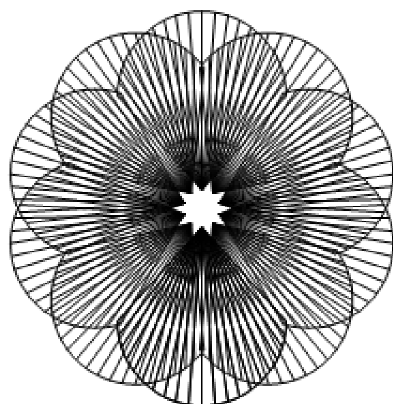
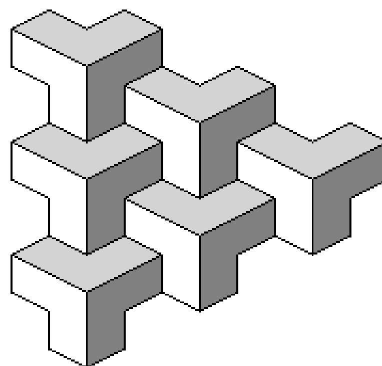
Г.Н.Гутман

Библиотека Tkinter: графика, геометрия и логические игры на Питоне

@ Гутман Г.Н. 2021

Язык программирования Python – один из современных языков программирования. За два десятилетия, прошедших с момента его создания он завоевал немало приверженцев среди программистов и используется для решения многих прикладных задач. Определяющими факторами является удачный синтаксис языка и наличие большого числа готовых модулей и библиотек «на все случаи жизни».

Библиотека Tkinter предназначена для создания оконных приложений, обеспечивающих стандартный интерфейс пользователя с приложением. Это её основное назначение. Однако её графические возможности намного шире, что позволяет разрабатывать на её основе игры различной сложности. Надо заметить, что для разработки игр со сложной графикой чаще свего используются дополнительные библиотеки, в частности, библиотека Pygame. Однако для простых приложений достаточно и тех возможностей, которые предоставляет нам Tkinter в совокупности с другими встроенными модулями.

Автор книги предполагает, что читатель знаком с базовыми конструкциями языка программирования Python и стремится применить свои знания для разработки интересных проектов. Все проекты могут быть реализованы на Python'е версии не ниже 3.4.

Во всех проектах используется библиотека tkinter:

А) для иллюстрации математических и геометрических понятий: пропорции, прогрессии, декартовой и полярной систем координат, параллельного переноса, масштабирования, поворотной симметрии, зеркального отражения, построения геометрических узоров и мозаик;

Б) для управления геометрическими объектами, исполнителями, моделирования движения различных механизмов

В) для создания логических игр и головоломок с развитым интерфейсом, с использованием элементов управления (кнопок, полей ввода и т.п.)

1. Графика в Tkinter

Главное окно приложения создаётся вызовом конструктора Tk(). Полученный объект лежит в основе иерархии всех создаваемых в программе объектов. Кроме главного окна программист при необходимости может создавать дополнительные окна — объекты класса Toplevel. Любое из окон является контейнером для размещения элементов управления — надписей, полей ввода, кнопок и т.д. Для проектов, представленных в этой работе, нам достаточно будет одного главного окна. С методами окна приложения мы будем знакомиться по мере надобности, но подавляющее большинство этих методов нам не понадобится.

Разместим в окне экземпляр класса Canvas – это своего рода холст для отображения векторных и растровых графических объектов. К векторным графическим объектам относятся графические примитивы: линии, дуги, прямоугольники, полигоны, овалы и текст. К растровым — изображения — графические файлы в формате GIF.

Параметры холста можно задавать как при вызове конструктора, так и впоследствии. Синтаксис вызова

```
w = Canvas ( master, option=value, ... )
```

master — окно приложения;

option=value, ... – набор именованных параметров.

Все параметры имеют значения по умолчанию, поэтому минимальным является вызов Canvas(). В этом случае холст автоматически «прикрепляется» к главному окну приложения. Однако общепринятым является указание параметра master во всех случаях.

