задает тип окна (см. *разд. 18.2*).

Класс `QProgressDialog` наследует все методы базовых классов и дополнительно определяет следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qprogressdialog.html**):

- `setValue(<Значение>)` — задает новое значение индикатора, которое должно представлять собой целое число. Если диалоговое окно является модальным, при установке значения автоматически вызывается метод `processEvents()` объекта приложения. Метод является слотом;
- `value()` — возвращает текущее значение индикатора в виде целого числа;
- `setLabelText(<Текст над индикатором>)` — задает надпись, выводимую над индикатором. Метод является слотом;
- `setCancelButtonText(<Текст на кнопке Cancel>)` — задает надпись для кнопки **Cancel**. Метод является слотом;
- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное значения в виде целых чисел. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы являются слотами;
- `setMinimumDuration(<Значение>)` — задает промежуток времени в миллисекундах перед отображением окна (задается целым числом — значение по умолчанию: `4000`). Окно

может быть отображено ранее этого срока при установке значения. Метод является слотом;

- reset() — сбрасывает значение индикатора. Метод является слотом;
- cancel() — имитирует нажатие кнопки **Cancel**. Метод является слотом;
- setLabel(<QLabel>) — позволяет заменить объект надписи;
- setBar(<QProgressBar>) — позволяет заменить объект индикатора;
- setCancelButton(<QPushButton>) — позволяет заменить объект кнопки;
- setAutoClose(<Флаг>) — если в качестве параметра указано значение True, при сбросе значения окно скрывается;
- setAutoReset(<Флаг>) — если в качестве параметра указано значение True, при достижении максимального значения будет автоматически произведен сброс;
- wasCanceled() — возвращает значение True, если была нажата кнопка **Cancel**.

Класс QProgressDialog поддерживает сигнал canceled(), который генерируется при нажатии кнопки **Cancel**.

26.10. Создание многостраничного мастера

С помощью классов QWizard и QWizardPage можно создать *мастер* — диалоговое окно, в котором при нажатии кнопок **Back** (Назад) и **Next** (Далее) последовательно или в произвольном порядке отображаются различные страницы. Класс QWizard реализует контейнер для страниц, а отдельная страница описывается с помощью класса QWizardPage.

26.10.1. Класс *QWizard*

Класс QWizard реализует набор страниц, отображаемых последовательно или в произвольном порядке. Иерархия наследования для него выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QWizard
```

Формат конструктора класса QWizard:

```
<Объект> = QWizard([parent=None][, flags=0])
```

Помимо унаследованных методов, класс QWizard поддерживает следующие (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qwizard.html**):

- addPage(<QWizardPage>) — добавляет страницу в конец мастера и возвращает ее целочисленный идентификатор. В качестве параметра указывается экземпляр класса QWizardPage;
- setPage(<Идентификатор>, <QWizardPage>) — добавляет страницу в указанную позицию;
- removePage(<Идентификатор>) — удаляет страницу с указанным идентификатором;
- page(<Идентификатор>) — возвращает ссылку на страницу (экземпляр класса QWizardPage), соответствующую указанному идентификатору, или значение None, если такой страницы не существует;
- pageIds() — возвращает список с идентификаторами страниц;

- currentId() — возвращает идентификатор текущей страницы;
- currentPage() — возвращает ссылку на текущую страницу (экземпляр класса QWizardPage) или None, если страницы не существует;
- setStartId(<Идентификатор>) — задает идентификатор начальной страницы;
- startId() — возвращает идентификатор начальной страницы;
- visitedPages() — возвращает список с идентификаторами посещенных страниц или пустой список;
- hasVisitedPage(<Идентификатор>) — возвращает значение True, если страница была посещена, и False — в противном случае;
- back() — имитирует нажатие кнопки **Back**. Метод является слотом;
- next() — имитирует нажатие кнопки **Next**. Метод является слотом;
- restart() — перезапускает мастер. Метод является слотом;
- nextId(self) — этот метод следует переопределить в классе, наследующем класс QWizard, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next** и должен возвращать идентификатор следующей страницы или значение -1;
- initializePage(self, <Идентификатор>) — этот метод следует переопределить в классе, наследующем QWizard, если необходимо производить настройку свойств компонентов на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на предыдущей странице, но до отображения следующей страницы. Если установлена опция IndependentPages, метод вызывается только при первом отображении страницы;
- cleanupPage(self, <Идентификатор>) — этот метод следует переопределить в классе, наследующем QWizard, если необходимо контролировать нажатие кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на текущей странице, но до отображения предыдущей страницы. Если установлена опция IndependentPages, метод не вызывается;
- validateCurrentPage(self) — этот метод следует переопределить в классе, наследующем QWizard, если необходимо производить проверку данных, введенных на текущей странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен вернуть значение True, если данные корректны, и False — в противном случае. Если метод возвращает значение False, переход на следующую страницу не производится;
- setField(<Свойство>, <Значение>) — устанавливает значение указанного свойства. Создание свойства производится с помощью метода registerField() класса QWizardPage. Используя метод setField(), можно изменять значения компонентов, расположенных на разных страницах мастера;
- field(<Свойство>) — возвращает значение указанного свойства. Создание свойства производится с помощью метода registerField() класса QWizardPage. Используя метод field(), можно также получить значения компонентов, расположенных на разных страницах мастера;
- setWizardStyle(<Стиль>) — задает стилевое оформление мастера. В качестве параметра указываются следующие атрибуты класса QWizard:
  - ClassicStyle — 0;
  - ModernStyle — 1;

- MacStyle — 2;
- AeroStyle — 3;

♦ setTitleFormat(<Формат>) — задает формат отображения названия страницы. В качестве параметра указываются следующие атрибуты класса QtCore.Qt:

- PlainText — 0 — простой текст;
- RichText — 1 — форматированный текст;
- AutoText — 2 — автоматическое определение (режим по умолчанию). Если текст содержит HTML-теги, то используется режим RichText, в противном случае — режим PlainText;

♦ setSubTitleFormat(<Формат>) — задает формат отображения описания страницы. Допустимые значения см. в описании метода setTitleFormat();

♦ setButton(<Роль>, <QAbstractButton>) — добавляет кнопку для указанной роли. В первом параметре указываются следующие атрибуты класса QWizard:

- BackButton — 0 — кнопка **Back**;
- NextButton — 1 — кнопка **Next**;
- CommitButton — 2 — кнопка **Commit**;
- FinishButton — 3 — кнопка **Finish**;
- CancelButton — 4 — кнопка **Cancel** (если установлена опция NoCancelButton, кнопка не отображается);
- HelpButton — 5 — кнопка **Help** (чтобы отобразить кнопку, необходимо установить опцию HaveHelpButton);
- CustomButton1 — 6 — первая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton1);
- CustomButton2 — 7 — вторая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton2);
- CustomButton3 — 8 — третья пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию HaveCustomButton3);

♦ button(<Роль>) — возвращает ссылку на кнопку с указанной ролью;

♦ setButtonText(<Роль>, <Текст надписи>) — устанавливает текст надписи для кнопки с указанной ролью;

♦ buttonText(<Роль>) — возвращает текст надписи кнопки с указанной ролью;

♦ setButtonLayout(<Список с ролями>) — задает порядок отображения кнопок. В качестве параметра указывается список с ролями кнопок. Список может также содержать значение атрибута Stretch класса QWizard, который задает фактор растяжения;

♦ setPixmap(<Роль>, <QPixmap>) — добавляет изображение для указанной роли. В первом параметре указываются следующие атрибуты класса QWizard:

- WatermarkPixmap — 0 — изображение, которое занимает всю левую сторону страницы при использовании стилей ClassicStyle или ModernStyle;
- LogoPixmap — 1 — небольшое изображение, отображаемое в правой части заголовка при использовании стилей ClassicStyle или ModernStyle;

- BannerPixmap — 2 — фоновое изображение, отображаемое в заголовке страницы при использовании стиля ModernStyle;
- BackgroundPixmap — 3 — фоновое изображение при использовании стиля MacStyle;

♦ setOption(<Опция>[, on=True]) — если во втором параметре указано значение True, заданная в первом параметре опция будет установлена, а если указано значение False, сброшена. В первом параметре указываются следующие атрибуты класса QWizard:

- IndependentPages — страницы не зависят друг от друга. В этом случае метод initializePage() будет вызван только при первом отображении страницы, а метод cleanupPage() не вызывается;
- IgnoreSubTitles — не отображать описание страницы в заголовке;
- ExtendedWatermarkPixmap — изображение с ролью WatermarkPixmap будет занимать всю левую сторону страницы вплоть до нижнего края окна;
- NoDefaultButton — не делать кнопки **Next** и **Finish** кнопками по умолчанию;
- NoBackButtonOnStartPage — не отображать кнопку **Back** на стартовой странице;
- NoBackButtonOnLastPage — не отображать кнопку **Back** на последней странице;
- DisabledBackButtonOnLastPage — сделать кнопку **Back** неактивной на последней странице;
- HaveNextButtonOnLastPage — показать неактивную кнопку **Next** на последней странице;
- HaveFinishButtonOnEarlyPages — показать неактивную кнопку **Finish** на непоследних страницах;
- NoCancelButton — не отображать кнопку **Cancel**;
- CancelButtonOnLeft — поместить кнопку **Cancel** слева от кнопки **Back** (по умолчанию кнопка **Cancel** расположена справа от кнопок **Next** и **Finish**);
- HaveHelpButton — показать кнопку **Help**;
- HelpButtonOnRight — поместить кнопку **Help** у правого края окна;
- HaveCustomButton1 — показать пользовательскую кнопку с ролью CustomButton1;
- HaveCustomButton2 — показать пользовательскую кнопку с ролью CustomButton2;
- HaveCustomButton3 — показать пользовательскую кнопку с ролью CustomButton3;
- NoCancelButtonOnLastPage — не показывать кнопку **Cancel** на последней странице;

♦ setOptions(<Опции>) — устанавливает сразу несколько опций;

♦ setSideWidget(<QWidget>) — устанавливает заданный компонент в левую часть мастера. Если используется стиль ClassicStyle или ModernStyle, компонент будет выведен выше заданного с ролью WatermarkPixmap изображения.

Класс QWizard поддерживает следующие сигналы:

♦ currentIdChanged(<Идентификатор>) — генерируется при изменении текущей страницы. Внутри обработчика через параметр доступен целочисленный идентификатор текущей страницы;

♦ customButtonClicked(<Роль>) — генерируется при нажатии кнопок с ролями CustomButton1, CustomButton2 и CustomButton3. В параметре доступна целочисленная роль нажатой кнопки;

- helpRequested() — генерируется при нажатии кнопки **Help**;
- pageAdded(<Идентификатор>) — генерируется при добавлении страницы. В параметре передается целочисленный идентификатор добавленной страницы;
- pageRemoved(<Идентификатор>) — генерируется при удалении страницы. В параметре передается целочисленный идентификатор удаленной страницы.

26.10.2. Класс *QWizardPage*

Класс QWizardPage описывает одну страницу в многостраничном мастере. Его иерархия наследования такова:

```
(QObject, QPaintDevice) — QWidget — QWizardPage
```

Формат конструктора класса QWizardPage:

```
<Объект> = QWizardPage([parent=None])
```

Класс QWizardPage наследует все методы базовых классов и, помимо них, поддерживает также следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qwizardpage.html**):

- wizard() — возвращает ссылку на объект мастера (экземпляр класса QWizard) или значение None, если страница не была добавлена в какой-либо мастер;
- setTitle(<Текст>) — задает название страницы;
- title() — возвращает название страницы;
- setSubTitle(<Текст>) — задает описание страницы;
- subTitle() — возвращает описание страницы;
- setButtonText(<Роль>, <Текст надписи>) — устанавливает текст надписи для кнопки с указанной ролью (допустимые значения параметра <Роль> см. в описании метода setButton() класса QWizard);
- buttonText(<Роль>) — возвращает текст надписи кнопки с указанной ролью;
- setPixmap(<Роль>, <QPixmap>) — добавляет изображение для указанной роли (допустимые значения параметра <Роль> см. в описании метода setPixmap() класса QWizard);
- registerField() — регистрирует свойство, с помощью которого можно получить доступ к значению компонента с любой страницы мастера. Формат метода:

  ```
  registerField(<Свойство>, <QWidget>[, property=None][, changedSignal=0])
  ```

  В параметре <Свойство> указывается произвольное название свойства в виде строки. Если в конце строки указать символ *, компонент должен обязательно иметь значение (например, в поле должно быть введено какое-либо значение), иначе кнопки **Next** и **Finish** будут недоступны. Во втором параметре указывается ссылка на компонент. После регистрации свойства изменить значение компонента позволяет метод setField(), а получить значение — метод field().

  В параметре property может быть указано свойство для получения и изменения значения, а в параметре changedSignal — сигнал, генерируемый при изменении значения.

  Назначить эти параметры для определенного класса позволяет также метод setDefaultProperty(<Название класса>, property, changedSignal) класса QWizard. Для стандартных классов по умолчанию используются следующие свойства и сигналы:

Класс:	Свойство:	Сигнал:
QAbstractButton	checked	toggled()
QAbstractSlider	value	valueChanged()
QComboBox	currentIndex	currentIndexChanged()
QDateTimeEdit	dateTime	dateTimeChanged()
QLineEdit	text	textChanged()
QListWidget	currentRow	currentRowChanged()
QSpinBox	value	valueChanged()

- `setField(<Свойство>, <Значение>)` — устанавливает значение указанного свойства. С помощью этого метода можно изменять значения компонентов, расположенных на разных страницах мастера;
- `field(<Свойство>)` — возвращает значение указанного свойства. С помощью этого метода можно получить значения компонентов, расположенных на разных страницах мастера;
- `setFinalPage(<Флаг>)` — если в качестве параметра указано значение `True`, на странице будет отображаться кнопка **Finish**;
- `isFinalPage()` — возвращает значение `True`, если на странице будет отображаться кнопка **Finish**, и `False` — в противном случае;
- `setCommitPage(<Флаг>)` — если в качестве параметра указано значение `True`, на странице будет отображаться кнопка **Commit**;
- `isCommitPage()` — возвращает значение `True`, если на странице будет отображаться кнопка **Commit**, и `False` — в противном случае;
- `isComplete(self)` — этот метод вызывается, чтобы определить, должны ли кнопки **Next** и **Finish** быть доступными (метод возвращает значение `True`) или недоступными (метод возвращает значение `False`). Метод можно переопределить в классе, наследующем класс `QWizardPage`, и реализовать собственную проверку правильности ввода данных. При изменении возвращаемого значения необходимо генерировать сигнал `completeChanged()`;
- `nextId(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next**. Метод должен возвращать идентификатор следующей страницы или значение `-1`;
- `initializePage(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо производить настройку свойств компонентов на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на предыдущей странице, но до отображения следующей страницы. Если установлена опция `IndependentPages`, метод вызывается только при первом отображении страницы;
- `cleanupPage(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо контролировать нажатие кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на текущей странице, но до отображения предыдущей страницы. Если установлена опция `IndependentPages`, метод не вызывается;
- `validatePage(self)` — этот метод следует переопределить в классе, наследующем `QWizardPage`, если необходимо производить проверку данных, введенных на текущей странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен вернуть значение `True`, если данные корректны, и `False` — в противном случае. Если метод возвращает значение `False`, переход на следующую страницу не производится.

ГЛАВА 27

Создание SDI- и MDI-приложений

В PyQt включена поддержка создания двух типов приложений:

- *SDI-приложения* (Single Document Interface) — позволяют открыть один документ. Чтобы открыть новый документ, необходимо предварительно закрыть предыдущий или запустить другой экземпляр приложения. Типичными примерами таких приложений являются программы Блокнот, WordPad и Paint, поставляемые в составе операционной системы Windows. Чтобы создать SDI-приложение, следует с помощью метода `setCentralWidget()` класса `QMainWindow` установить компонент, отображающий содержимое документа;
- *MDI-приложения* (Multiple Document Interface) — позволяют открыть сразу несколько документов, каждый — в отдельном вложенном окне. Примером приложения такого рода является программа Adobe Photoshop, позволяющая редактировать сразу несколько фотографий одновременно. Чтобы создать MDI-приложение, следует с помощью метода `setCentralWidget()` класса `QMainWindow` в качестве центрального компонента установить компонент `QMdiArea`. Отдельное вложенное окно внутри MDI-области представляется классом `QMdiSubWindow`.

Все рассмотренные в этой главе классы объявлены в модуле `QtWidgets`, если не указано иное.

27.1. Главное окно приложения

Класс `QMainWindow` реализует главное окно приложения, содержащее меню, панели инструментов, прикрепляемые панели, центральный компонент и строку состояния. Иерархия наследования для этого класса выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMainWindow
```

Конструктор класса `QMainWindow` имеет следующий формат:

```
<Объект> = QMainWindow([parent=None][, flags=0])
```

В параметре `parent` указывается ссылка на родительское окно. Доступные значения параметра `flags` мы рассматривали в *разд. 18.2*.

Класс `QMainWindow` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-5/qmainwindow.html**):

- ♦ `setCentralWidget(<QWidget>)` — делает указанный компонент центральным для главного окна;
- ♦ `centralWidget()` — возвращает ссылку на центральный компонент или значение `None`, если таковой не был установлен;
- ♦ `setMenuBar(<QMenuBar>)` — позволяет установить пользовательское меню вместо стандартного;
- ♦ `menuBar()` — возвращает ссылку на главное меню (экземпляр класса `QMenuBar`);
- ♦ `setMenuWidget(<QWidget>)` — позволяет установить компонент главного меню;
- ♦ `menuWidget()` — возвращает ссылку на компонент (экземпляр класса `QWidget`), в котором расположено главное меню;
- ♦ `createPopupMenu()` — создает контекстное меню с пунктами, позволяющими отобразить или скрыть панели инструментов и прикрепляемые панели, и возвращает ссылку на это меню (экземпляр класса `QMenu`). Контекстное меню по умолчанию отображается при щелчке правой кнопкой мыши в области меню, панели инструментов или прикрепляемых панелей. Переопределив этот метод, можно реализовать собственное контекстное меню;
- ♦ `setStatusBar(<QStatusBar>)` — позволяет заменить стандартную строку состояния;
- ♦ `statusBar()` — возвращает ссылку (экземпляр класса `QStatusBar`) на строку состояния;
- ♦ `addToolBar()` — добавляет панель инструментов. Форматы метода:

  ```
  addToolBar(<QToolBar>)
  addToolBar(<Область>, <QToolBar>)
  addToolBar(<Название панели>)
  ```

  Первый формат добавляет панель инструментов в верхнюю часть окна. Второй формат дополнительно позволяет задать местоположение панели. В качестве параметра `<Область>` могут быть указаны следующие атрибуты класса `QtCore.Qt`: `LeftToolBarArea` (слева), `RightToolBarArea` (справа), `TopToolBarArea` (сверху) или `BottomToolBarArea` (снизу). Третий формат создает панель инструментов с указанным именем, добавляет ее в верхнюю область окна и возвращает ссылку на представляющий ее экземпляр класса `QToolBar`;
- ♦ `insertToolBar(<QToolBar 1>, <QToolBar 2>)` — добавляет панель `<QToolBar 2>` перед панелью `<QToolBar 1>`;
- ♦ `removeToolBar(<QToolBar>)` — удаляет панель инструментов из окна и скрывает ее. При этом объект панели инструментов не удаляется и далее может быть добавлен в другое место;
- ♦ `toolBarArea(<QToolBar>)` — возвращает местоположение указанной панели инструментов в виде значений атрибутов `LeftToolBarArea` (слева), `RightToolBarArea` (справа), `TopToolBarArea` (сверху), `BottomToolBarArea` (снизу) или `NoToolBarArea` (положение не определено) класса `QtCore.Qt`;
- ♦ `setToolButtonStyle(<Стиль>)` — задает стиль кнопок на панели инструментов. В качестве параметра указываются следующие атрибуты класса `QtCore.Qt`:
  - `ToolButtonIconOnly` — `0` — отображается только значок;
  - `ToolButtonTextOnly` — `1` — отображается только текст;

- `ToolButtonTextBesideIcon` — 2 — текст отображается справа от значка;
- `ToolButtonTextUnderIcon` — 3 — текст отображается под значком;
- `ToolButtonFollowStyle` — 4 — зависит от используемого стиля;

♦ `toolButtonStyle()` — возвращает стиль кнопок на панели инструментов;

♦ `setIconSize(<QSize>)` — задает размеры значков;

♦ `iconSize()` — возвращает размеры значков (экземпляр класса `QSize`);

♦ `setAnimated(<Флаг>)` — если в качестве параметра указано значение `False`, вставка панелей инструментов и прикрепляемых панелей в новое место по окончании перемещения будет производиться без анимации. Метод является слотом;

♦ `addToolBarBreak([area=TopToolBarArea])` — вставляет разрыв в указанное место после всех добавленных ранее панелей. По умолчанию панели добавляются друг за другом на одной строке. С помощью этого метода можно поместить панели инструментов на двух и более строках;

♦ `insertToolBarBreak(<QToolBar>)` — вставляет разрыв перед указанной панелью инструментов;

♦ `removeToolBarBreak(<QToolBar>)` — удаляет разрыв перед указанной панелью;

♦ `toolBarBreak(<QToolBar>)` — возвращает значение `True`, если перед указанной панелью инструментов существует разрыв, и `False` — в противном случае;

♦ `addDockWidget()` — добавляет прикрепляемую панель. Форматы метода:

```
addDockWidget(<Область>, <QDockWidget>)
addDockWidget(<Область>, <QDockWidget>, <Ориентация>)
```

Первый формат добавляет прикрепляемую панель в указанную область окна. В качестве параметра `<Область>` могут быть указаны следующие атрибуты класса `QtCore.Qt`: `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху) или `BottomDockWidgetArea` (снизу). Второй формат позволяет дополнительно указать ориентацию при добавлении панели. В качестве параметра `<Ориентация>` могут быть указаны следующие атрибуты класса `QtCore.Qt`: `Horizontal` или `Vertical`. Если указан атрибут `Horizontal`, добавляемая панель будет расположена справа от ранее добавленной панели, а если `Vertical` — снизу;

♦ `removeDockWidget(<QDockWidget>)` — удаляет панель из окна и скрывает ее. При этом объект панели не удаляется и впоследствии может быть добавлен в другую область;

♦ `dockWidgetArea(<QDockWidget>)` — возвращает местоположение указанной панели в виде значений атрибутов `LeftDockWidgetArea` (слева), `RightDockWidgetArea` (справа), `TopDockWidgetArea` (сверху), `BottomDockWidgetArea` (снизу) или `NoDockWidgetArea` (положение не определено) класса `QtCore.Qt`;

♦ `setDockOptions(<Опции>)` — устанавливает опции для прикрепляемых панелей. Значение по умолчанию: `AnimatedDocks | AllowTabbedDocks`. В качестве значения указывается комбинация (через оператор `|`) следующих атрибутов класса `QMainWindow`:

- `AnimatedDocks` — если опция установлена, вставка панелей в новое место по окончании перемещения будет производиться с анимацией;
- `AllowNestedDocks` — если опция установлена, в любую область можно будет поместить несколько панелей;

- AllowTabbedDocks — если опция установлена, панели могут быть наложены одна на другую. Для переключения между накладывающимися панелями будет использоваться панель с вкладками;
- ForceTabbedDocks — если опция установлена, панели не могут быть расположены рядом друг с другом. При этом опция AllowNestedDocks игнорируется;
- VerticalTabs — если опция установлена, заголовки вкладок будут отображаться с внешнего края области (если область справа, то и заголовки вкладок справа, если область слева, то и заголовки слева, если область сверху, то и заголовки вкладок сверху, если область снизу, то и заголовки вкладок снизу). Если опция не установлена, заголовки вкладок отображаются снизу. Опция AllowTabbedDocks должна быть установлена;
- GroupedDragging — если опция установлена, при перетаскивании любой панели все остальные панели, наложенные на нее, будут перетаскиваться вместе с ней. Опция AllowTabbedDocks должна быть установлена. Поддерживается, начиная с PyQt 5.6;

Внимание!

Опции необходимо устанавливать до добавления прикрепляемых панелей. Исключением являются опции AnimatedDocks и VerticalTabs.

- dockOptions() — возвращает комбинацию установленных опций;
- setDockNestingEnabled(<Флаг>) — если указано значение True, метод устанавливает опцию AllowNestedDocks, а если указано значение False — сбрасывает ее. Метод является слотом;
- isDockNestingEnabled() — возвращает значение True, если опция AllowNestedDocks установлена, и False — в противном случае;
- setTabPosition(<Область>, <Позиция>) — задает позицию отображения заголовков вкладок для указанной области. По умолчанию заголовки вкладок отображаются снизу. В качестве параметра <Позиция> могут быть указаны следующие атрибуты класса QTabWidget:
  - North — 0 — сверху;
  - South — 1 — снизу;
  - West — 2 — слева;
  - East — 3 — справа;
- tabPosition(<Область>) — возвращает позицию отображения заголовков вкладок прикрепляемых панелей для указанной области;
- setTabShape(<Форма>) — задает форму углов ярлыков вкладок. Могут быть указаны следующие атрибуты класса QTabWidget:
  - Rounded — 0 — скругленные углы (значение по умолчанию);
  - Triangular — 1 — треугольная форма;
- tabShape() — возвращает форму углов ярлыков вкладок в области заголовка;
- setCorner(<Угол>, <Область>) — позволяет закрепить указанный угол за определенной областью. По умолчанию верхние углы закреплены за верхней областью, а нижние — за нижней областью. В качестве параметра <Область> могут быть указаны следующие атрибуты класса QtCore.Qt: LeftDockWidgetArea (слева), RightDockWidgetArea (справа),

TopDockWidgetArea (сверху) или BottomDockWidgetArea (снизу). В параметре <Угол> указываются следующие атрибуты класса QtCore.Qt:

- TopLeftCorner — левый верхний угол;
- TopRightCorner — правый верхний угол;
- BottomLeftCorner — левый нижний угол;
- BottomRightCorner — правый нижний угол;

- corner(<Угол>) — возвращает область, за которой закреплен указанный угол;
- splitDockWidget(<QDockWidget 1>, <QDockWidget 2>, <Ориентация>) — разделяет область, занимаемую панелью <QDockWidget 1>, и добавляет панель <QDockWidget 1> в первую часть области, а панель <QDockWidget 2> — во вторую. Порядок расположения частей зависит от параметра <Ориентация>. В качестве параметра <Ориентация> могут быть указаны следующие атрибуты класса QtCore.Qt: Horizontal или Vertical. Если указан атрибут Horizontal, то панель <QDockWidget 2> будет расположена справа, а если Vertical — то снизу. Если панель <QDockWidget 1> расположена на вкладке, панель <QDockWidget 2> будет добавлена на новую вкладку, и разделения области при этом не произойдет;
- tabifyDockWidget(<QDockWidget 1>, <QDockWidget 2>) — размещает панель <QdockWidget 2> над панелью <QDockWidget 1>, создавая таким образом область с вкладками;
- tabifiedDockWidgets(<QDockWidget>) — возвращает список ссылок на панели (экземпляры класса QDockWidget), которые расположены на других вкладках в области панели, указанной в качестве параметра;
- resizeDocks(<Список QDockWidget>, <Список размеров>, <Ориентация>) — задает новые размеры для панелей, указанных в параметре <Список QDockWidhet>, при этом первый размер из указанных в параметре <Список размеров> будет применен к первой панели, второй — ко второй панели и т. д. Если третьим параметром передан атрибут Horizontal класса QtCore.Qt, <Список размеров> задаст значения ширины панелей, если указан атрибут Vertical — высоты. Размеры панелей, не указанных в первом списке, будут изменены таким образом, чтобы вместиться в оставшееся свободное пространство. Размеры самого окна при этом не изменяются. Метод поддерживается, начиная с PyQt 5.6;
- saveState([version=0]) — возвращает экземпляр класса QByteArray с размерами и положением всех панелей инструментов и прикрепляемых панелей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода restoreState(<QByteArray>[, version=0]).

  Обратите внимание, что все панели инструментов и прикрепляемые панели должны иметь уникальные объектные имена. Задать объектное имя можно с помощью метода setObjectName(<Имя в виде строки>) класса QObject:

```
self.dw = QtWidgets.QDockWidget("MyDockWidget1")
self.dw.setObjectName("MyDockWidget1")
```

  Чтобы сохранить размеры окна, следует воспользоваться методом saveGeometry() класса QWidget. Метод возвращает экземпляр класса QByteArray с размерами окна. Чтобы восстановить размеры окна, достаточно вызвать метод restoreGeometry(<QByteArray>);
- restoreDockWidget(<QDockWidget>) — восстанавливает состояние указанной панели, если она была создана после вызова метода restoreState(). Метод возвращает значение True, если состояние панели успешно восстановлено, и False — в противном случае.

В PyQt 5.8 появилась поддержка сигнала tabifiedDockWidgetActivated(<QDockWidget>), генерируемого при активизации вкладки щелчком на ее заголовке. Активизированная вкладка передается обработчику с единственным параметром.

27.2. Меню

Главное меню позволяет компактно поместить в главное окно множество команд, объединенных в логические группы. Оно состоит из горизонтальной панели (реализуемой классом QMenuBar), на которой расположены отдельные меню (реализуются с помощью класса QMenu) верхнего уровня. Каждое меню может содержать множество пунктов (представляемых классом QAction), разделители и вложенные меню. Пункт меню может содержать значок, текст и флажок, превращающий его в переключатель.

Помимо главного меню, в приложениях часто используются контекстные меню, которые обычно отображаются при щелчке правой кнопкой мыши в области компонента. Контекстное меню реализуется с помощью класса QMenu и отображается внутри метода с предопределенным именем contextMenuEvent() с помощью метода exec() (или оставленного для совместимости с предыдущей версией PyQt метода exec_()), в который передаются глобальные координаты щелчка мышью.

27.2.1. Класс *QMenuBar*

Класс QMenuBar описывает горизонтальную панель меню. Панель меню реализована в главном окне приложения по умолчанию. Получить ссылку на нее можно вызовом метода menuBar() класса QMainWindow. Установить свою панель меню позволяет метод setMenuBar(<QMenuBar>). Иерархия наследования для класса QMenuBar выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMenuBar
```

Конструктор класса QMenuBar имеет следующий формат:

```
<Объект> = QMenuBar([parent=None])
```

В параметре parent указывается ссылка на родительское окно.

Класс QMenuBar наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-5/qmenubar.html**):

- addMenu(<QMenu>) — добавляет меню на панель и возвращает экземпляр класса QAction, с помощью которого, например, можно скрыть меню (вызовом метода setVisible()) или сделать его неактивным (вызовом метода setEnabled());
- addMenu([<QIcon>,]<Название>) — создает меню, добавляет его на панель и возвращает ссылку на него (экземпляр класса QMenu). Внутри текста в параметре <Название> символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ &, будет — в качестве подсказки пользователю — подчеркнут. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню окажется выбранным. Чтобы вывести сам символ &, необходимо его удвоить. В первом параметре можно указать значок, который будет выведен левее надписи меню;
- insertMenu(<QAction>, <QMenu>) — добавляет меню <QMenu> перед пунктом <QAction>. Метод возвращает экземпляр класса QAction;

- `addAction()` — добавляет пункт в меню. Форматы метода:

  ```
  addAction(<QAction>)
  addAction(<Название>)                          -> QAction
  addAction(<Название>, <Обработчик>)            -> QAction
  ```

  Первый формат просто добавляет заданный в виде действия (экземпляра класса `QAction`) пункт в меню. Второй и третий создают действие, добавляют его в меню и возвращают в качестве результата;
- `clear()` — удаляет все действия из панели меню;
- `setActiveAction(<QAction>)` — делает активным указанное действие;
- `activeAction()` — возвращает активное действие (экземпляр класса `QAction`) или значение `None`;
- `setDefaultUp(<Флаг>)` — если в качестве параметра указано значение `True`, пункты меню будут отображаться выше панели меню, а не ниже;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `False`, панель меню будет скрыта. Значение `True` отображает панель меню.

Класс `QMenuBar` поддерживает следующие сигналы:

- `hovered(<QAction>)` — генерируется при наведении указателя мыши на пункт меню, который передается обработчику в качестве параметра;
- `triggered(<QAction>)` — генерируется при выборе пункта меню, который передается обработчику в качестве параметра.

27.2.2. Класс *QMenu*

Класс `QMenu` реализует отдельное меню на панели меню, а также вложенное, плавающее и контекстное меню. Его иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMenu
```

Форматы конструктора класса `QMenu`:

```
<Объект> = QMenu([parent=None])
<Объект> = QMenu(<Название>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент. Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ `&`, будет — в качестве подсказки пользователю — подчеркнут. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню окажется выбранным. Чтобы вывести сам символ `&`, необходимо его удвоить.

Помимо унаследованных из базовых классов, класс `QMenu` поддерживает ряд своих методов (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qmenu.html**):

- `addAction()` — добавляет пункт в меню. Форматы метода:

  ```
  addAction(<QAction>)
  addAction(<Название>)                          -> QAction
  addAction(<QIcon>, <Название>)                 -> QAction
  ```

```
addAction(<Название>, <Обработчик>[, shortcut=0])
                                        -> QAction
addAction(<QIcon>, <Название>, <Обработчик>[, shortcut=0])
                                        -> QAction
```

Внутри текста в параметре <Название> символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ &, будет — в качестве подсказки пользователю — подчеркнут. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню окажется выбранным. Чтобы вывести сам символ &, необходимо его удвоить. Нажатие комбинации клавиш быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным.

Параметр shortcut задает комбинацию «горячих» клавиш, нажатие которых аналогично выбору пункта в меню. Нажатие комбинации «горячих» клавиш сработает даже в том случае, если меню не является активным. Вот примеры указания значения в параметре shortcut:

```
QtGui.QKeySequence("Ctrl+R")
QtGui.QKeySequence(QtCore.Qt.CTRL + QtCore.Qt.Key_R))
QtGui.QKeySequence.fromString("Ctrl+R")
```

- addSeparator() — добавляет разделитель в меню и возвращает представляющий его экземпляр класса QAction;
- addSection() — добавляет в меню секцию, которая выглядит как разделитель с заголовком и, возможно, значком. Форматы метода:

  ```
  addSection(<Заголовок>)
  addSection(<Заголовок>, <QIcon>)
  ```

  Первый формат создает секцию с одним лишь текстовым заголовком, второй — с заголовком и значком. Метод возвращает экземпляр класса QAction, представляющий созданную секцию;
- insertMenu(<QAction>, <QMenu>) — добавляет вложенное меню <QMenu> перед пунктом <QAction>. Метод возвращает экземпляр класса QAction;
- insertSeparator(<QAction>) — добавляет разделитель перед указанным пунктом и возвращает представляющий разделитель экземпляр класса QAction;
- insertSection() — добавляет секцию перед пунктом <QAction>. Форматы метода:

  ```
  insertSection(<QAction>, <Заголовок>)
  insertSection(<QAction>, <Заголовок>, <QIcon>)
  ```

  Первый формат вставляет секцию с одним лишь текстовым заголовком, второй — с заголовком и значком. Метод возвращает экземпляр класса QAction, представляющий созданную секцию;
- addMenu(<QMenu>) — добавляет вложенное меню и возвращает представляющий его экземпляр класса QAction;
- addMenu([<QIcon>,]<Название>) — создает вложенное меню, добавляет его в меню и возвращает ссылку на него (экземпляр класса QMenu);
- clear() — удаляет все действия из меню;
- isEmpty() — возвращает значение True, если меню не содержит видимых пунктов, и False — в противном случае;

- `menuAction()` — возвращает объект действия (экземпляр класса `QAction`), связанный с данным меню. С помощью этого объекта можно скрыть меню (вызовом метода `setVisible()`) или сделать его неактивным (вызовом метода `setEnabled()`);
- `setTitle(<Название>)` — задает название меню;
- `title()` — возвращает название меню;
- `setIcon(<QIcon>)` — задает значок меню;
- `icon()` — возвращает значок меню (экземпляр класса `QIcon`);
- `setActiveAction(<QAction>)` — делает активным указанный пункт;
- `activeAction()` — возвращает активный пункт (экземпляр класса `QAction`) или значение `None`;
- `setDefaultAction(<QAction>)` — задает пункт по умолчанию;
- `defaultAction()` — возвращает пункт по умолчанию (экземпляр класса `QAction`) или значение `None`;
- `setTearOffEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, в начало меню добавляется пункт с пунктирной линией, с помощью щелчка на котором можно оторвать меню от панели и сделать его плавающим (отображаемым в отдельном окне, которое можно разместить в любой части экрана);
- `isTearOffEnabled()` — возвращает значение `True`, если меню может быть плавающим, и `False` — в противном случае;
- `isTearOffMenuVisible()` — возвращает значение `True`, если плавающее меню отображается в отдельном окне, и `False` — в противном случае;
- `showTearOffMenu([<QPoint>])` — принудительно открывает меню, для которого была указана такая возможность, и помещает его либо в точке, заданной (в виде экземпляра класса `QPoint`) в параметре, либо, если параметр не указан, в точке, где расположен курсор мыши. Поддерживается, начиная с PyQt 5.7;
- `hideTearOffMenu()` — скрывает плавающее меню;
- `setSeparatorsCollapsible(<Флаг>)` — если в качестве параметра указано значение `True`, вместо нескольких разделителей, идущих подряд, будет отображаться один. Кроме того, разделители, расположенные по краям меню, также будут скрыты;
- `setToolTipsVisible(<Флаг>)` — если передано значение `True`, заданная для меню всплывающая подсказка будет отображаться на экране, если `False` — не будет (поведение по умолчанию). Всплывающую подсказку для меню можно задать вызовом метода `setToolTip(<Текст>)`, унаследованным от класса `QWidget`;
- `popup(<QPoint>[, <QAction>])` — отображает меню по указанным глобальным координатам. Если указан второй параметр, меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню;
- `exec([<QPoint>][, action=None]])` — отображает меню по указанным глобальным координатам и возвращает экземпляр класса `QAction` (соответствующий выбранному пункту) или значение `None` (если пункт не выбран — например, нажата клавиша <Esc>). Если в параметре `action` указано действие (экземпляр класса `QAction`), меню отображается таким образом, чтобы по координатам был расположен соответствующий пункт меню;
- `exec(<Список с экземплярами класса QAction>, <QPoint>[, at=None][, parent=None])` — выводит меню по указанным глобальным координатам и возвращает экземпляр класса

QAction (соответствующий выбранному пункту) или значение None (если пункт не выбран — например, нажата клавиша <Esc>). Если в параметре at указано действие (экземпляр класса QAction), меню отображается таким образом, чтобы по координатам был расположен соответствующий пункт меню. В параметре parent можно указать ссылку на родительский компонент. Метод является статическим.

Класс QMenu поддерживает следующие сигналы:

- hovered(<QAction>) — генерируется при наведении указателя мыши на пункт меню, который передается в обработчик в параметре;
- triggered(<QAction>) — генерируется при выборе пункта меню, который передается в обработчик в параметре;
- aboutToShow() — генерируется перед отображением меню;
- aboutToHide() — генерируется перед скрытием меню.

27.2.3. Контекстное меню компонента

Чтобы создать для компонента контекстное меню, также можно воспользоваться соответствующими методами класса QWidget:

- addAction(<QAction>) — добавляет пункт в конец меню;
- addActions(<Список с экземплярами класса QAction>) — добавляет несколько пунктов в конец меню;
- insertAction(<QAction 1>, <QAction 2>) — добавляет пункт <QAction 2> перед пунктом <QAction 1>;
- insertActions(<QAction>, <Список с экземплярами класса QAction>) — добавляет несколько пунктов, указанных во втором параметре перед пунктом <QAction>;
- actions() — возвращает список с действиями (экземпляры класса QAction);
- removeAction(<QAction>) — удаляет указанное действие из меню.

Необходимо также наследовать класс компонента и переопределить метод с названием contextMenuEvent(self, <event>). Этот метод будет автоматически вызываться при щелчке правой кнопкой мыши в области компонента. Внутри метода через параметр <event> доступен экземпляр класса QContextMenuEvent, который позволяет получить дополнительную информацию о событии. Класс QContextMenuEvent поддерживает следующие основные методы:

- x() и y() — возвращают координаты по осям X и Y соответственно в пределах области компонента;
- pos() — возвращает экземпляр класса QPoint с целочисленными координатами в пределах области компонента;
- globalX() и globalY() — возвращают координаты по осям X и Y соответственно в пределах экрана;
- globalPos() — возвращает экземпляр класса QPoint с координатами в пределах экрана.

Чтобы вывести собственно контекстное меню, внутри метода contextMenuEvent() следует вызвать метод exec() (или exec_()) объекта меню и передать ему результат выполнения метода globalPos():

```
def contextMenuEvent(self, event):
    self.context_menu.exec_(event.globalPos())
```

В этом примере меню создано внутри конструктора класса и сохранено в атрибуте `context_menu`. Если контекстное меню постоянно обновляется, его можно создавать при каждом вызове внутри метода `contextMenuEvent()` непосредственно перед выводом.

27.2.4. Класс *QAction*

Класс `QAction` описывает объект действия, который можно добавить в меню, на панель инструментов или прикрепить к какому-либо компоненту в качестве пункта его контекстного меню. Один и тот же объект действия допускается добавлять в несколько мест — например, в меню и на панель инструментов, — это позволяет управлять видимостью и доступностью действия централизованно. Иерархия наследования для класса `QAction` следующая:

```
QObject — QAction
```

Форматы конструктора класса `QAction`:

```
<Объект> = QAction(<QObject>)
<Объект> = QAction(<Название>, <QObject>)
<Объект> = QAction(<QIcon>, <Название>, <QObject>)
```

В параметре `<QObject>` указывается ссылка на родительский компонент или значение `None` (начиная с PyQt 5.7, этот параметр является необязательным). Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ `&`, будет — в качестве подсказки пользователю — подчеркнут. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню окажется выбранным. Чтобы вывести сам символ `&`, необходимо его удвоить. Параметр `<QIcon>` устанавливает значок.

Класс `QAction` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qaction.html**):

- `setText(<Название>)` — задает название действия. Внутри текста в параметре `<Название>` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. Нажатие комбинации клавиш быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным;
- `text()` — возвращает название действия;
- `setIcon(<QIcon>)` — устанавливает значок;
- `icon()` — возвращает значок (экземпляр класса `QIcon`);
- `setIconVisibleInMenu(<Флаг>)` — если в качестве параметра указано значение `False`, значок в меню отображаться не будет;
- `setSeparator(<Флаг>)` — если в качестве параметра указано значение `True`, объект станет разделителем;
- `isSeparator()` — возвращает значение `True`, если объект является разделителем, и `False` — в противном случае;
- `setShortcut(<QKeySequence>)` — задает комбинацию «горячих» клавиш. Нажатие комбинации «горячих» клавиш по умолчанию сработает даже в том случае, если меню не является активным. Вот примеры указания значения:

```
"Ctrl+O"
QtGui.QKeySequence("Ctrl+O")
QtGui.QKeySequence(QtCore.Qt.CTRL + QtCore.Qt.Key_O))
QtGui.QKeySequence.fromString("Ctrl+O")
QtGui.QKeySequence.Open
```

♦ `setShortcuts()` — позволяет задать сразу несколько комбинаций «горячих» клавиш. Форматы метода:

```
setShortcuts(<Список с экземплярами класса QKeySequence>)
setShortcuts(<Стандартная комбинация клавиш>)
```

В параметре `<Стандартная комбинация клавиш>` указываются атрибуты из перечисления `StandardKey` класса `QKeySequence` (например, `Open`, `Close`, `Copy`, `Cut` и т. д. — полный их список можно найти в документации по классу `QKeySequence`);

♦ `setShortcutContext(<Область>)` — задает область действия комбинации «горячих» клавиш. В качестве параметра указываются следующие атрибуты класса `QtCore.Qt`:

- `WidgetShortcut` — `0` — комбинация доступна, если родительский компонент имеет фокус;
- `WidgetWithChildrenShortcut` — `3` — комбинация доступна, если фокус имеет родительский компонент или любой дочерний компонент;
- `WindowShortcut` — `1` — комбинация доступна, если окно, в котором расположен компонент, является активным;
- `ApplicationShortcut` — `2` — комбинация доступна, если любое окно приложения является активным;

♦ `setToolTip(<Текст>)` — задает текст всплывающей подсказки;

♦ `toolTip()` — возвращает текст всплывающей подсказки;

♦ `setWhatsThis(<Текст>)` — задает текст справки;

♦ `whatsThis()` — возвращает текст справки;

♦ `setStatusTip(<Текст>)` — задает текст, который будет отображаться в строке состояния при наведении указателя мыши на пункт меню;

♦ `statusTip()` — возвращает текст для строки состояния;

♦ `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, действие является переключателем, который может находиться в двух состояниях: установленном и неустановленном;

♦ `isCheckable()` — возвращает значение `True`, если действие является переключателем, и `False` — в противном случае;

♦ `setChecked(<Флаг>)` — если в качестве параметра указано значение `True`, действие-переключатель будет находиться в установленном состоянии. Метод является слотом;

♦ `isChecked()` — возвращает значение `True`, если действие-переключатель находится в установленном состоянии, и `False` — в противном случае;

♦ `setDisabled(<Флаг>)` — если в качестве параметра указано значение `True`, действие станет недоступным. Значение `False` делает действие вновь доступным. Метод является слотом;

- setEnabled(<Флаг>) — если в качестве параметра указано значение False, действие станет недоступным. Значение True делает действие вновь доступным. Метод является слотом;
- isEnabled() — возвращает значение True, если действие доступно, и False — в противном случае;
- setVisible(<Флаг>) — если в качестве параметра указано значение False, действие будет скрыто. Значение True вновь выводит действие на экран. Метод является слотом;
- isVisible() — возвращает значение False, если действие скрыто, и True — в противном случае;
- setMenu(<QMenu>) — устанавливает вложенное меню;
- menu() — возвращает ссылку на вложенное меню (экземпляр класса QMenu) или значение None, если вложенного меню нет;
- setFont(<QFont>) — устанавливает шрифт для текста;
- font() — возвращает экземпляр класса QFont с текущими параметрами шрифта;
- setAutoRepeat(<Флаг>) — если в качестве параметра указано значение True (значение по умолчанию), действие будет повторяться, пока удерживается нажатой комбинация «горячих» клавиш;
- setPriority(<Приоритет>) — задает приоритет действия. В качестве параметра указываются атрибуты: LowPriority (низкий), NormalPriority (нормальный) или HighPriority (высокий) класса QAction;
- priority() — возвращает текущий приоритет действия;
- setData(<Данные>) — позволяет сохранить пользовательские данные любого типа в объекте действия;
- data() — возвращает пользовательские данные, сохраненные ранее с помощью метода setData(), или значение None;
- setActionGroup(<QActionGroup>) — добавляет действие в указанную группу;
- actionGroup() — возвращает ссылку на группу (экземпляр класса QActionGroup) или значение None;
- showStatusText([widget=None]) — отправляет событие QEvent.StatusTip указанному в параметре widget компоненту и возвращает значение True, если событие успешно отправлено. Если компонент не указан, событие посылается родителю действия.

  Чтобы обработать событие, необходимо наследовать класс компонента и переопределить метод event(self, <event>). При событии QEvent.StatusTip через параметр <event> доступен экземпляр класса QStatusTipEvent. Получить текст для строки состояния можно через метод tip() объекта события. Вот пример обработки события в классе, наследующем класс QLabel:

```
def event(self, e):
    if e.type() == QtCore.QEvent.StatusTip:
        self.setText(e.tip())
        return True
    return QtWidgets.QLabel.event(self, e)
```

- hover() — посылает сигнал hovered(). Метод является слотом;

- `toggle()` — производит изменение состояния переключателя на противоположное. Метод является слотом;
- `trigger()` — посылает сигнал `triggered()`. Метод является слотом.

Класс `QAction` поддерживает следующие сигналы:

- `changed()` — генерируется при изменении действия;
- `hovered()` — генерируется при наведении указателя мыши на объект действия;
- `toggled(<Состояние>)` — генерируется при изменении состояния переключателя. Внутри обработчика через параметр доступно текущее состояние в виде логической величины;
- `triggered(<Состояние переключателя>)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов, нажатии комбинации клавиш или вызове метода `trigger()`. В параметре доступно текущее состояние действия-переключателя в виде логической величины — для обычных действий значение параметра всегда равно `False`.

27.2.5. Объединение переключателей в группу

Класс `QActionGroup` позволяет объединить несколько действий-переключателей в группу. По умолчанию внутри группы может быть установлен только один переключатель — при попытке установить другой переключатель ранее установленный будет автоматически сброшен. Иерархия наследования для этого класса выглядит так:

```
QObject — QActionGroup
```

Конструктор класса `QActionGroup` имеет следующий формат:

```
<Объект> = QActionGroup(<QObject>)
```

В параметре `<QObject>` указывается ссылка на родительский компонент или значение `None`. После создания объекта группы он может быть указан в качестве родителя при создании объектов действия — в этом случае действие автоматически добавляется в группу.

Класс `QActionGroup` поддерживает следующие основные методы:

- `addAction()` — добавляет объект действия в группу. Метод возвращает экземпляр класса `QAction`. Форматы метода:

  ```
  addAction(<QAction>)
  addAction(<Название>)
  addAction(<QIcon>, <Название>)
  ```

- `removeAction(<QAction>)` — удаляет объект действия из группы;
- `actions()` — возвращает список с экземплярами класса `QAction`, которые были добавлены в группу, или пустой список;
- `checkedAction()` — возвращает ссылку (экземпляр класса `QAction`) на установленный переключатель внутри группы при использовании эксклюзивного режима или значение `None`;
- `setExclusive(<Флаг>)` — если в качестве параметра указано значение `False`, внутри группы может быть установлено произвольное количество действий-переключателей. Значение `True` возвращает эксклюзивный режим, при котором только одно действие-переключатель может быть установлено. Метод является слотом;
- `setDisabled(<Флаг>)` — если в качестве параметра указано значение `True`, все действия в группе станут недоступными. Значение `False` делает действия вновь доступными. Метод является слотом;

- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `False`, все действия в группе станут недоступными. Значение `True` делает действия вновь доступными. Метод является слотом;
- `isEnabled()` — возвращает значение `True`, если действия в группе доступны, и `False` — в противном случае;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `False`, все действия в группе будут скрыты. Значение `True` вновь выводит действия на экран. Метод является слотом;
- `isVisible()` — возвращает значение `False`, если действия группы скрыты, и `True` — в противном случае.

Класс `QActionGroup` поддерживает сигналы:

- `hovered(<QAction>)` — генерируется при наведении указателя мыши на объект действия внутри группы. Внутри обработчика через параметр доступна ссылка на этот объект действия;
- `triggered(<QAction>)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов или комбинации клавиш. Внутри обработчика через параметр доступна ссылка на этот объект действия.

27.3. Панели инструментов

Панели инструментов предназначены для отображения часто используемых команд. Добавить панель инструментов в главное окно позволяют методы `addToolBar()` и `insertToolBar()` класса `QMainWindow`. Один из форматов метода `addToolBar()` позволяет указать область, к которой изначально прикреплена панель. По умолчанию с помощью мыши пользователь может переместить панель в другую область окна или отобразить панель в отдельном окне. Ограничить перечень областей, к которым можно прикрепить панель, позволяет метод `setAllowedAreas()` класса `QToolBar`, а запретить отображение панели в отдельном окне — метод `setFloatable()`.

27.3.1. Класс *QToolBar*

Класс `QToolBar` реализует панель инструментов, которую можно перемещать с помощью мыши. Иерархия наследования для него такова:

```
(QObject, QPaintDevice) — QWidget — QToolBar
```

Форматы конструктора класса `QToolBar`:

```
<Объект> = QToolBar([parent=None])
<Объект> = QToolBar(<Название>[, parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент, а в параметре `<Название>` — название панели, которое отображается в контекстном меню при щелчке правой кнопкой мыши в области меню, области панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить панель инструментов.

Класс `QToolBar` наследует все методы базовых классов и поддерживает следующие собственные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qtoolbar.html**):

- `addAction()` — добавляет действие на панель инструментов. Форматы метода:

```
addAction(<QAction>)
addAction(<Название>)                            -> QAction
addAction(<QIcon>, <Название>)                   -> QAction
addAction(<Название>, <Обработчик>)              -> QAction
addAction(<QIcon>, <Название>, <Обработчик>) -> QAction
```

- `addSeparator()` — добавляет разделитель и возвращает представляющий его экземпляр класса `QAction`;
- `insertSeparator(<QAction>)` — добавляет разделитель перед указанным действием и возвращает представляющий разделитель экземпляр класса `QAction`;
- `addWidget(<QWidget>)` — позволяет добавить компонент (например, раскрывающийся список). Метод возвращает представляющий компонент экземпляр класса `QAction`;
- `insertWidget(<QAction>, <QWidget>)` — добавляет компонент перед указанным действием и возвращает представляющий компонент экземпляр класса `QAction`;
- `widgetForAction(<QAction>)` — возвращает ссылку на компонент (экземпляр класса `QWidget`), который связан с указанным действием;
- `clear()` — удаляет все действия из панели инструментов;
- `setOrientation(<Ориентация>)` — задает ориентацию панели в виде следующих атрибутов класса `QtCore.Qt`:
  - `Horizontal` — `1` — горизонтальная (значение по умолчанию);
  - `Vertical` — `2` — вертикальная;
- `setAllowedAreas(<Области>)` — задает области, к которым можно прикрепить панель инструментов. В качестве параметра указываются атрибуты (или их комбинация через оператор `|`) `LeftToolBarArea` (слева), `RightToolBarArea` (справа), `TopToolBarArea` (сверху), `BottomToolBarArea` (снизу) или `AllToolBarAreas` (все области) класса `QtCore.Qt`;
- `setMovable(<Флаг>)` — если в качестве параметра указано значение `False`, будет невозможно перемещать панель с помощью мыши. Значение `True` вновь разрешает перемещение панели;
- `isMovable()` — возвращает значение `True`, если панель можно перемещать с помощью мыши, и `False` — в противном случае;
- `setFloatable(<Флаг>)` — если в качестве параметра указано значение `False`, панель нельзя будет вынести в отдельное окно. Значение `True` вновь разрешает выносить панель в окно;
- `isFloatable()` — возвращает значение `True`, если панель можно вынести в отдельное окно, и `False` — в противном случае;
- `isFloating()` — возвращает значение `True`, если панель в данный момент вынесена в отдельное окно, и `False` — в противном случае;
- `setToolButtonStyle(<Стиль>)` — задает стиль кнопок на панели инструментов. Метод является слотом. В качестве параметра указываются следующие атрибуты класса `QtCore.Qt`:
  - `ToolButtonIconOnly` — `0` — отображается только значок;
  - `ToolButtonTextOnly` — `1` — отображается только текст;

- `ToolButtonTextBesideIcon` — `2` — текст отображается справа от значка;
- `ToolButtonTextUnderIcon` — `3` — текст отображается под значком;
- `ToolButtonFollowStyle` — `4` — зависит от используемого стиля;

♦ `toolButtonStyle()` — возвращает стиль кнопок на панели инструментов;

♦ `setIconSize(<QSize>)` — задает размеры значков. Метод является слотом;

♦ `iconSize()` — возвращает размеры значков (экземпляр класса `QSize`);

♦ `toggleViewAction()` — возвращает объект действия (экземпляр класса `QAction`), с помощью которого можно скрыть или отобразить панель.

Класс `QToolBar` поддерживает следующие сигналы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qtoolbar.html**):

♦ `actionTriggered(<QAction>)` — генерируется при нажатии кнопки на панели. Внутри обработчика через параметр доступно действие, связанное с этой кнопкой;

♦ `visibilityChanged(<Флаг>)` — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение `True`, если панель видима, и `False` — если скрыта;

♦ `topLevelChanged(<Флаг>)` — генерируется при изменении состояния панели. Внутри обработчика через параметр доступно значение `True`, если панель вынесена в отдельном окне, и `False` — если прикреплена к области.