Основные параметры холста:

width – ширина холста в пикселях;

height – высота холста в пикселях;

bg или background — цвет холста (фона).

Цвет холста задаётся текстовым литералом — названием или шестнадцатеричным кодом. Набор цветов содержит большое количество именованных цветов, однако он может отличаться в разных операционных системах. Поэтому для «нестандартных» цветов лучше использовать второй вариант задания цвета.

Созданный элемент управления («виджет») нужно разместить в окне приложения с помощью так называемых менеджеров размещения. В библиотеке Tkinter их три. Наиболее простым и наиболее подходящим для наших целей является метод pack().

Из всех параметров метода выделим параметр side, который задаёт сторону контейнера, по которой выравнивается элемент управления:

side = ‘top’ (значение по умолчанию) — элемент управления выравнивается по верхней стороне контейнера;

side = ’bottom’ — выравнивание по нижней стороне;

side = ‘left’ — выравнивание по левой стороне;

side = ‘right’ — выравнивание по правой стороне.

Если в окне приложения мы размещаем единственный элемент, например, холст, то можно использовать метод pack() без параметров (т.е. со значениями по умолчанию).

Все графические объекты создаются методами класса Canvas. Начнём с самых простых.

Прямоугольник. Пусть canvas есть экземпляр класса Canvas. Тогда для создания прямоугольника нужно вызвать метод create_rectangle. Синтаксис вызова:

```
id = canvas.create_rectangle(x1,y1,x2,y2,
                             option=value,...)
```

где

id – идентификатор графического объекта. Идентификатор объекта есть уникальное целое число, которое позволяет идентифицировать объект;

x1,y1,x2,y2 – координаты двух вершин прямоугольника, лежащих на одной диагонали;

option=value,... – набор именованных параметров.

Параметры:

fill – цвет внутренней области прямоугольника. Если параметр fill отсутствует, или fill есть пустая строка, то внутренность прямоугольника останется не заполненной;

stipple – шаблон заполнения внутренней области. Если этот параметр отсутствует или равен пустой строке, то область заливается сплошным цветом. Чтобы получить полупрозрачное заполнение следует использовать одно из значений 'gray75', 'gray50', 'gray25', 'gray12'. Чем меньше число, тем более прозрачным будет заполнение;

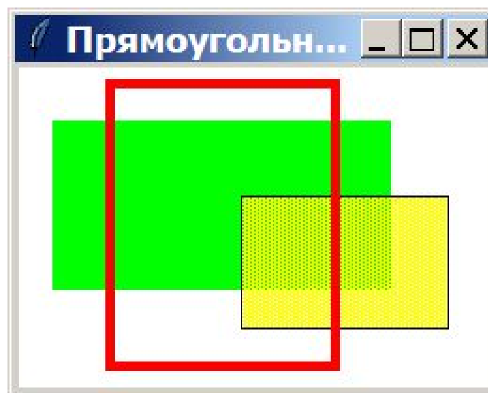
outline – цвет границы прямоугольника;

width – толщина границы, по умолчанию width=1. Если width = 0, то граница у прямоугольника отсутствует.

tags – набор тэгов графического объекта. Каждый тэг представляет собой текстовую строку, своего рода имя объекта. В отличие от идентификатора тэг не является уникальным значением, что позволяет объединять в одну группу несколько объектов.

Покажем использование описанных методов на примере.

Выведем на экран три прямоугольника разного вида.



Первый прямоугольник — без контура, второй — полупрозрачный и третий — только контур.