27.3.2. Класс *QToolButton*

При добавлении действия на панель инструментов автоматически создается кнопка, представляемая классом `QToolButton`. Получить ссылку на кнопку позволяет метод `widgetForAction()` класса `QToolBar`. Иерархия наследования для класса `QToolButton` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QToolButton
```

Конструктор класса `QToolButton` имеет следующий формат:

```
<Объект> = QToolButton([parent=None])
```

В параметре `parent` указывается ссылка на родительский компонент.

Класс `QToolButton`, помимо методов базовых классов, поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qtoolbutton.html**):

♦ `setDefaultAction(<QAction>)` — связывает объект действия с кнопкой. Метод является слотом;

♦ `defaultAction()` — возвращает ссылку на объект действия (экземпляр класса `QAction`), связанный с кнопкой;

♦ `setToolButtonStyle(<Стиль>)` — задает стиль кнопки. Метод является слотом. Допустимые значения параметра приведены в описании метода `setToolButtonStyle()` (см. *разд. 27.3.1*);

♦ `toolButtonStyle()` — возвращает стиль кнопки;

♦ `setMenu(<QMenu>)` — добавляет к кнопке меню;

- `menu()` — возвращает ссылку на меню (экземпляр класса `QMenu`), добавленное к кнопке, или значение `None`;
- `showMenu()` — отображает меню, связанное с кнопкой. Метод является слотом;
- `setPopupMode(<Режим>)` — задает режим отображения меню, связанного с кнопкой. В качестве параметра указываются следующие атрибуты класса `QToolButton`:
  - `DelayedPopup` — `0` — меню отображается при удержании кнопки, нажатой в течение некоторого промежутка времени;
  - `MenuButtonPopup` — `1` — справа от кнопки отображается кнопка со стрелкой, нажатие которой приводит к немедленному открытию меню;
  - `InstantPopup` — `2` — нажатие кнопки приводит к немедленному открытию меню. Сигнал `triggered()` при этом не генерируется;
- `popupMode()` — возвращает режим отображения меню, связанного с кнопкой;
- `setArrowType(<Тип значка>)` — позволяет вместо стандартного значка действия установить значок в виде стрелки, указывающей в заданном направлении. В качестве параметра задаются атрибуты `NoArrow` (значение по умолчанию), `UpArrow`, `DownArrow`, `LeftArrow` или `RightArrow` класса `QtCore.Qt`;
- `setAutoRaise(<Флаг>)` — если в качестве параметра указано значение `False`, кнопка будет отображаться с рамкой. По умолчанию кнопка сливается с фоном, а при наведении указателя мыши становится выпуклой.

Класс `QToolButton` поддерживает сигнал `triggered(<QAction>)`, который генерируется при нажатии кнопки или комбинации клавиш, а также при выборе пункта в связанном меню. Внутри обработчика через параметр доступно соответствующее действие.

27.4. Прикрепляемые панели

Если возможностей панелей инструментов недостаточно, и необходимо вывести на экран компоненты, занимающие много места (например, таблицу или иерархический список), можно воспользоваться прикрепляемыми панелями. Прикрепляемые панели реализуются с помощью класса `QDockWidget`.

Его иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDockWidget
```

Форматы конструктора класса `QDockWidget`:

```
<Объект> = QDockWidget([parent=None][, flags=0])
<Объект> = QDockWidget(<Название>[, parent=None][, flags=0])
```

В параметре `<Название>` задается название панели, которое отображается в заголовке панели и в контекстном меню при щелчке правой кнопкой мыши в области меню, области панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить прикрепляемую панель. В параметре `parent` указывается ссылка на родительское окно. Доступные значения параметра `flags` мы рассматривали в *разд. 18.2*.

Класс `QDockWidget` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только интересующие нас — полный их список можно найти на странице **https://doc.qt.io/qt-5/qdockwidget.html**):

- setWidget(<QWidget>) — устанавливает компонент, который будет отображаться на прикрепляемой панели;
- widget() — возвращает ссылку на компонент, расположенный на панели;
- setTitleBarWidget(<QWidget>) — позволяет указать пользовательский компонент, отображаемый в заголовке панели;
- titleBarWidget() — возвращает ссылку на пользовательский компонент, расположенный в заголовке панели, или значение None, если компонент не был установлен;
- setAllowedAreas(<Области>) — задает области, к которым можно прикрепить панель. В качестве параметра указываются атрибуты (или их комбинация через оператор |) LeftDockWidgetArea (слева), RightDockWidgetArea (справа), TopDockWidgetArea (сверху), BottomDockWidgetArea (снизу) или AllDockWidgetAreas (все области) класса QtCore.Qt;
- setFloating(<Флаг>) — если в качестве параметра указано значение True, панель отобразится в отдельном окне, а если указано значение False, она будет прикреплена к какой-либо области;
- isFloating() — возвращает значение True, если панель отображается в отдельном окне, и False — в противном случае;
- setFeatures(<Свойства>) — устанавливает свойства панели. В качестве параметра указывается комбинация (через оператор |) следующих атрибутов класса QDockWidget:
  - DockWidgetClosable — панель можно закрыть;
  - DockWidgetMovable — панель можно перемещать с помощью мыши;
  - DockWidgetFloatable — панель можно вынести в отдельное окно;
  - DockWidgetVerticalTitleBar — заголовок панели отображается с левой стороны, а не сверху;
  - NoDockWidgetFeatures — панель нельзя закрыть, переместить и вынести в отдельное окно;
- features() — возвращает комбинацию установленных свойств панели;
- toggleViewAction() — возвращает объект действия (экземпляр класса QAction), с помощью которого можно скрыть или отобразить панель.

Класс QDockWidget поддерживает следующие полезные сигналы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qdockwidget.html**):

- dockLocationChanged(<Область>) — генерируется при переносе панели в другую область. Внутри обработчика через параметр доступен целочисленный идентификатор области, на которую была перенесена панель;
- visibilityChanged(<Флаг>) — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение True, если панель видима, и False — если скрыта;
- topLevelChanged(<Флаг>) — генерируется при изменении состояние панели. Внутри обработчика через параметр доступно значение True, если панель вынесена в отдельное окно, и False — если прикреплена к области.

27.5. Управление строкой состояния

Класс `QStatusBar` реализует строку состояния, в которой можно выводить различные сообщения. Помимо текстовой информации, туда можно добавить различные компоненты, например индикатор хода выполнения процесса. Строка состояния состоит из трех секций:

- *секция для временных сообщений.* Реализована по умолчанию. В эту секцию, в частности, при наведении указателя мыши на пункт меню или кнопку на панели инструментов выводятся сообщения, сохраненные в объекте действия с помощью метода `setStatusTip()`. Вывести пользовательское сообщение во временную секцию можно с помощью метода `showMessage()`;
- *обычная секция.* При выводе временного сообщения содержимое обычной секции скрывается. Чтобы отображать сообщения в этой секции, необходимо предварительно добавить туда компоненты вызовом метода `addWidget()` или `insertWidget()`. Добавленные компоненты выравниваются по левой стороне строки состояния;
- *постоянная секция.* При выводе временного сообщения содержимое постоянной секции не скрывается. Чтобы отображать там сообщения, необходимо предварительно добавить туда компоненты методом `addPermanentWidget()` или `insertPermanentWidget()`. Добавленные компоненты выравниваются по правой стороне строки состояния.

Получить ссылку на строку состояния, установленную в главном окне, позволяет метод `statusBar()` класса `QMainWindow`, а установить пользовательскую панель вместо стандартной можно с помощью метода `setStatusBar(<QStatusBar>)`. Иерархия наследования для класса `QStatusBar` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QStatusBar
```

Формат конструктора класса `QStatusBar`:

```
<Объект> = QStatusBar([parent=None])
```

В параметре `parent` указывается ссылка на родительское окно.

Класс `QStatusBar` наследует все методы базовых классов и поддерживает следующие дополнительные методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qstatusbar.html**):

- `showMessage(<Текст>[, msecs=0])` — выводит временное сообщение. Во втором параметре можно указать время в миллисекундах, на которое показывается сообщение: если указано значение `0`, то сообщение показывается, пока не будет выведено новое сообщение или вызван метод `clearMessage()`. Метод является слотом;
- `currentMessage()` — возвращает временное сообщение, отображаемое в текущий момент;
- `clearMessage()` — удаляет временное сообщение. Метод является слотом;
- `addWidget(<QWidget>[, stretch=0])` — добавляет указанный компонент в конец обычной секции. В параметре `stretch` может быть указан фактор растяжения;
- `insertWidget(<Индекс>, <QWidget>[, stretch=0])` — добавляет компонент в указанную позицию обычной секции и возвращает индекс позиции. В параметре `stretch` может быть указан фактор растяжения;
- `addPermanentWidget(<QWidget>[, stretch=0])` — добавляет указанный компонент в конец постоянной секции. В параметре `stretch` может быть указан фактор растяжения;

- insertPermanentWidget(<Индекс>, <QWidget>[, stretch=0]) — добавляет компонент в указанную позицию постоянной секции и возвращает индекс позиции. В параметре stretch может быть указан фактор растяжения;
- removeWidget(<QWidget>) — удаляет компонент из обычной или постоянной секций. При этом сам компонент не удаляется, а только скрывается, лишается родителя и в дальнейшем может быть помещен в другое место;
- setSizeGripEnabled(<Флаг>) — если в качестве параметра указано значение True, в правом нижнем углу строки состояния будет отображаться маркер изменения размера. Значение False скрывает маркер.

Класс QStatusBar поддерживает сигнал messageChanged(<Сообщение>), генерируемый при изменении текста во временной секции. Внутри обработчика через параметр доступно новое сообщение или пустая строка.

27.6. MDI-приложения

MDI-приложения (Multiple Document Interface) позволяют, как уже было отмечено ранее, открыть несколько документов одновременно в разных вложенных окнах. Чтобы создать MDI-приложение, следует в качестве центрального компонента установить компонент QMdiArea вызовом метода setCentralWidget() класса QMainWindow. Отдельное окно внутри MDI-области представляется классом QMdiSubWindow.

27.6.1. Класс *QMdiArea*

Класс QMdiArea реализует MDI-область, внутри которой могут располагаться вложенные окна (экземпляры класса QMdiSubWindow). Иерархия наследования для класса QMdiArea выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QMdiArea
```

Конструктор класса QMdiArea имеет следующий формат:

```
<Объект> = QMdiArea([parent=None])
```

В параметре parent указывается ссылка на родительское окно.

Класс QMdiArea поддерживает следующие методы (здесь приведены только основные — полный их список можно найти на странице **https://doc.qt.io/qt-5/qmdiarea.html**):

- addSubWindow(<QWidget>[, flags=0]) — создает вложенное окно с флагами flags (см. *разд. 18.2*), добавляет в него заданный в первом параметре компонент и возвращает ссылку на созданное окно (экземпляр класса QMdiSubWindow):

  ```
  w = MyWidget()
  sWindow = self.mdi_area.addSubWindow(w)
  sWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose)
  # ... Задаем параметры окна
  sWindow.show()
  ```

  Чтобы окно автоматически удалялось при закрытии, необходимо установить атрибут WA_DeleteOnClose, а чтобы отобразить окно, — вызвать метод show().

  В первом параметре этого метода также можно указать ссылку на существующее вложенное окно (экземпляр класса QMdiSubWindow):

```
w = MyWidget()
sWindow = QtWidgets.QMdiSubWindow()
self.mdi_area.addSubWindow(sWindow)
sWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose)
# ... Производим настройку свойств окна
sWindow.setWidget(w)
sWindow.show()
```

Также можно добавить вложенное окно в MDI-область, указав последнюю в качестве родителя при создании объекта вложенного окна:

```
sWindow = QtWidgets.QMdiSubWindow(self.mdi_area)
```

- `activeSubWindow()` — возвращает ссылку на активное вложенное окно (экземпляр класса `QMdiSubWindow`) или значение `None`;
- `currentSubWindow()` — возвращает ссылку на текущее вложенное окно (экземпляр класса `QMdiSubWindow`) или значение `None`. Результат выполнения этого метода аналогичен таковому у метода `activeSubWindow()`, если MDI-область находится в активном окне;
- `subWindowList([order=CreationOrder])` — возвращает список со ссылками на все вложенные окна (экземпляры класса `QMdiSubWindow`), добавленные в MDI-область, или пустой список. В параметре `order` указываются следующие атрибуты класса `QMdiArea`:
  - `CreationOrder` — `0` — окна в списке располагаются в порядке их создания;
  - `StackingOrder` — `1` — окна в списке располагаются в порядке размещения их на экране. Последний элемент в списке будет содержать ссылку на самое верхнее окно, а последний — ссылку на самое нижнее окно;
  - `ActivationHistoryOrder` — `2` — окна в списке располагаются в порядке истории получения фокуса. Последний элемент в списке будет содержать ссылку на окно, получившее фокус последним;
- `removeSubWindow(<QWidget>)` — удаляет вложенное окно из MDI-области;
- `setActiveSubWindow(<QMdiSubWindow>)` — делает указанное вложенное окно активным. Если задать значение `None`, все вложенные окна станут неактивными. Метод является слотом;
- `setActivationOrder(<Порядок>)` — задает порядок передачи фокуса при использовании методов `activatePreviousSubWindow()`, `activateNextSubWindow()` и др. В параметре указываются те же атрибуты, что и в параметре метода `subWindowList()`;
- `activationOrder()` — возвращает порядок передачи фокуса;
- `activatePreviousSubWindow()` — делает активным предыдущее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`;
- `activateNextSubWindow()` — делает активным следующее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`;
- `closeActiveSubWindow()` — закрывает активное вложенное окно. Метод является слотом;
- `closeAllSubWindows()` — закрывает все вложенные окна. Метод является слотом;
- `cascadeSubWindows()` — выстраивает вложенные окна в виде стопки. Метод является слотом;

- `tileSubWindows()` — выстраивает вложенные окна в виде мозаики. Метод является слотом;
- `setViewMode(<Режим>)` — задает режим отображения вложенных окон в MDI-области. В параметре `<Режим>` указываются следующие атрибуты класса `QMdiArea`:
  - `SubWindowView` — `0` — в виде собственно окон (по умолчанию);
  - `TabbedView` — `1` — в виде отдельных вкладок на панели с вкладками;
- `viewMode()` — возвращает режим отображения вложенных окон в MDI-области;
- `setTabPosition(<Позиция>)` — задает позицию отображения заголовков вкладок при использовании режима `TabbedView`. По умолчанию заголовки вкладок отображаются сверху. В качестве параметра `<Позиция>` могут быть указаны следующие атрибуты класса `QTabWidget`:
  - `North` — `0` — сверху;
  - `South` — `1` — снизу;
  - `West` — `2` — слева;
  - `East` — `3` — справа;
- `tabPosition(<Область>)` — возвращает позицию отображения заголовков вкладок при использовании режима `TabbedView`;
- `setTabShape(<Форма>)` — задает форму углов ярлыков вкладок при использовании режима `TabbedView`. Могут быть указаны следующие атрибуты класса `QTabWidget`:
  - `Rounded` — `0` — скругленные углы (значение по умолчанию);
  - `Triangular` — `1` — треугольная форма;
- `tabShape()` — возвращает форму углов ярлыков вкладок в области заголовка при использовании режима `TabbedView`;
- `setTabsMovable(<Флаг>)` — если передано значение `True`, вкладки на панели при использовании режима `TabbedView` можно перемещать мышью;
- `setTabsClosable(<Флаг>)` — если передано значение `True`, вкладки на панели при использовании режима `TabbedView` можно закрывать щелчком на расположенной в заголовке вкладки кнопке закрытия;
- `setBackground(<QBrush>)` — задает кисть для заполнения фона MDI-области;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, заданная в первом параметре опция будет установлена, если `False` — сброшена. В параметре `<Опция>` может быть указан атрибут `DontMaximizeSubWindowOnActivation` класса `QMdiArea`, в случае установки запрещающий автоматически разворачивать следующее вложенное окно при переключении на него из уже развернутого окна;
- `testOption(<Опция>)` — возвращает значение `True`, если указанная опция установлена, и `False` — в противном случае.

Класс `QMdiArea` поддерживает сигнал `subWindowActivated(<QMdiSubWindow>)`, генерируемый при активизации другого вложенного окна. В обработчике через параметр доступна ссылка на активизированное вложенное окно или значение `None`.

27.6.2. Класс *QMdiSubWindow*

Класс `QMdiSubWindow` реализует окно, которое может быть отображено внутри MDI-области. Иерархия наследования для него выглядит следующим образом:

```
(QObject, QPaintDevice) — QWidget — QMdiSubWindow
```

Формат конструктора класса `QMdiSubWindow`:

```
<Объект> = QMdiSubWindow([parent=None][, flags=0])
```

В параметре `parent` указывается ссылка на родительское окно. Доступные значения параметра `flags` мы рассматривали в *разд. 18.2*.

Класс `QMdiSubWindow` наследует все методы базовых классов и дополнительно поддерживает следующие методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qmdisubwindow.html**):

- `setWidget(<QWidget>)` — помещает компонент в окно;
- `widget()` — возвращает ссылку на помещенный в окно компонент;
- `mdiArea()` — возвращает ссылку на MDI-область (экземпляр класса `QMdiArea`) или значение `None`;
- `setSystemMenu(<QMenu>)` — позволяет установить пользовательское системное меню окна вместо стандартного;
- `systemMenu()` — возвращает ссылку на системное меню окна (экземпляр класса `QMenu`) или значение `None`;
- `showSystemMenu()` — открывает системное меню окна. Метод является слотом;
- `setKeyboardSingleStep(<Значение>)` — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками. Чтобы изменить размеры окна или его положение с клавиатуры, необходимо в системном меню окна выбрать пункт **Переместить** или **Размер**. Значение по умолчанию — `5`;
- `setKeyboardPageStep(<Значение>)` — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками при удержании нажатой клавиши <Shift>. Значение по умолчанию — `20`;
- `showShaded()` — сворачивает содержимое окна, оставляя только заголовок. Метод является слотом;
- `isShaded()` — возвращает значение `True`, если отображается только заголовок окна, и `False` — в противном случае;
- `setOption(<Опция>[, on=True])` — если во втором параметре указано значение `True`, заданная в первом параметре опция будет установлена, а если `False` — сброшена. В параметре `<Опция>` могут быть указаны следующие атрибуты класса `QMdiSubWindow`:
  - `RubberBandResize` — если опция установлена, при изменении размеров окна станут изменяться размеры вспомогательного компонента, а не самого окна. По окончании изменения размеров будут изменены размеры окна;
  - `RubberBandMove` — если опция установлена, при изменении положения окна станет перемещаться вспомогательный компонент, а не само окно. По окончании перемещения будет изменено положение окна;
- `testOption(<Опция>)` — возвращает значение `True`, если указанная опция установлена, и `False` — в противном случае.

Класс `QMdiSubWindow` поддерживает следующие сигналы:

- `aboutToActivate()` — генерируется перед активацией вложенного окна;
- `windowStateChanged(<Старое состояние>, <Новое состояние>)` — генерируется при изменении состояния окна. Внутри обработчика через параметры доступны целочисленные обозначения старого и нового состояний.

27.7. Добавление значка приложения в область уведомлений

Класс `QSystemTrayIcon` позволяет добавить значок приложения в область уведомлений, расположенную в правой части панели задач Windows. Иерархия наследования для этого класса выглядит так:

```
QObject — QSystemTrayIcon
```

Форматы конструктора класса `QSystemTrayIcon`:

```
<Объект> = QSystemTrayIcon([parent=None])
<Объект> = QSystemTrayIcon(<QIcon>[, parent=None])
```

В параметре `parent` указывается ссылка на родительское окно.

Класс `QSystemTrayIcon` поддерживает следующие основные методы:

- `isSystemTrayAvailable()` — возвращает значение `True`, если область уведомлений доступна, и `False` — в противном случае. Метод является статическим;
- `setIcon(<QIcon>)` — устанавливает значок. Задать значок также можно в конструкторе класса;
- `icon()` — возвращает значок (экземпляр класса `QIcon`);
- `setContextMenu(<QMenu>)` — устанавливает контекстное меню, отображаемое при щелчке правой кнопкой мыши на значке;
- `contextMenu()` — возвращает ссылку на контекстное меню (экземпляр класса `QMenu`);
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `toolTip()` — возвращает текст всплывающей подсказки;
- `setVisible(<Флаг>)` — если в качестве параметра указано значение `True`, значок будет выведен на экран, а если `False` — то скрыт. Метод является слотом;
- `show()` — отображает значок. Метод является слотом;
- `hide()` — скрывает значок. Метод является слотом;
- `isVisible()` — возвращает значение `True`, если значок присутствует на экране, и `False` — в противном случае;
- `geometry()` — возвращает экземпляр класса `QRect` с размерами и координатами значка на экране;
- `showMessage()` — позволяет отобразить сообщение в области уведомлений. Форматы метода:

  ```
  showMessage(<Заголовок>, <Текст сообщения>[, icon=Information][, msecs=10000])
  showMessage(<Заголовок>, <Текст сообщения>, <QIcon>[, msecs=10000])
  ```

В первом формате необязательный параметр `icon` задает значок, который отображается слева от заголовка сообщения. В качестве значения можно указать атрибуты `NoIcon`, `Information`, `Warning` или `Critical` класса `QSystemTrayIcon`.

Во втором формате, поддерживаемом, начиная с PyQt 5.9, третьим параметром, можно указать произвольный значок.

Необязательный параметр `msecs` задает промежуток времени, в течение которого сообщение будет присутствовать на экране. Обратите внимание, что сообщение может не показываться вообще, кроме того, значение параметра `msecs` в некоторых операционных системах игнорируется.

Метод является слотом;

- `supportsMessages()` — возвращает значение `True`, если поддерживается вывод сообщений, и `False` — в противном случае. Метод является статическим.

Класс `QSystemTrayIcon` поддерживает следующие сигналы:

- `activated(<Причина>)` — генерируется при щелчке мышью на значке. Внутри обработчика через параметр доступна причина в виде целочисленного значения одного из следующих атрибутов класса `QSystemTrayIcon`:
  - `Unknown` — `0` — неизвестная причина;
  - `Context` — `1` — нажата правая кнопка мыши;
  - `DoubleClick` — `2` — двойной щелчок мышью;
  - `Trigger` — `3` — нажата левая кнопка мыши;
  - `MiddleClick` — `4` — нажата средняя кнопка мыши;
- `messageClicked()` — генерируется при щелчке мышью на сообщении.

ГЛАВА 28

Мультимедиа

Библиотека PyQt 5 предоставляет развитые средства воспроизведения звука и видео. Для этого она задействует мультимедийную подсистему Windows Media Foundation, входящую в состав Windows. Следовательно, все форматы, поддерживаемые операционной системой, будут гарантированно поддерживаться и PyQt 5.

Воспроизведение звука обеспечивает класс `QMediaPlayer`, реализующий полноценный мультимедийный проигрыватель (но, к сожалению, без какого бы то ни было пользовательского интерфейса). Чтобы воспроизвести видео, в дополнение к классу `QMediaPlayer` следует использовать класс `QVideoWidget`, который реализует панель для вывода собственно видео. Класс `QMediaPlaylist` обеспечивает функциональность списков воспроизведения (*плейлистов*). Класс `QAudioRecorder` позволяет записывать аудио с микрофона и сохранять в файле. А класс `QSoundEffect` — наилучший выбор в случае, если необходимо воспроизвести *эффект* — короткий звуковой фрагмент, например, в качестве оповещения о каком-либо событии.

Все описанные в этой главе классы объявлены в модуле `QtMultimedia`, если не указано иное.

28.1. Класс *QMediaPlayer*

Класс `QMediaPlayer` реализует полнофункциональный мультимедийный проигрыватель, позволяющий воспроизводить как звук, так и видео из файлов и сетевых источников. Единственный его недостаток заключается в том, что он не предоставляет никакого интерфейса для управления воспроизведением: ни кнопок запуска, остановки и паузы, ни регулятора громкости, ни индикатора текущей позиции воспроизведения. Все это нам придется делать самим.

Иерархия наследования для класса `QMediaPlayer` выглядит так:

```
QObject - QMediaObject - QMediaPlayer
```

А формат вызова его конструктора таков:

```
<Объект> = QMediaPlayer([parent=None][, flags=0])
```

В параметре `parent` может быть указана ссылка на родительский компонент. Параметр `flags` позволяет указать дополнительные настройки проигрывателя и должен быть задан в виде одного из значений следующих атрибутов класса `QMediaPlayer` или их комбинации через оператор `|`:

- `LowLatency` — `1` — воспроизведение должно начаться с минимальной задержкой. Используется при воспроизведении мультимедиа, закодированного в относительно простых форматах (например, звук PCM, сохраненный в файле WAV);
- `StreamPlayback` — `2` — настраивает проигрыватель для воспроизведения потокового мультимедиа;
- `VideoSurface` — `4` — настраивает проигрыватель для использования вместе с обработчиками мультимедиа (создание таких обработчиков описано в документации).

Класс `QMediaPlayer` поддерживает следующие наиболее полезные для нас методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qmediaplayer.html**):

- `setMedia(<QMediaContent>)` — задает источник (мультимедийный файл или сетевой ресурс) для воспроизведения. Источник указывается в виде экземпляра класса `QMediaContent`. Если передать значение `None`, заданный ранее источник будет выгружен. Примеры задания источника-файла:

  ```
  mp = QtMultimedia.QMediaPlayer()
  mc = QtMultimedia.QMediaContent(QtCore.QUrl.fromLocalFile(r"c:\media\song1.mp3"))
  mp.setMedia(mc)
  ```

  Пример задания сетевого источника:

  ```
  mp.setMedia(QtCore.QUrl(r"http://www.someradio.ru/playlist.m3u8"))
  ```

  Метод является слотом;
- `media()` — возвращает заданный источник (экземпляр класса `QMediaContent`);
- `setPlaylist(<QMediaPlaylist>)` — задает в качестве источника плейлист (экземпляр класса `QMediaPlaylist`). Если передать значение `None`, заданный ранее плейлист будет выгружен. Метод является слотом;
- `playlist()` — возвращает заданный плейлист (экземпляр класса `QMediaPlaylist`);
- `currentMedia()` — возвращает текущий источник мультимедиа. Если был задан один источник вызовом метода `setMedia()`, возвращаемое значение то же, что и у метода `media()`. Если был задан плейлист, возвращает позицию плейлиста, воспроизводящуюся в текущий момент. В любом случае возвращаемое значение — экземпляр класса `QMediaContent`;
- `play()` — начинает или возобновляет воспроизведение мультимедиа. Метод является слотом;
- `stop()` — останавливает воспроизведение мультимедиа и перемещает позицию воспроизведения на начало. Метод является слотом;
- `pause()` — приостанавливает воспроизведение мультимедиа. Метод является слотом;
- `state()` — возвращает текущее состояние проигрывателя в виде значения одного из следующих атрибутов класса `QMediaPlayer`:
  - `StoppedState` — `0` — воспроизведение либо остановлено, либо еще не начиналось;
  - `PlayingState` — `1` — идет воспроизведение;
  - `PausedState` — `2` — воспроизведение поставлено на паузу;
- `duration()` — возвращает продолжительность воспроизведения мультимедиа в миллисекундах;

- position() — возвращает текущую позицию воспроизведения мультимедиа в миллисекундах;
- isSeekable() — возвращает True, если имеется возможность изменить позицию воспроизведения, и False, если источник этого не поддерживает;
- setPosition(<Позиция воспроизведения>) — задает текущую позицию воспроизведения в миллисекундах. Метод является слотом;
- setVolume(<Громкость>) — задает громкость звука в виде числа от 0 (звук отключен) до 100 (максимальная громкость). Метод является слотом;
- volume() — возвращает громкость звука в виде числа от 0 до 100;
- setMuted(<Флаг>) — если передать значение True, звук будет отключен. Значение False восстанавливает прежнюю громкость. Метод является слотом;
- isMuted() — возвращает True, если звук в настоящее время отключен, и False — в противном случае;
- setPlaybackRate(<Скорость воспроизведения>) — задает скорость воспроизведения мультимедиа. Значение параметра должно представлять собой вещественное число, умножаемое на стандартную скорость воспроизведения, чтобы получить ее новую величину. Значение 1.0 задает стандартную скорость, меньшее 1.0 — уменьшенную скорость, а большее 1.0 — увеличенную. Метод является слотом;
- playbackRate() — возвращает текущую скорость воспроизведения мультимедиа;
- setVideoOutput(<QVideoWidget>) — задает панель для вывода видео, представляемую экземпляром класса QVideoWidget. Используется лишь при воспроизведении видео;
- mediaStatus() — возвращает состояние источника в виде значения одного из следующих атрибутов класса QMediaPlayer:
  - UnknownMediaStatus — 0 — состояние определить не удается;
  - NoMedia — 1 — указанный источник отсутствует;
  - LoadingMedia — 2 — идет загрузка данных источника;
  - LoadedMedia — 3 — загрузка данных источника закончена;
  - StalledMedia — 4 — воспроизведение приостановлено из-за опустошения внутреннего буфера проигрывателя или невозможности загрузить следующую порцию данных;
  - BufferingMedia — 5 — идет заполнение данными внутреннего буфера проигрывателя, однако данных достаточно для продолжения воспроизведения;
  - BufferedMedia — 6 — внутренний буфер заполнен, идет воспроизведение;
  - EndOfMedia — 7 — воспроизведение источника закончено;
  - InvalidMedia — 8 — источник не может быть воспроизведен (неподдерживаемый формат мультимедийных данных, поврежденный файл и т. п.);
- isAudioAvailable() — возвращает True, если источник содержит звук, и False — в противном случае. Обычно используется при воспроизведении видеороликов, чтобы узнать, имеют ли они звуковое сопровождение;
- isVideoAvailable() — возвращает True, если источник содержит видео, и False — в противном случае;
- bufferStatus() — возвращает процент заполнения внутреннего буфера проигрывателя в виде числа от 0 до 100;

- `error()` — возвращает ошибку, возникшую при воспроизведении мультимедиа, в виде значения одного из следующих атрибутов класса `QMediaPlayer`:
  - `NoError` — `0` — никакой ошибки не возникло;
  - `ResourceError` — `1` — не удалось загрузить источник;
  - `FormatError` — `2` — неподдерживаемый формат звука или видео;
  - `NetworkError` — `3` — сетевая ошибка;
  - `AccessDeniedError` — `4` — недостаточно прав для загрузки источника;
  - `ServiceMissingError` — `5` — ошибка в мультимедийной подсистеме;
- `errorString()` — возвращает текстовое описание ошибки;
- `isMetaDataAvailable()` — возвращает `True`, если доступны метаданные источника, описывающие хранящиеся в нем мультимедийные данные, и `False` — в противном случае;
- `metaData(<Ключ>)` — возвращает метаданные, относящиеся к заданному в виде строки ключу. Список доступных ключей, которые могут нам пригодиться, приведен далее (полный их список можно найти на странице **https://doc.qt.io/qt-5/qmediametadata.html**). Если не указано обратное, все значения представляют собой строки, а слово «список» означает обычный список Python (класс `list`).
  - `Title` — название произведения;
  - `Author` — список авторов произведения;
  - `Comment` — примечание;
  - `Description` — описание;
  - `Genre` — список жанров, к которым относится произведение;
  - `Year` — год выпуска произведения в виде целого числа;
  - `Date` — дата выпуска произведения в виде экземпляра класса `datetime.date`;
  - `Language` — язык произведения;
  - `Copyright` — авторские права создателей произведения;
  - `Size` — размер мультимедийного файла в байтах, представленный целым числом;
  - `Duration` — продолжительность воспроизведения мультимедийного файла в миллисекундах, заданная целым числом;
  - `AudioBitRate` — битрейт звука в виде целого числа, измеряемый в битах в секунду;
  - `AudioCodec` — кодек, с помощью которого кодировался звук;
  - `ChannelCount` — количество каналов звука в виде целого числа;
  - `AlbumTitle` — название альбома, в который входит произведение;
  - `AlbumArtist` — автор альбома;
  - `ContributingArtist` — список исполнителей, занятых в записи альбома;
  - `Composer` — список композиторов, произведения которых входят в альбом;
  - `Conductor` — дирижер, занятый в записи альбома;
  - `Lyrics` — тексты песен, входящих в альбом;
  - `TrackNumber` — порядковый номер произведения в альбоме в виде целого числа;

- `TrackCount` — количество произведений в альбоме в виде целого числа;
- `Resolution` — размер видео в виде экземпляра класса `QSize`;
- `VideoFrameRate` — частота кадров видео в виде вещественного числа, измеряемая в кадрах в секунду;
- `VideoBitRate` — битрейт видео в виде целого числа, измеряемый в битах в секунду;
- `VideoCodec` — кодек, с помощью которого кодировалось видео;
- `Director` — режиссер фильма;
- `LeadPerformer` — список актеров, занятых в фильме;
- `Writer` — список авторов сценария фильма.

Следует отметить, что в источнике совсем не обязательно будут присутствовать все эти данные;

- `availableMetaData()` — возвращает список ключей, доступных в источнике метаданных.

Класс `QMediaPlayer` поддерживает большое количество сигналов, из которых для нас представляют интерес лишь приведенные далее (полный их список можно найти на странице **https://doc.qt.io/qt-5/qmediaplayer.html**).

- `mediaChanged(<QMediaContent>)` — генерируется при указании другого источника. Новый источник передается в параметре;
- `currentMediaChanged(<QMediaContent>)` — генерируется при указании другого источника или переходе на новую позицию плейлиста. Новый источник передается в параметре;
- `durationChanged(<Продолжительность>)` — генерируется при первом получении или изменении продолжительности ролика. Новое значение продолжительности, заданное целым числом в миллисекундах, передается обработчику в параметре;
- `stateChanged(<Состояние>)` — генерируется при изменении состояния проигрывателя. В параметре обработчику передается новое состояние в виде целочисленного обозначения;
- `positionChanged(<Позиция>)` — генерируется при изменении позиции воспроизведения. В параметре обработчику передается новая позиция в виде целого числа в миллисекундах.

  По умолчанию такой сигнал генерируется каждые 1000 миллисекунд (1 секунду). Мы можем изменить этот интервал времени, указав нужное значение в вызове метода `setNotifyInterval(<Интервал>)` класса `QMediaPlayer`;
- `volumeChanged(<Громкость>)` — генерируется при изменении громкости. В параметре передается новое значение громкости в виде целого числа;
- `mutedChanged(<Флаг>)` — генерируется при отключении или, напротив, включении звука. В параметре передается значение `True`, если звук отключен, или `False`, если вновь включен;
- `playbackRateChanged(<Скорость воспроизведения>)` — генерируется при изменении скорости воспроизведения. В параметре передается новое значение скорости воспроизведения, представленное вещественным числом;
- `mediaStatusChanged(<Состояние источника>)` — генерируется при изменении состояния источника. Новое целочисленное обозначение состояния передается в параметре;

- `bufferStatusChanged(<Состояние буфера>)` — генерируется при изменении процента заполнения внутреннего буфера проигрывателя. Новая величина процента заполнения в виде целого числа передается в параметре;
- `error(<Ошибка>)` — генерируется при возникновении ошибки. Целочисленное обозначение ошибки передается в параметре;
- `metaDataAvailableChanged(<Флаг>)` — генерируется при получении метаданных, записанных в ролике. В параметре передается `True`, если метаданные доступны, или `False` — в противном случае;
- `metaDataChanged()` — генерируется при изменении метаданных ролика;
- `metaDataChanged(<Ключ>, <Значение>)` — генерируется при изменении какого-либо значения метаданных ролика. В параметрах передаются строковый ключ изменившегося значения и само это значение.

Простейший аудиопроигрыватель (листинг 28.1) предоставляет кнопку для открытия звукового файла, кнопки пуска, паузы и остановки воспроизведения, регулятор текущей позиции воспроизведения, регулятор громкости и кнопку временного отключения звука (рис. 28.1).

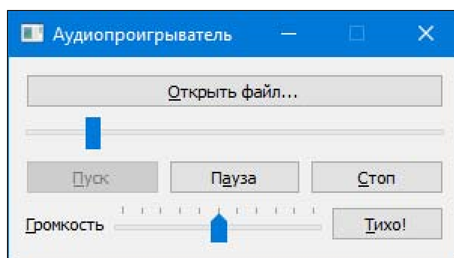

Рис. 28.1. Интерфейс простейшего аудиопроигрывателя

Листинг 28.1. Простейший аудиопроигрыватель

```
from PyQt5 import QtCore, QtWidgets, QtMultimedia
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
        QtWidgets.QWidget.__init__(self, parent, flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Аудиопроигрыватель")
        # Создаем сам проигрыватель
        self.mplPlayer = QtMultimedia.QMediaPlayer()
        self.mplPlayer.setVolume(50)
        self.mplPlayer.mediaStatusChanged.connect(self.initPlayer)
        self.mplPlayer.stateChanged.connect(self.setPlayerState)
        vbox = QtWidgets.QVBoxLayout()
        # Создаем кнопку открытия файла
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
```

```
        # Создаем компоненты для управления воспроизведением.
        # Делаем их изначально недоступными
        self.sldPosition = QtWidgets.QSlider(QtCore.Qt.Horizontal)
        self.sldPosition.setMinimum(0)
        self.sldPosition.valueChanged.connect(self.mplPlayer.setPosition)
        self.mplPlayer.positionChanged.connect(self.sldPosition.setValue)
        self.sldPosition.setEnabled(False)
        vbox.addWidget(self.sldPosition)
        hbox = QtWidgets.QHBoxLayout()
        self.btnPlay = QtWidgets.QPushButton("&Пуск")
        self.btnPlay.clicked.connect(self.mplPlayer.play)
        self.btnPlay.setEnabled(False)
        hbox.addWidget(self.btnPlay)
        self.btnPause = QtWidgets.QPushButton("П&ауза")
        self.btnPause.clicked.connect(self.mplPlayer.pause)
        self.btnPause.setEnabled(False)
        hbox.addWidget(self.btnPause)
        self.btnStop = QtWidgets.QPushButton("&Стоп")
        self.btnStop.clicked.connect(self.mplPlayer.stop)
        self.btnStop.setEnabled(False)
        hbox.addWidget(self.btnStop)
        vbox.addLayout(hbox)
        # Создаем компоненты для управления громкостью
        hbox = QtWidgets.QHBoxLayout()
        lblVolume = QtWidgets.QLabel("&Громкость")
        hbox.addWidget(lblVolume)
        sldVolume = QtWidgets.QSlider(QtCore.Qt.Horizontal)
        sldVolume.setRange(0, 100)
        sldVolume.setTickPosition(QtWidgets.QSlider.TicksAbove)
        sldVolume.setTickInterval(10)
        sldVolume.setValue(50)
        lblVolume.setBuddy(sldVolume)
        sldVolume.valueChanged.connect(self.mplPlayer.setVolume)
        hbox.addWidget(sldVolume)
        btnMute = QtWidgets.QPushButton("&Тихо!")
        btnMute.setCheckable(True)
        btnMute.toggled.connect(self.mplPlayer.setMuted)
        hbox.addWidget(btnMute)
        vbox.addLayout(hbox)
        self.setLayout(vbox)
        self.resize(300, 100)

    # Для открытия файла используем метод getOpenFileUrl() класса
    # QFileDialog, т. к. для создания экземпляра класса
    # QMediaContent нам нужен путь к файлу, заданный в виде
    # экземпляра класса QUrl
    def openFile(self):
        file = QtWidgets.QFileDialog.getOpenFileUrl(parent=self,
            caption = "Выберите звуковой файл",
            filter = "Звуковые файлы (*.mp3 *.ac3)")
        self.mplPlayer.setMedia(QtMultimedia.QMediaContent(file[0]))
```

```
    def initPlayer(self, state):
        if state == QtMultimedia.QMediaPlayer.LoadedMedia:
            # После загрузки файла подготавливаем проигрыватель
            # для его воспроизведения
            self.mplPlayer.stop()
            self.btnPlay.setEnabled(True)
            self.btnPause.setEnabled(False)
            self.sldPosition.setEnabled(True)
            self.sldPosition.setMaximum(self.mplPlayer.duration())
        elif state == QtMultimedia.QMediaPlayer.EndOfMedia:
            # По окончании воспроизведения файла возвращаем
            # проигрыватель в изначальное состояние
            self.mplPlayer.stop()
            self.sldPosition.setValue(0)
            self.sldPosition.setEnabled(False)
            self.btnPlay.setEnabled(False)
            self.btnPause.setEnabled(False)
            self.btnStop.setEnabled(False)
        elif state == QtMultimedia.QMediaPlayer.NoMedia or ↳
        state == QtMultimedia.QMediaPlayer.InvalidMedia:
            # Если файл не был загружен, отключаем компоненты,
            # управляющие воспроизведением
            self.sldPosition.setValue(0)
            self.sldPosition.setEnabled(False)
            self.btnPlay.setEnabled(False)
            self.btnPause.setEnabled(False)
            self.btnStop.setEnabled(False)

    # В зависимости от того, воспроизводится ли файл, поставлен
    # ли он на паузу или остановлен, делаем соответствующие кнопки
    # доступными или недоступными
    def setPlayerState(self, state):
        if state == QtMultimedia.QMediaPlayer.StoppedState:
            self.sldPosition.setValue(0)
            self.btnPlay.setEnabled(True)
            self.btnPause.setEnabled(False)
            self.btnStop.setEnabled(False)
        elif state == QtMultimedia.QMediaPlayer.PlayingState:
            self.btnPlay.setEnabled(False)
            self.btnPause.setEnabled(True)
            self.btnStop.setEnabled(True)
        elif state == QtMultimedia.QMediaPlayer.PausedState:
            self.btnPlay.setEnabled(True)
            self.btnPause.setEnabled(False)
            self.btnStop.setEnabled(True)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

В листинге 28.2 приведен код служебной утилиты, выводящей метаданные, которые хранятся в загруженном мультимедийном файле. В настоящий момент она выводит сведения об открытом в ней аудиофайле (рис. 28.2).

Метаданные

Открыть файл...

AlbumTitle: Steppenwolf
AudioBitRate: 160000
AudioCodec: MPEG Audio Layer-3 (MP3)
Author: ['Steppenwolf']
ChannelCount: 2
ContributingArtist: ['Steppenwolf']
Duration: 212872
Genre: ['Hard rock']
SampleRate: 44100
Title: Born to be wild
TrackNumber: 5
Year: 1968

Рис. 28.2. Интерфейс утилиты, выводящей метаданные мультимедийного файла

Листинг 28.2. Вывод метаданных мультимедийного файла

```
from PyQt5 import QtCore, QtWidgets, QtMultimedia
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
        QtWidgets.QWidget.__init__(self, parent, flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Метаданные")
        self.mplPlayer = QtMultimedia.QMediaPlayer()
        self.mplPlayer.setVolume(50)
        self.mplPlayer.mediaStatusChanged.connect(self.showMetadata)
        vbox = QtWidgets.QVBoxLayout()
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
        # Создаем доступное только для чтения многострочное текстовое
        # поле, в которое будет выводиться результат
        self.txtOutput = QtWidgets.QTextEdit()
        self.txtOutput.setReadOnly(True)
        vbox.addWidget(self.txtOutput)
        self.setLayout(vbox)
        self.resize(300, 250)

    def openFile(self):
        file = QtWidgets.QFileDialog.getOpenFileUrl(parent=self,
            caption = "Выберите звуковой файл",
```

```
            filter = "Звуковые файлы (*.mp3 *.ac3)")
        self.mplPlayer.setMedia(QtMultimedia.QMediaContent(file[0]))

    def showMetadata(self, state):
        self.txtOutput.clear()
        # Как только файл будет загружен...
        if state == QtMultimedia.QMediaPlayer.LoadedMedia:
            # ...извлекаем ключи всех доступных метаданных...
            keys = self.mplPlayer.availableMetaData()
            s = ""
            # ...перебираем их в цикле...
            for k in keys:
                v = self.mplPlayer.metaData(k)
                # ...на случай пустых списков проверяем, действительно
                # ли по этому ключу хранится какое-то значение...
                if v:
                    # ...если значение представляет собой список,
                    # преобразуем его в строку...
                    if v is list:
                        v = ", ".join(v)
                    # ...формируем на основе значений текст...
                    s += "<strong>" + k + "</strong>: " + str(v) + "<br>"
            # ...и выводим его в текстовое поле
            self.txtOutput.setHtml(s)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

28.2. Класс *QVideoWidget*

Класс `QVideoWidget`, определенный в модуле `QtMultimediaWidgets`, представляет панель вывода видео, которое воспроизводится мультимедийным проигрывателем. Задать панель вывода видео для мультимедийного проигрывателя можно вызовом метода `setVideoOutput()` класса `QMediaPlayer`.

Иерархия наследования класса `QVideoWidget` выглядит следующим образом:

```
(QObject, QPaintDevice) — (QWidget, QMediaBindableInterface) —
                          QVideoWidget
```

А формат вызова его конструктора таков:

```
<Объект> = QVideoWidget([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Методов класс `QVideoWidget` поддерживает относительно немного, и все они управляют выводом видео:

- `setFullScreen(<Флаг>)` — если передать значение `True`, воспроизводящееся видео займет весь экран. Если передать значение `False`, оно займет лишь саму панель. Метод является слотом;

- `isFullScreen()` — возвращает `True`, если воспроизводящееся видео занимает весь экран, и `False`, если оно выводится лишь в самой панели;
- `setAspectRatioMode(<Соотношение сторон>)` — задает режим управления соотношением сторон при масштабировании видео. В качестве результата указывается один из следующих атрибутов класса `QtCore.Qt`:
  - `IgnoreAspectRatio` — `0` — соотношение сторон при масштабировании видео не соблюдается, в результате чего видео может выводиться сжатым или вытянутым;
  - `KeepAspectRatio` — `1` — соотношение сторон соблюдается. Видео масштабируется так, чтобы полностью вместиться в панель или экран, но при этом какая-то часть панели или экрана может оказаться незанятой;
  - `KeepAspectRatioByExpanding` — `2` — соотношение сторон соблюдается. Видео масштабируется так, чтобы полностью занять панель (экран), но при этом какая-то часть видео может выйти за ее границы.

  Метод является слотом;
- `aspectRatioMode()` — возвращает обозначение режима управления соотношением сторон при масштабировании видео;
- `setBrightness(<Яркость>)` — задает яркость видео в виде числа от `-100` до `100`: отрицательные значения делают видео более темным, положительные — более светлым, а `0` устанавливает яркость по умолчанию. Метод является слотом;
- `brightness()` — возвращает яркость видео;
- `setContrast(<Контрастность>)` — задает контрастность видео в виде числа от `-100` до `100`: отрицательные значения делают видео менее контрастным, положительные — более контрастным, а `0` устанавливает значение этого параметра по умолчанию. Метод является слотом;
- `contrast()` — возвращает контрастность видео;
- `setHue(<Оттенок>)` — задает оттенок видео в виде числа от `-100` до `100`: значения, отличные от `0`, изменяют цветность видео, а `0` устанавливает значение этого параметра по умолчанию. Метод является слотом;
- `hue()` — возвращает значение оттенка видео;
- `setSaturation(<Насыщенность>)` — задает насыщенность видео в виде числа от `-100` до `100`: отрицательные значения делают видео менее насыщенным, положительные — более насыщенным, а `0` устанавливает значение этого параметра по умолчанию. Метод является слотом;
- `saturation()` — возвращает насыщенность видео;
- `mediaObject()` — возвращает ссылку (экземпляр класса `QMediaObject`) на мультимедийный проигрыватель, выводящий видео в эту панель, или `None`, если панель не привязана к проигрывателю.

Класс `QVideoWidget` поддерживает некоторое количество сигналов, генерируемых при изменении различных параметров видео (режима управления соотношением сторон, яркости, контрастности и т. д.). Описание этих сигналов приведено на странице **https://doc.qt.io/qt-5/qvideowidget.html**.

Мы можем превратить написанную ранее программу аудиопроигрывателя в инструмент для воспроизведения видео (рис. 28.3). Для этого достаточно внести в ее код некоторые измене-

ния (листинг 28.3) — добавленные и исправленные строки выделены здесь полужирным шрифтом.

Листинг 28.3. Простейший видеопроигрыватель

```
from PyQt5 import QtCore, QtWidgets, QtMultimedia, QtMultimediaWidgets
. . .
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
        . . .
        self.setWindowTitle("Видеопроигрыватель")
        . . .
        vbox.addWidget(btnOpen)
        vwg = QtMultimediaWidgets.QVideoWidget()
        vwg.setAspectRatioMode(QtCore.Qt.KeepAspectRatio)
        self.mplPlayer.setVideoOutput(vwg)
        vbox.addWidget(vwg)
        self.sldPosition = QtWidgets.QSlider(QtCore.Qt.Horizontal)
        . . .
        self.setLayout(vbox)
        self.resize(300, 300)
    . . .
    def openFile(self):
        file = QtWidgets.QFileDialog.getOpenFileUrl(parent=self,
            caption = "Выберите видеофайл",
            filter = "Видеофайлы (*.avi *.mp4)")
        self.mplPlayer.setMedia(QtMultimedia.QMediaContent(file[0]))
```

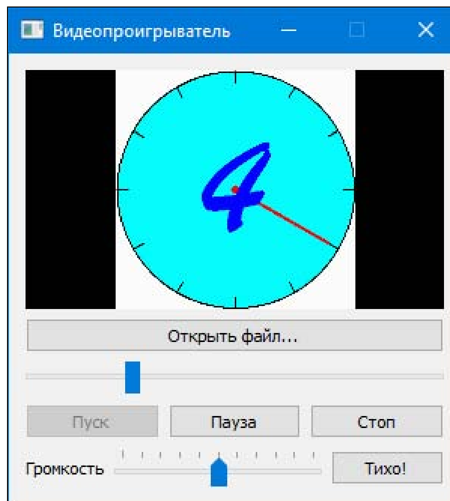

Рис. 28.3. Простейший видеопроигрыватель

28.3. Класс *QMediaPlaylist*

Класс `QMediaPlaylist` реализует поддержку списков воспроизведения (плейлистов). Установить плейлист в качестве источника мультимедийных данных позволяет метод `setPlaylist()` класса `QMediaPlayer`.

Иерархия наследования класса `QMediaPlaylist` выглядит так:

```
(QObject, QMediaBindableInterface) — QMediaPlaylist
```

А формат вызова его конструктора таков:

```
<Объект> = QMediaPlaylist([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Класс `QMediaPlaylist` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qmediaplaylist.html**):

- `addMedia()` — добавляет указанный источник или источники в конец плейлиста и возвращает `True`, если добавление прошло успешно, и `False` — в противном случае. Форматы метода:

  ```
  addMedia(<QMediaContent>)
  addMedia(<Список экземпляров класса QMediaContent>)
  ```

- `insertMedia()` — вставляет указанный источник или источники в позицию плейлиста, обозначенную заданным индексом, и возвращает `True`, если вставка прошла успешно, и `False` — в противном случае. Форматы метода:

  ```
  insertMedia(<Индекс>, <QMediaContent>)
  insertMedia(<Индекс>, <Список экземпляров класса QMediaContent>)
  ```

- `moveMedia(<Индекс 1>, <Индекс 2>)` — перемещает позицию, обозначенную `<Индексом 1>`, в местоположение, обозначенное `<Индексом 2>`. Возвращает `True`, если перемещение удалось, и `False` — в противном случае. Поддерживается, начиная с PyQt 5.7;
- `removeMedia(<Индекс>)` — удаляет обозначенную индексом позицию из плейлиста и возвращает `True`, если удаление прошло успешно, и `False` — в противном случае;
- `removeMedia(<Начальный индекс>, <Конечный индекс>)` — удаляет из плейлиста позиции, находящиеся между указанными индексами (позиция, обозначенная конечным индексом, не удаляется). Возвращает `True`, если удаление прошло успешно, и `False` — в противном случае;
- `clear()` — удаляет все позиции из плейлиста. Возвращает `True`, если удаление прошло успешно, и `False` — в противном случае;
- `isReadOnly()` — возвращает `True`, если содержимое плейлиста не может быть изменено, и `False` — в противном случае;
- `setPlaybackMode(<Порядок воспроизведения>)` — задает порядок воспроизведения позиций. В качестве параметра указывается значение одного из следующих атрибутов класса `QMediaPlaylist`:
  - `CurrentItemOnce` — `0` — текущая позиция воспроизводится однократно;
  - `CurrentItemInLoop` — `1` — текущая позиция воспроизводится снова и снова;
  - `Sequential` — `2` — содержимое плейлиста воспроизводится до самого конца однократно, начиная с текущей позиции;

   - `Loop` — `3` — содержимое плейлиста воспроизводится до самого конца снова и снова;
   - `Random` — `4` — позиции плейлиста воспроизводятся в случайном порядке;
- `mediaCount()` — возвращает количество позиций в плейлисте;
- `isEmpty()` — возвращает `True`, если плейлист пуст, и `False` — в противном случае;
- `media(<Индекс>)` — возвращает позицию (экземпляр класса `QMediaContent`), соответствующую указанному индексу, или `None`, если такой позиции нет;
- `setCurrentIndex(<Индекс>)` — начинает воспроизведение позиции с указанным индексом. Метод является слотом;
- `currentIndex()` — возвращает индекс текущей позиции (воспроизводящейся в настоящий момент);
- `currentMedia()` — возвращает текущую позицию в виде экземпляра класса `QMediaContent`;
- `previousIndex([steps=1])` — возвращает индекс позиции, отстоящей на текущей на заданное параметром `steps` число позиций, считая в направлении к началу плейлиста;
- `nextIndex([steps=1])` — возвращает индекс позиции, отстоящей на текущей на заданное параметром `steps` число позиций, считая в направлении к концу плейлиста;
- `previous()` — начинает воспроизведение предыдущей позиции в плейлисте. Метод является слотом;
- `next()` — начинает воспроизведение следующей позиции в плейлисте. Метод является слотом;
- `shuffle()` — упорядочивает позиции плейлиста случайным образом. Метод является слотом;
- `load(<QUrl>[, <Формат>])` — загружает плейлист из файла, заданного первым параметром. Содержимое загруженного плейлиста добавляется в конец текущего. Во втором параметре можно указать формат загружаемого плейлиста — если он не указан, формат будет определен по расширению файла.

  Из информации, собранной в Интернете, авторам удалось выяснить список поддерживаемых форматов плейлистов: M3U (формат плейлиста Winamp) и M3U8 (формат плейлиста Winamp с поддержкой UTF-8). Обозначение формата указывается во втором параметре в виде строки, набранной только строчными буквами: `m3u` или `m3u8`:

```
mpl = QtMultimedia.QMediaPlaylist()
file = QtCore.QUrl.fromLocalFile(r"c:\media\playlist.m3u8")
mpl.load(file)
player.setPlaylist(mpl)
```

- `save(<QUrl>[, <Формат>])` — сохраняет плейлист в файле, заданном первым параметром. Возвращает `True`, если сохранение прошло успешно, и `False` — в противном случае. Во втором параметре указывается формат сохраняемого плейлиста в виде строки `m3u` или `m3u8`:

```
file = QtCore.QUrl.fromLocalFile(r"c:\media\playlist.m3u8")
mpl.save(file, "m3u8")
```

> ***Внимание!***
>
> Метод `save()` даже при указании формата `m3u8` сохраняет плейлист в кодировке Windows-1251, а не в UTF-8, причем пути к файлам записываются в виде URL. Вероятно, это ошибка в библиотеке PyQt. Поэтому для сохранения плейлистов рекомендуется использовать инструменты Python, обеспечивающие работу с файлами (см. *главу 16*).

- `error()` — возвращает ошибку, возникшую при обработке плейлиста, в виде значения одного из следующих атрибутов класса `QMediaPlaylist`:
  - `NoError` — `0` — никакой ошибки не возникло;
  - `FormatError` — `1` — файл плейлиста поврежден, или это вообще не плейлист;
  - `FormatNotSupportedError` — `2` — неподдерживаемый формат плейлиста;
  - `NetworkError` — `3` — сетевая ошибка;
  - `AccessDeniedError` — `4` — недостаточно прав для загрузки плейлиста;
- `errorString()` — возвращает текстовое описание ошибки;
- `mediaObject()` — возвращает ссылку (экземпляр класса `QMediaObject`) на мультимедийный проигрыватель, воспроизводящий этот плейлист, или `None`, если плейлист не воспроизводится ни в каком проигрывателе.

Класс `QMediaPlaylist` поддерживает большое количество сигналов, из которых для нас представляют интерес лишь приведенные далее (полный их список можно найти на странице **https://doc.qt.io/qt-5/qmediaplaylist.html**).

- `currentIndexChanged(<Индекс>)` — генерируется при смене текущей позиции плейлиста. В параметре передается целочисленный индекс новой позиции;
- `currentMediaChanged(<QMediaContent>)` — генерируется при смене текущей позиции плейлиста. В параметре передается сама новая позиция;
- `loaded()` — генерируется после успешной загрузки плейлиста;
- `loadFailed()` — генерируется, если в процессе загрузки плейлиста возникла ошибка.

Для примера напишем совсем простое приложение аудиопроигрывателя, воспроизводящего плейлист (листинг 28.4). Оно будет создавать плейлист из четырех аудиофайлов, находящихся в той же папке, что и само приложение, воспроизводить его и позволит пользователю переключаться на предыдущую и следующую позицию плейлиста.