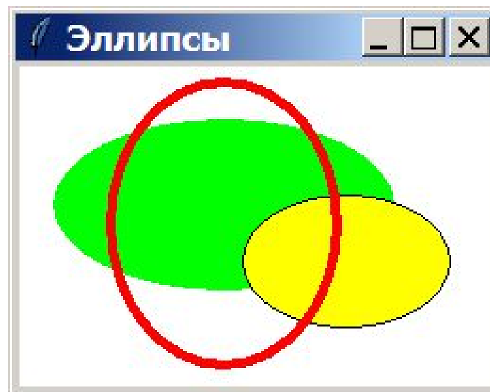
```
from tkinter import *
# создаём окно приложения и холст
root=Tk()
root.title('Прямоугольники')
canvas=Canvas(root,width=250,height=170,bg='white')
canvas.pack()
# рисуем три прямоугольника
r1 = canvas.create_rectangle(20,30,200,120,
                   width=0,fill='lime')
r2 = canvas.create_rectangle(120,70,230,140,
                   outline='black',fill='yellow',
                   stipple='gray75')
r3 = canvas.create_rectangle(50,10,170,160,
                   width=5,outline='red',fill='')
```

Легко заметить, что чем позже создан прямоугольник, тем ближе он к зрителю и поэтому может перекрывать ранее созданные объекты.

Эллипс. Эллипс по набору параметров идентичен прямоугольнику, он и задаётся описанным возле него прямоугольником. Эллипс создаётся методом create_oval.

```
id = canvas.create_oval(x1,y1,x2,y2, option=value,...)
```

Заменим в предыдущей программе метод create_rectangle на метод create_oval и мы получим следующую картину:



Заметим, что для эллипсов параметр stipple не работает.

Ломаная и сплайн. Ломаная линия задаётся списком своих вершин, соединённых в порядке перечисления. Ломаная может состоять из одного или нескольких отрезков прямых. В общем случае это незамкнутая фигура,

но можно получить замкнутую, если последняя точка будет совпадать с начальной.

Ломаная линия создаётся методом create_line:

id = canvas.create_line(x0,y0,x1,y1,..., xn,yn, option=value,...)

где

id – идентификатор графического объекта;

x0,y0,x1,y1,...,xn,yn – координаты вершин ломаной;

option=value,... – набор именованных параметров.

Часть именованных параметров нам уже известна:

fill – цвет линии;

width – толщина линии;

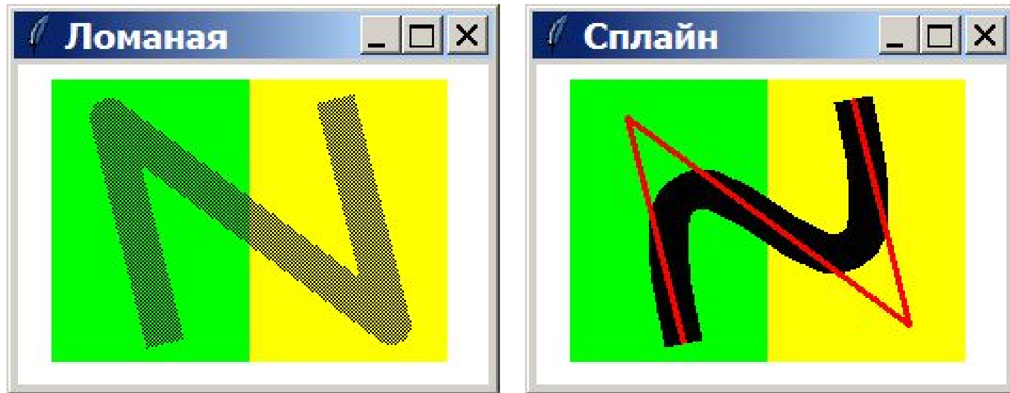
stipple – шаблон заполнения линии;

tags – набор тегов объекта.

Из других параметров самым интересным является параметр smooth. Если этот параметр не указывать или указать smooth = False, то будет создана ломаная линия. Но если задать smooth = True, то будет нарисована гладкая линия — так называемый сплайн. Эта линия уже не будет проходить через вершины ломаной, а будет сглаживать углы, образованные соседними звеньями ломаной.

Приведём примеры. Нарисуем полупрозрачную ломаную линию на фоне двух цветных прямоугольников.

```
from tkinter import *
# создаём окно приложения и холст
root=Tk()
root.title('Ломаная')
canvas=Canvas(root,width=250,height=170,bg='white')
canvas.pack()

r1=canvas.create_rectangle(20,10,135,160,
                 width=0,fill='lime')
r2=canvas.create_rectangle(135,10,230,160,
                 width=0,fill='yellow')
# рисуем ломаную
line = canvas.create_line(170,20,200,140,50,30,80,150,
         width=21,fill='black',stipple='gray50')
```



Заменим ломаную линию сплайном, построенным на той же последовательности вершин. Результат (кривая чёрного цвета) приведен на соседнем рисунке. Для наглядности на том же рисунке нарисована ломаная (красная линия), на основе которой построен сплайн.

Изменения в программе:

```
# рисуем сплайн
spline = canvas.create_line(170,20,200,140,50,30,80,150,
         width=21,fill='black',smooth=True)
line = canvas.create_line(170,20,200,140,50,30,80,150,
         width=3,fill='red')
```

Полигон. Полигон есть произвольный многоугольник. Его граница представляют собой замкнутую ломаную или кривую. Полигон задаётся списком своих вершин. Но в отличие от замкнутой ломаной, дублировать начальную вершин нет необходимости.

Полигон создаётся при вызове метода холста create_polygon:

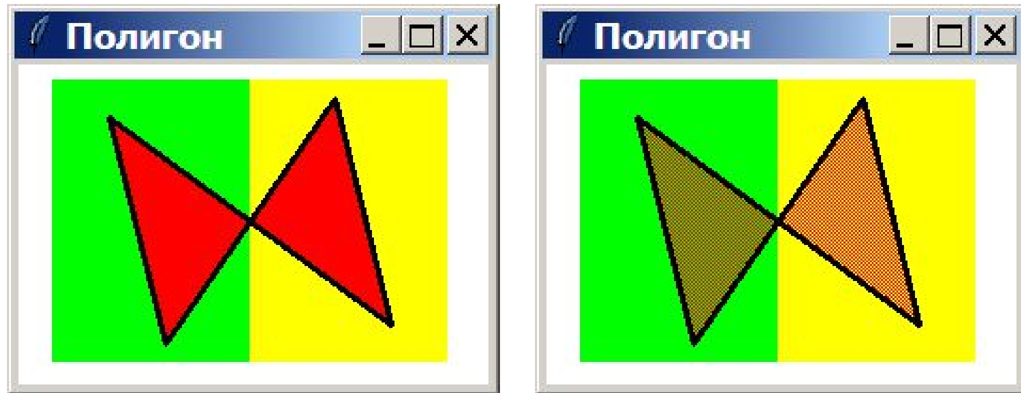
```
id = canvas.create_polygon(x0,y0,x1,y1,..., xn,yn,
                           option=value,...)
```

Основные именованные параметры полигона — уже известные нам fill, stipple, outline, width, tags, smooth.

Для иллюстрации заменим в программе рисования ломаной две последние строки Получим:

1) полигон со сплошным заполнением (левый рисунок)

```
poly = canvas.create_polygon(170,20,200,140,50,30,80,150,
     width=3,outline='black',fill='red')
```



2) полигон с шаблоном заполнения 'gray50' (правый рисунок)

```
poly = canvas.create_polygon(170,20,200,140,50,30,80,150,
     width=3,outline='black',fill='red',stipple='gray50')
```

Дуга, сектор и сегмент. Дуга в Питоне — это часть эллипса, поэтому она задаётся прямоугольной областью, начальным углом и углом самой дуги. Кроме того, концы дуги могут соединяться радиусами с её центром, образуя сектор (стиль по умолчанию) или напрямую соединяться друг с другом, образуя сегмент.