Листинг 28.4. Использование плейлистов

```
from PyQt5 import QtCore, QtWidgets, QtMultimedia
import sys, os

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
        QtWidgets.QWidget.__init__(self, parent, flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Плейлист")
        self.mplPlayer = QtMultimedia.QMediaPlayer()
        # Создаем плейлист и наполняем его файлами
        self.playlist = QtMultimedia.QMediaPlaylist()
        files = ["audio1.mp3", "audio2.mp3", "audio3.mp3", "audio4.mp3"]
        for f in files:
            fn = QtCore.QUrl.fromLocalFile(os.path.abspath(f))
            self.playlist.addMedia(QtMultimedia.QMediaContent(fn))
        # Поэкспериментируем со вставкой и удалением позиций в плейлисте
        # Вставляем вторую позицию в конец плейлиста
        self.playlist.insertMedia(4, self.playlist.media(1))
```

```
        # Удаляем вторую позицию
        self.playlist.removeMedia(1)
        self.playlist.setCurrentIndex(0)
        self.mplPlayer.setPlaylist(self.playlist)
        # Создаем элементы управления воспроизведением
        vbox = QtWidgets.QVBoxLayout()
        hbox = QtWidgets.QHBoxLayout()
        self.btnPlay = QtWidgets.QPushButton("&Пуск")
        self.btnPlay.clicked.connect(self.mplPlayer.play)
        hbox.addWidget(self.btnPlay)
        self.btnNext = QtWidgets.QPushButton("П&редыдущая")
        self.btnNext.clicked.connect(self.playlist.previous)
        hbox.addWidget(self.btnNext)
        self.btnPrevious = QtWidgets.QPushButton("&Следующая")
        self.btnPrevious.clicked.connect(self.playlist.next)
        hbox.addWidget(self.btnPrevious)
        vbox.addLayout(hbox)
        self.lblCurrent = QtWidgets.QLabel("")
        vbox.addWidget(self.lblCurrent)
        self.playlist.currentMediaChanged.connect(self.showFile)
        self.setLayout(vbox)
        self.resize(300, 70)

    # Выводим на экран имя файла, воспроизводящегося в данный момент
    # Метод canonicalUrl() класса QMediaContent возвращает
    # путь к файлу экземпляра класса QUrl, а метод fileName()
    # класса QUrl позволяет узнать имя файла
    def showFile(self, content):
        self.lblCurrent.setText(content.canonicalUrl().fileName())

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

28.4. Запись звука

Библиотека PyQt 5 предоставляет довольно богатые возможности записи звука и сохранения его в файле. Сейчас мы выясним, как это делается.

> ***Примечание***
>
> Помимо записи звука, PyQt 5 также позволяет с применением подключенной к компьютеру камеры производить фото- и видеосъемку. Как это выполняется, описано в документации по библиотеке.

28.4.1. Класс *QAudioRecorder*

Класс `QAudioRecorder` позволяет записывать звук с микрофона и сохранять его в аудиофайле в закодированном виде. Его иерархия наследования выглядит так:

```
(QObject, QMediaBindableInterface) — QMediaRecorder - QAudioRecorder
```

Формат вызова конструктора этого класса следующий:

```
<Объект> = QAudioRecorder([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Бóльшую часть полезных для нас методов `QAudioRecorder` наследует от базового класса — `QMediaRecorder`. Эти методы приведены далее (полный их список можно найти на страницах **https://doc.qt.io/qt-5/qmediarecorder.html** и **https://doc.qt.io/qt-5/qaudiorecorder.html**).

- `setOutputLocation(<QUrl>)` — задает путь к файлу, в котором будет сохраняться записываемый звук;
- `setAudioInput(<Звуковой вход>)` — задает вход, с которого будет сниматься записываемый звук. Наименование входа задается в виде строки;
- `setContainerFormat(<Формат файла>)` — задает MIME-тип файла, в котором будет сохраняться записываемый звук. Формат указывается в виде строки;
- `setAudioSettings(<QAudioEncoderSettings>)` — задает параметры кодирования записываемого звука. Параметры указываются в виде экземпляра класса `QAudioEncoderSettings`;
- `setVolume(<Уровень>)` — задает уровень записываемого звука в виде числа от `0` (звук отключен) до `100` (максимальный уровень). Метод является слотом;
- `record()` — запускает или возобновляет запись звука. Метод является слотом;
- `pause()` — приостанавливает запись звука. Метод является слотом;
- `stop()` — останавливает запись звука. Метод является слотом;
- `setMetaData(<Ключ>, <Значение>)` — заносит в метаданные сохраняемого файла заданное значение под заданным ключом. Доступные ключи были рассмотрены в описании метода `metaData()` (см. *разд. 28.1*);
- `defaultAudioInput()` — возвращает строку с наименованием звукового входа, используемого по умолчанию:

  ```
  ar = QtMultimedia.QAudioRecorder()
  print(ar.defaultAudioInput())
  ```

  У авторов вывел:

  ```
  Default Input Device
  ```
- `audioInputs()` — возвращает список наименований всех имеющихся в системе звуковых входов, заданных строками:

  ```
  for f in ar.audioInputs(): print(f)
  ```

  У авторов вывел:

  ```
  Микрофон (VIA HD Audio)
  Analog Mix (Line/CD/Aux/TAD/PC) (Creative SB Audigy)
  Microphone (Creative SB Audigy)
  ```

  Остальное пропущено;
- `supportedContainers()` — возвращает список всех поддерживаемых классом `QAudioRecorder` MIME-типов файлов, указанных в виде строк:

  ```
  for f in ar.supportedContainers(): print(f, end = " ")
  ```

  Выведет: `audio/x-wav audio/x-raw`

  Как видим, поддерживаются лишь форматы WAV и RAW;

- `supportedAudioCodecs()` — выводит список наименований всех поддерживаемых классом `QAudioRecorder` аудиокодеков, заданных в виде строк:

  ```
  for f in ar.supportedAudioCodecs(): print(f, end = " ")
  ```

  Выведет: `audio/pcm`

  Как можно видеть, поддерживается лишь кодек PCM — самый простой из имеющих хождение;

- `supportedAudioSampleRates([settings=QAudioEncoderSettings()])` — позволяет узнать поддерживаемые частоты дискретизации звука. Если в параметре `settings` указать экземпляр класса `QAudioEncoderSettings`, задающий параметры кодирования, будут возвращены только частоты, доступные для этих параметров.

  Возвращаемый результат представляет собой кортеж, первый элемент которого — список целочисленных значений частот дискретизации. Второй элемент будет значением `True`, если поддерживаются произвольные частоты дискретизации, или `False` — в противном случае:

  ```
  sr = ar.supportedAudioSampleRates()
  for f in sr[0]: print(f, end = " ")
  print()
  print(sr[1])
  ```

  Выведет:

  ```
  # 8000 11025 16000 22050 32000 44100 48000 88200 96000 192000
  # False
  ```

- `duration()` — возвращает текущую продолжительность записанного звука в миллисекундах;
- `volume()` — возвращает уровень записываемого звука в виде числа от `0` до `100`;
- `state()` — возвращает текущее состояние записи звука в виде значения одного из следующих атрибутов класса `QMediaRecorder`:
  - `StoppedState` — `0` — запись звука не начиналась или была остановлена;
  - `RecordingState` — `1` — идет запись звука;
  - `PausedState` — `2` — запись звука приостановлена;
- `status()` — возвращает текущее состояние подсистемы записи звука в виде значения одного из следующих атрибутов класса `QMediaRecorder`:
  - `UnavailableStatus` — `0` — состояние определить не удается, подсистема записи звука недоступна, или указаны не поддерживаемые параметры записи;
  - `UnloadedStatus` — `1` — подсистема записи звука неактивна;
  - `LoadingStatus` — `2` — подсистема записи звука в настоящий момент активизируется;
  - `LoadedStatus` — `3` — подсистема записи звука активна;
  - `StartingStatus` — `4` — запускается запись звука;
  - `RecordingStatus` — `5` — идет запись звука;
  - `PausedStatus` — `6` — запись звука приостановлена;
  - `FinalizingStatus` — `7` — запись звука остановлена, и идет сохранение аудиофайла;

- `error()` — возвращает ошибку, возникшую при записи звука, в виде значения одного из следующих атрибутов класса `QMediaRecorder`:
  - `NoError` — `0` — никакой ошибки не возникло;
  - `ResourceError` — `1` — подсистема записи звука недоступна или не готова к работе;
  - `FormatError` — `2` — заданы недопустимые параметры кодирования звука;
  - `OutOfSpaceError` — `3` — недостаточно места для сохранения аудиофайла;
- `errorString()` — возвращает текстовое описание ошибки.

Класс `QAudioRecorder` поддерживает довольно много сигналов, из которых нас интересуют лишь некоторые (полный их список можно найти на страницах **https://doc.qt.io/qt-5/qmediarecorder.html** и **https://doc.qt.io/qt-5/qaudiorecorder.html**):

- `durationChanged(<Продолжительность>)` — генерируется при изменении продолжительности записанного звука. В параметре передается новое значение продолжительности, заданное целым числом;
- `stateChanged(<Состояние>)` — генерируется при изменении состояния записи звука. В параметре передается целочисленный идентификатор нового состояния;
- `statusChanged(<Состояние>)` — генерируется при изменении состояния подсистемы записи звука. В параметре передается целочисленный идентификатор нового состояния;
- `volumeChanged(<Уровень звука>)` — генерируется при изменении уровня записываемого звука. В параметре передается новое значение уровня в виде целого числа.

28.4.2. Класс *QAudioEncoderSettings*

Класс `QAudioEncoderSettings` используется для настройки параметров кодирования записываемого звука. Формат вызова его конструктора очень прост:

```
<Объект> = QAudioEncoderSettings([<QAudioEncoderSettings>])
```

Если указать в качестве параметра экземпляр класса `QAudioEncoderSettings`, будет создана его копия.

Класс `QAudioEncoderSettings` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qaudioencodersettings.html**):

- `setCodec(<Кодек>)` — задает наименование используемого аудиокодека, указываемое в виде строки;
- `setSampleRate(<Частота дискретизации>)` — задает частоту дискретизации кодируемого звука, выраженную в герцах. Если указать значение `-1`, PyQt сама выберет оптимальную частоту дискретизации, исходя из возможностей звукового входа и выбранного кодека;
- `setChannelCount(<Количество каналов>)` — задает количество каналов кодируемого звука. Если указать значение `-1`, PyQt сама выберет оптимальное количество каналов, исходя из возможностей звукового входа и выбранного кодека;
- `setEncodingMode(<Режим кодирования>)` — задает режим кодирования звука. В качестве параметра указывается значение одного из следующих атрибутов класса `QMultimedia`:
  - `ConstantQualityEncoding` — `0` — режим кодирования с постоянным качеством. Качество задается методом `setQuality()`, а подходящий битрейт подбирается самой PyQt;

- `ConstantBitRateEncoding` — 1 — режим кодирования с постоянным битрейтом. Битрейт устанавливается вызовом метода `setBitRate()`, а значение качества соответственно выбирается самой PyQt;
- `AverageBitRateEncoding` — 2 — режим кодирования с переменным битрейтом. В этом случае метод `setBitRate()` задает величину среднего битрейта;
- `TwoPassEncoding` — 3 — режим кодирования в два прохода;

♦ `setBitRate(<Битрейт>)` — задает битрейт кодирования звука, выраженный в битах в секунду. Его имеет смысл указывать только в том случае, если задан режим кодирования с постоянным битрейтом (атрибут `ConstantBitRateEncoding`), с переменным битрейтом (атрибут `AverageBitRateEncoding`) или кодирование в два прохода (атрибут `TwoPassEncoding`);

♦ `setQuality(<Качество кодирования>)` — задает качество кодирования звука. В качестве параметра указывается значение одного из следующих атрибутов класса `QMultimedia`:

- `VeryLowQuality` — 0 — самое низкое качество;
- `LowQuality` — 1 — низкое качество;
- `NormalQuality` — 2 — нормальное качество;
- `HighQuality` — 3 — высокое качество;
- `VeryHighQuality` — 4 — наивысшее качество.

Качество кодирования имеет смысл указывать только в том случае, если задан режим кодирования с постоянным качеством (атрибут `ConstantQualityEncoding`);

♦ `setEncodingOption(<Параметр>, <Значение>)` — задает значение для указанного параметра кодирования. Наименование параметра кодирования задается в виде строки;

♦ `setEncodingOptions(<Словарь>)` — задает сразу несколько параметров кодирования звука. В параметре передается словарь, ключи которого представляют собой наименования параметров кодирования, а их значения — значения соответствующих параметров.

Набор поддерживаемых параметров кодирования зависит от выбранного звукового кодека.

Для примера напишем простую утилиту (листинг 28.5), осуществляющую запись звука с микрофона в минимальном качестве, — своего рода программный диктофон (рис. 28.4).

Листинг 28.5. Утилита для записи звука с микрофона

```
from PyQt5 import QtCore, QtWidgets, QtMultimedia
import sys, os

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
        QtWidgets.QWidget.__init__(self, parent, flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Запись звука")
        # Инициализируем подсистему для записи звука
        self.ardRecorder = QtMultimedia.QAudioRecorder()
        # Устанавливаем максимальную громкость
        self.ardRecorder.setVolume(100)
```

```
# Звук будет сохраняться в файле record.wav, находящемся
# в той же папке, где хранится программа
fn = QtCore.QUrl.fromLocalFile(os.path.abspath("record.wav"))
self.ardRecorder.setOutputLocation(fn)
# Устанавливаем звуковой вход по умолчанию
self.ardRecorder.setAudioInput(self.ardRecorder.defaultAudioInput())
# Указываем формат файла WAV
self.ardRecorder.setContainerFormat("audio/x-wav")
# Задаем параметры кодирования звука
aes = QtMultimedia.QAudioEncoderSettings()
aes.setCodec("audio/pcm")
aes.setSampleRate(8000)
aes.setChannelCount(1)
aes.setEncodingMode
(QtMultimedia.QMultimedia.ConstantQualityEncoding)
aes.setQuality(QtMultimedia.QMultimedia.VeryLowQuality)
self.ardRecorder.setAudioSettings(aes)
self.ardRecorder.statusChanged.connect(self.initRecorder)
self.ardRecorder.durationChanged.connect(self.showDuration)
# Создаем компоненты для запуска, приостановки и остановки
# записи звука и регулирования его уровня
vbox = QtWidgets.QVBoxLayout()
hbox = QtWidgets.QHBoxLayout()
self.btnRecord = QtWidgets.QPushButton("&Запись")
self.btnRecord.clicked.connect(self.ardRecorder.record)
hbox.addWidget(self.btnRecord)
self.btnPause = QtWidgets.QPushButton("П&ауза")
self.btnPause.clicked.connect(self.ardRecorder.pause)
self.btnPause.setEnabled(False)
hbox.addWidget(self.btnPause)
self.btnStop = QtWidgets.QPushButton("&Стоп")
self.btnStop.clicked.connect(self.ardRecorder.stop)
self.btnStop.setEnabled(False)
hbox.addWidget(self.btnStop)
vbox.addLayout(hbox)
hbox = QtWidgets.QHBoxLayout()
lblVolume = QtWidgets.QLabel("&Уровень записи")
hbox.addWidget(lblVolume)
sldVolume = QtWidgets.QSlider(QtCore.Qt.Horizontal)
sldVolume.setRange(0, 100)
sldVolume.setTickPosition(QtWidgets.QSlider.TicksAbove)
sldVolume.setTickInterval(10)
sldVolume.setValue(100)
lblVolume.setBuddy(sldVolume)
sldVolume.valueChanged.connect(self.ardRecorder.setVolume)
hbox.addWidget(sldVolume)
vbox.addLayout(hbox)
# Создаем надпись, в которую будет выводиться состояние программы
self.lblStatus = QtWidgets.QLabel("Готово")
vbox.addWidget(self.lblStatus)
```

```
        self.setLayout(vbox)
        self.resize(300, 100)

    # В зависимости от состояния записи звука делаем нужные
    # кнопки доступными или, напротив, недоступными и выводим
    # соответствующий текст в надписи
    def initRecorder(self, status):
        if status == QtMultimedia.QMediaRecorder.RecordingStatus:
            self.btnRecord.setEnabled(False)
            self.btnPause.setEnabled(True)
            self.btnStop.setEnabled(True)
            self.lblStatus.setText("Запись")
        elif status == QtMultimedia.QMediaRecorder.PausedStatus:
            self.btnRecord.setEnabled(True)
            self.btnPause.setEnabled(False)
            self.lblStatus.setText("Пауза")
        elif status == QtMultimedia.QMediaRecorder.FinalizingStatus:
            self.btnRecord.setEnabled(True)
            self.btnPause.setEnabled(False)
            self.btnStop.setEnabled(False)
            self.lblStatus.setText("Готово")

    # Выводим продолжительность записанного звука
    def showDuration(self, duration):
        self.lblStatus.setText("Записано " + str(duration // 1000) +
                               " секунд")

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

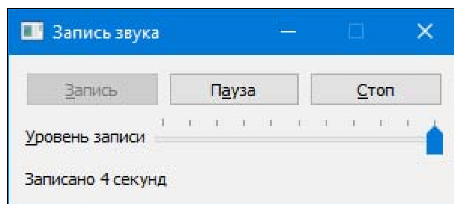

Рис. 28.4. Утилита для записи звука

28.5. Класс *QSoundEffect*

Класс `QSoundEffect` предоставляет средства воспроизведения коротких звуковых файлов, что может пригодиться, скажем, для информирования пользователя о каком-либо событии. Конструктор этого класса имеет следующий формат вызова:

```
<Объект> = QSoundEffect([parent=None])
```

В параметре `parent` может быть указана ссылка на родительский компонент.

Класс `QSoundEffect` поддерживает следующие основные методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qsoundeffect.html**):

- `setSource(<QUrl>)` — задает файл-источник воспроизводимого звука в виде экземпляра класса `QUrl`;
- `setVolume(<Громкость>)` — задает громкость воспроизводимого звука в виде вещественного числа от `0.0` (звук отключен) до `1.0` (максимальная громкость);
- `setLoopCount(<Количество повторений>)` — задает количество повторений звука при воспроизведении. Значение `0` или `1` предписывает воспроизвести звук всего один раз, а значение атрибута `Infinite` класса `QSoundEffect` или `-2` — воспроизводить его снова и снова бесконечное количество раз;
- `setCategory(<Категория>)` — задает наименование категории (указывается в виде строки), к которой относится воспроизводимый звук;
- `setMuted(<Флаг>)` — если передать значение `True`, звук при воспроизведении будет отключен. Значение `False` вновь включает звук;
- `play()` — запускает воспроизведение звука. Метод является слотом;
- `stop()` — останавливает воспроизведение звука. Метод является слотом;
- `isLoaded()` — возвращает `True`, если звук загружен из указанного в методе `setSource()` файла, и `False` — в противном случае;
- `isPlaying()` — возвращает `True`, если звук в текущий момент воспроизводится, и `False` — в противном случае;
- `loopsRemaining()` — возвращает количество оставшихся повторов воспроизведения звука или значение атрибута `Infinite` класса `QSoundEffect`, если было задано бесконечное воспроизведение;
- `volume()` — возвращает текущее значение громкости звука в виде вещественного числа от 0.0 до 1.0;
- `loopCount()` — возвращает количество повторений звука, заданное вызовом метода `setLoopCount()`;
- `isMuted()` — возвращает `True`, если воспроизводящийся звук в текущий момент отключен, и `False` — в противном случае;
- `status()` — возвращает текущее состояние звука в виде значения одного из следующих атрибутов класса `QSoundEffect`:
  - `Null` — `0` — источник звука не задан;
  - `Loading` — `1` — идет загрузка звука из файла-источника;
  - `Ready` — `2` — звук загружен и готов к воспроизведению;
  - `Error` — `3` — в процессе загрузки звука возникла ошибка;
- `supportedMimeTypes()` — возвращает список наименований поддерживаемых форматов звука в виде строк:

  ```
  for f in QtMultimedia.QSoundEffect.supportedMimeTypes():
      print(f, end = " ")
  ```

  Выведет: `audio/x-wav audio/wav audio/wave audio/x-pn-wav`

  Метод является статическим.

Как видим, поддерживается лишь формат WAV в виде четырех различных MIME-типов. Что касается аудиокодеков, то, по результатам экспериментов авторов, поддерживается лишь кодек PCM.

Класс `QSoundEffect` поддерживает следующие наиболее полезные для нас сигналы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qsoundeffect.html**):

- `statusChanged()` — генерируется при изменении состояния звука;
- `loadedChanged()` — генерируется при изменении состояния загрузки звука из файла-источника;
- `playingChanged()` — генерируется при изменении состояния воспроизведения звука;
- `loopsRemainingChanged()` — генерируется при каждом повторении воспроизводимого звука;
- `volumeChanged()` — генерируется при изменении громкости звука;
- `mutedChanged()` — генерируется при отключении звука или, наоборот, восстановлении его громкости через вызов метода `setMuted()`.

Примером к этому разделу станет небольшая программа, воспроизводящая звук, в зависимости от выбора пользователя: единожды, десять раз или бесконечно (листинг 28.6).

Листинг 28.6. Воспроизведение звуковых эффектов

```
from PyQt5 import QtCore, QtWidgets, QtMultimedia
import sys, os

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
        QtWidgets.QWidget.__init__(self, parent, flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Звуковые эффекты")
        # Инициализируем подсистему вывода звуковых эффектов
        self.sndEffect = QtMultimedia.QSoundEffect()
        self.sndEffect.setVolume(1)
        # Задаем файл-источник
        fn = QtCore.QUrl.fromLocalFile(os.path.abspath("effect.wav"))
        self.sndEffect.setSource(fn)
        self.sndEffect.loopsRemainingChanged.connect(self.showCount)
        self.sndEffect.playingChanged.connect(self.clearCount)
        vbox = QtWidgets.QVBoxLayout()
        # Создаем кнопки для запуска воспроизведения звука
        lblPlay = QtWidgets.QLabel("Воспроизвести...")
        vbox.addWidget(lblPlay)
        btnOnce = QtWidgets.QPushButton("...&один раз")
        btnOnce.clicked.connect(self.playOnce)
        vbox.addWidget(btnOnce)
        btnTen = QtWidgets.QPushButton("...&десять раз")
        btnTen.clicked.connect(self.playTen)
        vbox.addWidget(btnTen)
        btnInfinite = QtWidgets.QPushButton(
            "...&бесконечное количество раз")
```

```
        btnInfinite.clicked.connect(self.playInfinite)
        vbox.addWidget(btnInfinite)
        btnStop = QtWidgets.QPushButton("&Стоп")
        btnStop.clicked.connect(self.sndEffect.stop)
        vbox.addWidget(btnStop)
        self.lblStatus = QtWidgets.QLabel("")
        vbox.addWidget(self.lblStatus)
        self.setLayout(vbox)
        self.resize(200, 100)

    def playOnce(self):
        self.sndEffect.setLoopCount(1)
        self.sndEffect.play()

    def playTen(self):
        self.sndEffect.setLoopCount(10)
        self.sndEffect.play()

    def playInfinite(self):
        self.sndEffect.setLoopCount(QtMultimedia.QSoundEffect.Infinite)
        self.sndEffect.play()

    # Выводим количество повторений воспроизведения эффекта
    def showCount(self):
        self.lblStatus.setText("Воспроизведено " +
                               str(self.sndEffect.loopCount() -
                               self.sndEffect.loopsRemaining()) + " раз")

    # Если воспроизведение закончено, очищаем выведенное ранее
    # количество повторений эффекта
    def clearCount(self):
        if not self.sndEffect.isPlaying():
            self.lblStatus.setText("")

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

Примечание

Помимо описанных здесь возможностей, PyQt 5 поддерживает доступ к устройствам воспроизведения звука на низком уровне, а также получение непосредственно массива звуковых и видеоданных с целью их анализа и обработки. Соответствующие программные инструменты описаны в документации по этой библиотеке.

ГЛАВА 29

Печать документов

PyQt 5 поддерживает ряд развитых средств, позволяющих выполнить печать документов, их предварительный просмотр и экспорт в формат PDF.

Любой установленный в системе принтер представляется классом `QPrinter`. Поскольку он является подклассом класса `QPagedPaintDevice`, который, в свою очередь, наследует рассмотренный в *главе 24* класс `QPaintDevice`, мы можем для вывода документов на печать использовать методы последнего.

Класс `QPrintDialog` обеспечивает функциональность диалогового окна выбора принтера, а класс `QPageSetupDialog` — диалогового окна установки параметров страницы. Для предварительного просмотра печатаемых документов задействуются классы `QPrintPreviewDialog` и `QPrintPreviewWidget`: первый выводит документ в отдельное диалоговое окно, а второй — в специализированный компонент, который мы можем использовать вместе с любыми другими изученными нами компонентами. Класс `QPrinterInfo` позволяет выяснить, какие принтеры имеются в наличии, и узнать параметры любого из установленных принтеров.

PyQt 5 поддерживает и экспорт документов в формат Adobe PDF исключительно встроенными средствами, без привлечения каких бы то ни было сторонних программ. Для выполнения этой задачи служит класс `QPdfWriter`.

Все описанные в этой главе классы определены в модуле `QtPrintSupport`, если не указано иное.

29.1. Основные средства печати

Вывести документ на принтер в PyQt довольно просто. Почти все, что нам нужно для этого знать, было описано в *главе 24*.

29.1.1. Класс *QPrinter*

Как уже говорилось, класс `QPrinter` представляет установленный в системе принтер. Его иерархия наследования такова:

```
QPaintDevice - QPagedPaintDevice - QPrinter
```

Конструктор класса `QPrinter` имеет следующие форматы вызова:

```
<Объект> = QPrinter([mode=ScreenResolution])
<Объект> = QPrinter(<QPrinterInfo>[, mode=ScreenResolution])
```

Первый формат выбирает для печати принтер по умолчанию. В параметре `mode` может быть указано разрешение принтера, заданное в виде значения одного из следующих атрибутов класса `QPrinter`:

- `ScreenResolution` — `0` — разрешение экрана. Позволяет вывести документ максимально быстро, но в худшем качестве;
- `HighResolution` — `2` — разрешение принтера. Печать выполняется качественнее, но медленнее.

Второй формат позволяет выбрать произвольный принтер из числа установленных в системе. Этот принтер задается экземпляром класса `QPrinterInfo`, речь о котором пойдет далее.

Класс `QPrinter` поддерживает очень много методов, из которых мы рассмотрим лишь наиболее полезные (полный их список можно найти на странице **https://doc.qt.io/qt-5/qprinter.html**).

- `setPrinterName(<Имя принтера>)` — выполняет подключение к принтеру с заданным в виде строки именем. Если передать пустую строку, будет выполнено подключение к встроенной в PyQt подсистеме вывода документов в формате PDF;
- `printerName()` — возвращает строку с именем принтера, к которому выполнено подключение;
- `setOutputFileName(<Путь к файлу>)` — задает путь к файлу, в который будет выведен печатаемый документ (так называемая *печать в файл*). Если файл имеет расширение pdf, будет выполнен вывод в формате PDF. В противном случае файл будет сохранен в формате, установленном в вызове метода `setOutputFormat()`. Чтобы отключить вывод документа в файл, следует вызвать этот метод, передав ему в качестве параметра пустую строку;
- `outputFileName()` — возвращает путь к файлу, в который будет выведен документ вместо печати на бумаге;
- `setOutputFormat(<Формат вывода>)` — задает формат вывода документа при печати в файл. В качестве параметра передается значение одного из следующих атрибутов класса `QPrinter`:
  - `NativeFormat` — `0` — будет выполнен вывод во внутреннем формате принтера. Этот режим автоматически устанавливается при создании экземпляра класса `QPrinter` и вызове метода `setPrinterName()` с указанием имени существующего принтера. Если экземпляр класса подключен к подсистеме вывода в формате PDF, будет выполнено переподключение к принтеру по умолчанию;
  - `PdfFormat` — `1` — будет выполнен вывод в формате PDF. Этот режим автоматически устанавливается при вызове метода `setPrinterName()` с указанием пустой строки;
- `outputFormat()` — возвращает обозначение формата вывода документа;
- `isValid()` — возвращает `True`, если принтер действительно установлен в системе и готов к работе, и `False` — в противном случае.
- `setPageSize(<QPageSize>)` — задает размер страницы в виде экземпляра класса `QPageSize` из модуля `QtGui`. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае. Вот пример задания размера бумаги A4:

```
ps = QtGui.QPageSize(QtGui.QPageSize.A4)
printer.setPageSize(ps)
```

- `setPageOrientation(<Ориентация>)` — задает ориентацию страницы в виде значения атрибута `Portrait` (`0`, портретная) или `Landscape` (`1`, ландшафтная) класса `QPageLayout` из модуля `QtGui`. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setPageMargins(<Слева>, <Сверху>, <Справа>, <Снизу>, <Единица измерения>)` — задает отступы от краев страницы, соответственно, слева, сверху, справа и снизу в заданной единице измерения. Сами отступы указываются в виде вещественных чисел, а единица измерения — в виде значения одного из следующих атрибутов класса `QPageLayout`:
  - `Millimeter` — `0` — миллиметры;
  - `Point` — `1` — пункты;
  - `Inch` — `2` — дюймы;
  - `Pica` — `3` — пики;
  - `Didot` — `4` — дидо (0,375 мм);
  - `Cicero` — `5` — цицеро (4,5 мм).

  Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае. Вот пример задания отступов в 5 мм со всех сторон страницы:

  ```
  printer.setPageMargins(5, 5, 5, 5, QtGui.QPageLayout.Millimeter)
  ```
- `setPageLayout(<QPageLayout>)` — задает сразу все параметры страницы (размеры, ориентацию и отступы от краев страницы в виде экземпляра класса `QPageLayout`. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `pageLayout()` — возвращает экземпляр класса `QPageLayout`, хранящий сведения о размере бумаги, ориентации страницы и величинах отступов от краев страницы;
- `setCopyCount(<Количество копий>)` — задает количество копий печатаемого документа;
- `copyCount()` — возвращает количество копий печатаемого документа;
- `setCollateCopies(<Флаг>)` — если передано значение `True`, каждая копия документа будет отпечатана полностью, прежде чем начнется печать следующей копии. Если передать значение `False`, сначала будут отпечатаны все копии первой страницы, потом все копии второй и т. д.;
- `collateCopies()` — возвращает `True`, если каждая копия документа печатается полностью, и `False` — в противном случае;
- `setDuplex(<Режим двухсторонней печати>)` — задает режим двусторонней печати в виде значения одного из следующих атрибутов класса `QPrinter`:
  - `DuplexNone` — `0` — односторонняя печать;
  - `DuplexAuto` — `1` — двусторонняя печать с автоматическим определением стороны листа, вокруг которой следует его перевернуть;
  - `DuplexLongSide` — `2` — двусторонняя печать с переворачиванием листа вокруг длинной стороны;
  - `DuplexShortSide` — `3` — двусторонняя печать с переворачиванием листа вокруг короткой стороны;
- `duplex()` — возвращает обозначение заданного для принтера режима двусторонней печати;

- setPrintRange(<Диапазон печати>) — задает диапазон печати документа в виде значения одного из следующих атрибутов класса QPrinter:
  - AllPages — 0 — печатать все страницы;
  - Selection — 1 — печатать только выделенный фрагмент;
  - PageRange — 2 — печатать только заданный диапазон страниц;
  - CurrentPage — 3 — печатать только текущую страницу;
- setFromTo(<Начальная страница>, <Конечная страница>) — задает диапазон печатаемых страниц в документе;
- fromPage() — возвращает начальную страницу диапазона печати;
- toPage() — возвращает конечную страницу диапазона печати;
- setColorMode(<Цветовой режим>) — задает режим вывода цвета в виде значения атрибута Color (1, цветной) или Grayscale (0, черно-белый) класса QPrinter;
- colorMode() — возвращает обозначение режима вывода цвета, заданного для принтера;
- setResolution(<Разрешение>) — задает разрешение для принтера в виде целого числа в точках на дюйм. Если указано неподдерживаемое значение разрешения, будет выставлено разрешение, наиболее близкое к заданному;
- resolution() — возвращает заданное для принтера разрешение;
- setPaperSource(<Источник бумаги>) — задает источник бумаги для принтера в виде значения одного из следующих атрибутов класса QPrinter (здесь приведены лишь наиболее часто используемые источники — полный их список можно найти на странице **https://doc.qt.io/qt-5/qprinter.html#PaperSource-enum**):
  - OnlyOne — 0 — единственный лоток принтера или лоток, используемый по умолчанию;
  - Lower — 1 — нижний лоток;
  - Middle — 2 — средний лоток;
  - Manual — 3 — лоток для ручной подачи бумаги;
  - Envelope — 4 — лоток для конвертов;
  - EnvelopeManual — 5 — лоток для ручной подачи конвертов;
  - Auto — 6 — автоматический выбор источника;
- paperSource() — возвращает обозначение источника бумаги;
- setFontEmbeddingEnabled(<Флаг>) — если передано значение True, в создаваемый документ PDF будут внедрены все использованные в его тексте шрифты, если передано значение False — не будут;
- supportsMultipleCopies() — возвращает True, если принтер сам способен напечатать несколько копий документа, и False — в противном случае;
- supportedResolutions() — возвращает список разрешений, поддерживаемых принтером и измеряемых в точках на дюйм:

  ```
  printer = QtPrintSupport.QPrinter()
  for f in printer.supportedResolutions(): print(f, end = " ")
  ```

  У авторов вывел: 600

Как видим, принтер авторов, установленный по умолчанию, поддерживает лишь разрешение 600 точек на дюйм.

Внимание!

Для некоторых принтеров метод `supportResolutions()` по какой-то причине выводит пустой список.

29.1.2. Вывод на печать

Процесс вывода документа на печать средствами PyQt можно разбить на следующие этапы:

1. Создание принтера (экземпляра класса `QPrinter`) и задание его параметров:
   ```
   printer = QtPrintSupport.QPrinter()
   ```
2. Создание экземпляра класса `QPainter`:
   ```
   painter = QtGui.QPainter()
   ```
3. Вызов метода `begin(<QPaintDevice>)` класса `QPainter` и передача ему в качестве параметров только что созданного принтера:
   ```
   painter.begin(printer)
   ```
   Метод `begin()` инициирует процесс вывода графики на принтер. Он возвращает `True`, если инициализация прошла успешно, и `False` — в противном случае.
4. Выполнение вывода необходимой графики, составляющей собственно содержимое документа, средствами класса `QPainter` (см. *главу 24*).
5. В случае необходимости начать вывод новой страницы — вызов метода `newPage()` класса `QPrinter`:
   ```
   printer.newPage()
   ```
   Метод `newPage()` подготавливает принтер к выводу новой страницы и возвращает `True`, если подготовка увенчалась успехом, и `False` — в противном случае.
6. По окончании вывода документа — вызов метода `end()` класса `QPainter`:
   ```
   painter.end()
   ```
   Этот метод завершает рисование графики и возвращает `True`, если вывод графики был закончен успешно, и `False` — в противном случае. После его вызова выполняется собственно печать документа.

Перед началом вывода графики нам понадобится определить размеры всей страницы принтера, размеры области на ней, доступной для рисования, и некоторые другие параметры, касающиеся этих размеров. Для этого мы воспользуемся следующими методами класса `QPrinter` (полный список их можно найти на страницах **https://doc.qt.io/qt-5/qprinter.html** и **https://doc.qt.io/qt-5/qpaintdevice.html**):

- `width()` — возвращает ширину области, доступной для вывода графики, в пикселах;
- `height()` — возвращает высоту области, доступной для вывода графики, в пикселах;
- `widthMM()` — возвращает ширину области, доступной для вывода графики, в миллиметрах;
- `heightMM()` — возвращает высоту области, доступной для вывода графики, в миллиметрах;

- `colorCount()` — возвращает количество цветов, которые способен выводить принтер;
- `pageRect(<Единица измерения>)` — возвращает размеры области на странице, доступной для вывода графики, в виде экземпляра класса `QRectF`. Единица измерения задается в виде значения одного из следующих атрибутов класса `QPrinter`:
  - `Millimeter` — `0` — миллиметры;
  - `Point` — `1` — пункты;
  - `Inch` — `2` — дюймы;
  - `Pica` — `3` — пики;
  - `Didot` — `4` — дидо (0,375 мм);
  - `Cicero` — `5` — цицеро (4,5 мм);
  - `DevicePixel` — `6` — пикселы;
- `paperRect([<Единица измерения>])` — возвращает размеры страницы целиком в виде экземпляра класса `QRectF`. Единица измерения задается в виде значения одного из атрибутов класса `QPrinter`, приведенных ранее в описании метода `pageRect()`.

Начало координат находится в левом верхнем углу области, отведенной под вывод графики, — лишь в этой области мы можем выводить графику. Горизонтальная координатная ось направлена направо, а вертикальная — вниз.

Для примера напишем код, который будет выводить на установленный по умолчанию принтер документ из двух страниц (листинг 29.1). Первую страницу мы обведем точечной синей рамкой, а по ее центру расположим надпись «QPrinter». Вторая страница будет отпечатана в ландшафтной ориентации, и ее полностью займет графическое изображение.

Листинг 29.1. Использование класса `QPrinter`

```
from PyQt5 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys
app = QtWidgets.QApplication(sys.argv)
# Создаем принтер
printer = QtPrintSupport.QPrinter()
# Для целей отладки лучше выводить документ не на принтер,
# а в файл в формате PDF. Чтобы сделать это, достаточно
# раскомментировать следующую строчку кода:
# printer.setOutputFileName("output.pdf")
# Создаем поверхность рисования и привязываем ее к принтеру
painter = QtGui.QPainter()
painter.begin(printer)
# Рисуем рамку вокруг страницы
pen = QtGui.QPen(QtGui.QColor(QtCore.Qt.blue), 5,
                 style = QtCore.Qt.DotLine)
painter.setPen(pen)
painter.setBrush(QtCore.Qt.NoBrush)
painter.drawRect(0, 0, printer.width(), printer.height())
# Выводим надпись
color = QtGui.QColor(QtCore.Qt.black)
painter.setPen(QtGui.QPen(color))
painter.setBrush(QtGui.QBrush(color))
```

```
font = QtGui.QFont("Verdana", pointSize = 42)
painter.setFont(font)
painter.drawText(10, printer.height() // 2 - 100, printer.width() - 20,
                 50, QtCore.Qt.AlignCenter |
                 QtCore.Qt.TextDontClip, "QPrinter")
# Изменяем ориентацию страницы. Сделать это нужно перед вызовом
# метода newPage()
printer.setPageOrientation(QtGui.QPageLayout.Landscape)
# Переходим на новую страницу
printer.newPage()
# Выводим изображение
pixmap = QtGui.QPixmap("img.jpg")
pixmap = pixmap.scaled(printer.width(), printer.height(),
                       aspectRatioMode = QtCore.Qt.KeepAspectRatio)
painter.drawPixmap(0, 0, pixmap)
painter.end()
```

В листинге 29.2 приведен код класса `PrintList`, реализующий печать списков или содержимого таблиц баз данных в виде полноценного табличного отчета. Этот класс можно использовать для разработки бизнес-приложений.

Листинг 29.2. Класс `PrintList`, выводящий на печать табличные данные

```
from PyQt5 import QtCore, QtGui, QtPrintSupport

class PrintList:
    def __init__(self):
        self.printer = QtPrintSupport.QPrinter()
        self.headerFont = QtGui.QFont("Arial", pointSize = 10,
                                      weight = QtGui.QFont.Bold)
        self.bodyFont = QtGui.QFont("Arial", pointSize = 10)
        self.footerFont = QtGui.QFont("Arial", pointSize = 9, italic = True)
        self.headerFlags = QtCore.Qt.AlignHCenter | QtCore.Qt.TextWordWrap
        self.bodyFlags = QtCore.Qt.TextWordWrap
        self.footerFlags = QtCore.Qt.AlignHCenter | QtCore.Qt.TextWordWrap
        color = QtGui.QColor(QtCore.Qt.black)
        self.headerPen = QtGui.QPen(color, 2)
        self.bodyPen = QtGui.QPen(color, 1)
        self.margin = 5
        self._resetData()
    def _resetData(self):
        self.headers = None
        self.columnWidths = None
        self.data = None
        self._brush = QtCore.Qt.NoBrush
        self._currentRowHeight = 0
        self._currentPageHeight = 0
        self._headerRowHeight = 0
        self._footerRowHeight = 0
```

```
        self._currentPageNumber = 1
        self._painter = None

    def printData(self):
        self._painter = QtGui.QPainter()
        self._painter.begin(self.printer)
        self._painter.setBrush(self._brush)
        if self._headerRowHeight == 0:
            self._painter.setFont(self.headerFont)
            self._headerRowHeight = self._calculateRowHeight(
                self.columnWidths, self.headers)
        if self._footerRowHeight == 0:
            self._painter.setFont(self.footerFont)
            self._footerRowHeight = self._calculateRowHeight(
                [self.printer.width()], "Страница")
        for i in range(len(self.data)):
            height = self._calculateRowHeight(self.columnWidths, self.data[i])
            if self._currentPageHeight + height > self.printer.height() - ↳
            self._footerRowHeight - 2 * self.margin:
                self._printFooterRow()
                self._currentPageHeight = 0
                self._currentPageNumber += 1
                self.printer.newPage()
            if self._currentPageHeight == 0:
                self._painter.setPen(self.headerPen)
                self._painter.setFont(self.headerFont)
                self.printRow(self.columnWidths, self.headers,
                              self._headerRowHeight, self.headerFlags)
                self._painter.setPen(self.bodyPen)
                self._painter.setFont(self.bodyFont)
            self.printRow(self.columnWidths, self.data[i], height, self.bodyFlags)
        self._printFooterRow()
        self._painter.end()
        self._resetData()

    def _calculateRowHeight(self, widths, cellData):
        height = 0
        for i in range(len(widths)):
            r = self._painter.boundingRect(0, 0, widths[i] - 2 *
                self.margin, 50, QtCore.Qt.TextWordWrap, str(cellData[i]))
            h = r.height() + 2 * self.margin
            if height < h:
                height = h
        return height

    def printRow(self, widths, cellData, height, flags):
        x = 0
        for i in range(len(widths)):
            self._painter.drawText(x + self.margin,
                 self._currentPageHeight + self.margin,
```

```
                widths[i] - self.margin, height - 2 * self.margin,
                flags, str(cellData[i]))
            self._painter.drawRect(x, self._currentPageHeight,
                                   widths[i], height)
            x += widths[i]
        self._currentPageHeight += height

    def _printFooterRow(self):
        self._painter.setFont(self.footerFont)
        self._painter.drawText(self.margin, self.printer.height() -
             self._footerRowHeight - self.margin, self.printer.width() -
             2 * self.margin, self._footerRowHeight - 2 * self.margin,
             self.footerFlags, "Страница " + str(self._currentPageNumber))
```

Пользоваться этим классом очень просто. Сначала нужно создать его экземпляр, вызвав конструктор следующего формата:

```
<Объект> = PrintList()
```

После чего задать параметры печатаемого табличного отчета, воспользовавшись следующими атрибутами класса `PrintList`:

- `headers` — заголовки для столбцов табличного отчета в виде списка строк;
- `columnWidths` — значения ширины для столбцов, измеряемые в пикселах, в виде списка целочисленных величин;
- `data` — собственно выводимые данные. Они должны представлять собой список значений, выводимых в отдельных ячейках, — каждый из элементов этого списка задает данные для одной строки.

  Следующие свойства являются необязательными для указания:
- `printer` — задает принтер (ссылку на экземпляр класса `QPrinter`), через который отчет выводится на печать. Изначально хранит принтер, используемый по умолчанию;
- `headerFont` — шрифт, используемый для вывода текста «шапки» табличного отчета. По умолчанию — полужирный шрифт Arial размером 10 пунктов;
- `headerPen` — параметры рамки, рисуемой вокруг ячеек «шапки». По умолчанию — черная линия толщиной 2 пиксела;
- `headerFlags` — параметры вывода текста ячеек «шапки». По умолчанию — выравнивание по центру и перенос по словам;
- `bodyFont` — шрифт, используемый для вывода текста обычных строк. По умолчанию — шрифт Arial размером 10 пунктов;
- `bodyPen` — параметры рамки, рисуемой вокруг ячеек обычных строк. По умолчанию — черная линия толщиной 1 пиксел;
- `bodyFlags` — параметры вывода текста ячеек обычных строк. По умолчанию — перенос по словам;
- `footerFont` — шрифт, используемый для вывода текста «поддона» таблицы. По умолчанию — курсивный шрифт Arial размером 9 пунктов;
- `footerFlags` — параметры вывода текста «поддона». По умолчанию — выравнивание по центру и перенос по словам;

- `margin` — величина просвета между рамкой ячейки и ее содержимым. По умолчанию — 5 пикселов.

После задания всех необходимых параметров следует вызвать метод `printData()` класса `PrintList`, который и выполняет печать данных. Впоследствии, пользуясь тем же экземпляром этого класса, мы можем распечатать другой набор данных.

В листинге 29.3 приведен код тестового приложения, выводящего на экран числа от 1 до 100, их квадраты и кубы. Подразумевается, что код, приведенный в листинге 29.2, сохранен в файле PrintList.py.

Листинг 29.3. Тестовое приложение для проверки класса `PrintList`

```
from PyQt5 import QtWidgets
import sys
import PrintList
app = QtWidgets.QApplication(sys.argv)
pl = PrintList.PrintList()
# Если требуется вывести документ в файл формата PDF,
# следует раскомментировать эту строку:
# pl.printer.setOutputFileName("output.pdf")
data = []
for b in range(1, 101):
    data.append([b, b ** 2, b ** 3])
pl.data = data
pl.columnWidths = [100, 100, 200]
pl.headers = ["Аргумент", "Квадрат", "Куб"]
pl.printData()
```

В результате мы должны получить табличный документ из трех страниц с тремя столбцами и нумерацией в «поддоне».

29.1.3. Служебные классы

PyQt 5 предоставляет нам два служебных класса, позволяющих описать параметры используемой для печати бумаги, величины отступов от края страницы или же все это сразу.

29.1.3.1. Класс *QPageSize*

Класс `QPageSize`, объявленный в модуле `QtGui`, описывает размеры, или, говоря другими словами, *формат* бумаги, на которой осуществляется печать. Для указания размера бумаги, который будет использовать принтер, предназначен метод `setPageSize()` класса `QPrinter`.

Форматы конструкторов класса:

```
<Объект> = QPageSize()
<Объект> = QPageSize(<Идентификатор размера бумаги>)
<Объект> = QPageSize(<QSize>[, name=""][, matchPolicy=FuzzyMatch])
<Объект> = QPageSize(<QSizeF>, <Единица измерения>[, name=""][,
                     matchPolicy=FuzzyMatch])
<Объект> = QPageSize(<QPageSize>)
```

Первый формат создает «пустой» экземпляр класса, не хранящий данные ни о каком размере бумаги.

Второй формат позволяет указать размер сразу, в виде значения одного из следующих атрибутов класса `QPageSize` (здесь приведены только наиболее употребительные размеры — полный их список можно найти на странице **https://doc.qt.io/qt-5/qpagesize.html#PageSizeId-enum**): `A0` (5), `A1` (6), `A2` (7), `A3` (8), `A4` (0), `A5` (9), `Letter` (2), `Legal` (3).

Третий и четвертый форматы служат для создания нестандартных размеров бумаги, при этом третий формат принимает в качестве первого параметра экземпляр класса `QSize` (см. *разд. 18.5.2*), указывающий сами размеры в пунктах. В параметре `name` можно задать имя создаваемого размера бумаги — если он не указан, будет создано имя по умолчанию вида `Custom (<Ширина> x <Высота>)`.

Если заданные размеры близки к размерам какого-либо из стандартных форматов бумаги, будет использован этот формат. В параметре `matchPolicy` можно задать режим подбора стандартного формата бумаги в виде одного из следующих атрибутов класса `QPageSize`:

- `FuzzyMatch` — `0` — размеры стандартного формата должны быть близки к заданным в конструкторе размерам в определенных рамках с учетом ориентации;
- `FuzzyOrientationMatch` — `1` — размеры стандартного формата должны быть близки к заданным в конструкторе размерам в определенных рамках без учета ориентации;
- `ExactMatch` — `2` — размеры стандартного формата должны точно совпадать с заданными в конструкторе.

Четвертый формат принимает в качестве первого параметра экземпляр класса `QSizeF`, указывающий размеры бумаги. Вторым параметром должна быть задана единица измерения размеров в виде значения одного из следующих атрибутов класса `QPageSize`:

- `Millimeter` — `0` — миллиметры;
- `Point` — `1` — пункты;
- `Inch` — `2` — дюймы;
- `Pica` — `3` — пики;
- `Didot` — `4` — дидо (0,375 мм);
- `Cicero` — `5` — цицеро (4,5 мм).

Последний формат позволяет создать копию указанного в параметре экземпляра класса `QPageSize`.

Примеры:

```
# Задаем размер бумаги A5
ps = QtGui.QPageSize(QtGui.QPageSize.A5)
printer.setPageSize(ps)
# Задаем размер бумаги 100 x 100 пунктов с названием "Особый размер"
sz = QtCore.QSize(400, 300)
ps = QtGui.QPageSize(sz, name="Особый размер", ↳
matchPolicy=QtGui.QPageSize.FuzzyMatch)
pl.printer.setPageSize(ps)
```

Класс `QPageSize` поддерживает следующие полезные для нас методы (полный их список можно найти на странице **https://doc.qt.io/qt-5/qpagesize.html**):

- size(<Единица измерения>) и rect(<Единица измерения>) — возвращают размеры бумаги в заданных единицах измерения в виде экземпляра класса QSizeF или QRectF соответственно;
- size(<Идентификатор размера бумаги>, <Единица измерения>) — возвращает размеры бумаги для заданного идентификатора формата в заданных единицах измерения, представленные экземпляром класса QSizeF:

  ```
  s = QtGui.QPageSize.size(QtGui.QPageSize.A0, QtGui.QPageSize.Millimeter)
  print(s.width(), "x", s.height())
  ```

  Выведет: 841.0 x 1189.0

  Метод является статическим;
- sizePixels(<Разрешение>) и rectPixels(<Разрешение>) — возвращают размеры бумаги в пикселах для заданного разрешения, которое измеряется в точках на дюйм. Результат представляет собой экземпляр класса QSize или QRect соответственно:

  ```
  ps = QtGui.QPageSize(QtGui.QPageSize.A5)
  s = ps.sizePixels(600)
  print(s.width(), "x", s.height())
  ```

  Выведет: 3500 x 4958
- sizePixels(<Идентификатор размера бумаги>, <Разрешение>) — возвращает размеры бумаги в пикселах для заданных идентификатора формата и разрешения, которое измеряется в точках на дюйм. Результат представляет собой экземпляр класса QSize. Метод является статическим;
- sizePoints() и rectPoints() — возвращают размеры бумаги в пунктах в виде экземпляра класса QSize или QRect соответственно;
- sizePoints(<Идентификатор размера бумаги>) — возвращает размеры бумаги в пунктах для заданного идентификатора размера в виде экземпляра класса QSize. Метод является статическим;
- swap(<QPageSize>) — меняет размер бумаги, хранящийся в текущем экземпляре класса, на заданный в параметре;
- name() — возвращает имя формата бумаги;
- name(<Идентификатор размера бумаги>) — возвращает имя формата бумаги, заданного идентификатором. Метод является статическим;
- isEquivalentTo(<QPageSize>) — возвращает True, если размеры текущего формата бумаги равны размерам формата, переданного в параметре, и False — в противном случае.

Также класс QPageSize поддерживает операторы сравнения == и !=:

```
s1 = QtGui.QPageSize(QtGui.QPageSize.A4)
s2 = QtGui.QPageSize(QtGui.QPageSize.A4)
s3 = QtGui.QPageSize(QtGui.QPageSize.A3)
print(s1 == s2)                              # Выведет: True
print(s1 != s3)                              # Выведет: True
```

29.1.3.2. Класс *QPageLayout*

Класс QPageLayout, также объявленный в модуле QtGui, представляет сразу размеры, ориентацию страницы и величины отступов от краев страницы. Передать все эти сведения принтеру позволяет метод setPageLayout() класса QPrinter.

Форматы вызова конструктора этого класса:

```
<Объект> = QPageLayout()
<Объект> = QPageLayout(<QPageSize>, <Ориентация>, <QMarginsF>[,
                       units=Point][, minMargins=QMarginsF(0,0,0,0)]])
<Объект> = QPageLayout(<QPageLayout>)
```

Первый формат создает «пустой» объект, не хранящий никаких сведений. Второй формат позволяет задать сразу все необходимые сведения о бумаге. Назначение его параметров:

- размер бумаги в виде экземпляра класса `QPageSize`;
- ориентация бумаги в виде значения атрибута `Portrait` (`0`, портретная) или `Landscape` (`1`, ландшафтная) класса `QPageLayout`;
- размеры отступов от краев страницы в виде экземпляра класса `QMarginsF`;
- `units` — единицы измерения размеров в виде значения одного из атрибутов класса `QPageLayout` (они были приведены в описании метода `setPageMargins()` — см. *разд. 29.1.1*);
- `minMargins` — минимальные отступы от краев страницы, которые может соблюсти принтер, в виде экземпляра класса `QMarginsF`:

```
layout = QtGui.QPageLayout(QtGui.QPageSize(QtGui.QPageSize.A5),
                           QtGui.QPageLayout.Landscape,
                           QtCore.QMarginsF(10, 10, 10, 10),
                           units = QtGui.QPageLayout.Millimeter)
pl.printer.setPageLayout(layout)
```

Третий формат создает копию переданного в качестве параметра экземпляра класса `QPageLayout`.

Наиболее полезные для нас методы, поддерживаемые классом `QPageLayout`, приведены далее (полный их список можно найти на странице **https://doc.qt.io/qt-5/qpagelayout.html**).

- `setPageSize(<QPageSize>[, minMargins=QMarginsF(0, 0, 0, 0)])` — задает размеры бумаги в виде экземпляра класса `QPageSize`. В необязательном параметре `minMargins` можно указать минимальные величины отступов от краев страницы (экземпляр класса `QMarginsF`);
- `setOrientation(<Ориентация>)` — задает ориентацию страницы в виде значения атрибута `Portrait` (`0`, портретная) или `Landscape` (`1`, ландшафтная) класса `QPageLayout`;
- `setMargins(<QMarginsF>)` — задает величины отступов от краев страницы в виде экземпляра класса `QMarginsF`. Возвращает `True`, если операция прошла успешно, и `False` — в противном случае;
- `setLeftMargin(<Отступ>)`, `setTopMargin(<Отступ>)`, `setRightMargin(<Отступ>)` и `setBottomMargin(<Отступ>)` — задают величины отступов от левого, верхнего, правого и нижнего края страницы соответственно. Возвращают `True`, если операция прошла успешно, и `False` — в противном случае;
- `setUnits(<Единица измерения>)` — задает единицу измерения размеров в виде значения одного из атрибутов класса `QPageLayout` (они были приведены в описании метода `setPageMargins()` — см. *разд. 29.1.1*);
- `setMinimumMargins(<QMarginsF>)` — задает минимальные величины отступов от краев страницы в виде экземпляра класса `QMarginsF`;

- `swap(<QPageLayout>)` — меняет параметры бумаги, хранящиеся в текущем экземпляре класса, на заданные в параметре;
- `fullRect([<Единица измерения>])` — возвращает размеры страницы с учетом ориентации, но без учета отступов в виде экземпляра класса `QRectF`. Если единица измерения не указана, выведенные значения будут исчисляться в текущей единице измерения, заданной в конструкторе:

  ```
  layout = QtGui.QPageLayout(QtGui.QPageSize(QtGui.QPageSize.A5), ↳
  QtGui.QPageLayout.Landscape, QtCore.QMarginsF(10, 10, 10, 10), ↳
  units = QtGui.QPageLayout.Millimeter)
  r = layout.fullRect()
  print(r.width(), "x", r.height())
  ```

  Выведет: `210.0 x 148.0`
- `fullRectPixels(<Разрешение>)` — возвращает размеры страницы в пикселах с учетом ориентации, но без учета отступов для заданного разрешения, которое измеряется в точках на дюйм. Результат представляет собой экземпляр класса `QRect`;
- `fullRectPoints()` — возвращает размеры страницы в пунктах с учетом ориентации, но без учета отступов в виде экземпляра класса `QRect`;
- `paintRect([<Единица измерения>])` — возвращает размеры страницы с учетом ориентации и отступов (фактически — доступной для вывода графики области страницы) в виде экземпляра класса `QRectF`. Если единица измерения не указана, выведенные значения будут исчисляться в текущей единице измерения, заданной в конструкторе;
- `paintRectPixels(<Разрешение>)` — возвращает размеры страницы в пикселах с учетом ориентации и отступов для заданного разрешения, которое измеряется в точках на дюйм. Результат представляет собой экземпляр класса `QRect`;
- `paintRectPoints()` — возвращает размеры страницы в пунктах с учетом ориентации и отступов в виде экземпляра класса `QRect`.

Класс `QPageLayout` также поддерживает операторы сравнения == и `!=`.

29.2. Задание параметров принтера и страницы

Все профессионально выполненные приложения предоставляют пользователю возможность указать параметры принтера (выбрать собственно принтер из установленных в системе, задать число копий печатаемого документа и пр.) и страницы (указать размер бумаги, ориентацию и др.). Сейчас мы выясним, как это делается средствами PyQt 5.

29.2.1. Класс *QPrintDialog*

Класс `QPrintDialog` реализует функциональность стандартного диалогового окна выбора и настройки принтера, которое позволяет выбрать принтер, задать количество копий, диапазон печатаемых страниц и некоторые другие параметры (рис. 29.1).

Иерархия наследования этого класса:

```
(QObject, QPaintDevice) — QWidget — QDialog — QAbstractPrintDialog –
                                              QPrintDialog
```

Конструктор класса `QPrintDialog` имеет следующий формат:

```
<Объект> = QPrintDialog(<QPrinter>[, parent=None])
```

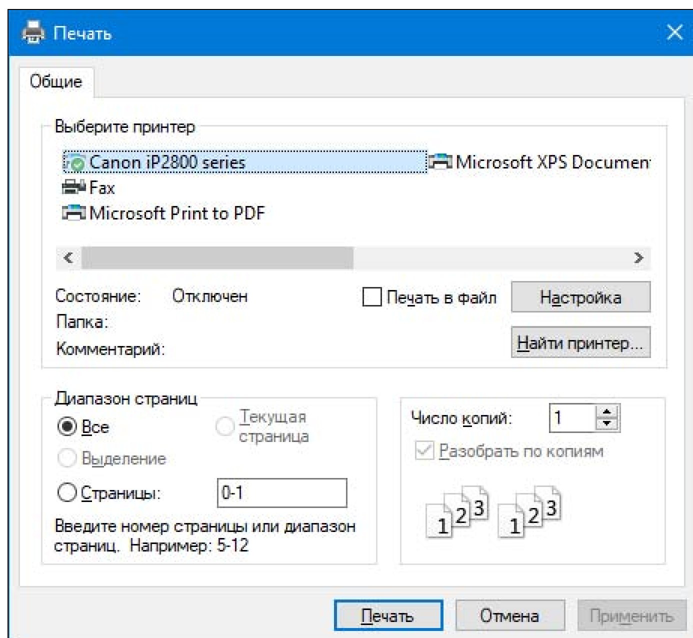

Рис. 29.1. Стандартное диалоговое окно выбора и настройки принтера

В первом параметре указывается принтер, настройки которого будут задаваться в диалоговом окне. Необязательный параметр `parent` может быть использован для задания родителя.