Позиционные параметры метода create_arc задают две противоположные вершины прямоугольника. Именованные параметры:

start – начальный угол дуги в градусах;

extent – размер дуги в градусах. Дуга всегда рисуется в направлении против часовой стрелки;

width – толщина линии дуги;

outline – цвет дуги;

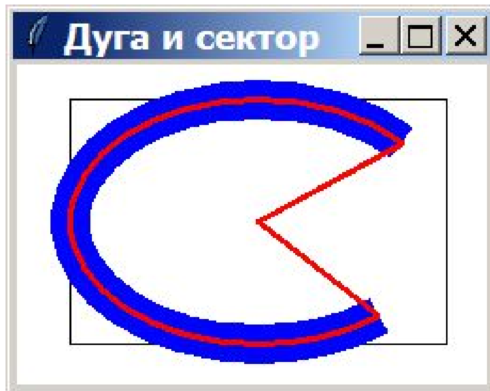
style – стиль графического объекта. По умолчанию style=PIESLICE (или 'pieslice') — рисуется сектор, если style=CHORD (или 'chord'), то рисуется сегмент, если style=ARC (или 'arc'), рисуется только дуга;

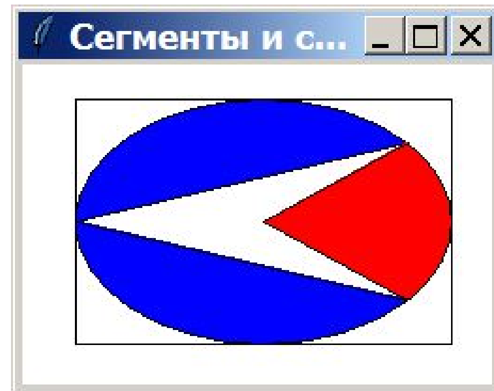
fill – цвет заполнения для сектора и сегмента. Если параметр fill отсутствует, то будет нарисован только контур графического объекта.

Пример 1: дуга и контур сектора.

```
from tkinter import *
# создаём окно приложения и холст
root=Tk()
root.title('Дуга и сектор')
canvas=Canvas(root,width=250,height=170,bg='white')
canvas.pack()

# рисуем описанный прямоугольник
```

```
rect=canvas.create_rectangle(30,20,230,150)
# рисуем дугу и сектор
arc = canvas.create_arc(30,20,230,150,
             start=40,extent=270,
             width=21,outline='blue',style='arc')
pie = canvas.create_arc(30,20,230,150,
             start=40,extent=270,
             width=3,outline='red')
```





Пример 2. Синие сегменты и красный сектор (правый рисунок)

```
ffrom tkinter import *
# создаём окно приложения и холст
root=Tk()
root.title('Сегменты и сектор')
canvas=Canvas(root, width=250, height=170, bg='white')
canvas.pack()

# рисуем описанный прямоугольник
rect=canvas.create_rectangle(30,20,230,150)
# рисуем два сегмента и сектор
seg1 = canvas.create_arc(30,20,230,150,
              start=40,extent=140,
              fill='blue',style='chord')
seg2 = canvas.create_arc(30,20,230,150,
              start=180,extent=140,
              fill='blue',style='chord')
pie  = canvas.create_arc(30,20,230,150,
              start=320,extent=80,fill='red')
```

Текст. Текст также является графическим объектом и создаётся методом create_text. Синтаксис вызова:

id = canvas.create_text(x,y, option=value,...)

где

id – идентификатор графического объекта;

x,y – координаты точки холста, к которой прикреплён графический объект. Расположение текста относительно этой точки задаётся параметром anchor (“якорь»).

anchor = CENTER (или 'center') - значение по умолчанию. Точка прикрепления совпадает с центром прямоугольника, в который вписан текст. Другие варианты имеют географическую мнемонику: N (north – север), W (west – запад), S (south – юг), E (east – восток) – середины соответствующих сторон прямоугольника. Точка прикрепления может также находиться в одном из углов прямоугольника. Угол задаётся комбинацией символов сторон, его образующих. Например, левый верхний угол будет точкой прикрепления при anchor=NW,

fill – цвет текста (по умолчанию — чёрный);

font – шрифт. Составной параметр, который задаёт гарнитуру, кегль (размер) и начертание символов текста. Например,

```
font=('Courier New', 28, 'italic')
```

задаёт гарнитуру Courier New, кегль 28 пунктов и курсивное начертание текста. Кроме курсива можно задать полужирное – bold, подчеркнутое – underline или зачеркнутое – overstrike начертание или их комбинацию.