Перед выводом диалогового окна настройки принтера можно указать для него значения по умолчанию, воспользовавшись методами класса `QPrinter`, описанными в *разд. 29.1.1*. Например, для указания размера бумаги следует вызвать метод `setPageSize()`, для задания количества копий — `setCopyCount()` и т. д.

Вывести диалоговое окно на экран можно вызовом методов `exec()` или `exec_()`, унаследованных от класса `QDialog`.

После закрытия диалогового окна все заданные в нем параметры при необходимости могут быть получены через соответствующие методы класса `QPrinter`. Так, выяснить размер бумаги, ее ориентацию и отступы от краев позволит метод `pageLayout()`, количество копий документа — метод `copyCount()` и т. д. Все эти методы были описаны в *разд. 29.1.1*.

Впрочем, надо сказать, что все настройки, заданные в этом диалоговом окне, будут применены к принтеру самой библиотекой PyQt. Так, если пользователь выберет другой принтер, печать будет выполнена на выбранном им принтере. А если он укажет напечатать документ в нескольких копиях, все эти копии будут напечатаны PyQt самостоятельно. Никакого кода нам самим для этого писать не придется.

Класс `QPrintDialog` поддерживает следующие основные методы (полный их список можно найти на страницах **https://doc.qt.io/qt-5/qabstractprintdialog.html** и **https://doc.qt.io/qt-5/qprintdialog.html**):

- `setOption(<Настройка принтера>[, on=True])` — активизирует указанную в первом параметре настройку принтера, если в параметре `on` задано значение `True`, и сбрасывает, если передано `False`. В первом параметре указывается одно из значений следующих атрибутов класса `QAbstractPrintDialog` или их комбинация через оператор `|`:

- `None` — `0` — все настройки принтера сброшены;
- `PrintToFile` — `1` — печать в файл;
- `PrintSelection` — `2` — выбор принтера;
- `PrintPageRange` — `4` — указание диапазона печатаемых страниц;
- `PrintShowPageSize` — `8` — указание размера страницы;
- `PrintCollateCopies` — `16` — указание режима печати копий документов (будет ли каждая копия печататься полностью, или сначала будут отпечатаны все копии первой страницы, потом — копии второй и т. д.);
- `PrintCurrentPage` — `64` — указание печати только текущей страницы;

- ♦ setOptions(<Настройки принтера>) — позволяет активизировать сразу несколько настроек принтера;
- ♦ testOption(<Настройка принтера>) — возвращает `True`, если заданная настройка принтера активизирована, и `False` — в противном случае;
- ♦ `options()` — возвращает комбинацию значений атрибутов класса `QAbstractPrintDialog` через оператор |, которые представляют активизированные настройки;
- ♦ `printer()` — возвращает принтер (экземпляр класса `QPrinter`), выбранный в диалоговом окне.

29.2.2. Класс *QPageSetupDialog*

Класс `QPageSetupDialog` реализует работу стандартного диалогового окна установки параметров страницы: размера, ориентации, отступов и др. Воочию его можно увидеть на рис. 29.2.

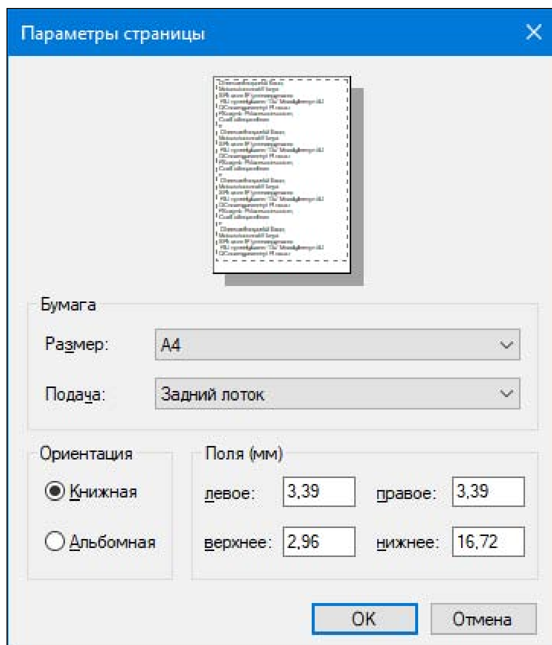

Рис. 29.2. Стандартное диалоговое окно установки параметров страницы

Иерархия наследования этого класса:

```
(QObject, QPaintDevice) — QWidget — QDialog - QPageSetupDialog
```

Конструктор класса `QPageSetupDialog` имеет следующий формат:

```
<Объект> = QPageSetupDialog([parent=None])
<Объект> = QPageSetupDialog(<QPrinter>[, parent=None])
```

Первый формат создает диалоговое окно, привязанное к используемому по умолчанию принтеру, второй формат позволяет указать нужный нам принтер в виде экземпляра класса `QPrinter`. Необязательный параметр `parent` может быть использован для задания родителя.

Принципы работы с диалоговым окном параметров страницы такие же, как и у его «коллеги», представляющего параметры принтера (см. *разд. 29.2.1*): мы задаем нужные параметры страницы, вызывая соответствующие методы класса `QPrinter`, и выводим диалоговое окно на экран вызовом метода `exec()` или `exec_()`. И, опять же, все заданные в этом диалоговом окне настройки будут применены к принтеру автоматически.

Из поддерживаемых классом `QPageSetupDialog` методов нас может заинтересовать разве что `printer()`. Он возвращает принтер (экземпляр класса `QPrinter`), указанный в вызове конструктора.

Практиковаться мы станем на утилите печати изображений, которую сейчас же и напишем. Ее интерфейс очень прост и включает лишь кнопки открытия файла, вывода диалоговых окон настройки принтера и параметров страницы и собственно печати (листинг 29.4).

Листинг 29.4. Использование классов `QPrintDialog` и `QPageSetupDialog`

```
from PyQt5 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
        QtWidgets.QWidget.__init__(self, parent, flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Печать изображений")
        self.printer = QtPrintSupport.QPrinter()
        self.printer.setPageOrientation(QtGui.QPageLayout.Landscape)
        self.file = None
        vbox = QtWidgets.QVBoxLayout()
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
        btnPageOptions = QtWidgets.QPushButton("Настройка &страницы...")
        btnPageOptions.clicked.connect(self.showPageOptions)
        vbox.addWidget(btnPageOptions)
        btnPrint = QtWidgets.QPushButton("&Печать...")
        btnPrint.clicked.connect(self.print)
        vbox.addWidget(btnPrint)
        self.setLayout(vbox)
        self.resize(200, 100)
```

```
    def openFile(self):
        self.file = QtWidgets.QFileDialog.getOpenFileName(parent=self,
            caption = "Выберите графический файл",
            filter = "Графические файлы (*.bmp *.jpg *.png)")[0]

    def showPageOptions(self):
        pd = QtPrintSupport.QPageSetupDialog(self.printer, parent=self)
        pd.exec()

    def print(self):
        pd = QtPrintSupport.QPrintDialog(self.printer, parent=self)
        pd.setOptions(QtPrintSupport.QAbstractPrintDialog.PrintToFile |
                      QtPrintSupport.QAbstractPrintDialog.PrintSelection)
        if pd.exec() == QtWidgets.QDialog.Accepted:
            painter = QtGui.QPainter()
            painter.begin(self.printer)
            pixmap = QtGui.QPixmap(self.file)
            pixmap = pixmap.scaled(self.printer.width(),
                                   self.printer.height(),
                                   aspectRatioMode=QtCore.Qt.KeepAspectRatio)
            painter.drawPixmap(0, 0, pixmap)
            painter.end()

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

29.3. Предварительный просмотр документов перед печатью

Еще одна возможность, предоставляемая серьезными приложениями, — предварительный просмотр документа перед собственно печатью. В PyQt 5 реализовать это проще простого.

29.3.1. Класс *QPrintPreviewDialog*

Класс `QPrintPreviewDialog` выводит документ в стандартном диалоговом окне предварительного просмотра (рис. 29.3).

Иерархия наследования этого класса:

```
(QObject, QPaintDevice) — QWidget — QDialog - QPrintPreviewDialog
```

Конструктор класса `QPrintPreviewDialog` имеет следующий формат:

```
<Объект> = QPrintPreviewDialog([parent=None][, flags=0])
<Объект> = QPrintPreviewDialog(<QPrinter>[, parent=None][, flags=0])
```

Первый формат создает диалоговое окно, привязанное к используемому по умолчанию принтеру, второй формат позволяет указать нужный нам принтер в виде экземпляра класса `QPrinter`. Необязательный параметр `parent` может быть использован для задания родителя, а необязательный параметр `flags` — для установки типа окна (см. *разд. 18.2*).

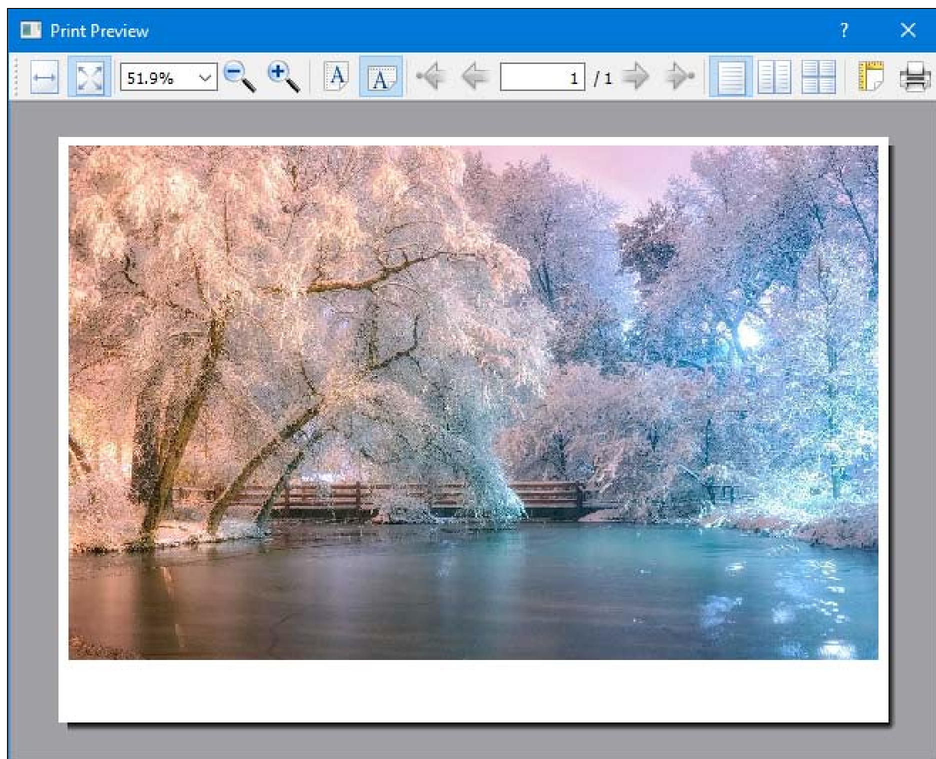

Рис. 29.3. Стандартное диалоговое окно предварительного просмотра документа

Вывод документа для предварительного просмотра с помощью класса `QPrintPreviewDialog` выполняется в три этапа:

1. Создание экземпляра класса `QPrintPreviewDialog`.
2. Назначение сигналу `paintRequested(<QPrinter>)` этого экземпляра обработчика, внутри которого и будет выполняться вывод документа. Как видим, обработчику сигнала в параметре передается указанный в вызове конструктора принтер, так что вывести документ мы сможем без труда.
3. Вывод диалогового окна на экран вызовом метода `exec()` или `exec_()` экземпляра класса `QPrintPreviewDialog`.

Переделаем утилиту печати изображений, чей код приведен в листинге 29.4, таким образом, чтобы дать пользователю возможность просматривать изображения перед печатью. Исправленный код можно увидеть в листинге 29.5 — поскольку переделки здесь довольно значительные, авторы привели его целиком.

Листинг 29.5. Использование класса `QPrintPreviewDialog`

```
from PyQt5 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent = None):
```

```
        QtWidgets.QWidget.__init__(self, parent, flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Печать изображений")
        self.printer = QtPrintSupport.QPrinter()
        self.printer.setPageOrientation(QtGui.QPageLayout.Landscape)
        self.file = None
        vbox = QtWidgets.QVBoxLayout()
        btnOpen = QtWidgets.QPushButton("&Открыть файл...")
        btnOpen.clicked.connect(self.openFile)
        vbox.addWidget(btnOpen)
        btnPageOptions = QtWidgets.QPushButton("Настройка &страницы...")
        btnPageOptions.clicked.connect(self.showPageOptions)
        vbox.addWidget(btnPageOptions)
        btnPreview = QtWidgets.QPushButton("П&росмотр...")
        btnPreview.clicked.connect(self.preview)
        vbox.addWidget(btnPreview)
        btnPrint = QtWidgets.QPushButton("&Печать...")
        btnPrint.clicked.connect(self.print)
        vbox.addWidget(btnPrint)
        self.setLayout(vbox)
        self.resize(200, 100)

    def openFile(self):
        self.file = QtWidgets.QFileDialog.getOpenFileName(parent=self,
            caption = "Выберите графический файл",
            filter = "Графические файлы (*.bmp *.jpg *.png)")[0]

    def showPageOptions(self):
        pd = QtPrintSupport.QPageSetupDialog(self.printer, parent=self)
        pd.exec()

    def preview(self):
        pp = QtPrintSupport.QPrintPreviewDialog(self.printer, parent=self)
        pp.paintRequested.connect(self._printImage)
        pp.exec()
    def print(self):
        pd = QtPrintSupport.QPrintDialog(self.printer, parent=self)
        pd.setOptions(QtPrintSupport.QAbstractPrintDialog.PrintToFile |
                      QtPrintSupport.QAbstractPrintDialog.PrintSelection)
        if pd.exec() == QtWidgets.QDialog.Accepted:
            self._printImage(self.printer)

    def _printImage(self, printer):
        painter = QtGui.QPainter()
        painter.begin(printer)
        pixmap = QtGui.QPixmap(self.file)
        pixmap = pixmap.scaled(printer.width(), printer.height(),
                               aspectRatioMode=QtCore.Qt.KeepAspectRatio)
        painter.drawPixmap(0, 0, pixmap)
        painter.end()
```

```
app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

Как видно из рис. 29.3, диалоговое окно предварительного просмотра дает возможности перехода со страницы на страницу, масштабирования выведенного документа, указания режима его вывода (постранично, по две страницы и т. п.), задания ориентации страницы, вызова диалогового окна настроек страницы, а также отправки документа на печать.

29.3.2. Класс *QPrintPreviewWidget*

Класс `QPrintPreviewWidget` представляет отдельный компонент — панель для предварительного просмотра документа. Ее можно воспринимать как центральную часть рассмотренного нами в предыдущем разделе диалогового окна.

Иерархия наследования этого класса:

```
(QObject, QPaintDevice) — QWidget — QPrintPreviewWidget
```

Конструктор класса `QPrintPreviewWidget` имеет следующий формат:

```
<Объект> = QPrintPreviewWidget([parent=None][, flags=0])
<Объект> = QPrintPreviewWidget(<QPrinter>[, parent=None][, flags=0])
```

Первый формат создает компонент, привязанный к используемому по умолчанию принтеру, второй формат позволяет указать нужный нам принтер в виде экземпляра класса `QPrinter`. Необязательный параметр `parent` может быть использован для задания родителя, а необязательный параметр `flags` — для установки типа окна (см. *разд. 18.2*).

Последовательность действий, необходимых для реализации предварительного просмотра с помощью класса `QPrintPreviewWidget`, почти такая же, что и в случае класса `QPrintPreviewDialog`: мы создаем экземпляр класса `QPrintPreviewWidget`, назначаем для его сигнала `paintRequested()` обработчик и пишем в этом обработчике код, который и выполнит вывод документа.

Класс `QPrintPreviewWidget` поддерживает ряд методов, предназначенных для выполнения различных действий над выведенным в панели документом. Рассмотрим их:

- `setOrientation(<Ориентация>)` — задает ориентацию страницы в виде значения атрибута `Portrait` (`0`, портретная) или `Landscape` (`1`, ландшафтная) класса `QPrinter`. Метод является слотом;
- `setPortraitOrientation()` — задает портретную ориентацию страницы. Метод является слотом;
- `setLandscapeOrientation()` — задает ландшафтную ориентацию страницы. Метод является слотом;
- `orientation()` — возвращает ориентацию страницы;
- `setViewMode(<Режим просмотра>)` — задает режим просмотра страниц документа в виде значения одного из следующих атрибутов класса `QPrintPreviewWidget`:
  - `SinglePageView` — `0` — постраничный режим (одновременно выводится только одна страница);

- `FacingPagesView` — `1` — режим просмотра разворота (одновременно отображаются две страницы);
- `AllPagesView` — `2` — выводятся сразу все страницы документа.

Метод является слотом;

- `setSinglePageViewMode()` — задает постраничный режим просмотра страниц. Метод является слотом;
- `setFacingPagesViewMode()` — задает режим просмотра разворота. Метод является слотом;
- `setAllPagesViewMode()` — задает режим просмотра всех страниц документа. Метод является слотом;
- `viewMode()` — возвращает обозначение режима просмотра страниц;
- `setZoomMode(<Режим масштабирования>)` — задает режим масштабирования страниц при просмотре в виде значения одного из следующих атрибутов класса `QPrintPreviewWidget`:
  - `CustomZoom` — `0` — произвольное масштабирование со значением масштаба заданным методом `setZoomFactor()`;
  - `FitToWidth` — `1` — выбирается такое значение масштаба, чтобы страница помещалась в панели по ширине;
  - `FitInView` — `2` — выбирается такое значение масштаба, чтобы страница помещалась в панели полностью.

  Метод является слотом;
- `fitToWidth()` — задает такой режим масштабирования, чтобы страница помещалась в панели по ширине. Метод является слотом;
- `fitInView()` — задает такой режим масштабирования, чтобы страница помещалась в панели полностью. Метод является слотом;
- `zoomMode()` — возвращает обозначение режима масштабирования страниц;
- `setZoomFactor(<Масштаб>)` — задает значение масштаба для выводимого в панели документа в виде вещественного числа при условии, что указан режим масштабирования `CustomZoom`. Значение `1.0` задает изначальный масштаб, меньшие значения уменьшают его, а бóльшие — увеличивают. Метод является слотом;
- `zoomIn([<Масштаб>])` — задает новый масштаб документа, увеличивая его. Если масштаб не задан, будет использовано значение `1.1`. Метод является слотом;
- `zoomOut([<Масштаб>])` — задает новый масштаб документа, уменьшая его. Если масштаб не задан, будет использовано значение `1.1`. Метод является слотом;
- `zoomFactor()` — возвращает значение масштаба;
- `pageCount()` — возвращает общее количество страниц в документе;
- `setCurrentPage(<Номер страницы>)` — задает номер страницы, выводимой в панели. Метод является слотом;
- `currentPage()` — возвращает номер страницы, выводящейся в панели в данный момент;
- `updatePreview()` — выполняет обновление содержимого панели. При этом будет снова сгенерирован сигнал `paintRequested()`. Метод является слотом;

- `print()` — выполняет печать документа, отображающегося в панели, с выводом на экран диалогового окна выбора и настройки принтера. Метод является слотом.

В дополнение к `paintRequested()`, класс `QPrintPreviewWidget` поддерживает сигнал `previewChanged()`. Он генерируется при изменении параметров панели просмотра: ориентации страницы, режима просмотра, масштаба и др.

29.4. Класс *QPrinterInfo*: получение сведений о принтере

Очень полезный класс `QPrinterInfo` позволяет получить сведения как обо всех установленных в системе принтерах, так и о конкретном принтере, указанном нами. Форматы вызова его конструктора следующие:

```
<Объект> = QPrinterInfo(<QPrinter>)
<Объект> = QPrinterInfo(<QPrinterInfo>)
```

Второй формат создает копию экземпляра класса `QPrinterInfo`, переданного в параметре.

Класс `QPrinterInfo` поддерживает следующие методы:

- `availablePrinters()` — возвращает список всех установленных в системе принтеров, представленных экземплярами класса `QPrinterInfo`. Метод является статическим;
- `availablePrinterNames()` — возвращает список имен всех установленных в системе принтеров, заданных строками:

  ```
  l = QtPrintSupport.QPrinterInfo.availablePrinterNames()
  for p in l: print(p)
  ```

  Выведет:

  ```
  Microsoft XPS Document Writer
  Microsoft Print to PDF
  Fax
  Canon iP2800 series
  ```

  Метод является статическим;
- `defaultPrinter()` — возвращает сведения о принтере, используемом по умолчанию, в виде экземпляра класса `QPrinterInfo`. Метод является статическим;
- `defaultPrinterName()` — возвращает имя принтера, используемого по умолчанию:

  ```
  print(QtPrintSupport.QPrinterInfo.defaultPrinterName())
  ```

  Выведет: `Canon iP2800 series`

  Метод является статическим;
- `printerInfo(<Имя принтера>)` — возвращает экземпляр класса `QPrinterInfo`, хранящий сведения о принтере с указанным именем. Если такого принтера нет, возвращается некорректный, «пустой» экземпляр класса:

  ```
  p = QtPrintSupport.QPrinterInfo.printerInfo("Canon iP2800 series")
  ```

  Метод является статическим;
- `isNull()` — возвращает `True`, если экземпляр объекта указывает на конкретный установленный в системе принтер, и `False`, если он является «пустым»;

- `printerName()` — возвращает имя принтера;
- `makeAndModel()` — возвращает развернутое наименование принтера с указанием его изготовителя, если таковые сведения имеются в наличии, или просто имя принтера — в противном случае;
- `description()` — возвращает развернутое описание принтера, если таковое имеется в наличии, или просто имя принтера — в противном случае;
- `location()` — возвращает указание на местоположение принтера, если таковое имеется в наличии, или пустую строку — в противном случае;
- `isDefault()` — возвращает `True`, если принтер помечен как используемый по умолчанию, и `False` — в противном случае;
- `isRemote()` — возвращает `True`, если это сетевой принтер, и `False`, если он локальный;
- `state()` — возвращает текущее состояние принтера в виде значения одного из следующих атрибутов класса `QPrinter`:
  - `Idle` — `0` — простаивает;
  - `Active` — `1` — идет печать;
  - `Aborted` — `2` — печать была прервана;
  - `Error` — `3` — возникла ошибка;
- `supportedPageSizes()` — возвращает список поддерживаемых принтером размеров бумаги, представляемых экземплярами класса `QPageSize`;
- `supportsCustomPageSizes()` — возвращает `True`, если принтер поддерживает указание произвольного размера бумаги, и `False` — в противном случае;
- `supportedResolutions()` — возвращает список разрешений, поддерживаемых принтером и измеряемых в точках на дюйм;
- `supportedDuplexModes()` — возвращает список поддерживаемых принтером режимов двусторонней печати. Каждый элемент списка является значением одного из атрибутов класса `QPrinter`, приведенных в описании метода `setDuplex()` (см. *разд. 29.1.1*);
- `defaultPageSize()` — возвращает установленный по умолчанию размер бумаги в виде экземпляра класса `QPageSize`:

  ```
  ps = p.defaultPageSize()
  print(ps.name())
  ```

  Выведет: `A4`

  ```
  s = ps.size(QtGui.QPageSize.Millimeter)
  print(s.width(),"x", s.height())
  ```

  Выведет: `210.0 x 297.0`
- `defaultDuplexMode()` — возвращает заданный по умолчанию режим двусторонней печати;
- `minimumPhysicalPageSize()` и `maximumPhysicalPageSize()` — возвращают, соответственно, минимальный и максимальный размеры бумаги, поддерживаемые принтером и заданные в виде экземпляров класса `QPageSize`:

  ```
  ps = p.maximumPhysicalPageSize()
  print(ps.name())
  ```

Выведет: `Custom (1190pt x 1916pt)`

```
s = ps.size(QtGui.QPageSize.Millimeter)
print(s.width(),"x", s.height())
```

Выведет: `419.81 x 675.92`

29.5. Класс *QPdfWriter*: экспорт в формат PDF

В *разд. 29.1.1* упоминалось о возможности сохранения печатаемого документа в файл формата PDF — для этого достаточно вызвать метод `setOutputFileName()`, передав ему в качестве параметра путь к файлу и указав у этого файла расширение pdf. Однако при этом задействуется подсистема печати Windows, что приводит к напрасному расходу системных ресурсов.

Библиотека PyQt 5 предоставляет возможность экспорта документов в формат PDF напрямую, без использования подсистемы печати. Это обеспечивает класс `QPdfWriter`, объявленный в модуле `QtGui`. Его иерархия наследования:

```
QPaintDevice - (QObject, QPagedPaintDevice) - QPdfWriter
```

Формат конструктора этого класса:

```
<Объект> = QPdfWriter(<Путь к файлу>)
```

В качестве параметра указывается путь к файлу, в который будет экспортирован документ.

Вывод документа в формат PDF выполняется так же, как и его печать (см. *разд. 29.1.2*). Как и `QPrinter`, класс `QPdfWriter` поддерживает метод `newPage()`, подготавливающий следующую страницу документа для вывода.

Класс `QPdfWriter` поддерживает также и следующие методы, которые будут нам полезны (полный их список можно найти на странице **https://doc.qt.io/qt-5/qpdfwriter.html**):

- `setPageSize(<QPageSize>)` — задает размер страницы в виде экземпляра класса `QPageSize`. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setPageOrientation(<Ориентация>)` — задает ориентацию страницы в виде значения атрибута `Portrait` (`0`, портретная) или `Landscape` (`1`, ландшафтная) класса `QPageLayout`. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setPageMargins(<QMarginsF>[, <Единица измерения>])` — задает отступы от краев страницы в заданной единице измерения. Сами отступы указываются в виде экземпляра класса `QMarginsF`. Единица измерения указывается в виде значения одного из следующих атрибутов класса `QPageLayout`:
  - `Millimeter` — `0` — миллиметры;
  - `Point` — `1` — пункты;
  - `Inch` — `2` — дюймы;
  - `Pica` — `3` — пики;
  - `Didot` — `4` — дидо (0,375 мм);
  - `Cicero` — `5` — цицеро (4,5 мм).

  Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;

- `setPageLayout(<QPageLayout>)` — задает сразу все параметры страницы (размеры, ориентацию, отступы от краев страницы и т. п.) в виде экземпляра класса `QPageLayout`. Возвращает `True`, если операция увенчалась успехом, и `False` — в противном случае;
- `setResolution(<Разрешение>)` — задает разрешение в виде целого числа в точках на дюйм;
- `setCreator(<Имя создателя>)` — задает имя создателя документа, которое будет записано в файл PDF;
- `setTitle(<Заголовок>)` — задает заголовок документа, который будет записан в файл PDF.

В качестве практического занятия возьмем код, приведенный в листинге 29.1, и немного переделаем его: пусть на первой странице он выводит вместо надписи «QPrinter» строку «QPdfWriter», а в качестве формата бумаги устанавливает A5 (листинг 29.6).

Листинг 29.6. Использование класса `QPdfWriter`

```
from PyQt5 import QtCore, QtWidgets, QtGui, QtPrintSupport
import sys
app = QtWidgets.QApplication(sys.argv)
writer = QtGui.QPdfWriter("output.pdf")
writer.setCreator("Владимир Дронов")
writer.setTitle("Тест")
# Заодно поэкспериментируем с указанием параметров бумаги с помощью
# класса QPageLayout
layout = QtGui.QPageLayout()
layout.setPageSize(QtGui.QPageSize(QtGui.QPageSize.A5))
layout.setOrientation(QtGui.QPageLayout.Portrait)
writer.setPageLayout(layout)
painter = QtGui.QPainter()
painter.begin(writer)
color = QtGui.QColor(QtCore.Qt.black)
painter.setPen(QtGui.QPen(color))
painter.setBrush(QtGui.QBrush(color))
font = QtGui.QFont("Verdana", pointSize=42)
painter.setFont(font)
painter.drawText(10, writer.height() // 2 - 50, writer.width() - 20,
        50, QtCore.Qt.AlignCenter | QtCore.Qt.TextDontClip, "QPdfWriter")
layout.setOrientation(QtGui.QPageLayout.Landscape)
writer.setPageLayout(layout)
writer.newPage()
pixmap = QtGui.QPixmap("img.jpg")
pixmap = pixmap.scaled(writer.width(), writer.height(),
                       aspectRatioMode=QtCore.Qt.KeepAspectRatio)
painter.drawPixmap(0, 0, pixmap)
painter.end()
```

У класса `QPdfWriter` есть недостаток — создаваемые с его помощью PDF-документы имеют очень большой размер. Так, у авторов после выполнения приведенного в листинге 29.6 кода получился файл объемом в 4,32 Мбайт. Вероятно, PyQt 5 создает документы максимально доступного в формате PDF качества, и понизить его, тем самым уменьшив размер результирующих файлов, мы не можем.

ГЛАВА 30

Взаимодействие с Windows

PyQt 5 предоставляет весьма богатые средства для взаимодействия с Windows. Мы можем задать значок, отображающийся на кнопке панели задач и перекрывающий значок приложения, создать у кнопки наложенный индикатор процесса выполнения, работать со списками быстрого доступа, формировать инструментальную панель на миниатюре, что возникает при наведении курсора на кнопку панели задач. Все это позволит придать нашим PyQt-приложениям подобающий им лоск.

Все описанные в этой главе классы определены в модуле `QtWinExtras`, если не указано иное.

30.1. Управление кнопкой в панели задач

Сначала рассмотрим инструменты для управления кнопкой в панели задач, которой представляется запущенное приложение: смены выводящегося на ней значка и создания наложенного индикатора процесса.

30.1.1. Класс *QWinTaskbarButton*

Класс `QWinTaskbarButton` создает кнопку, которая представляет текущее приложение на стандартной панели задач. Формат его конструктора:

```
<Объект> = QWinTaskbarButton([parent=None])
```

В параметре `parent` можно указать родителя.

Класс `QWinTaskbarButton` поддерживает следующие методы:

- `setWindow(<QWindow>)` — задает окно (экземпляр класса `QWindow`), которое должна представлять кнопка.

  Получить экземпляр упомянутого ранее класса, представляющий окно, можно вызовом метода `windowHandle()` класса `QWidget`. Однако здесь нужно иметь в виду, что этот метод способен вернуть корректное значение лишь после того, как окно будет выведено на экран. Поэтому метод `setWindow` кнопки панели задач следует вызывать внутри переопределенного метода `showEvent()` (за подробностями — к *разд. 19.7.1*):

```
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        . . .
```

```
        self.taskbarButton = QtWinExtras.QWinTaskbarButton(parent=self)
    . . .
    def showEvent(self, evt):
        self.taskbarButton.setWindow(self.windowHandle())
```

- `window()` — возвращает экземпляр класса `QWindow`, представляющий окно, с которым связана кнопка;
- `setOverlayIcon(<QIcon>)` — задает новый значок (экземпляр класса `QIcon`) для кнопки:

  ```
  taskbarButton = QtWinExtras.QWinTaskbarButton(window)
  . . .
  taskbarButton.setOverlayIcon(QtGui.QIcon("taskbar_icon.png"))
  ```

  Метод является слотом.

  Еще раз отметим, что заданный с применением этого метода значок будет выводиться на кнопке поверх стандартного значка приложения и перекрывать его. Обычно такие значки применяются для обозначения состояния, в котором находится приложение;
- `clearOverlayIcon()` — убирает заданный ранее перекрывающий значок. Метод является слотом;
- `overlayIcon()` — возвращает экземпляр класса `QIcon`, представляющий заданный для кнопки перекрывающий значок;
- `setOverlayAccessibleDescription(<Описание>)` — указывает описание для перекрывающего значка, заданное в виде строки. Метод является слотом;
- `overlayAccessibleDescription()` — возвращает описание перекрывающего значка;
- `progress()` — возвращает наложенный индикатор процесса (экземпляр класса `QWinTaskbarProgress`), заданный для кнопки.

Листинг 30.1 представляет код, создающий окно с тремя кнопками, две из которых задают для кнопки панели задач разные перекрывающие значки, а третья убирает значок.

Листинг 30.1. Использование класса `QWinTaskbarButton`

```
from PyQt5 import QtCore, QtWidgets, QtGui, QtWinExtras
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                          flags=QtCore.Qt.Window |
                          QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Класс QWinTaskbarButton")
        # Создаем оба значка
        self.icon1 = QtGui.QIcon("icon1.png")
        self.icon2 = QtGui.QIcon("icon2.png")
        # Создаем кнопку панели задач
        self.taskbarButton = QtWinExtras.QWinTaskbarButton(parent=self)
        # Создаем все три кнопки
        hbox = QtWidgets.QHBoxLayout()
        btnIcon1 = QtWidgets.QPushButton(self.icon1, "Значок &1")
        btnIcon1.clicked.connect(self.setIcon1)
```

```
        hbox.addWidget(btnIcon1)
        btnIcon2 = QtWidgets.QPushButton(self.icon2, "Значок &2")
        btnIcon2.clicked.connect(self.setIcon2)
        hbox.addWidget(btnIcon2)
        btnClearIcon = QtWidgets.QPushButton("&Убрать значок")
        btnClearIcon.clicked.connect(self.taskbarButton.clearOverlayIcon)
        hbox.addWidget(btnClearIcon)
        self.setLayout(hbox)
        self.resize(200, 50)

    # После вывода окна на экран привязываем его к кнопке панели задач
    def showEvent(self, evt):
        self.taskbarButton.setWindow(self.windowHandle())

    # Выполняем задание новых значков для кнопки панели задач при нажатии
    # кнопок
    def setIcon1(self):
        self.taskbarButton.setOverlayIcon(self.icon1)

    def setIcon2(self):
        self.taskbarButton.setOverlayIcon(self.icon2)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

30.1.2. Класс *QWinTaskbarProgress*

Класс `QWinTaskbarProgress` представляет индикатор процесса, выводящийся на кнопке панели задач поверх всех имеющихся на ней значков — как стандартного, так и перекрывающего.

Создавать экземпляр этого класса вызовом его конструктора нам не придется. Любая кнопка панели задач уже имеет такой индикатор, изначально невидимый. Получить его можно вызовом метода `progress()` класса `QWinTaskbarButton`.

Класс `QWinTaskbarProgress` поддерживает следующие методы:

- `setVisible(<Флаг>)` — если передать методу с параметром значение `True`, индикатор появится на экране. Значение `False` скрывает индикатор. Метод является слотом;
- `show()` — выводит индикатор на экран. Метод является слотом;
- `hide()` — скрывает индикатор. Метод является слотом;
- `isVisible()` — возвращает `True`, если индикатор присутствует на экране, и `False`, если он скрыт;
- `setRange(<Минимум>, <Максимум>)` — задает минимальное и максимальное значения для воображаемой шкалы индикатора, указанные в виде целых чисел. Если оба значения равны `0`, индикатор будет находиться в так называемом неопределенном состоянии, и по нему будет бежать бесконечная зеленая волна. Метод является слотом;

- setMinimum(<Минимум>) и setMaximum(<Максимум>) — задают, соответственно, минимальное и максимальное значения для воображаемой шкалы индикатора. Оба метода являются слотами;
- minimum() и maximum() — возвращают, соответственно, минимальное и максимальное значения воображаемой шкалы индикатора;
- setValue(<Значение>) — указывает значение, отображаемое индикатором в текущий момент. Метод является слотом;
- value() — возвращает текущее значение, отображаемое индикатором;
- pause() — ставит индикатор на паузу. Индикатор, поставленный на паузу, выводится желтым цветом. Метод является слотом;
- stop() — останавливает индикатор. Остановленный индикатор выводится красным цветом. Метод является слотом;
- resume() — вновь запускает поставленный на паузу или остановленный индикатор. Работающий индикатор выводится зеленым цветом. Метод является слотом;
- setPaused(<Флаг>) — если передать методу с параметром значение True, индикатор будет поставлен на паузу, если False — вновь запущен. Метод является слотом;
- reset() — сбрасывает индикатор, перемотав его на начальное значение воображаемой шкалы. Метод является слотом;
- isPaused() — возвращает True, если индикатор в текущий момент поставлен на паузу, и False — в противном случае;
- isStopped() — возвращает True, если индикатор в текущий момент остановлен, и False — в противном случае.

Теперь рассмотрим сигналы, поддерживаемые классом QWinTaskbarProgress:

- visibilityChanged(<Новое состояние>) — генерируется при выводе индикатора на экран или его скрытии. В параметре передается величина True, если индикатор выведен на экран, и False, если он скрыт;
- valueChanged(<Новое значение>) — генерируется при изменении значения, отображаемого индикатором. В параметре обработчику передается новое значение;
- minimumChanged(<Новое минимальное значение>) и maximumChanged(<Новое максимальное значение>) — генерируются при изменении, соответственно, минимального и максимального значения у воображаемой шкалы индикатора. В параметре обработчику передается новое значение.

Пример приложения, использующего индикатор на кнопке панели задач, приведен в листинге 30.2. Это приложение при нажатии кнопки **Пуск** последовательно, с секундным промежутком увеличивает значение индикатора от 0 до 10, позволяя ставить его на паузу, останавливать и запускать вновь.

Листинг 30.2. Использование класса QWinTaskbarProgress

```
from PyQt5 import QtCore, QtWidgets, QtWinExtras
import sys, time

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
```

```
        QtWidgets.QWidget.__init__(self, parent,
                          flags=QtCore.Qt.Window |
                          QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Класс QWinTaskbarProgress")
        # Создаем кнопку панели задач
        self.taskbarButton = QtWinExtras.QWinTaskbarButton(parent=self)
        # Получаем индикатор процесса, задаем его параметры
        # и делаем его видимым
        self.progress = self.taskbarButton.progress()
        self.progress.setRange(0, 10)
        self.progress.show()
        # Создаем необходимые кнопки
        vbox = QtWidgets.QVBoxLayout()
        btnStart = QtWidgets.QPushButton("&Пуск")
        btnStart.clicked.connect(self.start)
        vbox.addWidget(btnStart)
        btnPause = QtWidgets.QPushButton("Па&уза")
        btnPause.clicked.connect(self.progress.pause)
        vbox.addWidget(btnPause)
        btnStop = QtWidgets.QPushButton("&Стоп")
        btnStop.clicked.connect(self.progress.stop)
        vbox.addWidget(btnStop)
        btnResume = QtWidgets.QPushButton("П&родолжить")
        btnResume.clicked.connect(self.progress.resume)
        vbox.addWidget(btnResume)
        self.setLayout(vbox)
        self.resize(100, 100)

    # После вывода окна на экран привязываем его к кнопке панели задач
    def showEvent(self, evt):
        self.taskbarButton.setWindow(self.windowHandle())

    # При нажатии кнопки "Пуск" последовательно, с секундным промежутком
    # увеличиваем значение индикатора от 0 до 10, конечно, если индикатор
    # не поставлен на паузу и не остановлен
    def start(self):
        i = 0
        while i < 11:
            if not self.progress.isPaused() and ↳
            not self.progress.isStopped():
                self.progress.setValue(i)
                i += 1
            time.sleep(1)
            QtWidgets.qApp.processEvents()
        self.progress.reset()

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

Рис. 30.1 показывает три кнопки панели задач, принадлежащие трем разным копиям приложения, чей код представлен в листинге 30.2. Каждая кнопка имеет свой индикатор процесса: активный (показывающий процесс выполнения какой-либо операции), приостановленный и остановленный.

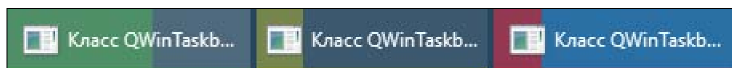

Рис. 30.1. Индикаторы процесса (в порядке слева направо): активный, приостановленный и остановленный

30.2. Списки быстрого доступа

Список быстрого доступа появляется при щелчке на кнопке панели задач правой кнопкой мыши и показывает типовые действия, которые можно выполнить с представляемым кнопкой приложением: закрепление на панели задач, закрытие и, наконец, активизация окна приложения. Однако мы можем программно добавить в этот список свои собственные пункты, предназначенные для открытия каких-либо избранных файлов или запуска каких-либо приложений.

Для работы со списками быстрого доступа PyQt предлагает три класса, один из которых представляет сам список, другой — отдельную категорию, к которой относятся те или иные его пункты, а третий — отдельный пункт списка.

30.2.1. Класс *QWinJumpList*

Класс `QWinJumpList` представляет сам список быстрого доступа. Конструктор этого класса имеет формат:

```
<Объект> = QWinJumpList([parent=None])
```

В параметре `parent` можно указать родителя.

Класс `QWinJumpList` поддерживает такие методы:

- `recent()` — возвращает стандартную категорию (экземпляр класса `QWinJumpListCategory`) недавно открывавшихся документов;
- `frequent()` — возвращает стандартную категорию (экземпляр класса `QWinJumpListCategory`) часто открываемых документов;
- `tasks()` — возвращает стандартную категорию (экземпляр класса `QWinJumpListCategory`) задач;
- `addCategory()` — добавляет новую произвольную (т. е. создаваемую самим разработчиком приложения) категорию в список быстрого доступа. Форматы метода:

  ```
  addCategory(<QWinJumpListCategory>)
  addCategory(<Заголовок>[, items=[]])
  ```

  Первый формат добавляет категорию, представленную экземпляром класса `QWinJumpListCategory`.

  Второй формат создает категорию с указанным `<Заголовком>` и пунктами, указанными в списке, который передан с параметром `items`, — эти пункты должны быть представлены экземплярами класса `QWinJumpListItem`. Если параметр `items` не указан, создается пустая

категория. Созданная категория добавляется в список. В качестве результата метод возвращает экземпляр класса `QWinJumpListCategory`, представляющий эту категорию;

- `categories()` — возвращает список экземпляров класса `QWinJumpListCategory`, представляющий произвольные категории, которые создаются самим разработчиком приложения;
- `clear()` — очищает список быстрого доступа. Метод является слотом;
- `setIdentifier(<Идентификатор>)` — задает для списка быстрого доступа уникальный идентификатор, указываемый в виде строки, которая не должна включать пробелы и превышать в длину 128 символов. Если указана пустая строка (это значение по умолчанию), будет использован идентификатор, сгенерированный самой системой;
- `identifier()` — возвращает уникальный идентификатор, заданный для списка быстрого доступа.

30.2.2. Класс *QWinJumpListCategory*

Класс `QWinJumpListCategory` представляет категорию списка быстрого доступа. Формат его конструктора:

```
<Объект> = QWinJumpListCategory([<Заголовок>])
```

В параметре можно сразу же указать заголовок создаваемой категории.

Класс `QWinJumpListCategory` поддерживает следующие методы:

- `addItem(<QWinJumpListItem>)` — добавляет в категорию пункт, представленный экземпляром класса `QWinJumpListItem`;
- `addDestination(<Путь к файлу>)` — добавляет в категорию пункт, указывающий на файл, чей `<Путь>` передан с параметром. В качестве названия пункта использует имя файла из этого параметра. Возвращает экземпляр класса `QWinJumpListItem`, представляющий созданный пункт. Этот метод используется для создания пунктов, открывающих какие-либо документы в текущем приложении;

  > **Внимание!**
  >
  > Чтобы пункты такого типа отображались в списке быстрого доступа, следует зарегистрировать их расширение, как обрабатываемое этим приложением.

- `addLink()` — добавляет в категорию пункт с заданным параметром `<Заголовок>`, указывающий на приложение, `<Путь>` к исполняемому файлу которого передан с параметром. Форматы метода:

  ```
  addLink(<Заголовок>, <Путь к файлу>[, arguments=[]])
  addLink(<QIcon>, <Заголовок>, <Путь к файлу>[, arguments=[]])
  ```

  Второй формат позволяет дополнительно указать значок, представленный экземпляром класса `QIcon`, который будет помечать создаваемый пункт.

  Дополнительно в параметре `arguments` можно указать список строковых параметров командной строки, которые будут переданы исполняемому файлу, указанному в параметре `<Путь к файлу>`, при его запуске.

  Метод возвращает экземпляр класса `QWinJumpListItem`, представляющий созданный пункт;

- `addSeparator()` — добавляет в категорию разделитель и возвращает представляющий его экземпляр класса `QWinJumpListItem`. Добавлять разделители можно только в категорию задач;
- `clear()` — очищает категорию;
- `items()` — возвращает список пунктов (экземпляров класса `QWinJumpListItem`), имеющихся в категории;
- `count()` — возвращает количество пунктов, содержащихся в категории;
- `isEmpty()` — возвращает `True`, если категория пуста, т. е. не содержит пунктов, и `False` — в противном случае;
- `setTitle(<Заголовок>)` — задает заголовок категории;
- `title()` — возвращает заголовок категории;
- `setVisible(<Флаг>)` — если передать значение `False`, категория не будет выводиться на экран. Чтобы вывести ее, нужно передать значение `True`;

> **Внимание!**
>
> Рекомендуется явно делать видимой каждую категорию, в которую будут программно добавляться пункты.

- `isVisible()` — возвращает `True`, если категория выводится на экран, и `False`, если она скрыта;
- `type()` — возвращает тип категории в виде одного из следующих атрибутов класса `QWinJumpListCategory`:
  - `Recent` — `1` — стандартная категория недавно открывавшихся документов;
  - `Frequent` — `2` — стандартная категория часто открываемых документов;
  - `Tasks` — `3` — стандартная категория задач;
  - `Custom` — `0` — произвольная, т. е. созданная самим разработчиком, категория.

30.2.3. Класс *QWinJumpListItem*

Класс `QWinJumpListItem` представляет отдельный пункт в списке быстрого доступа. Формат его конструктора:

```
<Объект> = QWinJumpListItem(<Тип>)
```

Единственным параметром методу передается тип создаваемого пункта в виде одного из следующих атрибутов класса `QWinJumpListItem`:

- `Destination` — `0` — пункт, указывающий на файл с документом (пункт такого типа также можно создать вызовом метода `addDestination()` класса `QWinJumpListCategory`);
- `Link` — `1` — пункт, указывающий на приложение (однотипный пункт создается вызовом метода `addLink()` класса `QWinJumpListCategory`);
- `Separator` — `2` — разделитель.

Класс `QWinJumpListItem` поддерживает следующие методы:

- `setFilePath(<Путь к файлу>)` — задает для пункта в параметре `<Путь к файлу>`: либо путь к документу (для пункта с типом `Destination`), либо путь к исполняемому файлу (для пункта типа `Link`);

- `filePath()` — возвращает путь к файлу, заданный для пункта;
- `setTitle(<Название>)` — задает `<Название>` для пункта с типом `Link`;
- `title()` — возвращает название пункта с типом `Link`;
- `setIcon(<QIcon>)` — задает значок (экземпляр класса `QIcon`) для пункта с типом `Link`;
- `icon()` — возвращает значок, представленный экземпляром класса `QIcon`, который был задан у пункта с типом `Link`;
- `setArguments(<Список аргументов>)` — задает аргументы командной строки, указываемые в виде списка строк, для пункта с типом `Link`;
- `arguments()` — возвращает список аргументов командной строки, указанных для пункта с типом `Link`;
- `setWorkingDirectory(<Путь к каталогу>)` — задает путь к рабочему каталогу для пункта с типом `Link`;
- `workingDirectory()` — возвращает путь к рабочему каталогу у пункта с типом `Link`;
- `setDescription(<Описание>)` — задает текстовое `<Описание>` для пункта с типом `Link`;
- `description()` — возвращает текстовое описание пункта с типом `Link`;
- `setType(<Тип>)` — устанавливает другой `<Тип>` для пункта. В качестве параметра `<Тип>` передается один из указанных ранее атрибутов класса `QWinJumpListItem`;
- `type()` — возвращает обозначение типа пункта.

Листинг 30.3 показывает код программы, которая формирует список быстрого доступа и добавляет в него следующие пункты:

- в категории последних открывавшихся документов — пункт, указывающий на файл с текстом лицензии Python;
- в категории часто открываемых документов — пункт, указывающий на файл со списком нововведений, появившихся в разных версиях Python;
- в категории задач — пункты, запускающие стандартные программы Блокнот и Write и отделенные друг от друга разделителем;
- во вновь созданной произвольной категории **Инструменты** — пункт, запускающий среду **Python Shell**.

Список быстрого доступа, созданный программой, показан на рис. 30.2.

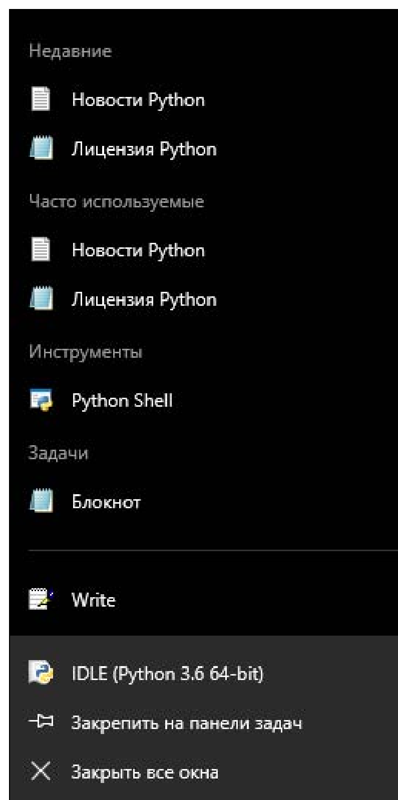

Рис. 30.2. Список быстрого доступа

Листинг 30.3. Формирование списка быстрого доступа

```
from PyQt5 import QtCore, QtWidgets, QtWinExtras
import sys, time

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                                   flags=QtCore.Qt.Window |
                                   QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Список быстрого доступа")
        # Создаем сам список быстрого доступа и очищаем его
        jumpList = QtWinExtras.QWinJumpList(parent=self)
        jumpList.clear()
        # Получаем категорию последних открывавшихся документоов
        recent = jumpList.recent()
        # Создаем пункт "Лицензия Python" и добавляем его в эту категорию
        item1 = QtWinExtras.QWinJumpListItem(
            QtWinExtras.QWinJumpListItem.Link)
        item1.setFilePath(r"C:\Windows\notepad.exe")
        item1.setTitle("Лицензия Python")
        item1.setArguments([r"C:/Python36/LICENSE.txt"])
        recent.addItem(item1)
        # Делаем категорию видимой
        recent.setVisible(True)
        # Получаем категорию часто открываемых документов, добавляем в
        # нее пункт "Новости Python" и делаем видимой
        frequent = jumpList.frequent()
        frequent.addLink("Новости Python", r"C:\Python36\NEWS.txt")
        frequent.setVisible(True)
        # Получаем категорию задач, добавляем в нее пункт "Блокнот",
        # разделитель, пункт "Write" и делаем видимой
        tasks = jumpList.tasks()
        tasks.addLink("Блокнот", r"C:\Windows\notepad.exe")
        tasks.addSeparator()
        tasks.addLink("Write", r"C:\Windows\write.exe")
        tasks.setVisible(True)
        # Создаем произвольную категорию "Инструменты"
        otherCat = QtWinExtras.QWinJumpListCategory()
        otherCat.setTitle("Инструменты")
        # Добавляем в нее пункт "Python" и также делаем видимой
        otherCat.addLink("Python Shell", r"C:\Python36\pythonw.exe")
        otherCat.setVisible(True)
        jumpList.addCategory(otherCat)
        self.resize(200, 50)

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

Здесь хорошо заметна непонятная особенность либо самих списков быстрого доступа, либо их реализации в PyQt: при добавлении нового пункта в категорию недавно открывавшихся документов этот же пункт появляется и в категории часто открываемых документов, и наоборот. Поэтому имеет смысл выводить в списке быстрого доступа какую-либо одну из этих двух категорий.

30.3. Панели инструментов, выводящиеся на миниатюрах

При наведении курсора мыши на кнопку, находящуюся в панели задач и представляющую запущенное приложение, на экране появляется небольшая миниатюра окна этого приложения. PyQt позволяет создать на нижней границе такой миниатюры небольшую панель инструментов с кнопками, посредством которых можно управлять запущенным приложением.

Функциональность таких панелей инструментов реализуют два класса PyQt: `QWinThumbnailToolBar` и `QWinThumbnailToolButton`. Первый класс создает саму панель инструментов, а второй — отдельную кнопку на ней.

30.3.1. Класс *QWinThumbnailToolBar*

Класс `QWinThumbnailToolBar` представляет саму инструментальную панель, выводящуюся на нижней границе миниатюры. Конструктор этого класса вызывается следующим образом:

```
<Объект> = QWinThumbnailToolBar([parent=None])
```

В параметре `parent` можно указать родительский компонент.

Класс `QWinThumbnailToolBar` поддерживает следующий набор методов, применяемых для формирования панели инструментов:

- `setWindow(<QWindow>)` — задает окно (экземпляр класса `QWindow`), на миниатюре которого должна выводиться панель инструментов.

  Получить экземпляр упомянутого ранее класса, представляющий окно, можно вызовом метода `windowHandle()` класса `QWidget`. Однако здесь нужно иметь в виду, что этот метод способен вернуть корректное значение лишь после вывода окна на экран. Поэтому метод `setWindow` панели инструментов следует вызывать внутри переопределенного метода `showEvent()` (за подробностями — к *разд. 19.7.1*):

```
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        . . .
        self.thumbnailToolbar = ↳
        QtWinExtras.QWinThumbnailToolBar(parent=self)
    . . .
    def showEvent(self, evt):
        self.thumbnailToolbar.setWindow(self.windowHandle())
```

- `window()` — возвращает экземпляр класса `QWindow`, представляющий окно, с которым связана панель инструментов;
- `addButton(<QWinThumbnailToolButton>)` — добавляет в панель инструментов кнопку, представленную экземпляром класса `QWinThumbnailToolButton`;

ВНИМАНИЕ!

На панели инструментов, выводящейся при миниатюре окна, может присутствовать не более семи кнопок.

- `setButtons(<Список кнопок>)` — помещает на панель инструментов кнопки, указанные (в виде экземпляров класса `QWinThumbnailToolButton`) в параметре `<Список кнопок>`. Кнопки, присутствовавшие ранее в панели инструментов, будут удалены;
- `removeButton(<QWinThumbnailToolButton>)` — удаляет указанную (в виде экземпляра класса `QWinThumbnailToolButton`) кнопку из панели инструментов;
- `clear()` — очищает панель инструментов. Метод является слотом;
- `buttons()` — возвращает список кнопок (экземпляров класса `QWinThumbnailToolButton`), имеющихся на панели инструментов;
- `count()` — возвращает количество кнопок, имеющихся на панели инструментов.

30.3.2. Класс *QWinThumbnailToolButton*

Класс `QWinThumbnailToolButton` представляет отдельную кнопку, присутствующую на панели инструментов в миниатюре окна. Вот формат вызова его конструктора:

```
<Объект> = QWinThumbnailToolBar([parent=None])
```

Необязательный параметр `parent` служит для задания родителя кнопки.

Класс `QWinThumbnailToolButton` поддерживает все необходимые для создания кнопки методы:

- `setIcon(<QIcon>)` — устанавливает на кнопку значок, заданный в виде экземпляра класса `QIcon`;
- `icon()` — возвращает экземпляр класса `QIcon`, представляющий установленный на кнопку значок;
- `setToolTip(<Текст подсказки>)` — задает для кнопки текст всплывающей подсказки;
- `toolTip()` — возвращает заданный для кнопки текст всплывающей подсказки;
- `setVisible(<Флаг>)` — если передать с параметром значение `False`, кнопка будет скрыта. Чтобы снова вывести ее на экран, нужно передать значение `True`;
- `isVisible()` — возвращает `True`, если кнопка присутствует на экране, и `False` — в противном случае;
- `setEnabled(<Флаг>)` — если передать с параметром значение `False`, кнопка станет недоступной для нажатия, — при этом она будет закрашена серым цветом. Чтобы вновь сделать кнопку доступной, следует передать значение `True`;
- `isEnabled()` — возвращает `True`, если кнопка доступна для нажатия, и `False`, если она недоступна и закрашена серым;
- `setInteractive(<Флаг>)` — если передать с параметром значение `False`, кнопка станет недоступной для нажатия, но при этом будет отображаться как обычно. Если нужно сделать кнопку вновь доступной, следует передать значение `True`;
- `isInteractive()` — возвращает `True`, если кнопка доступна для нажатия, и `False`, если она недоступна и отображается как обычно;
- `setDismissOnClick(<Флаг>)` — если передать с параметром значение `True`, кнопка после нажатия на нее будет возвращаться в исходное состояние. Если же передать значение

False, кнопка останется в нажатом состоянии и вернется в исходное только после вторичного нажатия (поведение по умолчанию);

- ♦ dismissOnClick() — возвращает True, если кнопка после нажатия сама возвращается в исходное состояние, и False — в противном случае;
- ♦ setFlat(<Флаг>) — если передать с параметром значение True, кнопка будет выводиться без рамки и фона — в виде одного лишь значка. Если же передать значение False, кнопка будет выводиться как обычно (поведение по умолчанию);
- ♦ isFlat() — возвращает True, если кнопка выводится без рамки и фона, и False — в противном случае;
- ♦ click() — имитирует нажатие кнопки. Если кнопка недоступна для нажатия, ничего не делает. Метод является слотом.

Если требуется создать промежуток между какими-либо кнопками, можно поместить туда кнопку, недоступную для нажатия, без значка, рамки и фона:

```
# Создаем кнопку, которая должна выводиться левее свободного пространства
buttonStart = QtWinExtras.QWinThumbnailToolButton(parent=self)
button2.setIcon(iconStart)
button2.setToolTip("Пуск")
. . .
thumbnailToolbar.addButton(buttonStart)
# Конструируем кнопку, которая создаст свободное пространство
buttonSpacer = QtWinExtras.QWinThumbnailToolButton(parent=self)
buttonSpacer.setInteractive(False)
buttonSpacer.setFlat(True)
thumbnailToolbar.addButton(buttonSpacer)
# Создаем кнопку, которая должна выводиться правее свободного
# пространства
buttonStop = QtWinExtras.QWinThumbnailToolButton(parent=self)
. . .
```

Класс QWinThumbnailToolButton поддерживает сигнал clicked(), генерируемый при щелчке на кнопке.

В качестве примера рассмотрим код приложения из листинга 30.4. Оно помещает в панель инструментов на миниатюре три кнопки, из которых первая выводится как обычно, вторая — без рамки и фона, а третья не возвращается в исходное состояние самостоятельно (первые две кнопки возвращаются). Между второй и третьей кнопок присутствует свободное пространство, созданное с помощью недоступной для нажатия кнопки без значка, рамки и фона. Получившуюся панель инструментов можно увидеть на рис. 30.3.

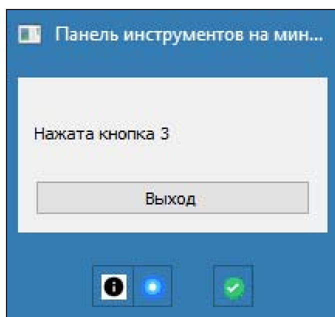

Рис. 30.3. Панель инструментов на миниатюре приложения

Листинг 30.4. Создание панели инструментов на миниатюре приложения

```
from PyQt5 import QtCore, QtWidgets, QtGui, QtWinExtras
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                          flags=QtCore.Qt.Window |
                          QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Панель инструментов на миниатюре")
        # Создаем значки для кнопок
        icon1 = QtGui.QIcon("icon3.png")
        icon2 = QtGui.QIcon("icon4.png")
        icon3 = QtGui.QIcon("icon5.png")
        # Создаем панель инструментов
        self.thumbnailToolbar = QtWinExtras.QWinThumbnailToolBar(
            parent=self)
        # Создаем первые две кнопки
        button1 = QtWinExtras.QWinThumbnailToolButton(parent=self)
        button1.setIcon(icon1)
        button1.setToolTip("Кнопка 1")
        button1.setDismissOnClick(True)
        button1.clicked.connect(self.button1Clicked)
        self.thumbnailToolbar.addButton(button1)
        button2 = QtWinExtras.QWinThumbnailToolButton(parent=self)
        button2.setIcon(icon2)
        button2.setToolTip("Кнопка 2")
        button2.setDismissOnClick(True)
        button2.clicked.connect(self.button2Clicked)
        self.thumbnailToolbar.addButton(button2)
        # Создаем кнопку, что сформирует свободное пространство
        # между второй и третьей кнопками
        button0 = QtWinExtras.QWinThumbnailToolButton(parent=self)
        button0.setInteractive(False)
        button0.setFlat(True)
        self.thumbnailToolbar.addButton(button0)
        # Создаем третью кнопку
        button3 = QtWinExtras.QWinThumbnailToolButton(parent=self)
        button3.setIcon(icon3)
        button3.setToolTip("Кнопка 3")
        button3.clicked.connect(self.button3Clicked)
        self.thumbnailToolbar.addButton(button3)
        # Создаем надпись, куда будут выводиться сообщения о нажатии
        # кнопок панели инструментов, и кнопку выхода
        vbox = QtWidgets.QVBoxLayout()
        self.lblOutput = QtWidgets.QLabel(parent=self)
        vbox.addWidget(self.lblOutput)
        btnClose = QtWidgets.QPushButton("&Выход")
```

```
        btnClose.clicked.connect(QtWidgets.qApp.quit)
        vbox.addWidget(btnClose)
        self.setLayout(vbox)
        self.resize(200, 100)

    # После вывода окна на экран привязываем его к панели инструментов
    def showEvent(self, evt):
        self.thumbnailToolbar.setWindow(self.windowHandle())

    # Выводим в надпись сообщения о нажатии кнопок
    def button1Clicked(self):
        self.lblOutput.setText("Нажата кнопка 1")

    def button2Clicked(self):
        self.lblOutput.setText("Нажата кнопка 2")

    def button3Clicked(self):
        self.lblOutput.setText("Нажата кнопка 3")

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

> ***Примечание***
>
> В PyQt также заявлена поддержка вывода произвольного изображения в качестве миниатюры, которая появляется при наведении курсора мыши на кнопку панели задач, представляющую приложение, и при переключении между приложениями нажатием комбинации клавиш <Alt>+<Tab>. Однако эта поддержка реализована не в полной мере и для указания некоторых ключевых параметров окна требует обращения непосредственно к Windows API, что средствами Python сделать невозможно.

30.4. Дополнительные инструменты по управлению окнами

Ряд инструментов, позволяющих управлять окнами, реализован в статических методах класса `QtWinExtras.QtWin`:

- `taskbarDeleteTab(<Окно>)` — удаляет из панели задач кнопку, представляющую указанное `<Окно>`, которое задается экземпляром класса `QWidget` или `QWindow`:

  ```
  QtWinExtras.QtWin.taskbarDeleteTab(window)
  ```

- `taskbarAddTab(<Окно>)` — вновь добавляет удаленную ранее из панели задач кнопку, представляющую указанное `<Окно>`, которое задается экземпляром класса `QWidget` или `QWindow`;
- `taskbarActivateTab(<Окно>)` — активизирует на панели задач кнопку, представляющую указанное `<Окно>`, не активизируя само окно. `<Окно>` должно быть указано в виде экземпляра класса `QWidget` или `QWindow`;

- `taskbarActivateTabAlt(<Окно>)` — помечает на панели задач кнопку, представляющую указанное `<Окно>`, как активную, не активизируя ни кнопку, ни само окно. `<Окно>` должно быть указано в виде экземпляра класса `QWidget` или `QWindow`;
- `setWindowExcludedFromPeek(<Окно>, <Флаг>)` — если вторым параметром передать значение `True`, при наведении курсора мыши на миниатюру панели задач, представляющую любое другое окно, окно, указанное в параметре `<Окно>`, будет выводиться как есть, а не в виде тонкой рамки. Чтобы вернуть окну поведение по умолчанию (вывод в виде тонкой рамки), следует передать вторым параметром значение `False`. `<Окно>` должно быть указано в виде экземпляра класса `QWidget` или `QWindow`:

  ```
  QtWinExtras.QtWin.setWindowExcludedFromPeek(window, True)
  ```
- `isWindowExcludedFromPeek(<Окно>)` — возвращает `True`, если `<Окно>` при наведении курсора мыши на миниатюру панели задач, представляющую любое другое окно, выводится как есть, и `False`, если оно выводится как обычно, в виде тонкой рамки. `<Окно>` должно быть указано в виде экземпляра класса `QWidget` или `QWindow`;
- `setWindowDisallowPeek(<Окно>, <Флаг>)` — если вторым параметром передать значение `True`, при наведении курсора мыши на миниатюру панели задач, представляющую `<Окно>`, все прочие окна будут выводиться как есть, а не в виде тонких рамок. Чтобы вернуть поведение по умолчанию (вывод остальных окон в виде тонких рамок), следует передать вторым параметром значение `False`. `<Окно>` должно быть указано в виде экземпляра класса `QWidget` или `QWindow`;
- `isWindowPeekDisallowed(<Окно>)` — возвращает `True`, если при наведении курсора мыши на миниатюру панели задач, представляющую `<Окно>`, все прочие окна выводятся как есть, и `False`, если они выводятся как обычно, в виде тонкой рамки. `<Окно>` должно быть указано в виде экземпляра класса `QWidget` или `QWindow`;
- `extendFrameIntoClientArea()` — расширяет полупрозрачную область заголовка, указанного в первом параметре `<Окно>` на его клиентскую область. Работает только в Windows Vista и 7. Форматы метода:

  ```
  extendFrameIntoClientArea(<Окно>, <Слева>, <Сверху>, <Справа>, <Снизу>)
  extendFrameIntoClientArea(<Окно>, <QMargins>)
  ```

  `<Окно>` должно быть указано в виде экземпляра класса `QWidget` или `QWindow`.

  Первый формат позволяет задать величины областей, на которые должна распространяться область полупрозрачности, непосредственно, в виде целых чисел, в пикселах. Второй формат указывает величины этих областей в виде экземпляра класса `QMargins`:

  ```
  QtWinExtras.QtWin.extendFrameIntoClientArea(window, 0, 20, 0, 20)
  ```

  Если указать в качестве всех величин областей, на которые распространяется область полупрозрачности, число `-1`, все окно станет полупрозрачным:

  ```
  QtWinExtras.QtWin.extendFrameIntoClientArea(window, -1, -1, -1, -1)
  ```
- `resetExtendedFrame(<Окно>)` — задает для окна, указанного в параметре `<Окно>`, область полупрозрачности по умолчанию, т. е. охватывающую только заголовок окна. `<Окно>` указывается в виде экземпляра класса `QWidget` или `QWindow`. Работает только в Windows Vista и 7;
- `enableBlurBehindWindow(<Окно>[, <QRegion>])` — создает снаружи окна, указанного в параметре `<Окно>`, область размытия. `<Окно>` задается экземпляром класса `QWidget` или `QWindow`. Вторым параметром можно передать размеры области размытия (экземпляр

класса QRegion), — если этот параметр отсутствует, будет создана область с размерами по умолчанию. Работает только в Windows Vista и 7;

- disableBlurBehindWindow(<Окно>) — убирает у окна, указанного в параметре <Окно>, созданную ранее область размытия. <Окно> задается экземпляром класса QWidget или QWindow. Работает только в Windows Vista и 7;
- setWindowFlip3DPolicy(<Окно>, <Поведение>) — задает для окна, указанного в параметре <Окно> (экземпляра класса QWidget или QWindow), <Поведение> при выводе трехмерного списка открытых окон. В качестве параметра <Поведение> может быть указан один из следующих атрибутов класса QtWinExtras.QtWin:
  - FlipDefault — 0 — окно выводится непосредственно в списке окон (поведение по умолчанию);
  - FlipExcludeAbove — 2 — окно выводится над списком окон, а не в нем;
  - FlipExcludeBelow — 1 — окно выводится под списком окон, а не в нем.

  Работает только в Windows Vista и 7;
- windowFlip3DPolicy(<Окно>) — выводит обозначение поведения окна, указанного в параметре <Окно> (экземпляра класса QWidget или QWindow), при выводе трехмерного списка окон в виде одного из указанных ранее атрибутов класса QtWinExtras.QtWin. Работает только в Windows Vista и 7.

30.5. Получение сведений об операционной системе

Часто бывает необходимо выяснить, на какой операционной системе запущено приложение, и узнать версию этой системы. Для такого случая PyQt предоставляет класс QSysInfo, объявленный в модуле QtCore.

Этот класс поддерживает ряд статических методов (здесь приведен сокращенный список методов, полный их список можно найти на странице **https://doc.qt.io/qt-5/qsysinfo.html**):

- productType() — возвращает строковое наименование системы. В случае Windows возвращает строку windows;
- productVersion() — возвращает строку с короткой версией системы:

```
>>> QtCore.QSysInfo.productVersion()
'10'
```

- prettyProductName() — возвращает строку с названием и короткой версией системы, заключенной в круглые скобки:

```
>>> QtCore.QSysInfo.prettyProductName()
'Windows 10 (10.0)'
```

- kernelVersion() — возвращает строку с полной версией системы:

```
>>> QtCore.QSysInfo.kernelVersion()
'10.0.16299'
```

- currentCpuArchitecture() — возвращает строковое обозначение процессорной архитектуры компьютера, на котором запущено приложение. Могут быть возвращены следующие строки (приведен сокращенный список):

- i386 — 32-разрядный процессор архитектуры *x*86;
- x86_64 — 64-разрядный процессор архитектуры *x*86;
- ia64 — процессор Intel Itanium;
- arm — 32-разрядный процессор архитектуры ARM;
- arm64 — 64-разрядный процессор архитектуры ARM.

Пример:

```
>>> QtCore.QSysInfo.currentCpuArchitecture()
'x86_64'
```

- machineHostName() — возвращает сетевое имя компьютера в виде строки.

30.6. Получение путей к системным каталогам

И напоследок выясним, как средствами PyQt получить пути к некоторым системным каталогам. А сделать это можно, вызвав один из следующих статических методов класса QDir, объявленного в модуле QtCore:

- homePath() — возвращает строковый путь к каталогу пользовательского профиля:

```
>>> QtCore.QDir.homePath()
'C:/Users/vlad'
```

- rootPath() — возвращает строковый путь к корневому каталогу системного диска:

```
>>> QtCore.QDir.rootPath()
'C:/'
```

- tempPath() — возвращает строковый путь к каталогу, предназначенному для хранения временных файлов:

```
>>> QtCore.QDir.tempPath()
'C:/Windows/Temp'
```

Пути, возвращаемые всеми этими тремя методами, включают в себя в качестве разделителей фрагментов прямые слэши. Чтобы преобразовать их в обратные слэши, применяемые в системе Windows, следует воспользоваться статическим методом toNativeSeparators(<Путь>) того же класса QDir:

```
>>> path = QtCore.QDir.homePath()
>>> path
'C:/Users/vlad'
>>> QtCore.QDir.toNativeSeparators(path)
'C:\\Users\\vlad'
```

ГЛАВА 31

Сохранение настроек приложений

Все профессионально выполненные приложения предусматривают возможность настройки их параметров. Эти настройки требуется где-то хранить, чтобы пользователю не пришлось задавать их при каждом запуске приложения. В этом PyQt идет навстречу разработчикам, предоставляя единое *хранилище настроек*, в качестве которого могут использоваться как реестр Windows (по умолчанию), так и INI-файл.

Для работы с таким хранилищем имеется удобный класс `QSettings`, объявленный в модуле `QtCore`.

31.1. Создание экземпляра класса *QSettings*

Рассмотрим все поддерживаемые форматы конструктора класса `QSettings`:

```
<Объект> = QSettings([parent=None])
<Объект> = QSettings(<Название организации>[, application=""][,
                      parent=None])
<Объект> = QSettings(<Диапазон>, <Название организации>[,
                      application=""][, parent=None])
<Объект> = QSettings(<Тип хранилища>, <Диапазон>,
                      <Название организации>[, application=""][,
                      parent=None])
<Объект> = QSettings(<Путь к файлу или ключу Реестра>, <Тип хранилища>[,
                      parent=None])
```

Первый формат — самый простой. Он лишь позволяет указать родителя, присвоив его необязательному параметру `parent`. При этом настройки будут храниться на уровне текущего пользователя в реестре Windows.

Чтобы настройки успешно сохранились, понадобится задать названия организации и приложения, и сделать это нужно еще до создания экземпляра класса `QSettings`. В этом нам помогут следующие два статических метода класса `QCoreApplication` из модуля `QtCore`:

- `setOrganizationName(<Название организации>)` — задает `<Название организации>`;
- `setApplicationName(<Название приложения>)` — задает `<Название приложения>`.

Пример:

```
QtCore.QCoreApplication.setOrganizationName("Прохоренок и Дронов")
QtCore.QCoreApplication.setApplicationName("Тестовое приложение")
settings = QtCore.QSettings()
```

Второй формат позволяет сразу задать <Название организации> — разработчика приложения. В необязательном параметре application можно указать название приложения. Настройки также будут храниться на уровне текущего пользователя в реестре Windows.

В третьем формате мы можем задать <Диапазон> хранения настроек, т. е. указать, будут ли они храниться на уровне текущего пользователя или на уровне системы (относиться ко всем зарегистрированным в системе пользователям). <Диапазон> указывается в виде одного из следующих атрибутов класса QSettings:

- UserScope — 0 — хранение настроек на уровне текущего пользователя;
- SystemScope — 1 — хранение настроек на уровне системы.

Эти настройки тоже будут храниться в реестре Windows.

Если нам нужно явно задать еще и <Тип хранилища> — будет ли это реестр или INI-файл, — мы воспользуемся четвертым форматом конструктора. <Тип хранилища> задается в виде одного из следующих атрибутов класса QSettings:

- NativeFormat — 0 — реестр Windows: его 32-разрядная копия, если приложение работает под управлением 32-разрядной редакции Python, и 64-разрядная — для 64-разрядной редакции языка;
- IniFormat — 1 — INI-файл;
- Registry32Format — 2 — для 64-разрядных приложений — 32-разрядная копия реестра, для 32-разрядных приложений аналогичен атрибуту NativeFormat. Поддерживается, начиная с PyQt 5.7;
- Registry64Format — 3 — для 32-разрядных приложений, запущенных под 64-разрядной Windows, — 64-разрядная копия реестра, для 32-разрядных приложений, работающих под 64-разрядной Windows, и 64-разрядных приложений аналогичен NativeFormat. Поддерживается, начиная с PyQt 5.7.

Пятый формат позволяет напрямую указать <Путь к файлу или ключу реестра>, где хранятся настройки (в частности, этот формат пригодится нам в случае, если нужно прочитать какие-либо системные настройки). В качестве параметра <Тип хранилища> следует указать один из следующих атрибутов класса QSettings:

- NativeFormat — 0 — реестр;
- IniFormat — 1 — INI-файл.

Класс QSettings поддерживает следующие методы, позволяющие узнать значения параметров, которые были заданы при создании экземпляра:

- organizationName() — возвращает название организации;
- applicationName() — возвращает название приложения;
- scope() — возвращает обозначение диапазона;
- format() — возвращает обозначение формата хранения настроек;
- fileName() — возвращает путь к ключу реестра или INI-файлу с настройками.

31.2. Запись и чтение данных

Создав экземпляр класса `QSettings`, мы можем приступить к сохранению настроек или чтению их.

31.2.1. Базовые средства записи и чтения данных

Для выполнения простейших операций по записи и чтению данных класс `QSettings` предоставляет следующие методы:

- `setValue(<Имя значения>, <Значение>)` — записывает заданное `<Значение>` под указанным в виде строки параметром `<Имя значения>`. `<Значение>` может быть любого типа;
- `value(<Имя значения>[, defaultValue=None][, type=None])` — считывает значение, указанное параметром `<Имя значения>`, и возвращает его в качестве результата. В необязательном параметре `defaultValue` можно указать значение, которое будет возвращено, если `<Имя значения>` не будет найдено. Если параметр не задан, возвращается значение `None`. В необязательном параметре `type` можно задать тип, в который должно быть преобразовано считанное значение. Если он не указан, значение возвращается в своем изначальном типе;
- `remove(<Имя значения>)` — удаляет значение, указанное параметром `<Имя значения>`. Если в качестве параметра была передана пустая строка, будут удалены все значения, что находятся в хранилище;
- `contains(<Имя значения>)` — возвращает `True`, если значение, указанное параметром `<Имя значения>`, существует, и `False` — в противном случае;
- `childKeys()` — возвращает список имен имеющихся в хранилище значений;
- `clear()` — удаляет все значения из хранилища;
- `sync()` — выполняет принудительную запись всех выполненных в хранилище изменений в реестр или файл.

В листинге 31.1 приведен код приложения, которое записывает в хранилище настроек (поскольку формат хранения не задан, в качестве хранилища выступает реестр) число, строку и экземпляр класса `QSize`. Далее оно считывает все сохраненные ранее значения, выводит их на экран и проверяет, присутствует ли в хранилище настроек значение, которое заведомо там не сохранялось. Наконец, оно очищает хранилище.

Листинг 31.1. Использование базовых средств хранения настроек

```
from PyQt5 import QtCore, QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
settings = QtCore.QSettings("Прохоренок и Дронов", "Тест 1")
v1 = 123
v2 = "Python"
v3 = QtCore.QSize(640, 480)
print(v1, v2, v3, sep=" | ")
print("Сохраняем настройки")
settings.setValue("Значение 2", v2)
settings.setValue("Значение 3", v3)
```

```
settings.sync()
print("Считываем настройки")
lv1 = settings.value("Значение 1")
lv2 = settings.value("Значение 2")
lv3 = settings.value("Значение 3")
print(lv1, lv2, lv3, sep=" | ")
if settings.contains("Значение 4"):
    print("Значение 4 в хранилище присутствует")
else:
    print("Значение 4 в хранилище отсутствует")
print("Очищаем хранилище")
settings.clear()
```

В консоли это приложение выведет:

```
123 | Python | PyQt5.QtCore.QSize(640, 480)
Сохраняем настройки
Считываем настройки
123 | Python | PyQt5.QtCore.QSize(640, 480)
Значение 4 в хранилище отсутствует
Очищаем хранилище
```

Теперь давайте прочитаем из реестра путь к системному каталогу **Документы**, для чего воспользуемся пятым форматом конструктора класса `QSettings` (см. *разд. 31.1*):

```
>>> settings = QtCore.QSettings("HKEY_CURRENT_USER\\Software\\↳
Microsoft\\Windows\\CurrentVersion\\Explorer\\Shell Folders", ↳
QtCore.QSettings.NativeFormat)
>>> settings.value("Personal")
'D:\\Data\\Документы'
```

31.2.2. Группировка сохраняемых значений. Ключи

PyQt позволяет нам объединять сохраняемые в хранилище настроек значения по какому-либо признаку в особые группы, называемые *ключами*. Каждый ключ может содержать произвольное количество значений, а разные ключи могут содержать значения с одинаковыми именами. Для создания ключей можно применить два способа: простой и сложный.

Простой способ заключается в том, что в первом параметре метода `setValue()` имя значения предваряется именем ключа, в которое его следует поместить, и отделяется от него прямым слэшем:

```
settings.setValue("Ключ 1/Значение 1", v1)
. . .
lv1 = settings.value("Ключ 1/Значение 1")
```

Ключи можно вкладывать внутрь других ключей:

```
settings.setValue("Ключ 2/Вложенный ключ 1/Значение 4", v4)
. . .
lv4 = settings.value("Ключ 2/ Вложенный ключ 1/Значение 4")
```

Сложный способ (который на самом деле не так уж и сложен) пригодится, если нам нужно сохранить в одном ключе сразу несколько значений или же прочитать ряд значений из одного и того же ключа. Для его реализации следует выполнить следующие шаги:

1. Вызвать метод `beginGroup(<Имя или путь ключа>)` класса `QSettings`. В качестве параметра методу передается имя ключа, к содержимому которого следует обратиться, или же целый «путь», если нужно создать вложенный ключ:

   ```
   settings.beginGroup("Ключ 1")
   settings.beginGroup("Ключ 2/Вложенный ключ 1")
   ```

2. Выполнить запись или чтение нужных значений с помощью знакомых нам методов `setValue()` и `value()`. В этом случае значения будут записаны в ключ, указанный в предыдущем вызове метода `beginGroup()`, или же прочитаны из этого ключа.

3. Вызвать метод `endGroup()` класса `QSettings`.

Использование методов `beginGroup()` и `endGroup()` имеет ряд особенностей, о которых нам обязательно следует знать.

- Методы `remove()`, `contains()` и `childKeys()`, если их вызовы помещены между вызовами упомянутых ранее методов, действуют только внутри указанного в вызове метода `beginGroup()` ключа. Так, метод `contains()` будет искать значение с указанным именем только в текущем ключе.
- Вызов метода `remove()` с передачей ему в качестве параметра пустой строки удалит все значения, сохраненные в текущем ключе, не затрагивая содержимое других ключей и «корня» хранилища.
- Список, возвращаемый методом `childKeys()`, будет включать не только имена значений, но и имена всех ключей, вложенных в текущий ключ.
- Для проверки существования какого-либо ключа можно использовать метод `contains()`, вызвав его в ключе предыдущего уровня или же в «корне» хранилища, если ключ никуда не вложен.
- Для удаления ключа можно использовать метод `remove()`, который в этом случае вызывается точно так же.

При использовании ключей для группировки записываемых в хранилище значений нам могут пригодиться три следующих метода класса `QSettings`:

- `group()` — возвращает строку с именем или путем текущего ключа;
- `childGroups()` — возвращает список с именами всех ключей, что имеются в текущем;
- `allKeys()` — возвращает список полных путей ко всем значениям, что имеются в хранилище, включая значения, которые сохранены в ключах.

В листинге 31.2 приведен код приложения, чье окно сохраняет свое местоположение при завершении и восстанавливает при запуске. Помимо этого, при нажатии специальной кнопки выполняется сохранение текста, занесенного пользователем в поле ввода. Местоположение окна и введенный текст сохраняются в разных группах.

Листинг 31.2. Использование ключей

```
from PyQt5 import QtCore, QtWidgets
import sys

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent,
                                   flags=QtCore.Qt.Window)
```

```
        self.setWindowTitle("Использование ключей")
        self.settings = QtCore.QSettings("Прохоренок и Дронов",
                                         "Использование ключей")
        vbox = QtWidgets.QVBoxLayout()
        self.txtLine = QtWidgets.QLineEdit(parent=self)
        vbox.addWidget(self.txtLine)
        btnSave = QtWidgets.QPushButton("&Сохранить текст")
        btnSave.clicked.connect(self.saveText)
        vbox.addWidget(btnSave)
        self.setLayout(vbox)
        if self.settings.contains("Окно/Местоположение"):
            self.setGeometry(self.settings.value("Окно/Местоположение"))
        else:
            self.resize(200, 50)
        if self.settings.contains("Данные/Текст"):
            self.txtLine.setText(self.settings.value("Данные/Текст"))

    def closeEvent(self, evt):
        self.settings.beginGroup("Окно")
        self.settings.setValue("Местоположение", self.geometry())
        self.settings.endGroup()

    def saveText(self):
        self.settings.beginGroup("Данные")
        self.settings.setValue("Текст", self.txtLine.text())
        self.settings.endGroup()

app = QtWidgets.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

31.2.3. Запись списков

Часто бывает необходимо записать в хранилище настроек список каких-либо значений, а потом прочитать его оттуда. Для таких случаев PyQt также предоставляет весьма удобные инструменты.

Для записи списка в хранилище настроек следует выполнить следующую последовательность действий:

1. Вызвать метод `beginWriteArray(<Имя или путь ключа>[, size=-1])` класса `QSettings`. Первым параметром методу передается `<Имя или путь ключа>`, в котором будут сохранены элементы списка. В необязательном параметре `size` можно указать размер списка — если задать значение `-1` или вообще опустить этот параметр, PyQt определит размер сохраняемого списка самостоятельно.
2. Перебрать в цикле весь сохраняемый список. Внутри цикла выполнить следующие действия:
   - вызвать метод `setArrayIndex(<Индекс записываемого элемента>)` класса `QSettings`, тем самым дав PyQt понять, что сейчас будет выполнена запись содержимого элемента списка с индексом, заданным в параметре `<Индекс записываемого элемента>`;

- собственно, записать значение элемента списка, индекс которого был передан методу `setArrayIndex()` на первом шаге.

3. Закончив запись элементов списка, уже по завершении выполнения цикла, в котором выполняется его перебор, вызвать метод `endArray()` класса `QSettings`. Это послужит сигналом окончания записи списка.

Прочитать список из хранилища настроек можно точно таким же образом, за единственным исключением. На *шаге 1* вместо метода `beginWriteArray()` следует вызвать метод `beginReadArray(<Имя или путь ключа>)` того же класса `QSettings`. Метод в качестве результата возвращает размер списка, ранее сохраненный методом `beginWriteArray()`. Этот размер понадобится нам для формирования цикла, который будет выполнять выборку элементов списка.

На *шаге 2*, после вызова метода `setArrayIndex()`, можно пользоваться методами `remove()` и `contains()` для удаления и проверки существования значения соответственно. Эти методы будут действовать только для элемента списка с указанным в вызове метода `setArrayIndex()` индексом.

При использовании описанного здесь подхода в хранилище настроек создается следующая структура ключей и значений:

- ♦ ключ, чье имя (путь) было задано в вызове метода `beginWriteArray()`. В этом ключе будут созданы:
  - значение `size` — хранящее размер записанного списка. Этот размер будет возвращен методом `beginReadArray()`;
  - ключ, чье имя совпадает с индексом, заданным очередным вызовом метода `setArrayIndex()` и увеличенным на единицу.

    В этом ключе будут созданы значения, записываемые последующими вызовами метода `setValue()`.

В качестве примера рассмотрим листинг 31.3, в котором приведен код, формирующий список строк, записывающий его в хранилище настроек, считывающий впоследствии и выводящий на экран.

Листинг 31.3. Запись и чтение списков

```
from PyQt5 import QtCore, QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
settings = QtCore.QSettings("Прохоренок и Дронов", "Тест 2")
l = ["Python", "Ruby", "PHP", "JavaScript"]
print(l)
print("Сохраняем список")
settings.beginWriteArray("Список")
for i, el in enumerate(l):
    settings.setArrayIndex(i)
    settings.setValue("Элемент", el)
settings.endArray()
settings.sync()
print("Считываем список")
```

```
ll = []
lSize = settings.beginReadArray("Список")
for i in range(lSize):
    settings.setArrayIndex(i)
    ll.append(settings.value("Элемент"))
settings.endArray()
print(ll)
settings.clear()
```

В консоли это приложение выведет следующее:

```
['Python', 'Ruby', 'PHP', 'JavaScript']
Сохраняем список
Считываем список
['Python', 'Ruby', 'PHP', 'JavaScript']
```

31.3. Вспомогательные методы класса *QSettings*

Теперь рассмотрим вспомогательные методы класса `QSettings`, которые могут пригодиться в некоторых случаях.

- `status()` — возвращает обозначение состояния, в котором пребывает хранилище настроек после выполнения очередной операции, выраженное в виде одного из следующих атрибутов класса `QSettings`:
  - `NoError` — `0` — операция была успешно выполнена, и никаких ошибок не возникло;
  - `AccessError` — `1` — возникла ошибка доступа к реестру или INI-файлу (возможно, была произведена попытка чтения недоступного для текущего пользователя ключа реестра или файла или же запись в файл, недоступный для записи);
  - `FormatError` — `2` — была выполнена попытка открыть некорректно сформированный INI-файл;
- `isWritable()` — возвращает `True`, если хранилище настроек доступно для записи, и `False` в противном случае (например, если хранилищем является INI-файл, на запись в который текущий пользователь не имеет прав).

31.4. Где хранятся настройки?

Осталось выяснить, где же хранятся настройки, которые мы записали в хранилище. Не будем рассматривать случай, когда экземпляр класса `QSettings` создан с применением пятого формата конструктора, в котором месторасположение хранилища указано напрямую — в виде конкретного пути к ключу реестра или INI-файлу (за подробностями — к *разд. 31.1*). Сосредоточимся на первых четырех форматах, в которых месторасположение хранилища не указано.

- Если выполняется сохранение в реестр Windows (в качестве типа хранилища указаны атрибуты `NativeFormat`, `Registry32Format` или `Registry64Format`):
  - если выполняется сохранение на уровне текущего пользователя (в качестве диапазона указан атрибут `UserScope`):
    - если название приложения указано (т. е. задан параметр `application` конструктора) — в ветви `HKEY_CURRENT_USER\Software\<Название организации>\<Название приложения>`;

    - если название приложения не указано (т. е. параметр application конструктора не задан) — в ветви HKEY_CURRENT_USER\Software\<Название организации>\OrganizationDefaults (похоже, что в таком случае PyQt считает, что заданные настройки применяются сразу ко всем приложениям, разработанным данной организацией);
  - если выполняется сохранение на уровне системы (в качестве диапазона указан атрибут SystemScope):
    - если название приложения указано — в ветви HKEY_LOCAL_MACHINE\Software\<Название организации>\<Название приложения>;
    - если название приложения не указано — в ветви HKEY_LOCAL_MACHINE\Software\<Название организации>\OrganizationDefaults.

  Приложения, работающие под управлением 32-разрядной редакции Python на 64-разрядной редакции Windows, если указан тип хранилища, отличный от Registry64Format, сохраняют настройки в ветвях HKEY_LOCAL_MACHINE\Software\WOW6432node\<Название организации>\<Название приложения> и HKEY_LOCAL_MACHINE\Software\WOW6432node\<Название организации>\OrganizationDefaults соответственно.

- Если выполняется сохранение в INI-файл (в качестве типа хранилища указан атрибут IniFormat):
  - если выполняется сохранение на уровне текущего пользователя:
    - если название приложения указано — в файле <каталог пользовательского профиля>\AppData\Roaming\<Название организации>\<Название приложения>.ini;
    - если название приложения не указано — в файле <каталог пользовательского профиля>\AppData\Roaming\<Название организации>.ini;
  - если выполняется сохранение на уровне системы:
    - если название приложения указано — в файле <корневой каталог системного диска>\ProgramData\<Название организации>\<Название приложения>.ini;
    - если название приложения не указано — в файле <корневой каталог системного диска>\ProgramData\<Название организации>.ini.

ГЛАВА 32

Приложение «Судоку»

В завершение в качестве примера напишем на языке Python с применением библиотеки PyQt приложение «Судоку». Оно предназначено для создания и решения головоломок судоку и позволит, помимо всего прочего, сохранять головоломки в файлах, загружать их из файлов и выводить на печать.

Изначально приложение было разработано Н. Прохоренком в 2011 году на языке C++. Для этой книги оно переписано на Python и несколько усовершенствовано В. Дроновым в конце 2017 года.

32.1. Правила судоку

Судоку — традиционная японская числовая головоломка (иногда ее неправильно называют магическим квадратом). Далее приведены правила ее решения (полное описание судоку можно найти по интернет-адресу **https://ru.wikipedia.org/wiki/Судоку**).

- Поле судоку представляет собой квадрат, разбитый на 81 ячейку (9 столбцов по 9 строк). Каждые 9 ячеек объединены в группу 3 × 3 — итого получается 9 групп.
- В каждую ячейку поля можно подставить только одну цифру от 1 до 9.
- Ячейки группы не должны содержать одинаковых цифр.
- Одна и та же цифра должна присутствовать в каждой строке и каждом столбце поля только один раз.

На рис. 32.1 приведен пример решенной судоку.

5	4	7	8	6	3	9	2	1
3	8	1	2	9	7	6	5	4
2	9	6	5	1	4	8	3	7
6	2	5	9	3	1	4	7	8
7	3	8	4	2	6	1	9	5
4	1	9	7	5	8	3	6	2
9	7	3	1	8	2	5	4	6
1	6	2	3	4	5	7	8	9
8	5	4	6	7	9	2	1	3

Рис. 32.1. Решенная судоку (группы выделены утолщенными рамками)

Как правило, судоку решают вдвоем: один игрок произвольно расставляет цифры в некоторых ячейках поля, а второй, собственно, решает головоломку.

32.2. Описание приложения «Судоку»

Приложение «Судоку» выполнено в традиционном ключе, присущем обычным Windows-приложениям. Ее окно (рис. 32.2) включает в себя главное меню, панель инструментов, основное содержимое — поле судоку и набор кнопок для установки цифр в ячейки, и строку состояния.

В самом поле судоку группы ячеек выделены различными цветами фона: оранжевым и светло-серым. Установленные в них цифры выводятся черным цветом.

Одна из ячеек является *активной* — именно с активной ячейкой осуществляется взаимодействие. Активная ячейка закрашена желтым цветом (на рис. 32.2 это ячейка с цифрой 4). Чтобы сделать какую-либо ячейку активной, следует:

- ♦ либо, пользуясь клавишами-стрелками, переместить на нее желтый фокус выделения;
- ♦ либо просто щелкнуть на ней мышью.

Чтобы установить в активную ячейку какую-либо цифру, следует:

- ♦ либо нажать соответствующую кнопку в наборе, расположенном ниже поля;
- ♦ либо нажать соответствующую цифровую клавишу.

Чтобы убрать цифру из активной ячейки, нужно:

- ♦ либо нажать кнопку **X**, что находится в расположенном под полем наборе;
- ♦ либо нажать клавишу пробела, <Backspace> или <Del>.

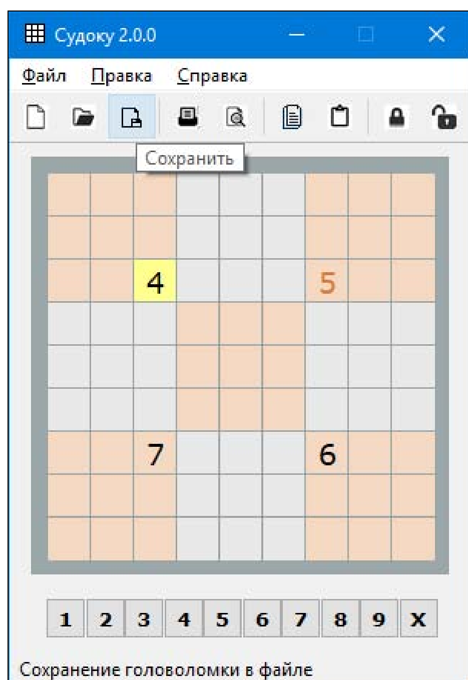

Рис. 32.2. Окно приложения «Судоку»

Чтобы случайно не занести в какую-либо ячейку другую цифру, есть возможность заблокировать ее. Для этого следует сделать нужную ячейку активной и нажать клавишу <F2>. В заблокированной ячейке цифра выводится красным шрифтом (на рис. 32.2 заблокирована ячейка с цифрой 5). Отметим, что блокировать можно только ячейки, содержащие цифры.

Снять блокировку с ячейки можно, сделав ее активной и нажав клавишу <F4>.

Главное меню приложения содержит следующие пункты:

- меню **Файл**:
  - **Новый** (с ним связана комбинация клавиш <Ctrl>+<N>) — очистка поля;
  - **Открыть** (<Ctrl>+<O>) — загрузка сохраненной ранее головоломки из выбранного пользователем файла;
  - **Сохранить** (<Ctrl>+<S>) — сохранение головоломки в файле с указанным пользователем именем.

    При выборе этого пункта производится сохранение в полном формате, т. е. для каждой ячейки, помимо находящейся в ней цифры, сохраняется признак того, заблокирована ли она;
  - **Сохранить компактно** — сохранение головоломки в файле в компактном формате.

    Компактный формат не предусматривает хранения признака блокировки ячейки. Вследствие этого файл, сохраненный в компактном формате, вдвое меньше полноформатного.

    При открытии файла, сохраненного в компактном формате, производится автоматическая блокировка всех ячеек, содержащих цифры;
  - **Печать** (<Ctrl>+<P>) — вывод головоломки на печать;
  - **Предварительный просмотр** — просмотр головоломки в том виде, в котором она будет выведена на печать;
  - **Параметры страницы** — настройка параметров печатаемой страницы;
  - **Выход** (<Ctrl>+<Q>) — завершение работы приложения;
- меню **Правка**:
  - **Копировать** (<Ctrl>+<C>) — копирование головоломки в буфер обмена в полном формате (аналогичном тому, в котором сохраняется файл при выборе пункта **Сохранить** меню **Файл**);
  - **Копировать компактно** — копирование головоломки в буфер обмена в компактном формате (аналогичном тому, в котором сохраняется файл при выборе пункта **Сохранить компактно** меню **Файл**);
  - **Копировать для Excel** — копирование головоломки в буфер обмена в формате, предназначенном для вставки в таблицу Microsoft Excel;
  - **Вставить** (<Ctrl>+<V>) — вставка головоломки, скопированной ранее в любом формате.

    Если головоломка была скопирована в компактном формате, все ячейки, содержащие цифры, будут автоматически заблокированы;
  - **Вставить из Excel** — вставка из буфера обмена головоломки, скопированной из таблицы Microsoft Excel;
  - **Блокировать** (<F2>) — блокировка активной ячейки;

   * **Блокировать все** (<F3>) — блокировка всех ячеек. Не забываем, что блокируются только ячейки, содержащие цифры;
   * **Разблокировать** (<F4>) — разблокировка активной ячейки;
   * **Разблокировать все** (<F5>) — разблокировка всех ячеек;
- меню **Справка**:
   * **О программе** — вывод окна со сведениями о приложении «Судоку»;
   * **О Qt** — вывод окна со сведениями о самой библиотеке Qt.

При наведении курсора мыши на любой пункт меню в строке состояния появляется его развернутое описание.

В панели инструментов присутствуют кнопки (в порядке слева направо): **Новый**, **Открыть**, **Сохранить**, **Печать**, **Предварительный просмотр**, **Копировать**, **Вставить**, **Блокировать все** и **Разблокировать все**. Они выполняют те же действия, что и одноименные пункты меню. При наведении курсора мыши на кнопку панели инструментов в строке состояния появляется ее развернутое описание, а рядом с кнопкой спустя небольшой промежуток времени появляется всплывающая подсказка (на рис. 32.2 курсор мыши наведен на кнопку **Сохранить**).

Сохранение головоломок выполняется в текстовых файлах с расширением svd. Форматы, в которых хранятся головоломки, будут описаны позже.

Окно приложения при закрытии сохраняет свое местоположение и впоследствии, после следующего запуска, восстанавливает его.

32.3. Программирование приложения

Определившись, что должно делать наше приложение, мы можем начать его разработку. Но сначала выполним некоторые подготовительные действия.

32.3.1. Подготовительные действия

Создадим где-либо каталог, в котором будут находиться все файлы нашего будущего приложения. Назовем его, скажем, sudoku. В этом каталоге создадим два вложенных каталога:

- images — для хранения значков, которые будут выводиться в пунктах меню и кнопках панели инструментов, а также значки самого приложения;
- modules — для хранения файлов с программными модулями Python, которые напишем впоследствии.

Найдем в Интернете значки для представления самого приложения, пунктов меню и кнопок панели инструментов. Сохраним их в каталоге images под именами svd.png (значок приложения), а также new.png, open.png, save.png, print.png, preview.png, copy.png, paste.png, lock.png и unlock.png (значки кнопок).

32.3.2. Класс *MyLabel*: ячейка поля судоку

Начнем мы разработку приложения с создания класса `MyLabel`, который будет представлять отдельную ячейку поля судоку, — 81 такой компонент и составят это поле.

Наш класс должен «уметь» выводить на экран цифру, занесенную в ячейку, отображать обычное, активное и заблокированное состояния, реагировать на щелчки мышью, чтобы

сообщить компоненту поля судоку, что та или иная ячейка стала активной. Помимо этого, ячейка должна иметь возможность принимать фоновый цвет — ведь именно разными цветами фона мы будем выделять группы ячеек на поле.

Визуально ячейка должна иметь размеры 30 × 30 пикселов и выводить цифру, выровненную по середине без отступов от границ компонента.

Бóльшую часть необходимой функциональности мы можем получить, просто сделав класс `MyLabel` производным от класса надписи `QLabel`. В самом деле, чтобы вывести на экран цифру, можно просто задать ее в качестве содержимого надписи, воспользовавшись унаследованным методом `setText()`, — также несложно будет указать необходимые размеры и выравнивание. А задать для текста и фона нужные цвета мы сможем, привязав к ячейке таблицу стилей, для чего воспользуемся методом `setStyleSheet()`, опять же, унаследованным.

Теперь подумаем, какие атрибуты должен поддерживать класс `MyLabel`:

- `colorYellow`, `colorOrange`, `colorGrey`, `colorBlack` и `colorRed` — будут хранить RGB-коды, соответственно, желтого, оранжевого, светло-серого, черного и красного цветов. Поскольку значения этих атрибутов будут одинаковыми для всех экземпляров класса ячейки, мы сделаем их атрибутами объекта класса. (Если же сделать их атрибутами экземпляра класса, каждый экземпляр будет хранить свой собственный набор этих атрибутов, что приведет к избыточному расходу оперативной памяти.);
- `isCellChange` — значение `True` сообщит о том, что ячейка разблокирована, а значение `False` — что она заблокирована;
- `fontColorCurrent` — текущий цвет текста: черный или красный;
- `bgColorDefault` — заданный при создании ячейки цвет фона: оранжевый или светло-серый. Он понадобится нам, чтобы перевести ячейку из активного в неактивное состояние;
- `bgColorCurrent` — текущий цвет фона: желтый, оранжевый или светло-серый;
- `id` — порядковый номер ячейки. Он понадобится нам, чтобы при щелчке мышью на ячейке сообщить компоненту поля судоку, какая ячейка стала активной.

Компонент `MyLabel` будет входить в состав компонента поля, который мы напишем потом. Он должен уведомлять компонент поля, когда текущая ячейка становится активной после щелчка мышью. Наилучший способ сделать это — объявить сигнал, который будет генерироваться при щелчке. Обрабатывая этот сигнал, поле всегда будет «в курсе», на какой ячейке был выполнен щелчок.

Мы объявим в ячейке сигнал `cellChangeFocus`. Он будет передавать обработчику единственный параметр целочисленного типа — номер ячейки, на которой пользователь щелкнул мышью.

Теперь определим набор необходимых методов нашего класса и представим в общих чертах, что должен делать каждый из них:

- `mousePressEvent()` — этот метод следует переопределить, чтобы получить возможность обрабатывать щелчки мышью. Внутри него мы будем генерировать сигнал `cellChangeFocus`;
- `showColorCurrent()` — задаст для ячейки цвета текста и фона, взятые из атрибутов `fontColorCurrent` и `bgColorCurrent` соответственно, и тем самым обновит визуальное состояние ячейки, или, как говорят программисты, перерисует ее. Этот метод мы будем

вызывать после перевода ячейки из неактивного состояния в активное, из разблокированного — в заблокированное и наоборот;

- `setCellFocus()` — переведет ячейку из неактивного состояния в активное. Сделать это очень просто — мы занесем в атрибут `bgColorCurrent` код желтого цвета (который хранится в атрибуте объекта класса `colorYellow`) и вызовем метод `showColorCurrent()`, чтобы обновить визуальное представление ячейки;
- `clearCellFocus()` — переведет ячейку из активного состояния в неактивное. Мы занесем в атрибут `bgColorCurrent` значение цвета фона, взятое из атрибута `bgColorDefault` (он, как мы помним, хранит код изначального цвета фона), и вызовем метод `showColorCurrent()`, который перерисует ячейку;
- `setCellBlock()` — переведет ячейку из разблокированного состояния в заблокированное. Здесь мы занесем в атрибут `isCellChange` значение `False`, тем самым указывая, что ячейка заблокирована, присвоим атрибуту `fontColorCurrent` код красного цвета, хранящийся в атрибуте объекта класса `colorRed`, и перерисуем ячейку вызовом метода `showColorCurrent()`;
- `clearCellBlock()` — переведет ячейку из заблокированного состояния в разблокированное. Для этого достаточно присвоить атрибуту `isCellChange` значение `True`, занести в атрибут `fontColorCurrent` значение черного цвета из атрибута объекта класса `colorBlack` и не забыть перерисовать ячейку с помощью метода `showColorCurrent()`;
- `setNewText()` — занесет в ячейку новое число, переданное ему в качестве единственного параметра в виде строки. Здесь нужно предварительно проверить, не заблокирована ли ячейка (не хранится ли в атрибуте `isCellChange` значение `False`), и уже потом вызывать унаследованный от класса `QLabel` метод `setText()`;
- конструктор должен принимать в качестве обязательных параметров номер создаваемой ячейки (он будет занесен в атрибут `id`), цвет ее фона (его мы присвоим атрибутам `bgColorDefault` и `bgColorCurrent`), а также необязательный параметр родителя `parent`. Еще он должен присвоить атрибуту `isCellChange` значение `True` — вновь созданная ячейка изначально должна быть разблокирована. Напоследок он перерисует ячейку, вызвав метод `showColorCurrent()`.

Код класса `MyLabel` относительно невелик, несложен и полностью приведен в листинге 32.1. Его следует сохранить в файле mylabel.py в каталоге modules.

> **Электронный архив**
>
> Напомним, что файлы с кодами разрабатываемого в этой главе приложения находятся в папке *sudoku* сопровождающего книгу электронного архива (см. *приложение*).

Листинг 32.1. Класс `MyLabel`

```
from PyQt5 import QtCore, QtWidgets

class MyLabel(QtWidgets.QLabel):
    colorYellow = "#FFFF90"
    colorOrange = "#F5D8C1"
    colorGrey = "#E8E8E8"
    colorBlack = "#000000"
    colorRed = "#D77A38"
```

```
    changeCellFocus = QtCore.pyqtSignal(int)

    def __init__(self, id, bgColor, parent=None):
        QtWidgets.QLabel.__init__(self, parent)
        self.setAlignment(QtCore.Qt.AlignCenter)
        self.setFixedSize(30, 30)
        self.setMargin(0)
        self.setText("")
        if id < 0 or id > 80:
            id = 0
        self.id = id
        self.isCellChange = True
        self.fontColorCurrent = self.colorBlack
        self.bgColorDefault = bgColor
        self.bgColorCurrent = bgColor
        self.showColorCurrent()

    def mousePressEvent(self, evt):
        self.changeCellFocus.emit(self.id)
        QtWidgets.QLabel.mousePressEvent(self, evt)

    def showColorCurrent(self):
        self.setStyleSheet("background-color:" + self.bgColorCurrent +
                           ";color:" + self.fontColorCurrent + ";")

    def setCellFocus(self):
        self.bgColorCurrent = self.colorYellow
        self.showColorCurrent()

    def clearCellFocus(self):
        self.bgColorCurrent = self.bgColorDefault
        self.showColorCurrent()

    def setCellBlock(self):
        self.isCellChange = False
        self.fontColorCurrent = self.colorRed
        self.showColorCurrent()

    def clearCellBlock(self):
        self.isCellChange = True
        self.fontColorCurrent = self.colorBlack
        self.showColorCurrent()

    def setNewText(self, text):
        if self.isCellChange:
            self.setText(text)
```

В принципе, объяснять здесь более особо нечего — все было рассказано ранее. Нужно лишь отметить пару чисто технических деталей:

- в методе `mousePressEvent()`, выполнив все необходимые действия, а именно, сгенерировав сигнал `changeCellFocus`, мы в обязательном порядке вызываем тот же метод суперкласса. Если этого не сделать, возможны проблемы;

- в методе `showColorCurrent()` мы вызовом метода `setStyleSheet()` привязываем к нашему компоненту таблицу стилей, которая установит для ячейки цвета текста и фона. (Прочие параметры для компонента мы установим в таблице стилей, привязанной к компоненту основного окна, которую создадим позже.)

32.3.3. Класс *Widget*: поле судоку

Класс `Widget` представит само поле судоку, составленное из 81-го компонента `MyLabel`, написанного ранее, и набора из 10-ти обычных кнопок, с помощью которых пользователь будет вставлять цифры в ячейки.

Сразу после создания компонент `Widget` должен вывести на экран поле судоку и набор кнопок. Он даст пользователю возможность делать ячейки активными с помощью клавиш-стрелок (активизация посредством щелчка мышью уже реализована нами в классе `MyLabel`), устанавливать в ячейки цифры либо клавишами, либо кнопками из набора и удалять цифры, опять же, клавишами или специальной кнопкой. Это основная функциональность компонента.

Что касается дополнительной функциональности, то ее мы будем реализовывать по частям. И сейчас мы сделаем лишь очистку поля, блокировку и разблокировку его ячеек.

Поскольку функциональности, специфической для какого-либо уже имеющегося в библиотеке компонента, нам не требуется, класс `Widget` мы сделаем производным от класса `QWidget`.

Необходимый нам набор атрибутов класса очень невелик:

- `cells` — массив ячеек поля судоку — экземпляров класса `MyLabel`. Мы сохраним этот массив в атрибуте, поскольку в других методах этого класса нам понадобится получать к нему доступ;
- `idCellInFocus` — порядковый номер ячейки, являющейся активной в настоящий момент. Его нужно знать в любом случае — хотя бы для того, чтобы визуально выделить активную ячейку, вызвав у нее метод `setCellFocus()` (см. *разд. 32.3.2*).

Набор методов, поддерживаемый классом `Widget`, будет более объемным (даже с учетом того, что мы еще не реализовали в классе дополнительную функциональность). Поскольку эти методы сложнее таковых у класса `MyLabel`, мы рассмотрим их вкратце, не углубляясь в технические детали — полное их описание будет приведено далее, вместе с их кодом:

- `onChangeCellFocus()` — обработчик сигналов `changeCellFocus` всех ячеек, что имеются в поле. (Упомянутый здесь сигнал, как мы помним, генерируется при щелчке на ячейке мышью.) Он получит с единственным параметром номер ячейки, на которой был выполнен щелчок, и активизирует ее;
- `keyPressEvent()` — этот метод следует переопределить, чтобы получить возможность обрабатывать нажатия клавиш клавиатуры. Здесь мы будем, в зависимости от нажатой клавиши, перемещать по полю фокус выделения, ставить цифры в ячейки и очищать их;
- `onBtn<N>Clicked()` (где `<N>` — число от 0 до 8 или буква `X`) — обработчики щелчков на кнопках **1**...**9** и **X**. Они будут ставить в ячейку соответствующую цифру или же очищать ячейку;
- `onClearAllCells()` — очистит поле судоку. Будет вызываться при выборе пункта **Новый** меню **Файл** или нажатии кнопки **Новый** панели инструментов;
- `onBlockCell()` — заблокирует активную ячейку, если она содержит цифру и еще не заблокирована. Будет вызываться при выборе пункта **Заблокировать** меню **Правка**;

- `onBlockCells()` — заблокирует все ячейки, содержащие цифры и не заблокированные. Будет вызываться при выборе пункта **Заблокировать все** меню **Правка** или нажатии кнопки **Заблокировать все** панели инструментов;
- `onClearBlockCell()` — разблокирует активную ячейку, если она заблокирована. Будет вызываться при выборе пункта **Разблокировать** меню **Правка**;
- `onClearBlockCells()` — разблокирует все заблокированные ячейки. Будет вызываться при выборе пункта **Разблокировать все** меню **Правка** или нажатии кнопки **Разблокировать все** панели инструментов;
- конструктор, который создаст все необходимые компоненты и выполнит привязку обработчиков к сигналам.

Весь код класса `Widget` мы сохраним в файле widget.py в каталоге modules. Поскольку этот код довольно велик, мы рассмотрим его по частям.

32.3.3.1. Конструктор класса *Widget*

Конструктор — самый сложный метод класса `Widget`. Поэтому мы не станем приводить его полный листинг, а рассмотрим его по фрагментам.

```
from PyQt5 import QtCore, QtGui, QtWidgets
```

Помимо модулей `QtCore` и `QtWidgets`, которые понадобятся нам уже сейчас, мы импортируем модуль `QtGui`. Объявленные в нем классы пригодятся нам позже, при реализации печати.

```
from modules.mylabel import MyLabel
```

Не забываем импортировать из модуля mylabel.py, что хранится в каталоге modules, класс `MyLabel`, представляющий отдельную ячейку и написанный нами ранее.

```
class Widget(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFocusPolicy(QtCore.Qt.StrongFocus)
```

По умолчанию экземпляр класса `QWidget` или производного от него класса не может принимать фокус ввода. Чтобы дать ему возможность принимать фокус ввода при щелчке мышью и переходе нажатием клавиши <Tab>, мы вызовем у него метод `setFocusPolicy()`, передав ему в качестве параметра атрибут `StrongFocus`.

```
        vBoxMain = QtWidgets.QVBoxLayout()
```

Поскольку само поле и набор кнопок будут располагаться друг над другом, мы используем для их размещения контейнер `QVBoxLayout`.

```
        frame1 = QtWidgets.QFrame()
        frame1.setStyleSheet(
            "background-color:#9AA6A7;border:1px solid #9AA6A7;")
```

Поле (которое будет создано контейнером-сеткой `QGridLayout`) мы поместим в панель с рамкой `QFrame`. Для этой панели с помощью таблицы стилей укажем тонкую рамку и фон одинакового темно-серого цвета. Панель с рамкой займет все выделенное под него пространство контейнера, а поле судоку поместится в центре этой панели. В результате этого будет казаться, что поле окружено толстой темно-серой рамкой, что выглядит весьма эффектно.

```
        grid = QtWidgets.QGridLayout()
        grid.setSpacing(0)
```

Создаем сетку `QGridLayout`, которая сформирует само поле.

```
idColor = (3, 4, 5, 12, 13, 14, 21, 22, 23,
           27, 28, 29, 36, 37, 38, 45, 46, 47,
           33, 34, 35, 42, 43, 44, 51, 52, 53,
           57, 58, 59, 66, 67, 68, 75, 76, 77)
```

Объявляем массив, хранящий номера ячеек, которые должны быть выделены светло-серым фоном.

```
self.cells = [MyLabel(i, MyLabel.colorGrey if i in idColor else ↳
MyLabel.colorOrange) for i in range(0, 81)]
```

Создаем список из 81-й ячейки `MyLabel`, сохранив его в атрибуте `cells` класса `Widget`. Здесь мы используем выражение генератора списка, которое позволит нам радикально упростить код. Если номер создаваемой ячейки имеется в объявленном ранее массиве, задаем для нее светло-серый цвет фона, в противном случае — оранжевый.