Что касается набора шрифтов, то он определяется набором шрифтов, установленных в системе.

text – выводимая строка символов;

tags – теги графического объекта;

width – ограничение ширины области вывода текста. Если этот параметр отсутствует текст будет выводиться в одну строку, даже если он выходит за пределы холста. Если же ширина области вывода задана, то текст будет выводиться в несколько строк. В этом случае дополнительно можно задать значение параметру justify:

justify = LEFT (или 'left') – значение по умолчанию — выравнивание по левому краю;

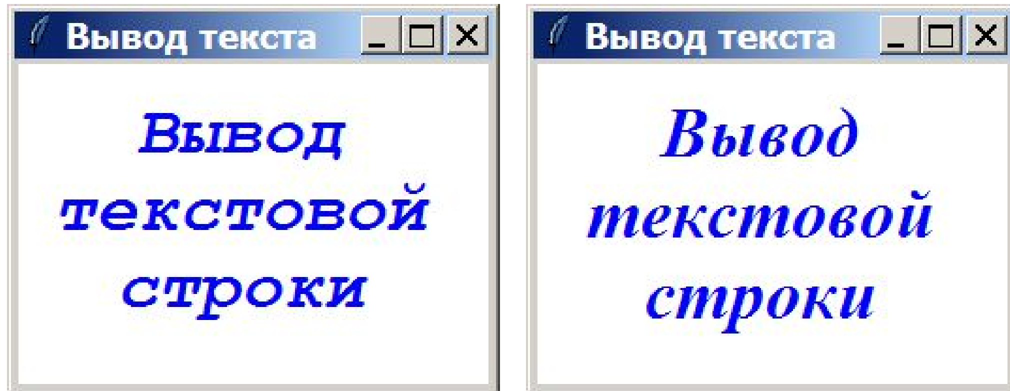
justify = RIGHT (или 'right') – выравнивание по правому краю;

justify = CENTER (или 'center') – выравнивание по центру.

Проиллюстрируем использование этих параметров на простом примере. Выведем в окно приложения текст в несколько строк и центрируем его.

```
from tkinter import *
# создаём окно приложения и холст
root=Tk()
root.title('Вывод текста')
canvas=Canvas(root, width=250, height=170, bg='white')
canvas.pack()
# выводим текст
```

```
text = canvas.create_text(120,80,
              text='Вывод текстовой строки',
              fill='blue', width=200, justify='center',
              font=('Courier New', 28, 'bold italic'))
```



Заменим теперь шрифт на Times New Roman (рисунок справа)

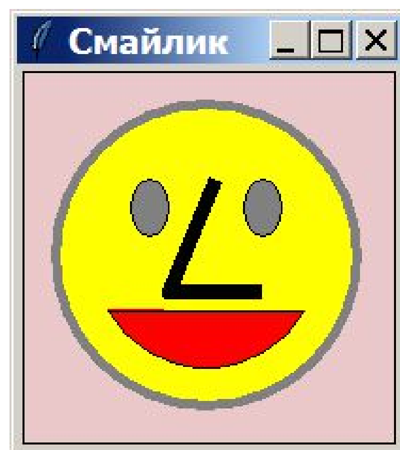
Кроме «обычных» символов шрифт может содержать символы других алфавитов и пиктограммы, например, математические знаки, шахматные фигуры и т.п. Чтобы вывести символы, отсутствующие на клавиатуре, нужно знать их код в кодировке Unicode. Например, белый король имеет шестнадцатеричный код \u2654, а чёрный ферзь - \u265b. Заметим, что слова «белый» и «чёрный» используется в шахматном смысле, так как мы можем в параметре fill указать любой цвет символа, например, синий.

```
from tkinter import *
# создаём окно приложения и холст
root=Tk()
root.title('Шахматы')
canvas=Canvas(root, width=250, height=120, bg='white')
canvas.pack()

# выводим шахматные фигуры
text = canvas.create_text(130,60,
            text='\u2654\u2655\u2656\u2657\u2658'+
            '\u2659\u265a\u265b\u265c\u265d\u265e\u265f',
            fill='blue', width=240, justify='center',
            font=('Courier New', 28, 'normal'))
```