```
self.cells[0].setCellFocus()
self.idCellInFocus = 0
```

Делаем активной ячейку с номером 0 и заносим тот же номер в атрибут `idCellInFocus` класса `Widget`. В результате изначально активной станет самая первая ячейка поля.

```
i = 0
for j in range(0, 9):
    for k in range(0, 9):
        grid.addWidget(self.cells[i], j, k)
        i += 1
```

Помещаем все созданные ячейки в сетку.

```
for cell in self.cells:
    cell.changeCellFocus.connect(self.onChangeCellFocus)
```

У всех ячеек задаем для сигнала `changeCellFocus` обработчик — метод `onChangeCellFocus()` класса `Widget`, пока еще не объявленный.

```
frame1.setLayout(grid)
vBoxMain.addWidget(frame1, alignment=QtCore.Qt.AlignHCenter)
```

Помещаем сетку в панель с рамкой и добавляем последнюю в контейнер `VBoxLayout`, указав для нее горизонтальное выравнивание по середине.

```
frame2 = QtWidgets.QFrame()
frame2.setFixedSize(272, 36)
```

Набор кнопок, с помощью которых будет выполняться занесение цифр в ячейки, мы поместим в другую панель с рамкой `QFrame`. Это позволит нам привязать к кнопкам таблицу стилей, задающую для них представление. Для панели мы обязательно зададим фиксированные размеры — иначе их установит сам PyQt согласно своему разумению, которое вряд ли совпадет с нашим.

```
hbox = QtWidgets.QHBoxLayout()
hbox.setSpacing(1)
```

Кнопки у нас будут выстроены по горизонтали, следовательно, наилучший вариант — поместить их в контейнер `QHBoxLayout`.

```
btns = []
for i in range(1, 10):
    btn = QtWidgets.QPushButton(str(i))
```

```
        btn.setFixedSize(27, 27)
        btn.setFocusPolicy(QtCore.Qt.NoFocus)
        btns.append(btn)
```

Создаем кнопки **1**...**9**, задаем для них размеры 27 × 27 пикселов и добавляем в специально созданный для этого список. Также для каждой кнопки мы указываем, что она не должна принимать фокус ввода (для чего вызовем у нее метод `setFocusPolicy()` с параметром `NoFocus`), — это нужно для того, чтобы поле судоку при нажатии любой из этих кнопок не теряло фокус, и пользователь смог продолжать манипулировать в нем с помощью клавиш.

```
    btn = QtWidgets.QPushButton("X")
    btn.setFixedSize(27, 27)
    btns.append(btn)
```

Таким же образом создаем кнопку **X**, которая уберет цифру из ячейки.

```
    for btn in btns:
        hbox.addWidget(btn)
```

Помещаем все кнопки в контейнер `QHBoxLayout`.

```
    btns[0].clicked.connect(self.onBtn0Clicked)
    btns[1].clicked.connect(self.onBtn1Clicked)
    btns[2].clicked.connect(self.onBtn2Clicked)
    btns[3].clicked.connect(self.onBtn3Clicked)
    btns[4].clicked.connect(self.onBtn4Clicked)
    btns[5].clicked.connect(self.onBtn5Clicked)
    btns[6].clicked.connect(self.onBtn6Clicked)
    btns[7].clicked.connect(self.onBtn7Clicked)
    btns[8].clicked.connect(self.onBtn8Clicked)
    btns[9].clicked.connect(self.onBtnXClicked)
```

Привязываем к сигналам `clicked` всех этих кнопок соответствующие обработчики — методы класса поля, которые объявим потом.

```
    frame2.setLayout(hbox)
    vBoxMain.addWidget(frame2, alignment=QtCore.Qt.AlignHCenter)
```

Помещаем контейнер с кнопками в панель с рамкой, а ее — во «всеобъемлющий» контейнер `VBoxLayout`, не забыв указать горизонтальное выравнивание по середине.

```
    self.setLayout(vBoxMain)
```

И помещаем этот контейнер в компонент поля.

32.3.3.2. Прочие методы класса *Widget*

Теперь напишем код остальных методов класса `Widget`. Они существенно проще конструктора, и мы можем не рассматривать их по частям.

Листинг 32.2. Метод `onChangeCellFocus()`

```
def onChangeCellFocus(self, id):
    if self.idCellInFocus != id and not (id < 0 or id > 80):
        self.cells[self.idCellInFocus].clearCellFocus()
        self.idCellInFocus = id
        self.cells[id].setCellFocus()
```

Метод `onChangeCellFocus()` станет обработчиком сигнала `changeCellFocus` ячейки `MyLabel`. В качестве единственного параметра он получит номер ячейки, ставшей активной.

Сначала мы проверим, не совпадает ли полученный номер с тем, что хранится в атрибуте `idCellInFocus` (не щелкнул ли пользователь на активной ячейке), не меньше ли он 0 и не больше ли 80 (т. е. не вышел ли он за диапазон номеров ячеек). Если это так, мы выполняем операцию по переносу фокуса на ячейку с полученным в параметре номером.

Предварительно нам следует деактивировать ячейку, бывшую активной ранее (номер этой ячейки в настоящий момент хранится в атрибуте `idCellInFocus`). Извлекаем номер, получаем из списка ячеек (он хранится в атрибуте `cells`) саму эту ячейку и переводим ее в неактивное состояние вызовом метода `clearCellFocus()`. Далее мы заносим полученный с параметром номер в атрибут `idCellInFocus`, тем самым указывая, что ячейка с этим номером сейчас активна, и делаем ее активной, вызвав у нее метод `setCellFocus()`.

Листинг 32.3. Метод `keyPressEvent()`

```
    def keyPressEvent(self, evt):
        key = evt.key()
        if key == QtCore.Qt.Key_Up:
            tid = self.idCellInFocus - 9
            if tid < 0:
                tid += 81
            self.onChangeCellFocus(tid)
        elif key == QtCore.Qt.Key_Right:
            tid = self.idCellInFocus + 1
            if tid > 80:
                tid -= 81
            self.onChangeCellFocus(tid)
        elif key == QtCore.Qt.Key_Down:
            tid = self.idCellInFocus + 9
            if tid > 80:
                tid -= 81
            self.onChangeCellFocus(tid)
        elif key == QtCore.Qt.Key_Left:
            tid = self.idCellInFocus - 1
            if tid < 0:
                tid += 81
            self.onChangeCellFocus(tid)
        elif key >= QtCore.Qt.Key_1 and key <= QtCore.Qt.Key_9:
            self.cells[self.idCellInFocus].setNewText(chr(key))
        elif key == QtCore.Qt.Key_Delete or ↳
        key == QtCore.Qt.Key_Backspace or key == QtCore.Qt.Key_Space:
            self.cells[self.idCellInFocus].setNewText("")
        QtWidgets.QWidget.keyPressEvent(self, evt)
```

Переопределенный метод `keyPressEvent()` будет обрабатывать нажатия клавиш. В качестве параметра он получит экземпляр класса, представляющий событие клавиатуры. Мы сразу же вызовем у этого экземпляра метод `key()`, чтобы получить код нажатой клавиши. После чего начнем последовательно сравнивать его с кодами различных клавиш, чтобы выяснить, какая из них была нажата.

Если была нажата клавиша <↑>, следует сделать активной ячейку, расположенную строкой выше. Мы можем с легкостью получить номер этой ячейки, вычтя из номера активной ячейки (он, как мы знаем, хранится в атрибуте `idCellInFocus`) число 9 — т. е. количество ячеек, помещающихся в строке поля. Если получившаяся разность меньше 0 (фокус выделения вышел за пределы верхней границы поля), мы прибавляем к разности 81 (количество ячеек в поле), в результате чего фокус окажется на самой последней строке поля, в том же столбце. И, наконец, вызываем метод `onChangeCellFocus()` класса поля, передав ему результирующий номер ячейки, чтобы сделать ее активной.

Теперь рассмотрим случай, когда была нажата клавиша <→>. Нам нужно сделать активной ячейку, расположенную правее. Понятно, что для получения ее номера нам следует прибавить к номеру активной ячейки единицу. Если же полученная сумма оказалась больше 80 (фокус выделения вышел за пределы верхней границы диапазона имеющихся ячеек), мы вычтем из нее то же число 81 — тогда фокус окажется на самой первой ячейке в поле. И не забываем напоследок вызвать метод `onChangeCellFocus()`.

Обработка нажатия клавиш <↓> и <←> производится аналогично. Вы можете сами разобраться, как работает выполняющий ее код.

Если была нажата клавиша <1>...<9>, мы формируем на основе ее кода соответствующий символ, воспользовавшись функцией `chr()`, вызываем у активной ячейки метод `setNewText()` и передаем ему этот символ. Так мы занесем в активную ячейку цифру, соответствующую нажатой клавише.

Случай нажатия клавиш <Backspace>, <Del> или <пробел> — самый простой. Мы вызываем у активной ячейки метод `setNewText()`, передав ему пустую строку — так мы уберем цифру с ячейки.

В самом конце, какая бы ни была нажата клавиша, мы в обязательном порядке вызываем метод `keyPressEvent()` базового класса. Если этого не сделать, приложение может повести себя непредсказуемо.

Листинг 32.4. Методы `onBtn0Clicked()`...`onBtn8Clicked()` и `onBtnXClicked()`

```
    def onBtn0Clicked(self):
        self.cells[self.idCellInFocus].setNewText("1")

    def onBtn1Clicked(self):
        self.cells[self.idCellInFocus].setNewText("2")

    def onBtn2Clicked(self):
        self.cells[self.idCellInFocus].setNewText("3")

    def onBtn3Clicked(self):
        self.cells[self.idCellInFocus].setNewText("4")

    def onBtn4Clicked(self):
        self.cells[self.idCellInFocus].setNewText("5")

    def onBtn5Clicked(self):
        self.cells[self.idCellInFocus].setNewText("6")

    def onBtn6Clicked(self):
        self.cells[self.idCellInFocus].setNewText("7")
```

```
    def onBtn7Clicked(self):
        self.cells[self.idCellInFocus].setNewText("8")

    def onBtn8Clicked(self):
        self.cells[self.idCellInFocus].setNewText("9")

    def onBtnXClicked(self):
        self.cells[self.idCellInFocus].setNewText("")
```

Методы `onBtn0Clicked()`...`onBtn8Clicked()` и `onBtnXClicked()` выполнят занесение в ячейку цифры, соответствующей нажатой кнопке **1**...**9**, или удаление цифры, если была нажата кнопка **X**. Как они работают, понятно без дополнительных пояснений.

Листинг 32.5. Метод `onClearAllCells()`

```
    def onClearAllCells(self):
        for cell in self.cells:
            cell.setText("")
            cell.clearCellBlock()
```

Метод `onClearAllCells()` очистит поле судоку. В нем мы перебираем все имеющиеся в поле ячейки, каждую очищаем от занесенной в нее цифры и переводим в разблокированное состояние вызовом метода `clearCellBlock()` класса `MyLabel`.

Листинг 32.6. Метод `onBlockCell()`

```
    def onBlockCell(self):
        cell = self.cells[self.idCellInFocus]
        if cell.text() == "":
            QtWidgets.QMessageBox.information(self, "Судоку",
                         "Нельзя блокировать пустую ячейку")
        else:
            if cell.isCellChange:
                cell.setCellBlock()
```

Метод `onBlockCell()` будет блокировать активную ячейку. Сначала он проверит, есть ли в ней цифра (получить ее можно вызовом унаследованного от суперкласса `QLabel` метода `text()`), поскольку блокировать можно только ячейки с цифрами. Если в ячейке нет цифры, на экран будет выведено информационное окно с соответствующим предупреждением. В противном случае будет выполнена проверка, хранится ли в атрибуте `isCellChange` блокируемой ячейки значение `True` (т. е. не заблокирована ли уже эта ячейка), и, если это так, ячейка блокируется вызовом метода `setCellBlock()` класса `MyLabel`.

В этом методе мы предварительно извлекаем из списка активную ячейку, сохраняем ее в переменной и в дальнейшем используем для доступа к активной ячейке именно эту переменную. Такой подход в случае, если нужно несколько раз обращаться к значению какого-либо атрибута класса или элемента списка, позволяет несколько повысить быстродействие, поскольку обращение к переменной выполняется быстрее, чем к атрибуту класса или элементу списка.

Листинг 32.7. Метод `onBlockCells()`

```
def onBlockCells(self):
    for cell in self.cells:
        if cell.text() and cell.isCellChange:
            cell.setCellBlock()
```

Метод `onBlockCells()`, блокирующий все ячейки, выполняет перебор всех ячеек и блокирует любую из них, если она содержит цифру и еще не заблокирована.

Листинг 32.8. Метод `onClearBlockCell()`

```
def onClearBlockCell(self):
    cell = self.cells[self.idCellInFocus]
    if not cell.isCellChange:
        cell.clearCellBlock()
```

Метод `onClearBlockCell()`, предназначенный для разблокирования активной ячейки, предварительно проверит, хранится ли в ее атрибуте `isCellChange` значение `False` (т. е. заблокирована ли эта ячейка). И только после этого он вызывает метод `clearCellBlock()` класса `MyLabel`, чтобы разблокировать ячейку.

Листинг 32.9. Метод `onClearBlockCells()`

```
def onClearBlockCells(self):
    for cell in self.cells:
        if not cell.isCellChange:
            cell.clearCellBlock()
```

Метод `onClearBlockCells()`, разблокирующий все ячейки поля, очень прост, и вы, уважаемые читатели, сами поймете, как он работает.

32.3.4. Класс *MainWindow*: основное окно приложения

Класс `MainWindow` представляет основное окно приложения «Судоку». Это окно включает компонент `Widget`, т. е. поле судоку, главное меню, панель инструментов и строку состояния.

Класс основного окна, в силу его сложности, мы также будем писать по частям. В настоящий момент мы реализуем в нем только часть всех функций приложения: операции очистки поля, выхода из приложения, блокировки и разблокировки ячеек и получения справочных сведений. Мы также реализуем сохранение и восстановление местоположения окна и начнем разработку функции печати. Остальная функциональность будет добавлена потом.

Класс `MainWindow` мы сделаем производным от класса главного окна `MainWindow`. Это позволит нам без проблем разместить в окно поле судоку, создать главное меню, панель инструментов и строку состояния.

Подумаем, какие атрибуты мы объявим в классе основного окна:

- `sudoku` — экземпляр класса `Widget`, представляющий компонент поля судоку. Нам придется работать с этим компонентом в других методах класса `MainWindow`;

- `settings` — экземпляр класса `QSettings`, предназначенный для загрузки и сохранения настроек приложения. Мы применим здесь подход, показанный в листинге 31.2, при котором сохранение настроек выполняется в методе `closeEvent()`;
- `printer` — экземпляр класса `QPrinter`, представляющий принтер. Мы сразу же создадим его в конструкторе класса окна, чтобы потом не вносить в код конструктора слишком много правок.

Что касается методов класса `MainWindow`, то мы объявим всего три:

- `closeEvent()` — этот метод следует переопределить, если нужно выполнять какие-либо действия непосредственно перед закрытием окна. Внутри его мы выполним сохранение местоположения окна;
- `aboutInfo()` — выведет на экран стандартное диалоговое окно со сведениями о приложении «Судоку»;
- конструктор сформирует интерфейс приложения, загрузит и установит сохраненное ранее местоположение окна, создаст принтер, а также привяжет к окну таблицу стилей, которая укажет специфическое оформление для ячеек поля судоку и набора кнопок, который находится ниже поля.

Весь код класса `MainWindow` мы сохраним в файле mainwindow.py в каталоге modules. Его мы также рассмотрим по частям.

32.3.4.1. Конструктор класса *MainWindow*

И в этом случае конструктор — самый сложный метод рассматриваемого класса. Разберем его по фрагментам.

```
from PyQt5 import QtCore, QtGui, QtWidgets, QtPrintSupport
```

Не забываем импортировать все нужные модули, включая модуль `QtPrintSupport`.

```
from modules.widget import Widget
```

Импортируем класс поля судоку `Widget` из модуля widget.py, который мы сохранили в каталоге modules.

```
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent,
                  flags=QtCore.Qt.Window |
                  QtCore.Qt.MSWindowsFixedSizeDialogHint)
        self.setWindowTitle("Судоку 2.0.0")
```

Для создаваемого окна указываем флаг `MSWindowsFixedSizeDialogHint`, запрещающий изменение его размеров. В самом деле, поле судоку у нас имеет фиксированный размер, и, если пользователь получит возможность изменять размеры окна, это будет выглядеть странно.

```
        self.setStyleSheet(
            "QFrame QPushButton {font-size:10pt;font-family:Verdana;"
            "color:black;font-weight:bold;}"
            "MyLabel {font-size:14pt;font-family:Verdana;"
            "border:1px solid #9AA6A7;}")
```

Указываем для окна таблицу стилей, которая задаст представление для следующих элементов управления:

- для кнопок из набора, расположенного под полем судоку, — черный полужирный шрифт Verdana размером 10 пунктов (так мы сделаем эти кнопки заметнее);

♦ для ячеек поля судоку — шрифт Verdana размером 14 пунктов и темно-серую сплошную рамку толщиной в 1 пиксел.

```
self.settings = QtCore.QSettings("Прохоренок и Дронов", "Судоку")
self.printer = QtPrintSupport.QPrinter()
```

Создаем объекты хранилища настроек и принтера.

```
self.sudoku = Widget()
self.setCentralWidget(self.sudoku)
```

Создаем экземпляр класса `Widget`, сохраняем его в атрибуте `sudoku` и помещаем в окно в качестве центрального.

```
menuBar = self.menuBar()
toolBar = QtWidgets.QToolBar()
```

Получаем доступ к уже имеющемуся в окне главному меню и создаем панель инструментов.

```
myMenuFile = menuBar.addMenu("&Файл")
```

Создаем меню **Файл**.

```
action = myMenuFile.addAction(QtGui.QIcon(r"images/new.png"),
                    "&Новый", self.sudoku.onClearAllCells,
                    QtCore.Qt.CTRL + QtCore.Qt.Key_N)
```

Создаем пункт **Новый** меню **Файл**.

Для создания пунктов меню мы используем разновидность метода `addAction()` класса `QMenu`, которая в качестве параметров принимает значок, название пункта, обработчик и комбинацию клавиш. На основе всего этого метод формирует действие (экземпляр класса `QAction`), создает связанный с ним пункт меню и возвращает это действие в качестве результата (подробности — в *разд. 27.2.2*). Мы сохраним полученное действие в переменной, чтобы впоследствии создать на панели инструментов связанную с ним кнопку и задать для него текст подсказки, выводящийся в строке состояния.

В качестве обработчика для этого действия мы указываем метод `onClearAllCells()` компонента поля судоку (класса `Widget`).

```
toolBar.addAction(action)
action.setStatusTip("Создание новой, пустой головоломки")
```

Создаем на основе только что подготовленного действия кнопку **Новый** панели инструментов и задаем для действия текст подсказки, выводящейся в строке состояния.

```
myMenuFile.addSeparator()
toolBar.addSeparator()

action = myMenuFile.addAction("&Выход", QtWidgets.qApp.quit,
                    QtCore.Qt.CTRL + QtCore.Qt.Key_Q)
action.setStatusTip("Завершение работы приложения")
```

Добавляем в меню и на панель инструментов разделители и точно таким же образом создаем пункт **Выход** меню **Файл**. В качестве обработчика указываем метод `quit()` приложения.

```
myMenuEdit = menuBar.addMenu("&Правка")

action = myMenuEdit.addAction("&Блокировать",
                   self.sudoku.onBlockCell, QtCore.Qt.Key_F2)
action.setStatusTip("Блокирование активной ячейки")
```

```
        action = myMenuEdit.addAction(QtGui.QIcon(r"images/lock.png"),
                          "Б&локировать все",
                          self.sudoku.onBlockCells, QtCore.Qt.Key_F3)
        toolBar.addAction(action)
        action.setStatusTip("Блокирование всех ячеек")

        action = myMenuEdit.addAction("&Разблокировать",
                          self.sudoku.onClearBlockCell,
                          QtCore.Qt.Key_F4)
        action.setStatusTip("Разблокирование активной ячейки")

        action = myMenuEdit.addAction(QtGui.QIcon(r"images/unlock.png"),
                          "Р&азблокировать все",
                          self.sudoku.onClearBlockCells,
                          QtCore.Qt.Key_F5)
        toolBar.addAction(action)
        action.setStatusTip("Разблокирование всех ячеек")
```

Создаем меню **Правка**, его пункты **Блокировать**, **Блокировать все**, **Разблокировать** и **Разблокировать все** и соответствующие им кнопки панели инструментов. В качестве обработчиков действий указываем, соответственно, методы `onBlockCell()`, `onBlockCells()`, `onClearBlockCell()` и `onClearBlockCells()` компонента поля судоку.

```
        myMenuAbout = menuBar.addMenu("&Справка")

        action = myMenuAbout.addAction("О &программе...", self.aboutInfo)
        action.setStatusTip("Получение сведений о приложении")

        action = myMenuAbout.addAction("О &Qt...",
                                QtWidgets.qApp.aboutQt)
        action.setStatusTip("Получение сведений о библиотеке Qt")
```

Создаем меню **Справка** и его пункты **О программе** и **О Qt**. Для действия, связанного с первым пунктом, указываем в качестве обработчика метод `aboutInfo()` класса `MainWindow`, который скоро напишем, а для действия, связанного с вторым пунктом, — статический метод `aboutQt()` приложения.

```
        toolBar.setMovable(False)
        toolBar.setFloatable(False)
        self.addToolBar(toolBar)
```

Запрещаем панели инструментов перемещаться внутри области, в которой она находится, и выноситься в отдельное окно (для нашего простого приложения это ни к чему), и добавляем ее в окно. Поскольку в вызове метода `addToolBar()` окна мы не указали область, панель будет помещена в верхнюю часть окна.

```
        statusBar = self.statusBar()
        statusBar.setSizeGripEnabled(False)
        statusBar.showMessage("\"Судоку\" приветствует вас", 20000)
```

Получаем доступ к строке состояния, убираем из нее маркер изменения размера (если его оставить, пользователь сможет изменить размеры окна, а это нам совсем не нужно) и выводим приветственное сообщение, которое будет отображаться в течение 20 секунд.

```
        if self.settings.contains("X") and self.settings.contains("Y"):
            self.move(self.settings.value("X"), self.settings.value("Y"))
```

Проверяем, находятся ли в хранилище настроек значения с именами `X` (горизонтальная координата левого верхнего угла окна) и `Y` (его вертикальная координата), и, если это так, извлекаем эти значения и позиционируем окно по этим координатам. (Размеры окна хранить не имеет смысла, поскольку они неизменны.)

32.3.4.2. Остальные методы класса *MainWindow*

Нам осталось рассмотреть два метода класса основного окна `MainWindow`.

Листинг 32.10. Метод `closeEvent()`

```
    def closeEvent(self, evt):
        g = self.geometry()
        self.settings.setValue("X", g.left())
        self.settings.setValue("Y", g.top())
```

Метод `closeEvent()` будет автоматически вызван при закрытии окна. В нем мы выполняем сохранение текущих координат левого верхнего угла окна.

Листинг 32.11. Метод `aboutInfo()`

```
    def aboutInfo(self):
        QtWidgets.QMessageBox.about(self, "О программе",
            "<center>\"Судоку\" v2.0.0<br><br>"
            "Программа для просмотра и редактирования судоку<br><br>"
            "(c) Прохоренок Н.А., Дронов В.А. 2011-2018 гг.")
```

Метод `aboutInfo()` выведет стандартное окно со сведениями о приложении. Здесь комментировать нечего.

32.3.5. Запускающий модуль

Теперь напишем запускающий модуль, который, собственно, и запустит приложение на выполнение. Мы сохраним его в файле start.pyw непосредственно в каталоге sudoku. Код запускающего модуля представлен в листинге 32.12.

Листинг 32.12. Запускающий модуль

```
from PyQt5 import QtGui, QtWidgets
import sys

from modules.mainwindow import MainWindow

app = QtWidgets.QApplication(sys.argv)
app.setWindowIcon(QtGui.QIcon(r"images/svd.png"))
window = MainWindow()
window.show()
sys.exit(app.exec_())
```

Здесь мы создаем экземпляр класса `QApplication`, представляющий приложение, указываем для него значок, подготовленный ранее, создаем экземпляр только что написанного класса `MainWindow`, представляющего основное окно, выводим окно на экран и запускаем приложение.

Выполним запуск приложения, запустив модуль **start.pyw** любым знакомым нам способом: щелчком мышью на самом файле или нажатием клавиши <F5> в окне IDLE, в котором открыт этот модуль. Проверим, работает ли активизация ячеек по щелчку мыши и посредством клавиш-стрелок поставим в какие-либо ячейки цифры и удалим их. Попробуем заблокировать и разблокировать ячейки. Наконец, очистим поле судоку, вызовем окна сведений о приложении и Qt и закроем приложение.

32.3.6. Копирование и вставка головоломок

Итак, базовые функции приложения мы реализовали. Настало время заняться дополнительными.

Начнем мы с реализации копирования головоломок в буфер обмена и их вставки оттуда. Код, который мы напишем для этого, будет впоследствии использован также для сохранения головоломок в файлы и их последующей загрузки.

32.3.6.1. Форматы данных

Но сначала следует определиться, в каких форматах головоломки будут копироваться в буфер обмена. В *разд. 32.2* мы решили, что таковых будет три: полный, компактный и предназначенный для Microsoft Excel.

В любом случае данные будут копироваться в виде строки, состоящей только из цифр от 0 до 9.

- *Полный формат* — строка длиной 162 символа. Каждая пара цифр, содержащаяся в ней, представляет сведения об одной ячейке:
  - первая цифра — обозначает состояние блокировки ячейки: 0 — разблокирована, 1 — заблокирована;
  - вторая цифра — это, собственно, цифра, которая установлена в ячейке, или 0, если ячейка не имеет цифры.

  Сведения о ячейках записываются последовательно, без каких-либо разделителей: первая пара цифр хранит сведения о ячейке с номером 0, вторая — о ячейке 1, третья — о ячейке 2 и т. д.
- *Компактный формат* — строка длиной 81 символ. Она содержит только цифры, установленные в ячейках; 0 обозначает отсутствие цифры в соответствующей ячейке. Первая цифра соответствует ячейке 0, вторая — ячейке 1 и т. д.
- *Формат для Excel* — более длинная строка. Каждую ячейку представляет одна цифра — та, что установлена в нее (состояние блокировки не сохраняется). Если ячейка не имеет цифры, сохраняется пустая строка. Цифры или пустые строки, соответствующие всем ячейкам одной строки поля судоку, отделяются друг от друга символами табуляции. Наборы цифр или пустых строк, соответствующие отдельным строкам, отделяются друг от друга последовательностями символов возврата каретки и перевода строки. Этой же последовательностью символов завершается сама строка с данными.

Чтобы реализовать копирование и вставку головоломок, нам потребуется добавить новые методы в классы `Widget` и `MainWindow`.

32.3.6.2. Реализация копирования и вставки в классе *Widget*

В классе `Widget` мы объявим четыре новых метода:

- `getDataAllCells()` — возвращает данные о головоломке в полном формате;
- `getDataAllCellsMini()` — возвращает данные о головоломке в компактном формате;
- `getDataAllCellsExcel()` — возвращает данные о головоломке в формате для Excel.

  Эти методы не будут непосредственно помещать данные о головоломке в буфер обмена, а станут лишь формировать их. Занесением готовых данных в буфер обмена займутся методы класса `MainWindow`, которые мы напишем потом.

  Методы `getDataAllCells()` и `getDataAllCellsMini()` мы впоследствии используем для подготовки данных, которые будут записываться в файлы;
- `setDataAllCells()` — принимает с единственным параметром данные о головоломке, представленные в полном или компактном формате (формат распознается автоматически), и выполняет их вставку.

  Опять же, этот метод не будет непосредственно извлекать данные из буфера обмена, а станет лишь выполнять вставку данных, извлеченных оттуда специальными методами класса `MainWindow`. Один из этих методов, помимо всего прочего, выполнит преобразование полученных из буфера обмена данных, представленных в формате Excel, в полный формат перед тем, как передать их методу `setDataAllCells()`.

  Позднее мы используем этот метод для вставки данных о головоломке, прочитанных из файла.

Листинг 32.13. Метод `getDataAllCells()`

```
def getDataAllCells(self):
    listAllData = []
    for cell in self.cells:
        listAllData.append("0" if cell.isCellChange else "1")
        s = cell.text()
        listAllData.append(s if len(s) == 1 else "0")
    return "".join(listAllData)
```

Метод `getDataAllCells()` возвращает строку с данными о головоломке в полном формате.

Формировать строку с копируемыми данными можно двумя способами. Первый способ заключается в том, что сначала объявляется переменная, хранящая пустую строку, а потом к этой строке постепенно добавляются символы, хранящие сведения о ячейках. Но в таком случае при очередном добавлении символов предыдущая строка останется в оперативной памяти — в результате эти «мусорные» строки будут постепенно накапливаться, засоряя память. Разумеется, рано или поздно они будут удалены особой подсистемой Python, носящей название *сборщика мусора*, но произойдет это только тогда, когда приложение будет простаивать.

Поэтому, чтобы избежать «замусоривания» памяти, мы используем второй способ: объявим пустой список, в который будем добавлять строки с символами, представляющие сведения об очередной ячейке, а под конец сформируем строку, составленную из элементов этого списка (для этого мы вызовем у пустой строки метод `join()`, передав ему наш список). Это и будут данные о головоломке, записанные в полном формате.

В самом методе `getDataAllCells()` нет ничего особо сложного. Мы перебираем ячейки поля судоку в цикле. У каждой ячейки мы выясняем состояние блокировки (оно, как мы помним, хранится в атрибуте `isCellChange` класса `MyLabel`). Если ячейка разблокирована, добавляем в список строку `"0"`, в противном случае — `"1"`. Далее мы вызовом унаследованного метода `text()` извлекаем текстовое содержимое ячейки, проверяем, равна ли ее длина единице (т. е. установлена ли в ячейку цифра), и, если так, добавляем в список строку с этим содержимым (т. е. с установленной в ячейку цифрой). В противном случае добавляем строку `"0"`. Под конец мы формируем строку, составленную из элементов списка и представляющую собой данные, которые должны быть помещены в буфер обмена. Эту строку мы возвращаем в качестве результата.

Листинг 32.14. Метод `getDataAllCellsMini()`

```
    def getDataAllCellsMini(self):
        listAllData = []
        for cell in self.cells:
            s = cell.text()
            listAllData.append(s if len(s) == 1 else "0")
        return "".join(listAllData)
```

Код метода `getDataAllCellsMini()`, возвращающий данные о головоломке в компактном формате, работает аналогично. Вы, уважаемые читатели, и сами разберетесь, как.

Листинг 32.15. Метод `getDataAllCellsExcel()`

```
    def getDataAllCellsExcel(self):
        numbers = (9, 18, 27, 36, 45, 54, 63, 72)
        listAllData = [self.cells[0].text()]
        for i in range(1, 81):
            listAllData.append("\r\n" if i in numbers else "\t")
            listAllData.append(self.cells[i].text())
        listAllData.append("\r\n")
        return "".join(listAllData)
```

Метод `getDataAllCellsExcel()`, что станет формировать данные для вставки в Excel, немногим сложнее. Мы создаем кортеж, содержащий номера ячеек, перед которыми в результирующую строку вместо символа табуляции нужно вставить возврат каретки и перевод строки, — как видим, это номера первых ячеек в каждой строке, кроме первой. Затем мы создаем список из единственного элемента — содержимого самой первой ячейки: цифры или пустой строки. Далее мы в цикле перебираем все ячейки от второй до последней. Если номер очередной ячейки входит в объявленный ранее кортеж (т. е. начинается новая строка поля судоку), мы добавляем в список возврат каретки и перевод строки, в противном случае — символ табуляции. Далее добавляем в список содержимое ячейки. Наконец, завершаем формируемые данные возвратом каретки и переводом строки и формируем строку, составленную из элементов списка.

Листинг 32.16. Метод `setDataAllCells()`

```
    def setDataAllCells(self, data):
        l = len(data)
        if l == 81:
```

```
            for i in range(0, 81):
                if data[i] == "0":
                    self.cells[i].setText("")
                    self.cells[i].clearCellBlock()
                else:
                    self.cells[i].setText(data[i])
                    self.cells[i].setCellBlock()
            self.onChangeCellFocus(0)
        elif l == 162:
            for i in range(0, 162, 2):
                if data[i] == "0":
                    self.cells[i // 2].clearCellBlock()
                else:
                    self.cells[i // 2].setCellBlock()
                self.cells[i // 2].setText("" if data[i + 1] == "0" ↳
                else data[i + 1])
            self.onChangeCellFocus(0)
```

Метод `setDataAllCells()`, вставляющий данные о головоломке в поле судоку, будет самым сложным. Ведь сначала он должен выяснять, в каком формате представлены данные — в полном или компактном.

Данные, предназначенные для вставки, метод получает с единственным параметром, и данные эти представлены в виде строки. Следовательно, узнать их формат мы можем, выяснив длину строки с данными:

- если длина строки с данными равна 81-му символу, данные представлены в компактном формате. В этом случае мы перебираем все символы в полученной строке. Если очередной символ представляет собой цифру `"0"`, мы очищаем и разблокируем соответствующую ячейку, в противном случае заносим очередной символ (которым станет цифра) в ячейку и блокируем ее;
- если же длина строки равна 162-м символам, данные представлены в полном формате. Мы точно так же перебираем в цикле символы этой строки, но уже через один, извлекая тем самым на каждом проходе каждый четный символ (для чего применим цикл `for i in range(0, 162, 2)`), — этим символом станет обозначение состояния блокировки соответствующей ячейки. Номер соответствующей символу ячейки мы можем получить, разделив номер очередного символа на 2 нацело (применив оператор `//`). Если извлеченный символ является цифрой `"0"`, мы разблокируем ячейку, в противном случае — блокируем. Далее мы извлекаем и проверяем следующий символ строки: если это цифра `"0"`, очищаем ячейку, а если цифра, отличная от `"0"`, заносим ее в ячейку.

В любом случае после вставки данных мы делаем активной самую первую (левую верхнюю) ячейку поля судоку.

32.3.6.3. Реализация копирования и вставки в классе *MainWindow*

В классе `MainWindow` мы внесем дополнения в код конструктора и объявим шесть новых методов:

- `onCopyData()` — копирует в буфер обмен данные в полном формате;
- `onCopyDataMini()` — копирует в буфер обмена данные в компактном формате;

- `onCopyDataExcel()` — копирует в буфер обмена данные в формате для Excel;
- `onPasteData()` — вставляет из буфера обмена данные, представленные в полном или компактном формате, предварительно проверив эти данные на корректность;
- `onPasteDataExcel()` — вставляет из буфера обмена данные, представленные в формате для Excel, предварительно проверив эти данные на корректность;
- `dataErrorMsg()` — выводит сообщение о том, что предназначенные для вставки данные имеют неправильный формат.

Но начнем мы с того, что в самое начало файла, где находятся выражения импорта, добавим выражение, импортирующее модуль для поддержки регулярных выражений:

```
import re
```

Регулярные выражения очень помогут нам реализовать проверку вставляемых данных на корректность.

Доработаем конструктор. Все необходимые добавления показаны в листинге 32.17 (добавленный код выделен полужирным шрифтом).

Листинг 32.17. Конструктор (дополнения)

```
    def __init__(self, parent=None):
        . . .
        myMenuEdit = menuBar.addMenu("&Правка")

        action = myMenuEdit.addAction(QtGui.QIcon(r"images/copy.png"),
                             "К&опировать", self.onCopyData,
                             QtCore.Qt.CTRL + QtCore.Qt.Key_C)
        toolBar.addAction(action)
        action.setStatusTip("Копирование головоломки в буфер обмена")

        action = myMenuEdit.addAction("&Копировать компактно",
                                      self.onCopyDataMini)
        action.setStatusTip("Копирование в компактном формате")

        action = myMenuEdit.addAction("Копировать &для Excel",
                             self.onCopyDataExcel)
        action.setStatusTip("Копирование в формате MS Excel")

        action = myMenuEdit.addAction(QtGui.QIcon(r"images/paste.png"),
                             "&Вставить", self.onPasteData,
                             QtCore.Qt.CTRL + QtCore.Qt.Key_V)
        toolBar.addAction(action)
        action.setStatusTip("Вставка головоломки из буфера обмена")

        action = myMenuEdit.addAction("Вставить &из Excel",
                             self.onPasteDataExcel)
        action.setStatusTip("Вставка головоломки из MS Excel")

        myMenuEdit.addSeparator()
        toolBar.addSeparator()
```

```
        action = myMenuEdit.addAction("&Блокировать",
                               self.sudoku.onBlockCell, QtCore.Qt.Key_F2)
        . . .
```

Здесь мы добавляем в начало меню **Правка** пять пунктов: **Копировать**, **Копировать компактно**, **Копировать для Excel**, **Вставить** и **Вставить из Excel**. Указываем для них в качестве обработчиков перечисленные ранее методы, а также добавляем нужные кнопки в панель инструментов.

Листинг 32.18. Методы `onCopyData()`, `onCopyDataMini()` и `onCopyDataExcel()`

```
    def onCopyData(self):
        QtWidgets.QApplication.clipboard().setText(
            self.sudoku.getDataAllCells())

    def onCopyDataMini(self):
        QtWidgets.QApplication.clipboard().setText(
            self.sudoku.getDataAllCellsMini())

    def onCopyDataExcel(self):
        QtWidgets.QApplication.clipboard().setText(
            self.sudoku.getDataAllCellsExcel())
```

Методы `onCopyData()`, `onCopyDataMini()` и `onCopyDataExcel()`, копирующие данные, очень просты. Они всего лишь помещают в буфер обмена результат, возвращенный, соответственно, методами `getDataAllCells()`, `getDataAllCellsMini()` и `getDataAllCellsExcel()` класса поля судоку (листинг 32.18).

Листинг 32.19. Метод `onPasteData()`

```
    def onPasteData(self):
        data = QtWidgets.QApplication.clipboard().text()
        if data:
            if len(data) == 81 or len(data) == 162:
                r = re.compile(r"[^0-9]")
                if not r.match(data):
                    self.sudoku.setDataAllCells(data)
                    return
        self.dataErrorMsg()
```

Метод `onPasteData()` вставляет данные в полном или компактном формате (листинг 32.19). Перед тем как вызвать метод `setDataAllCells()` класса поля судоку, передав ему данные для вставки, он выполняет их проверку. Сначала он удостоверяется, что данные для вставки вообще есть (не равны пустой строке), потом — что их длина равна 81-му или 162-м символам. Далее он выполняет последнюю проверку — выясняет, не присутствует ли в строке символ, отличный от цифр 0...9. Для этого он создает регулярное выражение, совпадающее с любым из таких символов (`[^0-9]`), и выполняет поиск в строке с данными посредством быстро выполняющегося метода `match()`. (Более подробно о работе с регулярными выражениями рассказывалось в *главе 7*.)

И только если все проверки выполнены, вызывается метод `setDataAllCells()`, и ему передается строка с данными для вставки. После чего сразу же выполняется выход из метода.

Если же какая-либо проверка завершилась неудачей, будет выполнено самое последнее выражение — вызов метода `dataErrorMsg()`, выводящего сообщение об ошибке. Мы напишем этот метод потом.

Листинг 32.20. Метод `onPasteDataExcel()`

```
    def onPasteDataExcel(self):
        data = QtWidgets.QApplication.clipboard().text()
        if data:
            data = data.replace("\r", "")
            r = re.compile(r"([0-9]?[\t\n]){81}")
            if r.match(data):
                result = []
                if data[-1] == "\n":
                    data = data[:-1]
                dl = data.split("\n")
                for sl in dl:
                    dli = sl.split("\t")
                    for sli in dli:
                        if len(sli) == 0:
                            result.append("00")
                        else:
                            result.append("0" + sli[0])
                data = "".join(result)
                self.sudoku.setDataAllCells(data)
                return
        self.dataErrorMsg()
```

Метод `onPasteDataExcel()` вставит из буфера обмена данные, представленные в формате для Excel, разумеется, также выполнив необходимые проверки (листинг 32.20). Сначала он убедится, что данные для вставки есть, и для удобства их дальнейшей проверки и обработки удалит из них символы возврата каретки (для этого можно использовать метод `replace()` класса `str`, указав у этого метода первым параметром удаляемый символ, а вторым — пустую строку).

Полученная нами строка представляет собой набор из строго 81-й комбинации двух символов: цифры от 0...9, которая может присутствовать в единственном числе или отсутствовать, и символа табуляции или перевода строки. Это правило прекрасно формализуется регулярным выражением `([0-9]?[\t\n]){81}`. Мы сравниваем с ним строку с данными и выполняем дальнейшие манипуляции, только если сравнение выполняется.

Сначала подготавливаем пустой список, в который будем помещать отдельные строки — фрагменты вставляемой головоломки. Удаляем из полученной строки с данными завершающий символ перевода строки, если он там есть. Разбиваем эту строку по символам перевода строки, воспользовавшись методом `split()` класса `str`, и получаем список строк, представляющих отдельные строки поля судоку. Перебираем этот список и каждую из имеющихся в нем строк тем же методом разбиваем по символам табуляции, получив список строк, каждая из которых представляет сведения об одной ячейке. Перебираем этот список.

Если очередной его элемент-строка пуст (т. е. ячейка не имеет цифры), добавляем в список строку "00", где первая цифра обозначает, что ячейка не заблокирована, а вторая — отсутствие цифры в ячейке. Если же элемент не пуст, значит, он представляет собой цифру, которую следует занести в ячейку, и мы добавляем в список строку вида "0<Эта цифра>. Наконец, объединяем все элементы списка в строку, передаем ее методу setDataAllCells() класса поля судоку и выполняем возврат из метода.

Последнее выражение, выполняющееся, если какая-либо проверка из описанных ранее завершилась неудачей, вызовет все тот же метод dataErrorMsg(), который выведет сообщение о неправильном формате данных.

Осталось написать этот метод. Его код приведен в листинге 32.21 — как видим, он очень прост.

Листинг 32.21. Метод dataErrorMsg()

```
    def dataErrorMsg(self):
        QtWidgets.QMessageBox.information(self, "Судоку",
                   "Данные имеют неправильный формат")
```

Запустим приложение и проверим, как работает копирование и вставка данных в разных форматах. Проще всего сделать это, занеся цифры в некоторые ячейки поля судоку, выполнив копирование в каком-либо формате, очистив поле и произведя вставку. После этого поле судоку должно выглядеть так же, как перед копированием.

32.3.7. Сохранение и загрузка данных

Настала пора заняться средствами для сохранения головоломок в файлах и загрузки их оттуда. Сохранять головоломки мы будем в тех же форматах, в каких они копировались в буфер обмена, — это позволит нам использовать написанные в *разд. 32.3.6.2* методы класса Widget, выполняющие копирование данных.

Чтобы дать нашему приложению возможность сохранять и загружать данные, мы добавим в класс MainWindow следующие методы:

- onOpenFile() — загрузит сохраненную в файле головоломку;
- onSave() — сохранит головоломку в файл в полном формате;
- onSaveMini() — сохранит головоломку в файл в компактном формате;
- saveSVDFile() — этот метод будет вызываться обоими предыдущими методами для выполнения собственно сохранения данных в файл. Эти данные он будет получать с единственным параметром.

И внесем исправления в код конструктора — их можно увидеть в листинге 32.22 (добавленный код выделен полужирным шрифтом).

Листинг 32.22. Конструктор (дополнения)

```
    def __init__(self, parent=None):
        . . .
        action = myMenuFile.addAction(QtGui.QIcon(r"images/new.png"),
                             "&Новый", self.sudoku.onClearAllCells,
                             QtCore.Qt.CTRL + QtCore.Qt.Key_N)
```

```
        toolBar.addAction(action)
        action.setStatusTip("Создание новой, пустой головоломки")

        action = myMenuFile.addAction(QtGui.QIcon(r"images/open.png"),
                             "&Открыть...", self.onOpenFile,
                             QtCore.Qt.CTRL + QtCore.Qt.Key_O)
        toolBar.addAction(action)
        action.setStatusTip("Загрузка головоломки из файла")

        action = myMenuFile.addAction(QtGui.QIcon(r"images/save.png"),
                             "Со&хранить...", self.onSave,
                             QtCore.Qt.CTRL + QtCore.Qt.Key_S)
        toolBar.addAction(action)
        action.setStatusTip("Сохранение головоломки в файле")

        action = myMenuFile.addAction("&Сохранить компактно...",
                                      self.onSaveMini)
        action.setStatusTip(
            "Сохранение головоломки в компактном формате")

        myMenuFile.addSeparator()
        toolBar.addSeparator()
        . . .
```

Этот код добавит в меню **Файл**, между пунктом **Новый** и разделителем, пункты **Открыть**, **Сохранить** и **Сохранить компактно**. В качестве обработчиков указаны описанные ранее методы. Также выполняется добавление еще двух кнопок в панель инструментов.

Листинг 32.23. Метод `onOpenFile()`

```
    def onOpenFile(self):
        fileName = QtWidgets.QFileDialog.getOpenFileName(self,
                   "Выберите файл", QtCore.QDir.homePath(),
                   "Судоку (*.svd)")[0]
        if fileName:
            data = ""
            try:
                with open(fileName, newline="") as f:
                    data = f.read()
            except:
                QtWidgets.QMessageBox.information(self, "Судоку",
                          "Не удалось открыть файл")
                return
            if len(data) > 2:
                if data[-1] == "\n":
                    data = data[:-1]
                if len(data) == 81 or len(data) == 162:
                    r = re.compile(r"[^0-9]")
```

```
                if not r.match(data):
                    self.sudoku.setDataAllCells(data)
                    return
            self.dataErrorMsg()
```

В методе `onOpenFile()`, загружающем данные из файла (листинг 32.23), мы выводим стандартное диалоговое окно открытия файла, указав в качестве начального каталог пользовательского профиля. Если пользователь выбрал файл и нажал кнопку **Открыть**, мы в блоке обработки исключения открываем этот файл для чтения и читаем его содержимое. Если файл прочитать не удалось, и было сгенерировано исключение, мы выводим соответствующее сообщение и выполняем возврат из метода.

Если данные были прочитаны, мы проверяем, имеют ли они длину 81 или 162 символа и не включают ли в себя символы, отличные от цифр 0...9. Если это так, мы передаем загруженные данные все тому же методу `setDataAllCells()` класса поля судоку и выполняем возврат.

Если же все эти проверки не увенчаются успехом, выполняется последнее выражение, которое вызовет метод `dataErrorMsg()` класса `MainWindow`, написанный нами ранее.

Листинг 32.24. Методы `onSave()` и `onSaveMini()`

```
    def onSave(self):
        self.saveSVDFile(self.sudoku.getDataAllCells())

    def onSaveMini(self):
        self.saveSVDFile(self.sudoku.getDataAllCellsMini())
```

Методы `onSave()` и `onSaveMini()`, сохраняющие данные в файл (листинг 32.24), очень просты — они лишь вызывают метод `saveSVDFile()`, передав ему результат, возвращенный методами, соответственно, `getDataAllCells()` и `getDataAllCellsMini()` класса поля судоку.

Листинг 32.25. Метод `saveSVDFile()`

```
    def saveSVDFile(self, data):
        fileName = QtWidgets.QFileDialog.getSaveFileName(self,
                    "Выберите файл", QtCore.QDir.homePath(),
                    "Судоку (*.svd)")[0]
        if fileName:
            try:
                with open(fileName, mode="w", newline="") as f:
                    f.write(data)
                self.statusBar().showMessage("Файл сохранен", 10000)
            except:
                QtWidgets.QMessageBox.information(self, "Судоку",
                          "Не удалось сохранить файл")
```

Метод `saveSVDFile()`, непосредственно выполняющий сохранение данных (листинг 32.25), также несложен — мы выводим стандартное диалоговое окно сохранения файла. Если пользователь задал имя файла для сохранения и нажал кнопку **Сохранить**, мы открываем

файл на запись (если файл с заданным именем отсутствует, он будет создан), записываем в него данные, полученные с параметром, и выводим в строке состояния сообщение об успехе. Открытие файла и запись в него мы выполняем в блоке обработки исключений — в случае возникновения исключения на экран будет выведено сообщение об ошибке записи.

Запустим приложение, поставим в некоторые ячейки цифры, сохраним головоломку в полном формате, очистим поле судоку и загрузим сохраненную головоломку. После чего попробуем сохранить и загрузить головоломку в компактном формате.

32.3.8. Печать и предварительный просмотр

Последнее, что мы добавим в приложение «Судоку», — это средства для печати, предварительного просмотра головоломок и настройки печатной страницы. Здесь нам понадобится внести изменения в классы `Widget`, `MainWindow` и определить новый класс `PreviewDialog`, представляющий диалоговое окно предварительного просмотра.

Условимся, что головоломка будет печататься в том же виде, в каком представлена на экране. Ячейки будут иметь размеры 30 × 30 пикселов, иметь темно-серую рамку, оранжевый или светло-серый цвет фона. Цифры в ячейках будут выводиться черным цветом, шрифтом Verdana размером 14 пунктов и выравниваться по середине.

32.3.8.1. Реализация печати в классе *Widget*

Все действия по формированию печатного представления головоломки мы будем выполнять в классе поля судоку `Widget`. Для этого мы определим в нем метод `print()`, который в качестве единственного параметра получит принтер, на котором должна быть выполнена печать и который представляется экземпляром класса `QPrinter`.

Код метода `print()` не очень велик, но требует развернутых пояснений. Мы рассмотрим его по частям.

```
def print(self, printer):
    penText = QtGui.QPen(QtGui.QColor(MyLabel.colorBlack), 1)
    penBorder = QtGui.QPen(QtGui.QColor(QtCore.Qt.darkGray), 1)
    brushOrange = QtGui.QBrush(QtGui.QColor(MyLabel.colorOrange))
    brushGrey = QtGui.QBrush(QtGui.QColor(MyLabel.colorGrey))
```

Сразу же создаем два пера: для вывода цифр (черное) и рамок ячеек (темно-серое) и две кисти: оранжевую и светло-серую.

```
    painter = QtGui.QPainter()
    painter.begin(printer)
```

Начинаем печать.

```
    painter.setFont(QtGui.QFont("Verdana", pointSize=14))
```

Указываем шрифт для вывода цифр в ячейках.

```
    i = 0
```

Объявляем переменную, в которой будет храниться номер печатаемой в настоящий момент ячейки.

```
    for j in range(0, 9):
```

Запускаем цикл, который будет перебирать числа из диапазона 0...8 включительно. Эти числа будут представлять номера печатаемых строк поля судоку.

```
        for k in range(0, 9):
```

Внутри этого цикла запускаем другой, аналогичный, который будет перебирать номера ячеек текущей строки.

```
                x = j * 30
                y = k * 30
```

Вычисляем координаты левого верхнего угла печатаемой в настоящий момент ячейки. Горизонтальную координату мы можем получить, взяв номер текущей строки и умножив его на ширину ячейки (30 пикселов). Вертикальная координата вычисляется аналогично на основе номера текущей ячейки текущей строки и высоты ячейки (также 30 пикселов).

```
                painter.setPen(penBorder)
```

Теперь нам нужно вывести сам квадратик, создающий ячейку. Задаем темно-серое перо для печати рамки этого квадратика.

```
                painter.setBrush(brushGrey if ↳
                self.cells[i].bgColorDefault == MyLabel.colorGrey ↳
                else brushOrange)
```

Если для фона ячейки задан светло-серый цвет, задаем светло-серое перо, в противном случае — оранжевое.

```
                painter.drawRect(x, y, 30, 30)
```

Выводим квадратик.

```
                painter.setPen(penText)
```

Задаем черное перо, которым будет выведена цифра.

```
                painter.drawText(x, y, 30, 30, QtCore.Qt.AlignCenter,
                                 self.cells[i].text())
```

Выводим поверх квадратика цифру, установленную в ячейку.

```
                i += 1
```

Увеличиваем значение номера текущей ячейки на единицу, чтобы на следующем проходе цикла напечатать следующую ячейку.

```
        painter.end()
```

И завершаем печать.

32.3.8.2. Класс *PreviewDialog*: диалоговое окно предварительного просмотра

Класс `PreviewDialog` реализует функциональность диалогового окна предварительного просмотра головоломки перед печатью (рис. 32.3). Это окно позволит нам просматривать головоломку в масштабе 1:1, увеличивать, уменьшать масштаб и сбрасывать его к изначальному значению.

Код класса `PreviewDialog` мы сохраним в файле previewdialog.py в каталоге modules. Он довольно велик и использует примечательные приемы программирования, о которых следует поговорить, поэтому давайте рассмотрим его по частям.

```
from PyQt5 import QtCore, QtWidgets, QtPrintSupport

class PreviewDialog(QtWidgets.QDialog):
```

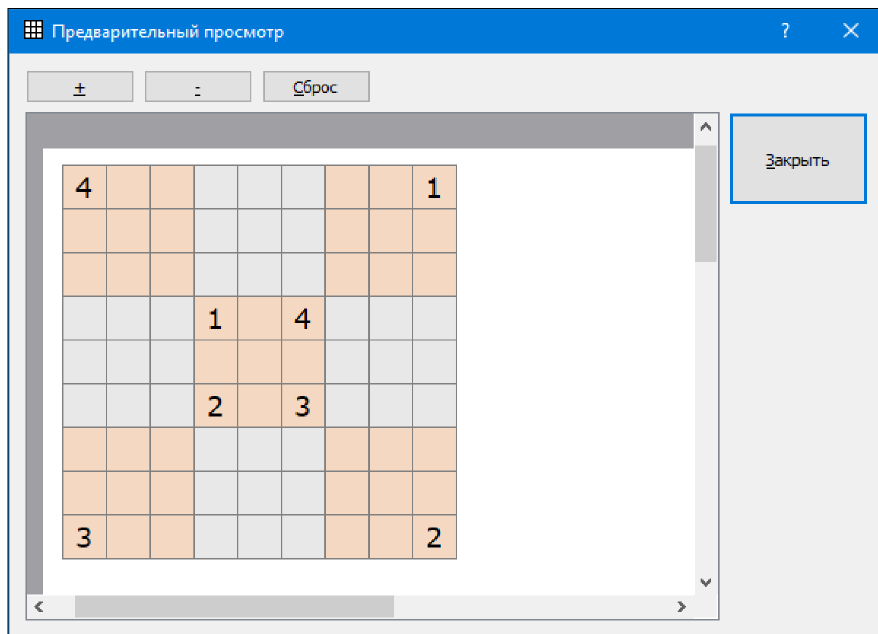

Рис. 32.3. Диалоговое окно предварительного просмотра

Окно предварительного просмотра мы делаем подклассом класса `Dialog`. Это позволит нам без особых проблем сделать размеры окна неизменяемыми, а само окно — модальным.

```
    def __init__(self, parent=None):
        QtWidgets.QDialog.__init__(self, parent)
        self.setWindowTitle("Предварительный просмотр")
        self.resize(600, 400)
        vBox = QtWidgets.QVBoxLayout()
```

Интерфейс окна включит две горизонтальные группы элементов управления, расположенные друг над другом (см. рис. 32.3). Поэтому для размещения групп мы создадим вертикальный контейнер `QVBoxlayout`.

```
        hBox1 = QtWidgets.QHBoxLayout()
```

Верхняя группа будет содержать три кнопки: для увеличения, уменьшения и сброса масштаба. Поскольку элементы в группе должны располагаться по горизонтали, используем для их расстановки контейнер `QHBoxLayout`.

```
        btnZoomIn = QtWidgets.QPushButton("&+")
        btnZoomIn.setFocusPolicy(QtCore.Qt.NoFocus)
        hBox1.addWidget(btnZoomIn, alignment=QtCore.Qt.AlignLeft)
        btnZoomOut = QtWidgets.QPushButton("&-")
        btnZoomOut.setFocusPolicy(QtCore.Qt.NoFocus)
        hBox1.addWidget(btnZoomOut, alignment=QtCore.Qt.AlignLeft)
        btnZoomReset = QtWidgets.QPushButton("&Сброс")
        btnZoomReset.setFocusPolicy(QtCore.Qt.NoFocus)
        btnZoomReset.clicked.connect(self.zoomReset)
        hBox1.addWidget(btnZoomReset, alignment=QtCore.Qt.AlignLeft)
```

Создаем все эти три кнопки и добавляем их в контейнер. Для каждой кнопки указываем, что она не может принимать фокус ввода, вызвав у нее метод `setFocusPolicy()` с параметром `NoFocus`, — таким образом, мы создадим в нашем диалоговом окне подобие панели инструментов. Также для всех трех кнопок указываем выравнивание по левому краю.

У кнопки сброса масштаба мы сразу же указываем в качестве обработчика сигнала `clicked` метод `zoomReset()` класса `PreviewDialog`. У остальных кнопок мы пока не указываем обработчики этого сигнала.

```
hBox1.addStretch()
```

Добавляем в горизонтальный контейнер растягивающуюся область, чтобы все кнопки оказались прижатыми к левому краю контейнера.

```
vBox.addLayout(hBox1)
```

Добавляем сам горизонтальный контейнер в вертикальный.

```
hBox2 = QtWidgets.QHBoxLayout()
```

Создаем еще один горизонтальный контейнер, в котором будут выводиться панель предварительного просмотра и кнопка **Закрыть**.

```
self.ppw = QtPrintSupport.QPrintPreviewWidget(parent.printer)
self.ppw.paintRequested.connect(parent.sudoku.print)
hBox2.addWidget(self.ppw)
```

Создаем панель предварительного просмотра (экземпляр класса `QPrintPreviewWidget`) и связываем его сигнал `paintRequested` с методом `print()` компонента поля судоку, иначе эта панель ничего не выведет. Компонент поля судоку хранится в атрибуте `sudoku` основного окна приложения, а основное окно мы без проблем получим с параметром `parent` конструктора. Напоследок добавляем панель просмотра во второй горизонтальный контейнер.

```
btnZoomIn.clicked.connect(self.ppw.zoomIn)
btnZoomOut.clicked.connect(self.ppw.zoomOut)
```

Создав панель предварительного просмотра, связываем с ее методами `zoomIn()` и `zoomOut()` сигналы `clicked` кнопок увеличения и уменьшения масштаба.

```
box = QtWidgets.QDialogButtonBox(
    QtWidgets.QDialogButtonBox.Close, QtCore.Qt.Vertical)
```

Создаем контейнер для кнопок, которые обычно выводятся в диалоговом окне, добавляем в него кнопку закрытия и располагаем по вертикали.

```
btnClose = box.button(QtWidgets.QDialogButtonBox.Close)
btnClose.setText("&Закрыть")
btnClose.setFixedSize(96, 64)
btnClose.clicked.connect(self.accept)
```

Получаем только что созданную в контейнере кнопку, задаем для нее надпись **Закрыть**, увеличенные размеры и связываем ее сигнал `clicked` с методом `accept()`, унаследованным нашим диалоговым окном от класса `Dialog`.

```
hBox2.addWidget(box,
      alignment=QtCore.Qt.AlignRight | QtCore.Qt.AlignTop)
```

Добавляем контейнер с кнопками во второй горизонтальный контейнер, указав выравнивание по правой и верхней границам, т. е. расположение в верхнем правом углу.

```
vBox.addLayout(hBox2)
self.setLayout(vBox)
```

Добавляем второй горизонтальный контейнер в вертикальный контейнер и помещаем последний в окно.

```
        self.zoomReset()
```

Указываем масштаб по умолчанию — 1:1, вызвав метод `zoomReset()` окна.

```
    def zoomReset(self):
        self.ppw.setZoomFactor(1)
```

И, не откладывая дела в долгий ящик, определим этот метод.

32.3.8.3. Реализация печати в классе *MainWindow*

Теперь внесем необходимые дополнения в код класса `MainWindow`, чтобы наше приложение наконец-то овладело печатным мастерством. Нам понадобится добавить в конструктор код, создающий необходимые пункты меню и кнопки панели инструментов, и три метода:

- `onPrint()` — выполняет печать головоломки;
- `onPreview()` — выводит на экран созданное ранее окно предварительного просмотра;
- `onPageSetup()` — производит настройку параметров страницы.

Поскольку предварительный просмотр у нас будет выполняться в только что созданном окне `PreviewDialog`, нам следует добавить в самое начало кода класса `MainWindow` выражение, выполняющее импорт класса `PreviewDialog`:

```
from modules.previewdialog import PreviewDialog
```

Дополнения, которые следует внести в код конструктора класса `MainWindow`, показаны в листинге 32.26 (добавленный код, как обычно, выделен полужирным шрифтом).

Листинг 32.26. Конструктор (дополнения)

```
    def __init__(self, parent=None):
        . . .
        action = myMenuFile.addAction("&Сохранить компактно...",
                                      self.onSaveMini)
        action.setStatusTip(
            "Сохранение головоломки в компактном формате")

        myMenuFile.addSeparator()
        toolBar.addSeparator()

        action = myMenuFile.addAction(QtGui.QIcon(r"images/print.png"),
                           "&Печать...", self.onPrint,
                           QtCore.Qt.CTRL + QtCore.Qt.Key_P)
        toolBar.addAction(action)
        action.setStatusTip("Печать головоломки")

        action = myMenuFile.addAction(QtGui.QIcon(r"images/preview.png"),
                           "П&редварительный просмотр...",
                           self.onPreview)
        toolBar.addAction(action)
        action.setStatusTip("Предварительный просмотр головоломки")
```

```
        action = myMenuFile.addAction("П&араметры страницы...",
                              self.onPageSetup)
        action.setStatusTip("Задание параметров страницы")

        myMenuFile.addSeparator()
        toolBar.addSeparator()

        action = myMenuFile.addAction("&Выход", QtWidgets.qApp.quit,
                              QtCore.Qt.CTRL + QtCore.Qt.Key_Q)
        . . .
```

Между пунктами главного меню, вызывающими файловые операции, мы вставляем разделитель и пункты **Печать**, **Предварительный просмотр** и **Параметры страницы**. Не забываем добавить соответствующие кнопки на панель инструментов.

Листинг 32.27. Методы `onPrint()`, `onPreview()` и `onPageSetup()`

```
    def onPrint(self):
        pd = QtPrintSupport.QPrintDialog(self.printer, parent=self)
        pd.setOptions(QtPrintSupport.QAbstractPrintDialog.PrintToFile |
                      QtPrintSupport.QAbstractPrintDialog.PrintSelection)
        if pd.exec() == QtWidgets.QDialog.Accepted:
            self.sudoku.print(self.printer)

    def onPreview(self):
        pd = PreviewDialog(self)
        pd.exec()

    def onPageSetup(self):
        pd = QtPrintSupport.QPageSetupDialog(self.printer, parent=self)
        pd.exec()
```

Код методов `onPrint()`, `onPreview()` и `onPageSetup()`, производящих печать, предварительный просмотр и настройку страницы, очень прост (листинг 32.27). Единственная деталь, достойная рассмотрения: в диалоговом окне печати мы задаем только возможность указания печати в файл и выбора принтера, на котором будет выполняться печать. Остальные параметры, в частности выбор печатаемых страниц, в нашем случае не нужны.

Запустим приложение, создадим какую-либо головоломку и проверим, как работает печать, предварительный просмотр и настройка параметров страницы.

На этом работу над приложением «Судоку» можно считать завершенной. Собственно, подошел конец и книге, посвященной замечательному языку Python и не менее замечательной библиотеке PyQt.

Заключение

Вот и закончилось наше путешествие в мир Python 3 и PyQt 5. Материал книги описывает лишь базовые возможности этих замечательных программных платформ. А здесь мы расскажем, где найти дополнительную информацию, чтобы продолжить изучение.

Самыми важными источниками информации являются официальные сайты **https://www.python.org/** и **https://riverbankcomputing.com/** — на них вы найдете дистрибутивы, новости, а также ссылки на все другие ресурсы в Интернете.

На сайте **https://docs.python.org/** имеется документация по Python, которая обновляется в режиме реального времени. Язык постоянно совершенствуется, появляются новые функции, изменяются параметры, добавляются модули и т. д. Регулярно посещайте этот сайт, и вы получите самую последнюю информацию.

Поскольку библиотека PyQt является надстройкой над библиотекой Qt, следует регулярно наведываться на сайт **https://doc.qt.io/** — только там можно найти полную документацию по Qt.

В пакет установки Python входит большое количество модулей, позволяющих выполнять наиболее часто встречающиеся задачи. Однако этим возможности Python не исчерпываются — мир Python включает множество самых разнообразных модулей и целых библиотек, созданных сторонними разработчиками и доступных для свободного скачивания, — вы легко найдете их на сайте **https://pypi.python.org/pypi**. Имейте только в виду, что при выборе модуля необходимо учитывать версию Python, которая обычно указывается в составе названия дистрибутива.

Особенно необходимо отметить библиотеку PySide (**https://wiki.qt.io/PySide**), созданную специалистами компании Nokia, а ныне поддерживаемую независимыми разработчиками. Эта библиотека является полным аналогом PyQt и распространяется по лицензии LGPL. Не забывайте также о существовании других библиотек для создания графического интерфейса: wxPython (**https://www.wxpython.org/**), PyGTK (**http://www.pygtk.org/**), PyWin32 (**https://github.com/mhammond/pywin32**) и pyFLTK (**http://pyfltk.sourceforge.net/**). Обратите внимание и на библиотеку pygame (**http://www.pygame.org/**), позволяющую разрабатывать игры, и на фреймворк Django (**https://www.djangoproject.com/**), с помощью которого можно создавать веб-сайты.

Если в процессе изучения у вас возникнут какие-либо вопросы, вспомните, что в Интернете можно найти решения самых разнообразных проблем, — достаточно лишь набрать свой вопрос в строке запроса того или иного поискового портала (например, **https://www.bing.com/** или **https://www.google.ru/**).

Засим авторы прощаются с вами, уважаемые читатели, и желают успехов в нелегком, но таком увлекательном деле, как программирование!

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив с материалами, сопровождающими книгу, можно скачать с FTP-сервера издательства по ссылке **ftp://ftp.bhv.ru/9785977539784.zip** или со страницы книги на сайте **www.bhv.ru**.

Структура архива представлена в табл. П.1.

Таблица П.1. *Структура электронного архива*

Файл или каталог	Описание
Sudoku	Каталог с исходными текстами приложения «Судоку»
Readme.txt	Описание электронного архива
Listings.doc	Все листинги, представленные в книге
PyQt.doc	Более 750 дополнительных листингов, демонстрирующих возможности библиотеки PyQt 5 (к *части II* книги)

Предметный указатель

@

@abc 263
@abstractmethod 259, 263
@classmethod 258
@staticmethod 258

_

__abs__() 256
__add__() 255
__all__ 236, 241
__and__() 256
__annotations__ 230
__bases__ 251
__bool__() 254
__call__() 253
__cause__ 275
__class__ 286
__complex__() 254
__contains__() 257, 279
__debug__ 276
__del__() 248
__delattr__() 254
__delitem__() 279
__dict__ 234, 254
__doc__ 38, 39, 85
__enter__() 269
__eq__() 257
__exit__() 269, 270
__file__ 290
__float__() 254
__floordiv__() 256
__ge__() 257
__getattr__() 253, 260
__getattribute__() 254, 260
__getitem__() 278, 279
__gt__() 257
__hash__() 255
__iadd__() 255
__iand__() 257
__ifloordiv__() 256
__ilshift__() 257
__imod__() 256
__import__() 234
__imul__() 256
__index__() 254
__init__() 247
__int__() 254
__invert__() 256
__ior__() 257
__ipow__() 256
__irshift__() 257
__isub__() 255
__iter__() 278
__itruediv__() 256
__ixor__() 257
__le__() 257
__len__() 254, 278
__lshift__() 257
__lt__() 257
__mod__() 256
__mro__ 252
__mul__() 255
__name__ 231, 286
__ne__() 257
__neg__() 256
__next__() 44, 220, 273, 278, 296, 304
__or__() 257
__pos__() 256
__pow__() 256
__radd__() 255
__rand__() 257
__repr__() 255, 278
__rfloordiv__() 256
__rlshift__() 257

__rmod__() 256
__rmul__() 255
__ror__() 257
__round__() 254
__rpow__() 256
__rrshift__() 257
__rshift__() 257
__rsub__() 255
__rtruediv__() 256
__rxor__() 257
__setattr__() 254, 260
__setitem__() 279
__slots__ 261
__str__() 255, 278
__sub__() 255
__truediv__() 256
__xor__() 257

A

A0 709
A1 709
A2 709
A3 709
A4 709
A5 709
AA_UseStyleSheetPropagationInWidgetStyles 393
abc 259
Abort 617, 620
Aborted 722
about() 625
aboutQt() 625
aboutToActivate 672
aboutToHide 657
aboutToShow 657
abs() 76, 190, 256
AbsoluteSize 568
abspath() 288, 290, 310
Accept 634
accept() 349, 390, 407, 420, 421, 424, 429, 430, 607, 610, 615, 616
accepted 616, 619
Accepted 615, 616, 626, 632
acceptHoverEvents() 609
AcceptOpen 632
acceptProposedAction() 429, 430, 610, 611
acceptRichText() 470
AcceptRole 618, 619, 621
AcceptSave 632
access() 306
AccessDeniedError 677, 688
AccessError 750
AccessibleDescriptionRole 503
AccessibleTextRole 502
accumulate() 173
acos() 78
actionChanged() 427
actionGroup() 660
ActionRole 618, 621
actions() 657, 661
actionTriggered 491, 664
activated 487, 501, 516, 673
activateNextSubWindow() 669
activatePreviousSubWindow() 669
activateWindow() 379, 616
ActivationChange 409
ActivationHistoryOrder 669
activationOrder() 669
Active 384, 722
activeAction() 654, 656
ActivePython 23
activeSubWindow() 669
ActiveWindowFocusReason 414
actualSize() 582
add() 165
addAction() 654, 657, 661, 663
addActions() 657
addBindValue() 539, 540
addButton() 617, 621, 735
addCategory() 730
addDatabase() 532
addDestination() 731
addDockWidget() 650
addEllipse() 585
addFile() 581
addItem() 453, 498, 584, 731
addItems() 498
addLayout() 437, 440
addLine() 584
addLink() 731
addMedia() 686
addMenu() 653, 655
addPage() 642
addPath() 585
addPermanentWidget() 667
addPixmap() 582, 585
addPolygon() 585
addRect() 585
addRow() 442
addSection() 655
addSeparator() 655, 663, 732
addSimpleText() 585
addSpacing() 437

addStretch() 437
addSubWindow() 668
addTab() 449
addText() 585
addToGroup() 603
addToolBar() 649
addToolBarBreak() 650
AddToSelection 588
addWidget() 330, 436, 437, 439, 443, 455, 585, 663, 667
Adjust 517
adjust() 377
adjusted() 377
adjustSize() 366
AdjustToContents 500
AdjustToContentsOnFirstShow 500
AdjustToMinimumContentsLength 500
AdjustToMinimumContentsLengthWithIcon 500
AeroStyle 644
AlignAbsolute 438
AlignBaseline 438
AlignBottom 359, 438, 570
AlignCenter 359, 438, 570
AlignHCenter 332, 359, 437, 447, 570
AlignJustify 438
AlignLeft 359, 437, 438, 447, 570
alignment() 447, 474
AlignRight 359, 437, 438, 447, 570
AlignTop 359, 438, 570
AlignVCenter 359, 438, 570
all() 157
AllDockWidgetAreas 666
AllEditTriggers 515
AllFonts 502
allKeys() 747
AllLayers 589
AllNonFixedFieldsGrow 443
AllowNestedDocks 650, 651
AllowTabbedDocks 651
AllPages 702
AllPagesView 720
AllTables 534
AllToolBarAreas 663
alpha() 557, 558, 559
alphaF() 558, 559
AltModifier 419
AmPmSection 485
anchorClicked 481
and 61
angleDelta() 423
animateClick() 461
AnimatedDocks 650
answerRect() 430
Antialiasing 569, 591
any() 156
AnyFile 632
AnyKeyPressed 515
append() 117, 142, 143, 153, 470, 563
appendColumn() 507, 511
appendRow() 507, 511
appendRows() 511
ApplicationActivate 409
ApplicationDeactivate 409
ApplicationModal 382, 616
applicationName() 744
ApplicationShortcut 417, 659
Apply 617, 620
ApplyRole 618, 621
arguments() 733
argv 36, 329
ArrowCursor 424
as 233, 235, 240, 269
as_integer_ratio() 77
AscendingOrder 506, 509, 513, 520, 522, 527, 544, 546, 587
ASCII 123, 127
ascii() 96, 100
asctime() 187
asin() 78
assert 272, 276
AssertionError 272, 276
at() 541
atan() 78
atBlockEnd() 479
atBlockStart() 479
atEnd() 479
atStart() 479
AttributeError 232, 245, 254, 261, 272
audioInputs() 690
Auto 702
AutoAll 473
AutoBulletList 472
AutoConnection 398
AutoNone 472
AutoText 459, 621, 644
availableGeometry() 369, 370
availableMetaData() 678
availablePrinterNames() 721
availablePrinters() 721
availableRedoSteps() 475
availableSizes() 582
availableUndoSteps() 475
AverageBitRateEncoding 693

B

back() 494, 643
BackButton 421, 644
Background 384
backgroundColor() 497
BackgroundLayer 589
BackgroundPixmap 645
BackgroundRole 502
backspace() 464
BacktabFocusReason 414
backward() 480
backwardAvailable 481
backwardHistoryCount() 481
BannerPixmap 645
BaseException 272
basename() 311
baseSize() 366
Batched 518
beforeDelete 549
beforeInsert 549
beforeUpdate 549
begin() 566, 703
beginEditBlock() 480
beginGroup() 747
beginReadArray() 749
beginWriteArray() 748
BevelJoin 560
Bin 489
bin() 75, 254
bindValue() 539, 540
black 359, 556
Black 473, 476, 564
black() 558
blackF() 558
blake2b() 121
blake2s() 121
block() 479
blockCount() 476
blockCountChanged 477
BlockingIOError 324
BlockingQueuedConnection 398
blockNumber() 479
blockSignals() 399
BlockUnderCursor 479
blue 556
blue() 557
blueF() 558
blurRadius() 604, 605
blurRadiusChanged 605
Bold 473, 476, 564
bold() 565
BOM 27, 292
bool 41
Bool 536
bool() 48, 59, 254
bottom() 377
BottomDockWidgetArea 650, 652, 666
bottomLeft() 377
BottomLeftCorner 652
bottomRight() 377
BottomRightCorner 652
BottomToolBarArea 649, 663
BottomToTop 438, 490
boundedTo() 373
boundingRect() 564, 566, 570, 594
BoundingRectShape 601
boundValue() 543
boundValues() 543
Box 448
break 66, 70, 71
brightness() 684
bspTreeDepth() 584
BspTreeIndex 584
buffer 299
BufferedMedia 676
BufferingMedia 676
bufferStatus() 676
bufferStatusChanged 679
builtins 37
BusyCursor 424
button() 421, 422, 608, 618, 622, 644
buttonClicked 622
buttonDownPos() 608
buttonDownScenePos() 608
buttonDownScreenPos() 608
buttonRole() 618, 622
buttons() 421, 423, 608, 610, 611, 618, 622, 736
buttonText() 644, 646
Byte Order Mark 27, 292
bytearray 42, 83, 112, 116
ByteArray 536
bytearray() 49, 116
bytes 42, 83, 112
bytes() 49, 112, 113
BytesIO 305

C

CacheBackground 590
CacheNone 590
calendar 184, 201
Calendar 201

calendar() 206
Cancel 617, 619
cancel() 642
CancelButton 644
CancelButtonOnLeft 645
canceled 642
canPaste() 470
capitalize() 105
cascadeSubWindows() 669
casefold() 105
CaseInsensitive 475, 527, 528
CaseSensitive 527, 528
categories() 731
ceil() 78
cellRect() 440
center() 94, 377, 562
centerOn() 593
centralWidget() 649
chain() 173
changed 589, 661
changeEvent() 410
changeOverrideCursor() 424
Char 536
characterCount() 476
chdir() 290, 319
checkBox() 622
Checked 463, 502, 512
checkedAction() 661
checkOverflow() 488
checkState() 463, 512
CheckStateRole 502
child() 504, 512
ChildAdded 409
childGroups() 747
childItems() 597
childKeys() 745
ChildPolished 409
ChildRemoved 409
chmod() 307
choice() 80, 157
chr() 105
Cicero 701, 704, 709, 723
class 244
ClassicStyle 643–645
cleanText() 483
cleanupPage() 643, 645, 647
Clear 526
clear() 155, 166, 182, 318, 428, 431, 450, 460, 464, 470, 474, 482, 499, 508, 543, 564, 578, 585, 619, 654, 655, 663, 686, 731, 732, 736, 745
ClearAndSelect 526
clearEditText() 500
clearFocus() 414, 587, 596
clearHistory() 481
clearMessage() 359, 667
clearOverlayIcon() 726
clearSelection() 479, 514, 526, 588
clearSpans() 519
clearUndoRedoStacks() 475
clearValues() 548
click() 403, 462, 737
clicked 447, 461, 462, 488, 516, 619, 737
clicked() 330
clickedButton() 622
ClickFocus 415
Clipboard 408
clipboard() 431
clone() 513
cloneDatabase() 534
Close 390, 408, 617, 620, 659
close() 294, 301, 302, 317, 390, 413, 534
closeActiveSubWindow() 669
closeAllSubWindows() 669
closeAllWindows() 390
closed 298, 302
ClosedHandCursor 424
closeEvent() 390, 413
cmath 77
Cmyk 559
CoarseTimer 404
collapse() 522
collapseAll() 522
collapsed 522
collateCopies() 701
collidesWithItem() 598
collidingItems() 586, 597
Color 702
color() 604, 606
color0 556, 577
color1 556, 577
colorChanged 605, 606
colorCount() 704
colorMode() 702
colorNames() 556
column() 504, 510
columnCount() 440, 507, 510
columnIntersectsSelection() 525
Columns 526
columnSpan() 519
columnWidth() 519, 521
combinations() 169
combinations_with_replacement() 170
combine() 198

commit() 533
CommitButton 644
compile() 122, 131, 132, 136, 137
completeChanged 647
CompleteHtmlSaveFormat 496
complex 42, 73
complex() 254
compress() 171
connect() 330, 395, 398, 403
ConnectionError 324, 537
connectionName() 533
connectionNames() 534
ConstantBitRateEncoding 693
ConstantQualityEncoding 692
contains() 378, 534, 535, 597, 745
ContainsItemBoundingRect 586–588, 591
ContainsItemShape 586–588, 591
contentsChange 477
contentsChanged 477
contentsSize() 496
Context 673
ContextMenu 409
contextMenu() 672
contextMenuEvent() 466, 470, 657
ContiguousSelection 514
continue 71
contrast() 684
ControlModifier 419
convertFromImage() 576
convertTo() 559
convertToFormat() 580
copy 144, 177, 183
Copy 419, 659
copy() 144, 165, 167, 177, 183, 307, 466, 470, 576, 580
copy2() 308
CopyAction 426
copyAvailable 471
copyCount() 701
copyfile() 307
corner() 652
cos() 77
count() 69, 106, 156, 168, 444, 451, 454, 456, 499, 523, 535, 564, 732, 736
CoverWindow 364
createEditor() 528
createItemGroup() 585, 603
createMaskFromColor() 576
createPopupMenu() 649
createStandardContextMenu() 466, 470
CreationOrder 669
Critical 619, 673
critical() 624
CrossCursor 424
CrossPattern 386, 561
cssclasses 204
ctime() 187, 193, 200
Current 526
currentChanged 444, 451, 454, 526, 634
CurrentChanged 514
currentCharFormat() 474
currentCharFormatChanged 471
currentColumnChanged 526
currentCpuArchitecture() 741
currentFont() 473, 502
currentFontChanged 502
currentId() 643
currentIdChanged 645
currentIndex() 443, 451, 453, 499, 503, 513, 526, 687
currentIndexChanged 501, 688
CurrentItemInLoop 686
CurrentItemOnce 686
currentMedia() 675, 687
currentMediaChanged 678, 688
currentMessage() 667
CurrentPage 702
currentPage() 643, 720
currentPageChanged 488
currentRowChanged 526
currentSection() 485
currentSectionIndex() 485
currentSubWindow() 669
currentText() 499
currentUrlChanged 634
currentWidget() 443, 451, 454
cursor() 424
cursorBackward() 466
cursorForPosition() 477
cursorForward() 466
cursorPosition() 466
cursorPositionChanged 467, 471, 477
cursorWordBackward() 466
cursorWordForward() 466
Custom 732
CustomButton1 644, 645
CustomButton2 644, 645
CustomButton3 644, 645
customButtonClicked 645
CustomDashLine 560
customEvent() 433
CustomizeWindowHint 364
CustomZoom 720
Cut 659

cut() 466, 470
cyan 556
cyan() 558
cyanF() 558
cycle() 169

D

darkBlue 556
darkCyan 556
darker() 558
darkGray 556
darkGreen 556
darkMagenta 556
darkRed 556
darkYellow 556
DashDotDotLine 520, 560
DashDotLine 520, 560
DashLine 520, 560
data() 428, 503, 505, 508, 512, 544, 660
database() 534
databaseText() 537
dataChanged 431, 549
date 189, 191
Date 536
date() 199, 484
dateChanged 485
datetime 184, 188, 189, 196
DateTime 536
dateTime() 484
dateTimeChanged 485
day 192, 198
day_abbr 207
day_name 207
days 189, 190
DaySection 485
dbm 317
Dec 489
decimal 54, 74
decode() 115, 118
DecorationRole 502
deepcopy() 144, 177, 183
def 210, 245
defaultAction() 427, 656, 664
defaultAudioInput() 690
defaultDuplexMode() 722
defaultPageSize() 722
defaultPrinter() 721
defaultPrinterName() 721
defaultSectionSize() 523
defaultTextColor() 602
defaultValue() 536
degrees() 78
del 50, 118, 155, 179
del() 464
delattr() 246
DelayedPopup 665
deleteChar() 480
deletePreviousChar() 480
deleter() 262
delta() 609
DemiBold 473, 476, 564
Dense1Pattern 386, 561
Dense2Pattern 386, 561
Dense3Pattern 386, 561
Dense4Pattern 386, 561
Dense5Pattern 386, 561
Dense6Pattern 386, 561
Dense7Pattern 386, 561
depth() 576, 580
DescendingOrder 506, 509, 513, 520, 522, 527, 544, 546, 587
description() 722, 733
Deselect 526
deselect() 465
Designer 335
Desktop 364
desktop() 368
Destination 732
destroyItemGroup() 585, 603
DestructiveRole 618, 621
Detail 632
DevicePixel 704
Dialog 364
dict 42
dict() 175, 177
dict_items 181
dict_keys 66, 180
dict_values 180
Didot 701, 704, 709, 723
difference() 163, 167
difference_update() 163
digest() 120
digest_size 120
dir() 39, 234
DirectConnection 398
Directory 632
directory() 633
directoryEntered 634
directoryUrl() 633
directoryUrlEntered 634
DirEntry 322
dirname() 290, 311
disableBlurBehindWindow() 741

Disabled 384
DisabledBackButtonOnLastPage 645
Discard 617, 620
discard() 165
disconnect() 399
dismissOnClick() 737
display() 488
DisplayRole 502
displayText() 464
divmod() 77
dockLocationChanged 666
dockOptions() 651
dockWidgetArea() 650
DockWidgetClosable 666
DockWidgetFloatable 666
DockWidgetMovable 666
DockWidgetVerticalTitleBar 666
Document 479
document() 474, 602
documentMargin() 476
documentTitle() 470
done() 615, 616
DontConfirmOverwrite 633
DontMaximizeSubWindowOnActivation 670
DontUseNativeDialog 639
DontWrapRows 442
DOTALL 122, 124
DotLine 520, 560
Double 536
DoubleClick 673
doubleClicked 516
DoubleClicked 514
doubleClickInterval() 420
DoubleInput 627, 628
doubleValue() 627
doubleValueChanged 628
doubleValueSelected 628
Down 478
DownArrow 665
dragCursor() 427
DragDrop 515
DragEnter 408
dragEnterEvent() 429, 430, 610
DragLeave 408
dragLeaveEvent() 429, 610
dragMode() 591
DragMove 408
dragMoveEvent() 429, 610
DragOnly 515
draw() 603, 604
drawArc() 569
drawBackground() 584
drawChord() 569
drawEllipse() 569
Drawer 364
drawForeground() 584
drawImage() 572, 578
drawLine() 567
drawLines() 568
drawPicture() 574
drawPie() 569
drawPixmap() 571, 575, 577
drawPoint() 567
drawPoints() 567
drawPolygon() 569
drawPolyline() 568
drawRect() 568
drawRoundedRect() 568
drawText() 570
drivers() 534
driverText() 537
Drop 409
dropAction() 430, 611
dropEvent() 429, 430, 610
DropOnly 515
dropwhile() 171
dump() 315, 316
dumps() 119, 316
dup() 301
duplex() 701
DuplexAuto 701
DuplexLongSide 701
DuplexNone 701
DuplexShortSide 701
duration() 675, 691
durationChanged 678, 692
dx() 562
dy() 562

E

e 77
East 450, 651, 670
Eclipse 16
edit() 514
editingFinished 467, 482
EditKeyPressed 515
EditRole 502
editTextChanged 501
ElideLeft 450, 515
ElideMiddle 450, 515
ElideNone 450, 515
ElideRight 450, 515
elif 64

ellipsis 43
Ellipsis 43
else 66, 70, 268
emit() 345, 401
Empty 352
empty() 353
enableBlurBehindWindow() 740
enabledChanged 604
encode() 112
encoding 299
End 478
end() 134, 466, 566, 567, 703
endArray() 749
endEditBlock() 480
endGroup() 747
EndOfBlock 478
EndOfLine 478
EndOfMedia 676
EndOfWord 478
endpos 133
endswith() 107
ensureCursorVisible() 470
EnsureVisible 515
ensureVisible() 456, 593, 598
ensureWidgetVisible() 456
Enter 408
entered 516
enterEvent() 423
Enum 282
enumerate() 69, 149
EnumMeta 284
env 28
Envelope 702
EnvelopeManual 702
EOFError 272, 313
error 679
Error 696, 722
error() 677, 688, 692
errorString() 677, 688, 692
escape() 140
eval() 35
event() 409, 417, 420, 433, 660
eventFilter() 431, 432
ExactMatch 709
exc_info() 266
except 265–268, 273
Exception 272, 274
exec() 426, 538, 543, 614, 615, 620, 626, 632, 656
exec_() 331, 342, 349, 426, 539, 543, 615, 620, 626, 632
execBatch() 540, 543
executedQuery() 543
ExistingFile 632
ExistingFiles 632
exists() 308
exit() 329, 342, 349
exp() 78
expand() 135, 139, 521
expandAll() 522
expanded 522
expandedTo() 373
Expanding 442, 445
ExpandingFieldsGrow 442
expandtabs() 94
expandToDepth() 522
extend() 117, 154
ExtendedSelection 514
ExtendedWatermarkPixmap 645
extendFrameIntoClientArea() 740
ExtraButton1 421
ExtraButton2 421

F

F_OK 307
fabs() 78
FacingPagesView 720
factorial() 78
False 41, 59
families() 565
family() 564
FastTransformation 577, 580, 601
fdopen() 301
features() 666
field() 535, 536, 643, 647
fieldIndex() 547
fieldName() 535
FieldsStayAtSizeHint 442
FileExistsError 291, 300, 324
FileName 633
fileName() 744
fileno() 297
FileNotFoundError 291, 324
filePath() 733
fileSelected 634
filesSelected 634
FileType 633
fill() 576, 579
Filled 488
fillRect() 568
filter() 152, 546
filterfalse() 171
filterSelected 634

FinalizingStatus 691
finally 268
find() 105, 470, 475
findall() 136
FindBackward 471, 476, 495
findBlock() 476
findBlockByLineNumber() 476
findBlockByNumber() 476
FindCaseSensitively 471, 476, 495
FindChildrenRecursively 437
findData() 500
FindDirectChildrenOnly 437
findItems() 509
finditer() 136, 137
findText() 494, 500
FindWholeWords 471, 476
finish() 359
FinishButton 644
finished 616
finished() 344
first() 541
firstBlock() 476
firstweekday() 205
FitInView 720
fitInView() 594, 720
FitToWidth 720
fitToWidth() 720
Fixed 445, 517, 523
FixedColumnWidth 472
FixedPixelWidth 472
flags 133
flags() 504, 512, 596
Flat 488
FlatCap 560
FlipDefault 741
FlipExcludeAbove 741
FlipExcludeBelow 741
float 42, 73, 77
float() 48, 75, 254
floor() 78
flush() 292, 297, 304
fmod() 78
FocusIn 408, 415
focusInEvent() 415, 607
focusItem() 587
focusItemChanged 590
focusNextChild() 414
focusNextPrevChild() 415
FocusOut 408, 415
focusOutEvent() 415, 607
focusPolicy() 415
focusPreviousChild() 415
focusProxy() 414, 607
focusWidget() 414, 415
fold 195, 197, 198
font() 601, 602, 660
fontFamily() 473
fontItalic() 473
fontPointSize() 473
FontRole 502
fontUnderline() 473
fontWeight() 473
for 34, 44, 65, 89, 148, 162, 166, 179, 183, 296, 304
ForbiddenCursor 424
ForceTabbedDocks 651
Foreground 384
ForegroundLayer 589
ForegroundRole 502
ForeignWindow 364
format() 95, 97, 580, 744
Format_ARGB32 579
Format_ARGB32_Premultiplied 578, 579
format_exception() 266
format_exception_only() 266
Format_Indexed8 579
Format_Invalid 578
Format_Mono 578
Format_MonoLSB 579
Format_RGB32 579
FormatError 677, 688, 692, 750
formatmonth() 202, 204
FormatNotSupportedError 688
formats() 428
formatyear() 203, 204
formatyearpage() 204
forward() 480, 494
forwardAvailable 481
ForwardButton 421
forwardHistoryCount() 481
fractions 74
frameGeometry() 367, 368
FramelessWindowHint 364
frameSize() 367
Free 517
Frequent 732
frequent() 730
Friday 487
FRIDAY 202
from 234, 239–241, 275
from_iterable() 173
fromCmyk() 558
fromCmykF() 558
fromData() 579

fromhex() 113, 117
fromHsv() 559
fromHsvF() 559
fromImage() 576, 578
fromkeys() 176
fromordinal() 192, 198
fromPage() 702
fromRgb() 557
fromRgba() 557
fromRgbF() 557
fromtimestamp() 192, 197
frozenset 43, 166
frozenset() 166
fsum() 78
Full 352
full() 353
fullmatch() 132
fullRect() 712
fullRectPixels() 712
fullRectPoints() 712
function 43, 212
functools 153, 403
FuzzyMatch 709
FuzzyOrientationMatch 709

G

geometry() 366, 368, 672
get() 178, 181, 318, 352
get_nowait() 352
getatime() 309
getattr() 232, 245
getbuffer() 305
getCmyk() 558
getCmykF() 558
getColor() 638
getCoords() 377
getctime() 309
getcwd() 319
getDouble() 630
getExistingDirectory() 634
getExistingDirectoryUrl() 634
getFont() 639
getHsv() 559
getHsvF() 559
getInt() 629
getItem() 630
getlocale() 104
getmtime() 309
getMultiLineText() 631
getOpenFileName() 635
getOpenFileNames() 636
getOpenFileUrl() 635
getOpenFileUrls() 636
getRect() 377
getrecursionlimit() 224
getrefcount() 46
getRgb() 557
getRgbF() 557
getSaveFileName() 637
getSaveFileUrl() 637
getsize() 308
getter() 262
getText() 628
getvalue() 302
glob 321
glob() 321
global 225
globalPos() 421, 423, 657
globalPosF() 423
globals() 227
globalX() 421, 423, 657
globalY() 421, 423, 657
gmtime() 184, 207
gotFocus() 415
grabKeyboard() 415, 420, 596, 607
grabMouse() 422, 596, 608
grabShortcut() 417
graphicsEffect() 603
gray 556
Grayscale 702
green 556
green() 557
greenF() 558
group() 134, 603, 747
groupdict() 134
GroupedDragging 651
groupindex 133
groups 133
groups() 134
GroupSwitchModifier 419

H

hasAcceptableInput() 468, 469
hasAlpha() 577
hasAlphaChannel() 577
hasattr() 232, 246
hasChildren() 509, 512
hasComplexSelection() 479
hasFocus() 414, 587, 596
hasFormat() 428
hashlib 120
hasHtml() 428

hasImage() 428
hasSelectedText() 460, 465
hasSelection() 479, 494, 525
hasText() 428
hasTracking() 491
hasUrls() 428
hasVisitedPage() 643
HaveCustomButton1 644, 645
HaveCustomButton2 644, 645
HaveCustomButton3 644, 645
HaveFinishButtonOnEarlyPages 645
HaveHelpButton 644, 645
HaveNextButtonOnLastPage 645
header() 521, 523
headerData() 509, 545
height() 366, 368, 372, 377, 566, 576, 579, 584, 703
heightForWidth() 445
heightMM() 703
Help 617, 620
help() 37, 38, 39
HelpButton 644
HelpButtonOnRight 645
helpRequested 619, 646
HelpRole 618, 621
HeuristicMaskShape 601
Hex 489
hex() 75, 115, 254
HexArgb 557
hexdigest() 120
HexRgb 557
hiddenSectionCount() 524
Hide 408
hide() 363, 596, 615, 616, 672, 727
hideColumn() 519, 521
hideEvent() 410
HideNameFilterDetails 633
hidePopup() 500
hideRow() 519
hideSection() 524
hideTearOffMenu() 656
HideToParent 408
HighestPriority 344
HighEventPriority 432
highlighted 481, 501
HighPriority 344, 660
HighQuality 693
HighResolution 700
historyChanged 481
historyTitle() 481
historyUrl() 481
HLine 448
home() 466, 480
homePath() 742
Horizontal 454, 489, 490, 493, 509, 523, 545, 609, 617, 650, 652, 663
horizontalHeader() 519, 522
horizontalHeaderItem() 509
horizontalScrollBar() 457
hotSpot() 427
hour 194, 195, 196, 198
hours 189
HourSection 485
hover() 660
hovered 654, 657, 661, 662
hoverEnterEvent() 608
hoverLeaveEvent() 608
hoverMoveEvent() 609
Hsl 559
Hsv 559
hsvHue() 559
hsvHueF() 559
hsvSaturation() 559
hsvSaturationF() 559
html() 428
HTMLCalendar 202, 204
hue() 684

I

IBeamCursor 424
icon() 494, 656, 658, 672, 733, 736
iconChanged 495
IconMode 517
iconSize() 650, 664
identifier() 731
Idle 722
IDLE 22, 26, 31
IdlePriority 344
if…else 62, 64, 65
Ignore 617, 620
ignore() 349, 390, 407, 420, 421, 424, 430, 607
IgnoreAction 426
IgnoreAspectRatio 372, 577, 580, 589, 684
IGNORECASE 122
Ignored 445
IgnoreSubTitles 645
image() 431
imageData() 428
imp 238
import 28, 34, 231, 234, 236, 240–242
ImportError 272
in 56, 60, 89, 156, 161, 164, 178, 181, 257
In 539

Inactive 384
Inch 701, 704, 709, 723
IndentationError 272
IndependentPages 643, 645, 647
index() 69, 106, 156, 503, 505, 508, 513, 544
IndexError 87, 134, 146, 155, 272, 279
indexesMoved 518
indexFromItem() 508
indexOf() 444, 451, 454, 456, 535
indexWidget() 514
Infinite 696
Information 619, 673
information() 622
InheritPriority 344
IniFormat 744
initializePage() 643, 645, 647
InnerJoin 552
input() 27, 35, 272
insert() 118, 154, 464, 563
insertAction() 657
insertActions() 657
InsertAfterCurrent 499
InsertAlphabetically 500
InsertAtBottom 499
InsertAtCurrent 499
InsertAtTop 499
InsertBeforeCurrent 499
insertBlock() 480
insertColumn() 507, 511
insertColumns() 507, 511
insertFragment() 480
insertFrame() 480
insertHtml() 469, 480
insertImage() 480
insertItem() 453, 498
insertItems() 499
insertLayout() 437
insertList() 480
insertMedia() 686
insertMenu() 653, 655
insertPermanentWidget() 668
insertPlainText() 469
insertRecord() 547
insertRow() 442, 507, 511
insertRows() 505, 507, 511, 546
insertSection() 655
insertSeparator() 499, 655, 663
insertSpacing() 437
insertStretch() 437
insertTab() 450
insertTable() 480
insertText() 480
insertToolBar() 649
insertToolBarBreak() 650
insertWidget() 436, 437, 443, 455, 663, 667
installEventFilter() 432
installSceneEventFilter() 611
InstantPopup 665
int 42, 73
Int 536
int() 48, 75, 254
IntEnum 283, 285
Interactive 523
InternalMove 515
InterruptedError 324
intersected() 378
intersection() 163, 167
intersection_update() 163
intersects() 378
IntersectsItemBoundingRect 586–588, 591
IntersectsItemShape 586–588, 591
interval() 406
IntInput 627, 628
intValue() 488, 627
intValueChanged 628
intValueSelected 628
Invalid 536, 559
invalidate() 589
invalidateScene() 594
InvalidMedia 676
InvalidRole 618, 621, 622
invertPixels() 581
InvertRgb 581
InvertRgba 581
invisibleRootItem() 508
io 298, 301, 305
is 45, 61, 143
is not 61
is_dir() 323
is_file() 322
is_integer() 77
is_symlink() 323
isabs() 311
isAccepted() 408
isActive() 406, 543, 567, 588
isActiveWindow() 380
IsADirectoryError 324
isalnum() 109
isalpha() 109
isAmbiguous() 417
isatty() 314
isAudioAvailable() 676
isAudioMuted() 497
isAutoRepeat() 419

isAutoValue() 536
isBackwardAvailable() 481
isCheckable() 447, 512, 659
isChecked() 447, 462–464, 659
isColumnHidden() 520, 521
isColumnSelected() 525
isCommitPage() 647
isComplete() 647
isdecimal() 109
isDefault() 722
isDescending() 537
isdigit() 109
isdir() 321
isDirty() 547
isdisjoint() 165
isDockNestingEnabled() 651
isDown() 462
isDriverAvailable() 534
isEmpty() 372, 378, 474, 535, 564, 655, 687, 732
isEnabled() 400, 462, 596, 604, 660, 662, 736
isEquivalentTo() 710
isExpanded() 522
isfile() 321
isFinalPage() 647
isFinished() 345
isFlat() 447, 737
isFloatable() 663
isFloating() 663, 666
isForwardAvailable() 481
isFullScreen() 380, 684
isGenerated() 548
isHeaderHidden() 521
isHidden() 363
isidentifier() 110
isIndexHidden() 520, 521
isinstance() 47
isInteractive() 591, 736
isItemEnabled() 453
iskeyword() 110
isleap() 206
islink() 321
isLoaded() 696
islower() 109
isMaximized() 380
isMetaDataAvailable() 677
isMinimized() 380
isModal() 382, 616
isModified() 465, 475
isMovable() 663
isMuted() 676, 696
isNull() 371, 372, 378, 475, 478, 542, 548, 562, 575, 579, 581, 721
isnumeric() 109
isocalendar() 194, 200
isoformat() 193, 196, 200
isOpen() 533
isOpenError() 533
isoweekday() 194, 200
ISOWeekNumbers 487
isPaused() 728
isPlaying() 696
isprintable() 110
isQBitmap() 576
isReadOnly() 465, 472, 536, 686
isRecursive() 355
isRedoAvailable() 466, 475
isRemote() 722
isRowHidden() 518, 519, 521
isRowSelected() 525
isRunning() 345
isSectionHidden() 524
isSeekable() 676
isSelect() 542
isSelected() 525, 596
isSeparator() 658
isShaded() 671
isSingleShot() 406
isSizeGripEnabled() 615
isslice() 172
isspace() 110
isStopped() 728
issubset() 164, 167
issuperset() 164, 167
isSystemTrayAvailable() 672
isTabEnabled() 451
isTearOffEnabled() 656
isTearOffMenuVisible() 656
istitle() 110
isTristate() 464, 512
isUndoAvailable() 466, 475
isUndoRedoEnabled() 470, 475
isupper() 109
isValid() 372, 378, 503, 541, 556, 700
isValidColor() 556
isVideoAvailable() 676
isVisible() 363, 596, 660, 662, 672, 727, 732, 736
isWindowExcludedFromPeek() 740
isWindowPeekDisallowed() 740
isWritable() 750
italic() 565
item() 508
itemAt() 586, 592
itemChange() 612
itemChanged 509

ItemChildAddedChange 613
ItemChildRemovedChange 613
ItemCursorChange 612
ItemCursorHasChanged 612
itemData 499
ItemDoesntPropagateOpacityToChildren 595
ItemEnabledChange 612
ItemEnabledHasChanged 612
ItemFlagsChange 612
ItemFlagsHaveChanged 612
itemFromIndex() 508
ItemIgnoresParentOpacity 595
ItemIgnoresTransformations 595
ItemIsDragEnabled 504
ItemIsDropEnabled 504
ItemIsEditable 504
ItemIsEnabled 504
ItemIsFocusable 587, 595, 596, 607
ItemIsMovable 595
ItemIsPanel 595
ItemIsSelectable 504, 588, 591, 595, 596
ItemIsTristate 504
ItemIsUserCheckable 504
ItemLayer 589
ItemNeverHasChildren 504
ItemOpacityChange 612
ItemOpacityHasChanged 613
ItemParentChange 613
ItemParentHasChanged 613
ItemPositionChange 612
ItemPositionHasChanged 612
items() 181, 317, 586, 592, 732
itemsBoundingRect() 584
ItemSceneChange 613
ItemSceneHasChanged 613
ItemScenePositionHasChanged 612
ItemSelectedChange 612
ItemSelectedHasChanged 612
ItemSendsGeometryChanges 612
ItemSendsScenePositionChanges 612
ItemStacksBehindParent 595
itemText() 453, 499
ItemToolTipChange 612
ItemToolTipHasChanged 612
ItemTransformChange 612
ItemTransformHasChanged 612
ItemVisibleChange 612
ItemVisibleHasChanged 612
ItemZValueChange 612
ItemZValueHasChanged 612
iter() 44
itertools 172

J

join() 102, 103, 160, 312, 352
joinPreviousEditBlock() 480

K

KeepAnchor 478
KeepAspectRatio 372, 577, 580, 589, 684
KeepAspectRatioByExpanding 372, 577, 580, 589, 684
kernelVersion() 741
key() 417, 419
KeyboardInterrupt 70, 272
keyboardModifiers() 430
KeyError 165, 166, 177, 181, 182, 272, 318
KeypadModifier 419
KeyPress 408
keyPressEvent() 419, 607
KeyRelease 408
keyReleaseEvent() 419, 607
keys() 66, 179, 180, 317
keyword 110
killTimer() 404

L

lambda 218
Landscape 701, 711, 719
last() 541
lastBlock() 476
lastError() 533, 543, 544
lastgroup 133
lastindex 133
lastInsertId() 543
lastPos() 608, 609
lastQuery() 543
lastScenePos() 608, 609
lastScreenPos() 608, 609
LC_ALL 103
LC_COLLATE 103
LC_CTYPE 104
LC_MONETARY 104
LC_NUMERIC 104
LC_TIME 104
leapdays() 206
Leave 408
leaveEvent() 423
Left 478
left() 377
LeftArrow 665
LeftButton 421

LeftDockWidgetArea 650, 651, 666
LeftJoin 552
LeftToolBarArea 649, 663
LeftToRight 438, 517
Legal 709
len() 68, 88, 101, 146, 162, 179, 254
length() 536
Letter 709
LifoQueue 351
Light 473, 476, 564
lighter() 558
lightGray 556
line() 599
lineCount() 476
LineUnderCursor 479
Link 732
LinkAction 426
linkActivated 603
linkActivated() 460
linkHovered 603
linkHovered() 461
LinksAccessibleByKeyboard 460, 472
LinksAccessibleByMouse 460, 472
list 42
List 632
list() 50, 103, 142, 144, 157
listdir() 319, 320, 321
ListMode 517
ListView 513
ljust() 95
load() 315, 316, 494, 574, 575, 579, 687
loaded 688
loadedChanged 697
LoadedMedia 676
LoadedStatus 691
loadFailed 688
loadFinished 495
loadFromData() 576, 579
Loading 696
LoadingMedia 676
LoadingStatus 691
loadProgress 495
loads() 119, 317
loadStarted 495
loadUi() 337
loadUiType() 338
locale 103
LOCALE 123
localeconv() 104
LocaleHTMLCalendar 202, 204
LocaleTextCalendar 202
localPos() 421
locals() 227
LocalTime 485
localtime() 185
location() 722
lock() 355
log() 78
log10() 78
log2() 78
logicalIndex() 524
LogoPixmap 644
LongDayNames 487
LongLong 536
LookIn 633
Loop 687
loopCount() 696
loopsRemaining() 696
loopsRemainingChanged 697
lostFocus() 415
Lower 702
lower() 104
LowestPriority 344
LowEventPriority 432
LowLatency 675
LowPriority 344, 660
LowQuality 693
lseek() 301
lstrip() 101

M

machineHostName() 742
MacStyle 644, 645
magenta 556
magenta() 558
magentaF() 558
makeAndModel() 722
maketrans() 108
manhattanLength() 371
Manual 702
ManualWrap 472
map() 150
mapFrom() 422
mapFromGlobal() 422
mapFromParent() 422
mapFromScene() 592
mapTo() 422
mapToGlobal() 422
mapToParent() 422
mapToScene() 592
marginsAdded() 375
marginsRemoved(<QMargins>) 376
mask() 387, 576

MaskInColor 576
MaskOutColor 576
MaskShape 601
match() 131, 133
MatchCaseSensitive 501
MatchContains 501
MatchEndsWith 501
matches() 419
MatchExactly 501
MatchFixedString 501
MatchRecursive 501
MatchRegExp 501
MatchStartsWith 501
MatchWildcard 501
MatchWrap 501
math 77
max 191, 194, 196, 201
max() 76, 156
Maximum 445
maximum() 728
maximumBlockCount() 476
maximumChanged 728
maximumHeight() 366
maximumPhysicalPageSize() 722
maximumSectionSize() 523
maximumSize() 366
maximumWidth() 366
MaxUser 409
MAXYEAR 191, 192, 196, 198
md5() 120
mdiArea() 671
MDI-приложение 648, 668
media() 675, 687
mediaChanged 678
mediaCount() 687
mediaObject() 684, 688
mediaStatus() 676
mediaStatusChanged 678
MemoryError 272
memoryview() 305
menu() 463, 660, 665
menuAction() 656
menuBar() 649
MenuBarFocusReason 414
MenuButtonPopup 665
menuWidget() 649
mergeBlockCharFormat() 480
mergeBlockFormat() 480
mergeCharFormat() 480
messageChanged 668
messageClicked 673
metaData() 677
metaDataAvailableChanged 679
metaDataChanged 679
MetaModifier 419
microsecond 194, 195, 197, 198
microseconds 189, 190
MidButton 421
Middle 702
MiddleButton 421
MiddleClick 673
Millimeter 701, 704, 709, 723
milliseconds 189
mimeData() 426, 429, 431, 610
MimeHtmlSaveFormat 496
min 191, 194, 196, 201
min() 76, 156
Minimum 445
minimum() 728
minimumChanged 728
MinimumExpanding 442, 445
minimumHeight() 366
minimumPhysicalPageSize() 722
minimumSectionSize() 523
minimumSize() 366
minimumSizeHint() 366
minimumWidth() 366
minute 194, 195, 196, 198
minutes 189
MinuteSection 485
MINYEAR 191, 192, 196, 198
mirrored() 581
MiterJoin 560
mkdir() 319
mktime() 185
mode 298
model() 504, 513
ModernStyle 643–645
modificationChanged 477
modifiers() 419, 421, 424, 608–611
module 43
modules 234
Monday 487
MONDAY 202
MonospacedFonts 502, 639
month 192, 198
month() 205
month_abbr 208
month_name 207
monthcalendar() 205
monthrange() 205
MonthSection 485
monthShown() 486
MouseButtonDblClick 408

MouseButtonPress 408
MouseButtonRelease 408
mouseButtons() 430
mouseDoubleClickEvent() 420, 608
MouseFocusReason 414
mouseGrabberItem() 589, 608
MouseMove 408
mouseMoveEvent() 422, 426, 608
mousePressEvent() 420, 425, 607
mouseReleaseEvent() 420, 607
Move 408
move() 308, 367
MoveAction 426
MoveAnchor 478
moveBottom() 376
moveBottomLeft() 376
moveBottomRight() 376
moveBy() 595
moveCenter() 376
moveCursor() 477
moveEvent() 411
moveMedia() 686
movePosition() 478
moveRight() 376
moveSection() 524
moveTo() 376
moveTop() 376
moveTopLeft() 376
moveTopRight() 376
MSecSection 485
msleep() 347
MSWindowsFixedSizeDialogHint 364
MULTILINE 122, 124
MultiSelection 514
mutedChanged 678, 697

N

name 284, 298, 299, 322
name() 536, 537, 557, 710
NameError 273
nativeErrorCode() 537
NativeFormat 700, 744
Netbeans 16
NetworkError 677, 688
newPage() 703
next() 44, 69, 541, 643, 687
NextBlock 478
NextButton 644
NextCell 478
NextCharacter 478
nextId() 643, 647
nextIndex() 687
NextRow 478
NextWord 478
No 617, 620
NoArrow 665
NoBackButtonOnLastPage 645
NoBackButtonOnStartPage 645
NoBrush 386, 561, 568
NoButton 421, 422, 617–619, 622
NoButtons 482, 628, 639
NoCancelButton 644, 645
NoCancelButtonOnLastPage 645
NoDefaultButton 645
NoDockWidgetArea 650
NoDockWidgetFeatures 666
NoDrag 591
NoDragDrop 515
NoDropShadowWindowHint 364
NoEcho 465, 627
NoEditTriggers 514
NoError 537, 677, 688, 692, 750
NoFocus 415
NoFrame 448
NoHorizontalHeader 487
NoIcon 619, 673
NoIndex 584
NoInsert 499
NoItemFlags 504
NoMedia 676
NoModifier 419
NoMove 477
None 41, 60, 714
NoneType 41
nonlocal 229
NonModal 382, 616
NonRecursive 355
NonScalableFonts 502, 639
NoPen 520, 560, 568
Normal 384, 465, 473, 476, 564, 627
NormalEventPriority 432
normalized() 378
NormalPriority 344, 660
NormalQuality 693
normcase() 320
normpath() 312
NoRole 618, 619, 621
North 450, 651, 670
NoSection 485
NoSelection 487, 514
not 61
not in 56, 60, 89, 156, 161, 164, 178, 181
NotADirectoryError 324

Notepad++ 16, 28
NoTextInteraction 460, 471
NoTicks 491
NotImplementedError 273
NoToAll 617, 620
NoToolBarArea 649
NoUpdate 526
NoVerticalHeader 487
now() 197
NoWrap 472
Null 696
numRowsAffected() 543

O

O_APPEND 299
O_BINARY 300
O_CREAT 300
O_EXCL 300
O_RDONLY 299
O_RDWR 299
O_SHORT_LIVED 300
O_TEMPORARY 300
O_TEXT 300
O_TRUNC 300
O_WRONLY 299
object 250
objectName() 391
Oct 489
oct() 75, 254
OddEvenFill 569
offset() 601, 605
offsetChanged 605
OffsetFromUTC 485
Ok 617, 619
oldPos() 411
oldSize() 411
oldState() 410
OnFieldChange 546
OnlyOne 702
OnManualSubmit 546, 547, 549
OnRowChange 546
opacity() 596, 606
opacityChanged 606
opacityMask() 606
opacityMaskChanged 606
Open 617, 620, 659
open() 287, 291, 293, 299, 317, 533, 615
OpenHandCursor 424
Optional 536
options() 714
or 62
ord() 105
organizationName() 744
orientation() 609, 719
os 290, 299, 306–310, 319, 322
os.path 288, 290, 308, 310, 320
OSError 273, 287, 299, 307–310, 324
OTabWidget 670
OtherFocusReason 414
Outline 488
OutOfSpaceError 692
outputFileName() 700
outputFormat() 700
overflow 489
OverflowError 185, 273
overlayAccessibleDescription() 726
overlayIcon() 726
overline() 565
overrideCursor() 425
overwriteMode() 472

P

p1() 562
p2() 562
page() 495, 642
pageAdded 646
pageCount() 720
pageIds() 642
pageLayout() 701
PageRange 702
pageRect() 704
pageRemoved 646
Paint 408
paint() 594
paintEvent() 412, 555, 567
paintRectPixels() 712
paintRectPoints() 712
paintRequested 717
palette() 384, 385
Panel 448
paperRect() 704
paperSource() 702
parent() 504, 509, 512
parentItem() 597
parentWidget() 363
partial() 403
PartiallyChecked 463, 502, 512
partition() 102
pass 210
Password 465, 627
PasswordEchoOnEdit 465, 627
paste() 466, 470

path 322
pattern 133
pause() 675, 690, 728
PausedState 675, 691
PausedStatus 691
pbkdf2_hmac() 121
PdfFormat 700
PEP-8 16
PermissionError 324
permutations() 170
pi 77
Pica 701, 704, 709, 723
pickle 119, 315–317
pickle.DEFAULT_PROTOCOL 316
pickle.HIGHEST_PROTOCOL 316
Pickler 316
pip3 327
pixel() 580
pixelDelta() 423
pixelSize() 565
pixmap() 359, 426, 431, 582, 601
Plain 448
PlainText 459, 621, 644
play() 675, 696
playbackRate() 676
playbackRateChanged 678
playingChanged 697
PlayingState 675
playlist() 675
PlusMinus 482
Point 701, 704, 709, 723
point() 563
PointingHandCursor 424
pointSize() 564
pointSizeF() 565
Polish 409
PolishRequest 409
polygon() 600
pop() 118, 155, 166, 181, 318
popitem() 182, 318
Popup 364
popup() 656
PopupFocusReason 414
popupMode() 665
Portrait 701, 711, 719
pos 133
pos() 368, 411, 421–423, 425, 429, 595, 608, 609, 611, 657
posF() 423, 430
position() 479, 676
PositionAtBottom 515
PositionAtCenter 515
PositionAtTop 515
positionChanged 678
positionInBlock() 479
possibleActions() 430, 611
postEvent() 432
pow() 76, 78
prcal() 206
PreciseTimer 404
precision() 536
Preferred 445
prepare() 539
prepareGeometryChange() 595
prepend() 563
pressed 462, 516
prettyProductName() 741
previewChanged 721
previous() 541, 687
PreviousBlock 478
PreviousCell 478
PreviousCharacter 478
previousIndex() 687
PreviousRow 478
PreviousWord 478
primaryIndex() 534, 537
primaryKey() 547
primeInsert 549
print() 32, 33, 255, 313, 471, 475, 496, 721
print_exception() 266
print_tb() 266
PrintCollateCopies 714
PrintCurrentPage 714
printer() 714, 715
printerInfo() 721
printerName() 700, 722
PrintPageRange 714
PrintSelection 714
PrintShowPageSize 714
PrintToFile 714
printToPdf() 496
priority() 344, 660
PriorityQueue 352
prmonth() 203, 205
processEvents() 343, 641
product() 170
productType() 741
productVersion() 741
progress() 726
property() 261
ProportionalFonts 502, 639
proposedAction() 430, 611
pryear() 203
purge() 140

put() 352
put_nowait() 352
PyDev 16
pydoc 37
PYQT_VERSION_STR 327
PyQt5 340
pyqtSignal() 346, 401
pyqtSlot() 397
PyScripter 16, 28
Python Shell 26
python.exe 21
PYTHONPATH 237
pythonw.exe 21
PythonWin 16
pyuic5 339

Q

QAbstractButton 461, 462, 463
QAbstractGraphicsShapeItem 599
QAbstractItemView 503, 513–516, 525, 529
QAbstractListModel 505
QAbstractPrintDialog 713
QAbstractProxyModel 527
QAbstractScrollArea 457
QAbstractSlider 490, 491, 492, 493
QAbstractSpinBox 482, 484
QAction 418, 657, 658, 660, 661
QActionGroup 660–662
qApp 329
QApplication 329, 342, 343, 383, 385, 391, 415, 420, 424, 426, 431, 564
QAudioEncoderSettings 692
QAudioRecorder 674, 689, 692
QBitmap 576–578
QBoxLayout 438
QBrush 386, 561
QButtonGroup 463
QByteArray 341, 383
QCalendarWidget 486, 487
QCheckBox 463
QClipboard 431
QCloseEvent 390, 413
QColor 384, 556, 557
QColorDialog 638
QComboBox 498–501, 504, 513
QCompleter 465
QConicalGradient 561
QContextMenuEvent 657
QCoreApplication 343, 392, 432
QCursor 424, 425
QDate 342
QDateEdit 483, 485
QDateTime 342
QDateTimeEdit 483–485
QDesktopWidget 368
QDial 492
QDialog 362, 364, 614, 616
QDialogButtonBox 616– 619
QDir 742
QDockWidget 665, 666
QDoubleSpinBox 481, 483
QDoubleValidator 468
QDrag 426, 427
QDragEnterEvent 429
QDragLeaveEvent 429
QDragMoveEvent 429, 430
QDropEvent 429
QErrorMessage 640
QEvent 407, 429, 432
QFileDialog 632–634
QFocusEvent 415, 607
QFont 473, 564
QFontComboBox 501, 502
QFontDatabase 565
QFontDialog 639
QFontMetrics 566
QFontMetricsF 566
QFormLayout 441, 442
QFrame 362, 447, 448
QGradient 386, 561
QGraphicsBlurEffect 605
QGraphicsColorizeEffect 605
QGraphicsDropShadowEffect 604, 605
QGraphicsEffect 603, 604
QGraphicsEllipseItem 585, 600
QGraphicsItem 583, 587, 588, 594, 595, 597, 603, 607, 609–612
QGraphicsItemGroup 585, 603
QGraphicsLineItem 584, 598
QGraphicsOpacityEffect 606
QGraphicsPathItem 585
QGraphicsPixmapItem 585, 600
QGraphicsPolygonItem 585, 599
QGraphicsProxyWidget 585
QGraphicsRectItem 585, 599
QGraphicsScene 583–586, 588, 589, 603
QGraphicsSceneDragDropEvent 610
QGraphicsSceneEvent 608
QGraphicsSceneHoverEvent 608, 609
QGraphicsSceneMouseEvent 607, 608
QGraphicsSceneWheelEvent 609
QGraphicsSimpleTextItem 585, 601
QGraphicsTextItem 585, 602, 603

QGraphicsView 583, 590–593
QGridLayout 438, 440
QGroupBox 445–447, 463
QHBoxLayout 435
QHeaderView 519, 521–524
QHideEvent 410
QIcon 581
QImage 555, 566, 572, 578, 579
QImageReader 383, 574
QImageWriter 575
QInputDialog 626–628
QIntValidator 468
QItemDelegate 528
QItemSelection 526
QItemSelectionModel 525, 526
QKeyEvent 419, 607
QKeySequence 417, 419, 659
QLabel 329, 387, 417, 458, 460
QLCDNumber 488, 489
QLine 562
QLinearGradient 561
QLineEdit 464, 467, 627
QLineF 562
QListView 504, 516–518
QListWidget 513
QMainWindow 362, 648, 650
QMargins 375
QMdiArea 648, 668–670
QMdiSubWindow 648, 668, 671, 672
QMediaContent 675
QMediaPlayer 674– 678
QMediaPlaylist 674, 686, 688
QMediaRecorder 690–692
QMenu 654, 657
QMenuBar 653, 654
QMessageBox 619–622
QMimeData 426, 427, 429
QModelIndex 503, 505, 544
QMouseEvent 421, 422
QMoveEvent 411
QMultimedia 692, 693
QMutex 355
QMutexLocker 357, 358
QObject 331, 395, 399, 401, 404
QPagedPaintDevice 471, 475, 555, 566, 699
QPageLayout 701, 710–712
QPageSetupDialog 699, 714
QPageSize 708–710
QPaintDevice 555, 566, 567
QPainter 555, 566, 567, 569–571, 573–575, 577, 578, 591, 592, 703
QPainterPath 594
QPaintEvent 412
QPalette 384, 385
QPdfWriter 471, 475, 555, 699, 723
QPen 559, 560
QPersistentModelIndex 504
QPicture 555, 566, 573
QPixmap 387, 555, 566, 571, 575
QPlainTextEdit 469
QPoint 370
QPointF 370
QPolygon 562, 563
QPolygonF 564
QPrintDialog 699, 712, 713
QPrinter 471, 475, 555, 699–703, 719, 722
QPrinterInfo 699, 721
QPrintPreviewDialog 699, 716
QPrintPreviewWidget 699, 719–721
QProgressBar 489, 490
QProgressDialog 641, 642
QPushButton 330, 461, 462
QRadialGradient 561
QRadioButton 463
QRect 374
QRectF 370
QRegExp 527
QRegExpValidator 468
QRegion 412
QResizeEvent 411
QScrollArea 456, 457
QScrollBar 457, 492, 493
QSettings 743
QShortcut 418
QShortcutEvent 417
QShowEvent 410
QSize 372
qsize() 352
QSizeF 370
QSizePolicy 444, 445
QSlider 490, 491
QSortFilterProxyModel 527
QSoundEffect 674, 695–697
QSpinBox 481, 483
QSplashScreen 359, 364
QSplitter 454, 455
QSqlDatabase 532, 533–535
QSqlError 533, 537
QSqlField 535, 536
QSqlIndex 534, 537
QSqlQuery 538, 540, 541, 543
QSqlQueryModel 544
QSqlRecord 534–536, 547, 548
QSqlRelation 551, 552

QSqlRelationalDelegate 554
QSqlRelationalTableModel 551, 552
QSqlTableModel 545, 546, 549
QStackedLayout 443, 444
QStackedWidget 444
QStandardItem 506, 509
QStandardItemModel 506, 507
QStatusBar 667, 668
QStatusTipEvent 660
QStringListModel 504
QStyle 384, 582
QStyledItemDelegate 528
QStyleOption 529
QStyleOptionViewItem 529
QSysInfo 741
QSystemTrayIcon 672, 673
Qt 341
QT_VERSION_STR 327
Qt4CompatiblePainting 591
QTableView 506, 513, 518, 522
QTableWidget 513
QTabWidget 448, 449–451, 651, 670
QtCore 332, 340, 344, 346, 355
QTextBlock 476
QTextBrowser 480, 481
QTextCharFormat 474, 487
QTextCursor 475, 477, 478, 479
QTextDocument 471, 474–477
QTextDocumentFragment 479
QTextEdit 469, 471–474
QTextOption 472, 570
QTextStream 342
QtGui 340
QtHelp 341
QThread 344, 347
QTime 342
QTimeEdit 483, 485
QTimer 405, 407
QtMultimedia 340
QtMultimediaWidgets 341
QtNetwork 341
QToolBar 662, 664
QToolBox 452, 453
QToolButton 664, 665
QtPrintSupport 341
QTransform 573, 593, 597
QTreeView 506, 513, 520, 522
QTreeWidget 513
QtSql 341, 532
QtSvg 341
QtWebEngineCore 340
QtWebEngineWidgets 340, 493
QtWidgets 329, 340, 359
QtWinExtras 341
QtWinExtras.QtWin 739
QtXml 341
QtXmlPatterns 341
query() 544
Question 619
question() 623
queue 351
Queue 351
QueuedConnection 346, 398
quit() 330, 342, 349, 397
QUrl 342
QValidator 468
QVariant 341, 402, 536
QVBoxLayout 435
QVideoWidget 674, 683, 684
QWebEngineDownloadItem 496
QWebEnginePage 495, 496
QWebEnginePage.FindFlags 495
QWebEngineView 493–495
QWheelEvent 423
QWidget 329, 331, 332, 362, 363, 379, 381–383, 388, 390, 391, 400, 412, 414, 417, 422, 424, 429, 435, 438, 444, 458, 555, 566, 616, 657
QWindowStateChangeEvent 410
QWinJumpList 730
QWinJumpListCategory 731
QWinJumpListItem 732
QWinTaskbarButton 725
QWinTaskbarProgress 727, 728
QWinThumbnailToolBar 735
QWinThumbnailToolButton 736, 737
QWizard 642–645
QWizardPage 642, 643, 646

R

R_OK 307
radians() 78
raise 273–276
raise_() 616
Raised 448
randint() 79
random 79, 157, 159
Random 687
random() 79, 80
randrange() 79
range 42
range() 68, 149, 159, 167
rangeChanged 491

raw_input() 35
re 122, 133
read() 295, 300, 303
readline() 295, 303
readlines() 296, 303
ReadOnly 633
Ready 696
reason() 416
Recent 732
recent() 730
record() 534, 535, 542, 544, 690
RecordingState 691
RecordingStatus 691
rect() 366, 412, 576, 579, 599, 600, 710
rectPixels() 710
rectPoints() 710
RecursionError 224, 273
Recursive 355
red 556
red() 557
redF() 558
redo() 466, 470, 475, 480
redoAvailable 471, 476
RedoStack 475
reduce() 153
region() 412
registerEventType() 409, 432
registerField() 643, 646
Registry32Format 744
Registry64Format 744
Reject 634
reject() 615, 616
rejected 616, 619
Rejected 615, 616, 626, 632
RejectRole 618, 619, 621
RelativeSize 568
released 462
releaseKeyboard() 415, 420
releaseMouse() 422
releaseShortcut() 417
reload() 238, 480, 494
relock() 358
remainingTime() 406
remove() 118, 155, 165, 308, 564, 745
removeAction() 657, 661
removeButton() 619, 622, 736
removeColumn() 511
removeColumns() 508, 511
removeDatabase() 534
removeDockWidget() 650
removeEventFilter() 432
removeFormat() 428
removeFromGroup() 603
removeItem() 453, 499, 585
removeMedia() 686
removePage() 642
removeRow() 511
removeRows() 505, 508, 511, 546
removeSceneEventFilter() 611
removeSelectedText() 479
removeSubWindow() 669
removeTab() 450
removeToolBar() 649
removeToolBarBreak() 650
removeWidget() 437, 668
rename() 308
render() 589, 594
repaint() 412, 555
repeat() 169, 209
replace() 107, 193, 195, 199
ReplaceSelection 588
replaceWidget() 437
repr() 96, 100, 191, 255
Required 536
requiredStatus() 536
Reset 617, 620
reset() 489, 642, 728
resetCachedContent() 590
resetExtendedFrame() 740
ResetRole 618, 621
resetTransform() 593, 597
Resize 408
resize() 329, 365
resizeColumnsToContents() 519
resizeColumnToContents() 519, 521
resizeDocks() 652
resizeEvent() 411
resizeRowsToContents() 519
resizeRowToContents() 519
resizeSection() 523
ResizeToContents 523
resolution 191, 194, 196, 201
resolution() 702
ResourceError 677, 692
restart() 643
restore() 573
RestoreDefaults 617, 620
restoreDockWidget() 652
restoreGeometry() 652
restoreOverrideCursor() 424, 425
restoreState() 455, 524, 634, 652
result() 615
resume() 728
retrieveData() 428

Retry 617, 620
return 210
returnPressed 467
reverse() 118, 157
reversed() 157
revert() 547
revertAll() 547
rfind() 106
Rgb 559
rgb() 557
rgba() 557
RichText 459, 621, 644
Right 478
right() 377
RightArrow 665
RightButton 421
RightDockWidgetArea 650, 651, 666
RightToLeft 438
RightToolBarArea 649, 663
rindex() 106
rjust() 95
rmdir() 319, 320
rmtree() 320
rollback() 533
rootIndex() 514
rootPath() 742
rotate() 573, 593
rotation() 597
round() 76, 254
RoundCap 560
Rounded 451, 651, 670
RoundJoin 560
row() 504, 510
rowCount() 440, 505, 507, 510, 544
rowHeight() 519, 521
rowIntersectsSelection() 525
Rows 526
rowSpan() 519
rpartition() 102
rsplit() 102
rstrip() 101
RubberBandDrag 591
RubberBandMove 671
RubberBandResize 671
run() 344
RuntimeError 224, 273

S

sample() 80, 158, 159
saturation() 684
Saturday 487
SATURDAY 202
Save 617, 620
save() 496, 573, 574, 576, 579, 687
SaveAll 617, 620
saveGeometry() 652
saveState() 455, 524, 634, 652
ScalableFonts 502, 639
scale() 372, 573, 593, 597
scaled() 373, 576, 580
scaledToHeight() 577, 581
scaledToWidth() 577, 580
scandir() 322
scene() 590, 595
sceneBoundingRect() 595
sceneEvent() 611
sceneEventFilter() 611
scenePos() 595, 608, 609, 611
sceneRect() 584, 590
sceneRectChanged 589
sceneTransform() 597
ScientificNotation 468
scope() 744
screenGeometry() 369
screenPos() 421, 608–611
ScreenResolution 700
ScrollBarAlwaysOff 457
ScrollBarAlwaysOn 457
ScrollBarAsNeeded 457
ScrollHandDrag 591
scrollPosition() 496
scrollTo() 515
scrollToBottom() 515
scrollToTop() 515
SDI-приложение 648
search() 131, 133
second 194–196, 198
seconds 189, 190
SecondSection 485
sectionAt() 485
sectionClicked 524
sectionCount() 485
sectionDoubleClicked 524
sectionMoved 525
sectionPressed 524
sectionResized 525
sectionsHidden() 524
sectionSize() 523
sectionsMovable() 524
sectionText() 485
seed() 79
seek() 298, 302, 541
SEEK_CUR 298, 301

SEEK_END 298, 301
SEEK_SET 298, 301
seekable() 298
Select 526
select() 479, 526, 546
selectAll() 465, 470, 482, 514
selectColumn() 518
SelectColumns 514
SelectCurrent 526
SelectedClicked 515
selectedColumns() 525
selectedDate() 486
selectedFiles() 633
selectedIndexes() 513, 525
selectedItems() 588
selectedRows() 525
selectedText() 460, 465, 479, 494
selectedUrls() 633
selectFile() 633
Selection 702
selection() 479, 526
selectionArea() 588
selectionChanged 467, 471, 488, 495, 526, 590
selectionEnd() 479
selectionModel() 514, 525
selectionStart() 460, 465, 479
SelectItems 514
selectRow() 518, 547
SelectRows 514
selectUrl() 633
self 245
sendEvent() 432
sep 288
Separator 732
Sequential 686
ServiceMissingError 677
set 42, 166
set() 162
setAcceptDrops() 429, 610
setAccepted() 408
setAcceptedMouseButtons() 607
setAcceptHoverEvents() 609
setAcceptMode() 632
setAcceptRichText() 470
setActionGroup() 660
setActivationOrder() 669
setActiveAction() 654, 656
setActiveSubWindow() 669
setAlignment() 332, 447, 456, 459, 465, 474, 482, 590
setAllowedAreas() 663, 666
setAllPagesViewMode() 720
setAlpha() 557
setAlphaF() 557
setAlternatingRowColors() 514
setAnimated() 522, 650
setApplicationName() 743
setArguments() 733
setArrayIndex() 748
setArrowType() 665
setAspectRatioMode() 684
setattr() 245
setAttribute 392
setAttribute() 383, 390, 422
setAudioInput() 690
setAudioMuted() 496
setAudioSettings() 690
setAutoClose() 642
setAutoDefault() 462, 616
setAutoExclusive() 462
setAutoFillBackground() 385
setAutoFormatting() 472
setAutoRaise() 665
setAutoRepeat() 461, 660
setAutoReset() 642
setAutoScroll() 515
setAutoScrollMargin() 515
setBackground() 512, 670
setBackgroundBrush() 584, 590
setBackgroundColor() 497
setBar() 642
setBaseSize() 366
setBatchSize() 518
setBinMode() 489
setBitRate() 693
setBlue() 557
setBlueF() 557
setBlurRadius() 604, 605
setBold() 565
setBottom() 375
setBottomLeft() 375
setBottomMargin() 711
setBottomRight() 375
setBrightness() 684
setBrush() 385, 560, 567, 599
setBspTreeDepth() 584
setBuddy() 417, 459
setButton() 644
setButtonLayout() 644
setButtons() 736
setButtonSymbols() 482
setButtonText() 644, 646
setCacheMode 590
setCalendarPopup() 485

setCancelButton() 642
setCancelButtonText() 626, 641
setCapStyle() 561
setCascadingSectionResizes() 523
setCaseSensitivity() 475
setCategory() 696
setCenterButtons() 619
setCentralWidget() 648, 649
setChannelCount() 692
setCheckable() 447, 462, 512, 659
setCheckBox() 622
setChecked() 447, 462–464, 659
setCheckState() 463, 512
setChild() 510
setChildrenCollapsible() 455
setClearButtonEnabled() 466
setCmyk() 558
setCmykF() 558
setCodec() 692
setCollapsible() 455
setCollateCopies() 701
setColor() 384, 560, 561, 604, 605
setColorMode() 702
setColumnCount() 507, 510
setColumnHidden() 519, 521
setColumnMinimumWidth() 440
setColumnStretch() 440
setColumnWidth() 519, 521
setComboBoxEditable() 627
setComboBoxItems() 627
setCommitPage() 647
setCompleter() 465, 500
setConnectOptions() 533
setContainerFormat() 690
setContentsMargins() 438, 440, 443
setContextMenu() 672
setContrast() 684
setCoords() 375
setCopyCount() 701
setCorner() 651
setCornerButtonEnabled() 520
setCreator() 724
setCurrentCharFormat() 474
setCurrentFont() 473, 502
setCurrentIndex() 443, 451, 453, 499, 513, 526, 687
setCurrentPage() 486, 720
setCurrentSection() 485
setCurrentSectionIndex() 485
setCurrentText() 499
setCurrentWidget() 443, 451, 453
setCursor() 424, 596
setCursorPosition() 466
setCursorWidth() 473
setData() 428, 505, 508, 512, 546, 660
setDatabaseName() 533
setDate() 484
setDateRange() 484, 486
setDateTextFormat() 487
setDateTime() 484
setDateTimeRange() 484
setDecimals() 483
setDecMode() 489
setdefault() 178, 181, 318
setDefault() 462, 616
setDefaultAction() 656, 664
setDefaultAlignment() 524
setDefaultButton() 622
setDefaultFont() 476
setDefaultProperty() 646
setDefaultSectionSize() 523
setDefaultStyleSheet() 476
setDefaultSuffix() 633
setDefaultTextColor() 602
setDefaultUp() 654
setDescription() 733
setDetailedText() 621
setDigitCount() 489
setDirection() 438
setDirectory() 633
setDirectoryUrl() 633
setDisabled() 334, 400, 659, 661
setDismissOnClick() 736
setDisplayFormat() 485
setDockNestingEnabled() 651
setDockOptions() 650
setDocument() 474, 602
setDocumentMargin() 476
setDocumentMode() 451
setDocumentTitle() 470
setDoubleClickInterval() 420
setDoubleDecimals() 627
setDoubleMaximum() 627
setDoubleMinimum() 627
setDoubleRange() 627
setDoubleValue() 627
setDown() 462
setDragCursor() 427
setDragDropMode() 515
setDragEnabled() 465, 512, 515
setDragMode() 591
setDropAction() 429, 430, 610, 611
setDropEnabled() 513
setDropIndicatorShown() 515

setDuplex() 701
setDuplicatesEnabled() 500
setDynamicSortFilter() 528
setEchoMode() 465
setEditable() 499, 512
setEditorData() 529
setEditStrategy() 546
setEditText() 500
setEditTriggers() 514
setElideMode() 450
setEnabled() 400, 462, 513, 596, 604, 660, 662, 736
setEncodingMode() 692
setEncodingOption() 693
setEncodingOptions() 693
setEscapeButton() 622
setExclusive() 661
setExpanded() 521
setExpandsOnDoubleClick() 522
setFacingPagesViewMode() 720
setFamily() 564
setFeatures() 666
setField() 643, 647
setFieldGrowthPolicy() 442
setFileMode() 632
setFilePath() 732
setFilter() 546
setFilterCaseSensitivity() 528
setFilterFixedString() 527
setFilterKeyColumn() 528
setFilterRegExp() 527
setFilterRole() 528
setFiltersChildEvents() 611
setFilterWildcard() 527
setFinalPage() 647
setFirstColumnSpanned() 522
setFirstDayOfWeek() 487
setfirstweekday() 202, 205
setFixedHeight() 365
setFixedSize() 365
setFixedWidth() 365
setFlag() 587, 588, 595
setFlags() 512, 596
setFlat() 447, 462, 737
setFloatable() 663
setFloating() 666
setFlow() 517
setFocus() 414, 587, 596
setFocusItem() 587
setFocusPolicy() 415, 420
setFocusProxy() 414, 607
setFont() 512, 564, 584, 601, 602, 660
setFontEmbeddingEnabled() 702
setFontFamily() 473
setFontFilters() 502
setFontItalic() 473
setFontPointSize() 473
setFontUnderline() 473
setFontWeight() 473
setForeground() 512
setForegroundBrush() 584, 590
setFormAlignment() 442
setFormat() 490
setForwardOnly() 541
setFrame() 465, 482, 500
setFrameShadow() 448
setFrameShape() 448
setFrameStyle() 448
setFromTo() 702
setFullScreen() 683
setGenerated() 548
setGeometry() 365, 368
setGraphicsEffect() 603
setGreen() 557
setGridSize() 517
setGridStyle() 520
setGridVisible() 487
setGroup() 603
setHandleWidth() 455
setHeader() 523
setHeaderData() 509, 544
setHeaderHidden() 521
setHeaderTextFormat() 487
setHeight() 372, 375
setHeightForWidth() 445
setHexMode() 489
setHighlightSections() 524
setHistory() 633
setHorizontalHeader() 523
setHorizontalHeaderFormat() 487
setHorizontalHeaderItem() 509
setHorizontalHeaderLabels() 508
setHorizontalPolicy() 445
setHorizontalScrollBarPolicy() 457
setHorizontalSpacing() 440, 443
setHorizontalStretch() 445
setHostName() 532
setHotSpot() 427
setHsv() 558
setHsvF() 559
setHtml() 428, 469, 474, 494, 602
setHue() 684
setIcon() 461, 512, 620, 656, 658, 672, 733, 736
setIconPixmap() 620

setIconSize() 461, 500, 515, 650, 664
setIconVisibleInMenu() 658
setIdentifier() 731
setImage() 431
setImageData() 428
setIndent() 460
setIndentation() 522
setIndexWidget() 514
setInformativeText() 620
setInputMask() 467
setInputMode() 627
setInsertPolicy() 499
setInteractive() 591, 736
setInterval() 406
setIntMaximum() 627
setIntMinimum() 627
setIntRange() 627
setIntStep() 627
setIntValue() 627
setInvertedAppearance() 490, 491
setInvertedControls() 491
setItalic() 565
setItem() 507
setItemData() 499
setItemDelegate() 529
setItemDelegateForColumn() 530
setItemDelegateForRow() 530
setItemEnabled() 453
setItemIcon() 453, 499
setItemIndexMethod() 584
setItemsExpandable() 522
setItemText() 453, 499
setItemToolTip() 453
setJoinMode() 552
setJoinStyle() 561
setKeepPositionOnInsert() 480
setKeyboardPageStep() 671
setKeyboardSingleStep() 671
setLabel() 642
setLabelAlignment() 442
setLabelText() 626, 633, 641
setLandscapeOrientation() 719
setLayout() 330, 435
setLayoutDirection() 438
setLayoutMode() 517
setLeft() 374
setLeftMargin() 711
setLine() 562, 599
setLineWidth() 448
setLineWrapColumnOrWidth() 472
setLineWrapMode() 472
setlocale() 103
setLoopCount() 696
setMargin() 460
setMargins() 711
setMask() 387, 576
setMaxCount() 500
setMaximum() 483, 489, 490, 641, 728
setMaximumBlockCount() 476
setMaximumDate() 484, 486
setMaximumDateTime() 484
setMaximumHeight() 366
setMaximumSectionSize() 523
setMaximumSize() 366
setMaximumTime() 484
setMaximumWidth() 366
setMaxLength() 465
setMaxVisibleItems() 500
setMedia() 675
setMenu() 462, 660, 664
setMenuBar() 649
setMenuWidget() 649
setMetaData() 690
setMidLineWidth() 448
setMimeData() 426, 427, 431
setMinimum() 483, 489, 490, 641, 728
setMinimumContentsLength() 500
setMinimumDate() 484, 486
setMinimumDateTime() 484
setMinimumDuration() 641
setMinimumHeight() 366
setMinimumMargins() 711
setMinimumSectionSize() 523
setMinimumSize() 365
setMinimumTime() 484
setMinimumWidth() 366
setModal() 615
setMode() 488
setModel() 513
setModelColumn() 517
setModelData() 529
setModified() 465, 474
setMouseTracking() 422
setMovable() 451, 663
setMovement() 517
setMovie() 460
setMuted() 676, 696
setNamedColor() 556
setNameFilter() 633
setNameFilters() 633
setNavigationBarVisible() 487
setNotation() 468
setNotchesVisible() 492
setNotchTarget() 492

setNotifyInterval() 678
setNull() 548
setNum() 459
setObjectName() 391
setOctMode() 489
setOffset() 601, 604
setOkButtonText() 626
setOpacity() 596, 606
setOpacityMask() 606
setOpaqueResize() 455
setOpenExternalLinks() 459, 602
setOpenLinks() 481
setOption() 627, 632, 645, 670, 671, 713
setOptions() 628, 633, 645, 714
setOrganizationName() 743
setOrientation() 455, 489, 490, 617, 711, 719
setOrientation(<Ориентация>) 663
setOutputFileName() 700
setOutputFormat() 700
setOutputLocation() 690
setOverlayAccessibleDescription() 726
setOverlayIcon() 726
setOverline() 565
setOverrideCursor() 424, 425
setOverwriteMode() 472
setP1() 562
setP2() 562
setPage() 642
setPageLayout() 701, 724
setPageMargins() 701, 723
setPageOrientation() 701, 723
setPageSize() 700, 711, 723
setPageStep() 491
setPalette() 384, 386
setPaperSource() 702
setParent() 363
setParentItem() 597
setPassword() 533
setPaused() 728
setPen() 567, 599
setPicture() 459
setPixel() 580
setPixelSize() 565
setPixmap() 359, 387, 426, 431, 459, 601, 644, 646
setPlaceholderText() 465
setPlainText() 469, 474, 602
setPlaybackMode() 686
setPlaybackRate() 676
setPlaylist() 675
setPoint() 563
setPoints() 562, 563
setPointSize() 564
setPointSizeF() 564
setPolygon() 600
setPopupMode() 665
setPort() 532
setPortraitOrientation() 719
setPos() 425, 595
setPosition() 478, 676
setPrefix() 483
setPrinterName() 700
setPrintRange() 702
setPriority() 344, 660
setQuality() 693
setQuery() 544
setQuitOnLastWindowClosed() 342
setRange() 483, 489, 490, 641, 727
setReadOnly() 465, 472, 482
setRecord() 547
setRect() 375, 599, 600
setRed() 557
setRedF() 557
setRelation() 551
setRenderHint() 569, 571, 591
setRenderHints() 592
setResizeMode() 517
setResolution() 702, 724
setResult() 615
setRgb() 557
setRgba() 557
setRgbF() 557
setRight() 375
setRightMargin() 711
setRootIndex() 513
setRootIsDecorated() 522
setRotation() 597
setRowCount() 507, 510
setRowHeight() 519
setRowHidden() 518, 519, 521
setRowMinimumHeight() 440
setRowStretch() 440
setRowWrapPolicy() 442
setRubberBandSelectionMode() 591
setSampleRate() 692
setSaturation() 684
setScale() 597
setScaledContents() 460
setScene() 590
setSceneRect() 584, 590
setSectionHidden() 523
setSectionResizeMode() 523
setSectionsClickable() 524
setSectionsMovable() 524

setSegmentStyle() 488
setSelectable() 512
setSelected() 588, 596
setSelectedDate() 486
setSelectedSection() 485
setSelection() 460, 465
setSelectionArea() 588
setSelectionBehavior() 514
setSelectionMode() 487, 514
setSelectionModel() 514, 525
setSelectionRectVisible() 518
setSeparator() 658
setSeparatorsCollapsible() 656
setShapeMode() 601
setShortcut() 461, 658
setShortcutContext() 659
setShortcutEnabled() 417
setShortcuts() 659
setShowGrid() 520
setSidebarUrls() 633
setSideWidget() 645
setSinglePageViewMode() 720
setSingleShot() 406
setSingleStep() 483, 491
setSize() 375
setSizeAdjustPolicy() 500
setSizeGripEnabled() 615, 668
setSizePolicy() 444
setSizes() 456
setSliderPosition() 490
setSmallDecimalPoint() 489
setSort() 546
setSortCaseSensitivity() 527
setSortingEnabled() 520, 522, 527
setSortLocaleAware() 527
setSortRole() 509, 527
setSource() 480, 696
setSourceModel() 527
setSpacing() 438, 440, 443, 518
setSpan() 519
setSpanAngle() 600
setSpecialValueText() 482
setStackingMode() 443
setStandardButtons() 617, 621
setStartAngle() 600
setStartDragDistance() 426
setStartDragTime() 426
setStartId() 643
setStatusBar() 649
setStatusTip() 659
setStickyFocus() 587
setStrength() 606
setStretchFactor() 455
setStretchLastSection() 523
setStrikeOut() 565
setStringList() 505
setStyle() 561
setStyleSheet() 385, 390
setSubTitle() 646
setSubTitleFormat() 644
setSuffix() 483
setSystemMenu() 671
setTabChangesFocus() 473, 602
setTabEnabled() 451
setTabIcon() 450
setTabKeyNavigation() 515
setTable() 546
setTabOrder() 415
setTabPosition() 450, 651, 670
setTabsClosable() 451, 670
setTabShape() 451, 651, 670
setTabsMovable() 670
setTabStopWidth() 473
setTabText() 450
setTabToolTip() 451
setTabWhatsThis() 451
setTearOffEnabled() 656
setter() 262
setText() 334, 427, 431, 458, 461, 464, 465, 467, 469, 512, 601, 620, 658
setTextAlignment() 512
setTextBackgroundColor() 474
setTextColor() 473
setTextCursor() 477, 602
setTextDirection() 489
setTextEchoMode() 627
setTextElideMode() 515
setTextFormat() 459, 621
setTextInteractionFlags() 460, 471, 602
setTextMargins() 466
setTexture() 561
setTextureImage() 561
setTextValue() 627
setTextVisible() 489
setTextWidth() 602
setTickInterval() 492
setTickPosition() 491
setTime() 484
setTimeRange() 484
setTimerType() 406
setTimeSpec() 485
setTitle() 446, 646, 656, 724, 732, 733
setTitleBarWidget() 666
setTitleFormat() 644

setToolButtonStyle() 649, 663, 664
setToolTip() 389, 512, 596, 659, 672, 736
setToolTipDuration() 389
setToolTipsVisible() 656
setTop() 374
setTopLeft() 375
setTopMargin() 711
setTopRight() 375
setTracking() 491
setTransform() 573, 593, 597
setTransformationMode() 601
setTransformOriginPoint() 597
setTristate() 463, 512
setType() 733
setUnderline() 565
setUndoRedoEnabled() 470, 475
setUniformItemSizes() 518
setUniformRowHeights() 521
setUnits() 711
setUpdatesEnabled() 412
setupUi() 338
setUrl() 494
setUrls() 428
setUserName() 533
setUsesScrollButtons() 451
setValidator() 468, 500
setValue() 483, 489, 490, 548, 641, 728, 745, 746
setVerticalHeader() 523
setVerticalHeaderFormat() 487
setVerticalHeaderItem() 509
setVerticalHeaderLabels() 508
setVerticalPolicy() 445
setVerticalScrollBarPolicy() 457
setVerticalSpacing() 440, 443
setVerticalStretch() 445
setVideoOutput() 676
setViewMode() 517, 632, 670, 719
setViewport() 573
setVisible() 363, 596, 615, 616, 654, 660, 662, 672, 727, 732, 736
setVolume() 676, 690, 696
setWeekdayTextFormat() 487
setWeight() 565
setWhatsThis() 389, 512, 659
setWidget() 456, 666, 671
setWidgetResizable() 456
setWidth() 372, 375, 560
setWidthF() 560
setWindow() 573
setWindow(<QWindow>) 725, 735
setWindowDisallowPeek() 740
setWindowExcludedFromPeek() 740
setWindowFlags() 363
setWindowFlip3DPolicy() 741
setWindowIcon() 383
setWindowModality() 382, 616
setWindowOpacity() 381
setWindowState() 379
setWindowTitle() 329, 363, 620
setWizardStyle() 643
setWordWrap() 459, 517, 520, 522
setWordWrapMode() 472
setWorkingDirectory() 733
setWrapping() 482, 492, 517
setX() 371, 374, 595
setXOffset() 605
setY() 371, 374, 595
setYOffset() 605
setZoomFactor() 494, 720
setZoomMode() 720
setZValue() 595
sha1() 120
sha224() 120
sha256() 120
sha3_224() 120
sha3_256() 120
sha3_384() 120
sha3_512() 120
sha384() 120
sha512() 120
shake_128() 120
shake_256() 120
shape() 594
shear() 573, 593
Sheet 364
shelve 315, 317
ShiftModifier 419
Shortcut 417
ShortcutFocusReason 414
shortcutId() 417
ShortDayNames 487
Show 408
show() 359, 363, 596, 615, 672, 727
ShowAlphaChannel 639
showColumn() 519, 521
ShowDirsOnly 633
showEvent() 410
showFullScreen() 379
showMaximized() 379
showMenu() 463, 665
showMessage() 359, 640, 667, 672
showMinimized() 379
showNextMonth() 487

showNextYear() 487
showNormal() 379
showPopup() 500
showPreviousMonth() 487
showPreviousYear() 487
showRow() 519
showSection() 524
showSelectedDate() 486
showShaded() 671
showStatusText() 660
showSystemMenu() 671
showTearOffMenu() 656
showToday() 486
ShowToParent 408
shuffle() 80, 157, 687
shutil 307, 320
sibling() 504
signalsBlocked() 399
sin() 77
SingleHtmlSaveFormat 496
SingleLetterDayNames 487
SinglePageView 719
SinglePass 518
SingleSelection 487, 514
singleShot() 407
size() 366, 377, 411, 542, 543, 564, 576, 579, 710
SizeAllCursor 424
SizeBDiagCursor 424
SizeFDiagCursor 424
sizeHint() 366
SizeHintRole 503
SizeHorCursor 424
sizePixels() 710
sizePoints() 710
sizePolicy() 444
sizes() 456
SizeVerCursor 424
sleep() 188, 342, 346, 347
SliderMove 491
sliderMoved 491
SliderNoAction 491
SliderPageStepAdd 491
SliderPageStepSub 491
sliderPosition() 490
sliderPressed 491
sliderReleased 491
SliderSingleStepAdd 491
SliderSingleStepSub 491
SliderToMaximum 491
SliderToMinimum 491
SmoothPixmapTransform 591
smoothSizes() 565
SmoothTransformation 577, 580, 601
Snap 517
SolidLine 520, 560
SolidPattern 386, 561
sort() 158, 179, 506, 509, 527, 544
sortByColumn() 520, 522
sortChildren() 513
sorted() 67, 159, 179
source() 427, 430, 480, 611
sourceChanged 481
sourceModel() 527
South 450, 651, 670
span() 134
spanAngle() 600
spec() 559
SplashScreen 364
split() 101, 139, 312
splitDockWidget() 652
splitdrive() 312
splitext() 312
SplitHCursor 424
splitlines() 102
splitterMoved 456
SplitVCursor 424
spontaneous() 408
sqrt() 78
SquareCap 560
StackAll 443
stackingMode() 444
StackingOrder 669
StackOne 443
StalledMedia 676
standardButton() 618, 622
standardButtons() 618, 622
StandardNotation 468
starmap() 172
start 168
Start 477
start() 134, 344, 406
startAngle() 600
startDragDistance() 426
startDragTime() 426
started() 344
startId() 643
StartingStatus 691
StartOfBlock 477
StartOfLine 477
StartOfWord 478
startswith() 107
startTimer() 404
stat 307

stat() 309, 323
stat_result 309
state() 675, 691, 722
stateChanged 464, 678, 692
StatementError 537
Static 517
status() 691, 696, 750
statusBar() 649
statusChanged 692, 697
StatusTip 660
statusTip() 659
StatusTipRole 502
stderr 297
stdin 35, 297, 313
stdout 33, 297, 299, 314
step 168
stepBy() 482
stepUp() 482
stickyFocus() 587
stop 168
stop() 406, 494, 675, 690, 696, 728
StopIteration 69, 220, 273, 278, 296, 304
StoppedState 675, 691
str 42, 82
str() 48, 49, 83, 89, 96, 100, 115, 119, 191, 255
StreamPlayback 675
strength() 606
strengthChanged 606
Stretch 523, 644
strftime() 186, 187, 193, 196, 200
strikeOut() 565
string 133
String 536
StringIO 301
stringList() 505
strip() 101
StrongFocus 415
strptime() 186, 187, 198
struct_time 184–186
StyledPanel 448
styles() 565
sub() 137, 138
submit() 547, 549
submitAll() 547, 549
subn() 138
subTitle() 646
SubWindow 364
subWindowActivated 670
subWindowList() 669
SubWindowView 670
sum() 76
Sunday 487
SUNDAY 202
Sunken 448
super() 250
supportedActions() 427
supportedAudioCodecs() 691
supportedAudioSampleRates() 691
supportedContainers() 690
supportedImageFormats() 383, 574, 575
supportedMimeTypes() 696
supportedPageSizes() 722
supportedResolutions() 702, 722
supportsCustomPageSizes() 722
supportsMessages() 673
supportsMultipleCopies() 702
SvgMiterJoin 560
swap() 577, 710, 712
swapcase() 104
swapSections() 524
symmetric_difference() 163, 167
symmetric_difference_update() 164
sync() 745
SyntaxError 273
sys 34, 46, 224, 234, 266
sys.argv 36
sys.path 237
sys.stdin 35
sys.stdout 34
SystemError 273
systemMenu() 671
SystemScope 744
SystemTables 534

T

TabbedView 670
tabCloseRequested 451
tabCloseRequested() 452
TabError 273
TabFocus 415
TabFocusReason 414
tabifiedDockWidgetActivated 653
tabifiedDockWidgets() 652
tabifyDockWidget() 652
tableName() 546
Tables 534
tables() 533
tabPosition() 651, 670
tabShape() 651, 670
tabStopWidth() 473
tabText() 450
takeChild() 511
takeColumn() 508, 511

takeItem() 508
takeRow() 508, 511
takewhile() 171
takeWidget() 456
tan() 77
target() 427
targetChanged() 427
TargetMoveAction 426
task_done() 352
taskbarActivateTab() 739
taskbarActivateTabAlt() 740
taskbarDeleteTab() 739
Tasks 732
tasks() 730
tee() 173
tell() 297, 302
tempPath() 742
terminate() 349
testOption() 670, 671, 714
text() 419, 427, 431, 459, 461, 464, 482, 489, 512, 537, 601, 658
TextAlignmentRole 502
TextAntialiasing 571, 591
textBackgroundColor() 474
TextBrowserInteraction 460, 472
TextCalendar 201, 202
textChanged 467, 471
textColor() 474
textCursor() 477, 602
TextDontClip 570
TextDontPrint 570
TextEditable 460, 472
textEdited 467
TextEditorInteraction 460, 472
TextExpandTabs 570
TextHideMnemonic 570
TextIncludeTrailingSpaces 570
TextInput 627, 628
TextJustificationForced 570
TextSelectableByKeyboard 460, 471
TextSelectableByMouse 460, 471
TextShowMnemonic 570
TextSingleLine 570
textValue() 627
textValueChanged 628
textValueSelected 628
textWidth() 602
TextWordWrap 570
TextWrapAnywhere 570
Thursday 487
THURSDAY 202
TicksAbove 491
TicksBelow 492
TicksBothSides 491
TicksLeft 492
TicksRight 492
tileSubWindows() 670
time 184, 186, 188, 189, 194, 196, 207, 342
Time 536
time() 184, 199, 484
timeChanged 485
TimeCriticalPriority 344
timedelta 189
timegm() 207
timeit 184, 208
timeit() 208
timeout 406
TimeoutError 324
Timer 208, 408
timerId() 404, 406
timerType() 406
timestamp() 199, 421, 424
timetuple() 193, 200
timetz() 199
tip() 660
title() 105, 446, 494, 646, 656, 732, 733
titleBarWidget() 666
titleChanged 495
tobytes() 305
toCmyk() 559
today() 192, 197
Toggle 526
toggle() 462, 661
ToggleCurrent 526
toggled 447, 462–464, 661
toggleViewAction() 664, 666
toHsl() 559
toHsv() 559
toHtml() 470, 474, 479, 602
toImage() 576
tolist() 305
toNativeSeparators() 742
Tool 364
toolBarArea() 649
toolBarBreak() 650
ToolButtonFollowStyle 650, 664
ToolButtonIconOnly 649, 663
toolButtonStyle() 650, 664
ToolButtonTextBesideIcon 650, 664
ToolButtonTextOnly 649, 663
ToolButtonTextUnderIcon 650, 664
ToolTip 364
toolTip() 389, 659, 672, 736
toolTipDuration() 389
ToolTipRole 499, 502
toordinal() 192, 193, 198, 200

top() 377
toPage() 702
TopDockWidgetArea 650, 652, 666
toPlainText() 470, 474, 479, 602
topLeft() 377
TopLeftCorner 652
topLevelChanged 664, 666
topLevelItem() 597
topRight() 377
TopRightCorner 652
TopToBottom 438, 490, 517
TopToolBarArea 649, 663
toPyDate() 484
toPyDateTime() 484
toPyTime() 484
toRgb() 559
total_seconds() 190
traceback 266
transaction() 533
TransactionError 537
transform() 593, 597
transformed() 577, 578, 581
translate() 108, 376, 573, 593
translated() 377
transparent 556
transpose() 373
transposed() 373
Triangular 451, 651, 670
Trigger 673
trigger() 661
triggered 654, 657, 661, 662, 665
triggerPageAction() 495
True 41, 59
truncate() 297, 304
try 265
tryLock() 355
Tuesday 487
TUESDAY 202
tuple 42
tuple() 50, 160
TwoPassEncoding 693
type 43
type() 47, 408, 536, 537, 732, 733
TypeError 103, 155, 160, 273, 279
typeName() 342
tzinfo 189, 194, 195, 197, 198

U

uic 337
UInt 536
UliPad 16
ULongLong 536
UnavailableStatus 691
UnboundLocalError 225, 273
Unchecked 463, 502, 512
underline() 565
undo() 466, 470, 475, 480
UndoAndRedoStacks 475
undoAvailable 471, 476
undoCommandAdded 477
UndoStack 475
ungrabKeyboard() 596, 607
ungrabMouse() 596, 608
UNICODE 123
UnicodeDecodeError 49, 84, 273
UnicodeEncodeError 112, 116, 273
UnicodeTranslationError 273
uniform() 79
union() 162, 167
unique 283
UniqueConnection 398
united() 379
Unknown 536, 673
UnknownError 537
UnknownMediaStatus 676
unlink() 308
UnloadedStatus 691
unlock() 355, 358
Unpickler 316
unsetCursor() 424, 596
Up 477
UpArrow 665
UpArrowCursor 424
update() 120, 163, 182, 318, 413, 516, 555, 589, 598, 604
updateEditorGeometry() 529
updatePreview() 720
updateScene() 594
updateSceneRect() 594
UpDownArrows 482
upper() 104
url() 494
urlChanged 496
urls() 428
urlSelected 634
urlsSelected 634
UseListViewForComboBoxItems 628
UsePlainTextEditForTextInput 628
User 409
UserRole 503
UserScope 744
usleep() 347
UTC 485

utcfromtimestamp() 197
utcnow() 197
utctimetuple() 200
utime() 310

V

validateCurrentPage() 643
validatePage() 647
value 284
value() 341, 483, 488–490, 542, 548, 559, 641, 728, 745
valueChanged 483, 490, 491, 728
ValueError 69, 106, 118, 155, 156, 186, 191, 195, 197, 198, 273, 293
valueF() 559
values() 180, 317
ValuesAsRows 540
vars() 227
VERBOSE 123
Vertical 454, 489, 490, 493, 509, 523, 545, 609, 617, 650, 652, 663
verticalHeader() 519, 522
verticalHeaderItem() 509
verticalScrollBar() 457
VerticalTabs 651
VeryCoarseTimer 404
VeryHighQuality 693
VeryLowQuality 693
VideoSurface 675
viewMode() 670, 720
viewport() 573
viewportEntered 516
Views 534
views() 589
visibilityChanged 664, 666, 728
visitedPages() 643
visualIndex() 524
VLine 448
volume() 676, 691, 696
volumeChanged 678, 692, 697

W

W_OK 307
WA_DeleteOnClose 383, 390, 668
WA_NoMousePropagation 422
wait() 349
WaitCursor 424
walk() 319
Warning 619, 673
warning() 624
wasCanceled() 642
WatermarkPixmap 644, 645
Wednesday 487
WEDNESDAY 202
weekday() 194, 200, 207
weekheader() 206
weeks 189
weight() 565
West 450, 651, 670
whatsThis() 389, 659
WhatsThisCursor 424
WhatsThisRole 502
Wheel 408
wheelEvent() 423, 609
WheelFocus 415
while 29, 70, 72, 149
white 359, 556
Widget 363
widget() 444, 451, 454, 456, 666, 671
widgetForAction() 663
widgetRemoved 444
WidgetShortcut 417, 659
WidgetWidth 472
WidgetWithChildrenShortcut 417, 659
width() 366, 368, 372, 377, 566, 576, 579, 584, 703
widthMM() 703
WindingFill 569
Window 363, 384
window() 573, 726, 735
WindowActivate 408
WindowActive 380
WindowBlocked 409
WindowCloseButtonHint 365
WindowContextHelpButtonHint 365
WindowDeactivate 408
windowFlags() 365
windowFlip3DPolicy() 741
WindowFullScreen 380
windowHandle() 725, 735
WindowMaximizeButtonHint 365
WindowMaximized 380
WindowMinimizeButtonHint 365
WindowMinimized 380
WindowMinMaxButtonsHint 365
WindowModal 382, 616
windowModality() 382, 616
WindowNoState 380
windowOpacity() 381
windowPos() 421

WindowShortcut 417, 659
windowState() 380
WindowStateChange 409, 410
windowStateChanged 672
WindowStaysOnBottomHint 365
WindowStaysOnTopHint 359, 365
WindowSystemMenuHint 364
WindowText 384
windowTitle() 363
WindowTitleHint 364
windowType() 364
WindowUnblocked 409
WinPanel 448
with 269, 270, 271
wizard() 646
WordLeft 478
WordRight 478
WordUnderCursor 479
WordWrap 472
workingDirectory() 733
WrapAllRows 442
WrapAnywhere 472
WrapAtWordBoundaryOrAnywhere 472
WrapLongRows 442
writable() 294
write() 34, 294, 301, 302
writelines() 294, 302

X

x() 368, 371, 377, 421, 423, 595, 657
X_OK 307
x1() 562
x2() 562
XButton1 421
XButton2 421
xOffset() 605

Y

y() 368, 371, 377, 421, 423, 595, 657
y1() 562
y2() 562
year 192, 198
YearSection 485
yearShown() 486
yellow 556
yellow() 558
yellowF() 558
Yes 617, 619
YesRole 618, 619, 621
YesToAll 617, 620
yield 219, 220
yieldCurrentThread() 347
yOffset() 605

Z

ZeroDivisionError 273
zfill() 95
zip() 151, 176
zip_longest() 172
zoomFactor() 494, 720
zoomIn() 470, 720
zoomMode() 720
zoomOut() 470, 720
zValue() 595

В

Ввод 35
◊ перенаправление 312
Время 184
Вывод 33
◊ перенаправление 312
Выделение блоков 30
Выражение-генератор 150

Г

Генератор
◊ множества 166
◊ словаря 183
◊ списка 149

Д

Дата 184
◊ текущая 184
◊ форматирование 186
Декоратор класса 263
Делегат 503, 528
◊ связанный 553
Десериализация 119
Деструктор 248
Диапазон 141, 142, 167
Документация 36

З

Запуск программы 27, 36
Засыпание скрипта 188

И

Именование переменных 40
Индекс 141, 161
Индикатор выполнения процесса 314
Исключение 264
◊ возбуждение 273
◊ иерархия классов 271
◊ перехват всех исключений 267
◊ пользовательское 273
Итератор 277, 278

К

Календарь 201
◊ HTML 204
◊ текстовый 202
Каталог 319
◊ обход дерева 319
◊ очистка дерева каталогов 320
◊ права доступа 305
◊ преобразование пути 310
◊ создание 319
◊ список объектов 319
◊ текущий рабочий 288, 319
◊ удаление 319
Квантификатор 127
Класс 244
Ключ 746
Ключевые слова 40
Код символа 105
Кодировка 27, 29
Комментарий 31
Конструктор 247
Контейнер 277, 435
◊ перечисление 281
◊ последовательность 279
Кортеж 141, 160
◊ объединение 161
◊ повторение 161
◊ проверка на вхождение 161
◊ создание 160
◊ срез 161

Л

Локаль 103

М

Маска прав доступа 306
Мастер 642
Менеджер
◊ геометрии 435
◊ компоновки 435
Множества 162
Множество 141
◊ генератор 166
Модель 503
◊ выделения 503
◊ промежуточная 503
Модуль 231
◊ импорт модулей внутри пакета 241
◊ импортирование 231, 234
◊ относительный импорт 241
◊ повторная загрузка 238
◊ получение значения атрибута 232
◊ проверка существования атрибута 232

Модуль *(прод.)*
◊ пути поиска 237
◊ список всех идентификаторов 234
Мьютекс 355

Н

Наследование 248
◊ множественное 250

О

Объектно-ориентированное программирование 244
Окно: модальное 381
ООП 244
◊ абстрактный метод 259
◊ декоратор 263
◊ деструктор 248
◊ конструктор 247
◊ метод класса 258
◊ множественное наследование 250
◊ наследование 248
◊ определение класса 244
◊ перегрузка оператора 255
◊ примесь 252
◊ псевдочастный атрибут 260
◊ свойство класса 261
◊ создание атрибута класса 245
◊ создание метода класса 245
◊ создание экземпляра класса 245
◊ специальный метод 253
◊ статический метод 258
Оператор 52
◊ break 71
◊ continue 71
◊ pass 210
◊ ветвления 62, 64, 65
◊ двоичный 54
◊ для работы с последовательностями 55
◊ математический 52
◊ перегрузка 255
◊ приоритет выполнения 57
◊ присваивания 56
◊ сравнения 60
◊ условный 59
Отображение 44
Ошибка
◊ времени выполнения 264
◊ логическая 264
◊ синтаксическая 264

П

Пакет 239
Переменная 40
◊ глобальная 224
◊ локальная 224
◊ удаление 50
Перенаправление ввода/вывода 312
Перечисление 277, 282
Плейлист 674
Последовательность 44
◊ количество элементов 146
◊ объединение 147
◊ оператор 55
◊ перебор элементов 148
◊ повторение 147
◊ преобразование в кортеж 160
◊ преобразование в список 142
◊ проверка на вхождение 147
◊ сортировка 159
◊ срез 146
Права доступа 305
Представление 503
Примесь 252
Присваивание 44
◊ групповое 45
◊ позиционное 46
Псевдокласс 391
Путь к интерпретатору 28

Р

Разделитель 392
Регулярное выражение 122
◊ группировка 128
◊ замена 137
◊ квантификатор 127
◊ класс 127
◊ метасимвол 124
◊ обратная ссылка 128
◊ поиск всех совпадений 136
◊ поиск первого совпадения 131
◊ разбиение строки 139
◊ специальный символ 123
◊ флаг 122
◊ экранирование спецсимволов 140
Редактирование файла 27
Редактор 528
Рекурсия 223
◊ проход 224

С

Сборщик мусора 772
Селектор
◊ дополнительный 391
◊ основной 391
◊ универсальный 391
Сериализация 119
Сигнал 330, 395
Словарь 175
◊ генератор 183
◊ добавление элементов 182
◊ количество элементов 179
◊ перебор элементов 179
◊ поверхностная копия 177
◊ полная копия 177
◊ проверка существования ключа 178, 181
◊ создание 175
◊ список значений 180
◊ список ключей 180
◊ удаление элементов 179, 181
Слот 330, 397
Событие 395
Создание файла с программой 27
Специальный символ 86
Список 141
◊ быстрого доступа 730
◊ выбор элементов случайным образом 157
◊ генератор 149
◊ добавление элементов 153
◊ заполнение числами 159
◊ количество элементов 146
◊ максимальное значение 156
◊ минимальное значение 156
◊ многомерный 148
◊ объединение 147
◊ перебор элементов 148
◊ переворачивание 157
◊ перемешивание 157
◊ поверхностная копия 144
◊ поиск элемента 156
◊ полная копия 144
◊ преобразование в строку 160
◊ создание 142
◊ сортировка 158
◊ срез 146
◊ удаление элементов 155
Срез 88, 146
Строка 82
◊ длина 88, 101
◊ документирования 31, 38, 85
◊ замена 107
◊ изменение регистра 104
◊ кодирование 120
◊ конкатенация 89
  ▫ неявная 89
◊ неформатированная 86
◊ перебор символов 89
◊ повторение 89
◊ поиск 105
◊ преобразование объекта 119
◊ проверка на вхождение 89
◊ проверка типа содержимого 109
◊ разбиение 101
◊ соединение 89
◊ создание 83
◊ тип данных 82
◊ удаление пробельных символов 101
◊ форматирование 90, 95
◊ форматируемая 99
◊ шифрование 120
◊ экранирование спецсимвола 84
Структура программы 28

Т

Текущий рабочий каталог 288
Тип данных 41
◊ преобразование 48
◊ проверка 47

У

Установка
◊ PyQt 327
◊ Python 19

Ф

Файл 287
◊ абсолютный путь 287
◊ время последнего доступа 309
◊ время последнего изменения 309
◊ дата создания 309
◊ дескриптор 297
◊ закрытие 294, 301
◊ запись 294, 301
◊ копирование 307
◊ обрезание 297
◊ открытие 287, 299

Файл *(прод.)*
◊ относительный путь 287
◊ переименование 308
◊ перемещение 308
  ▫ указателя 298
◊ позиция указателя 297
◊ права доступа 305
◊ преобразование пути 310
◊ проверка существования 308
◊ размер 308
◊ режим открытия 291
◊ создание 287
◊ сохранение объекта 315
◊ удаление 308
◊ чтение 295, 300
Факториал 223
Функция 210
◊ аннотация 229
◊ анонимная 218, 226
◊ вложенная 228
◊ вызов 211
◊ генератор 219
◊ декоратор 221
◊ значение параметра по умолчанию 216
◊ лямбда 218
◊ необязательный параметр 214
◊ обратного вызова 212
◊ определение 210
◊ переменное число параметров 216
◊ расположение определений 213
◊ родитель 228
◊ создание 210
◊ сопоставление по ключам 214

Х

Хранилище настроек 743

Ц

Цикл
◊ for 65
◊ while 70
◊ переход на следующую итерацию 71
◊ прерывание 70, 71

Ч

Число 73
◊ абсолютное значение 76, 78
◊ вещественное 73
  ▫ точность вычислений 74
◊ возведение в степень 76, 78
◊ восьмеричное 73
◊ двоичное 73
◊ десятичное 73
◊ квадратный корень 78
◊ комплексное 73, 74
◊ логарифм 78
◊ округление 76, 78
◊ преобразование 75
◊ случайное 79
◊ факториал 78
◊ целое 73
◊ шестнадцатеричное 73
◊ экспонента 78

Э

Эффект 674

Я

Язык 103