Результат

Пример. Смайлик.

```
from tkinter import *
root=Tk()
root.title('Смайлик')
canvas = Canvas(root,width=200,height=200)
canvas.pack()
canvas.create_rectangle(3,3,200,200,
                   fill='pink',stipple='gray50')
canvas.create_oval(20,20,180,180,
                   fill='yellow',width=5,outline='gray')
canvas.create_oval(60,60,80,90,
                   fill='gray')
canvas.create_oval(120,60,140,90,
                   fill='gray')
canvas.create_arc(40,40,160,160,
                  start=210,extent=120,style=CHORD,
                  fill='red')
canvas.create_line(105,60,80,120,130,120,width=8)
```



Недостаток написанной программы — её привязка к конкретному месту и фиксированный размер. В следующем примере мы нарисуем несколько

праздничных воздушных шариков в разных местах и разного вида. Для этого напишем функцию ballon, которая получает координаты, размер и другие параметры, определяющие внешний вид шарика.

```
def ballon(x,y,size,**kw)
```

Параметры функции:

x,y – координаты начальной точки;
size – базовый размер;
**kw – набор ключевых параметров (пар option=value).

Через ключевые параметры будем передавать цвет шарика, текст и цвет надписи. Например, вызов

```
ballon(50,140,25,fill='red',text='МИР',ctext='white')
```

нарисует красный шарик с белой надписью, а вызов

```
ballon(100,130,30,fill='yellow',text='МАЙ')
```

нарисует желтый шарик с чёрной надписью.

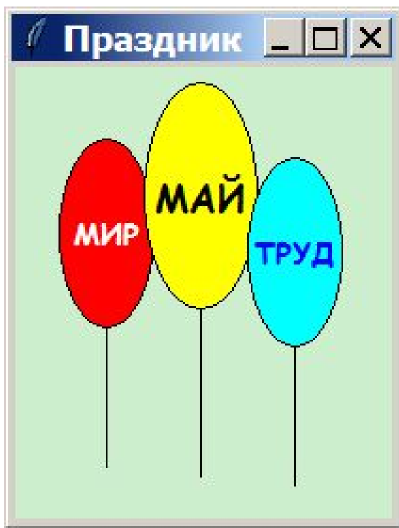
Напишем программу/

```
from tkinter import *
root=Tk()
root.title('Праздник')
canvas=Canvas(root,width=200,height=240,bg='#cceecc')
canvas.pack()

def ballon(x,y,size,**kw):
   line=canvas.create_line(x,y,x,y+3*size)
   oval=canvas.create_oval(x-size,y,x+size,
                           y-4*size,fill='')
   text=canvas.create_text(x,y-2*size,
               text='',fill='black',
               font=('Comic Sans MS',size//2,'bold'))
   if kw.get('fill')!=None:
      canvas.itemconfig(oval,fill=kw['fill'])
   if kw.get('text')!=None:
      canvas.itemconfig(text,text=kw['text'])
   if kw.get('ctext')!=None:
      canvas.itemconfig(text,fill=kw['ctext'])

ballon(50,140,25,fill='red',text='МИР',ctext='white')
ballon(100,130,30,fill='yellow',text='МАЙ')
ballon(150,150,25,fill='cyan',text='ТРУД',ctext='blue')
```

Заметим, что функция ballon сначала создаёт все графические составляющие объекта с некоторыми значениями параметров (это значения

по умолчанию). Затем анализируется словарь с ключевыми параметрами. Если при вызове функции будет указан параметр fill='цвет', то вызывается метод канвы itemconfig, который устанавливает цвет овала в соответствии со значением kw['fill']. Аналогично для остальных параметров.

Результат выполнения программы:



Задания для самостоятельного выполнения.

1) Напишите функцию roundrect, которая рисует прямоугольник со скруглёнными вершинами. Функция должна получать координаты вершин описанного прямоугольника и радиус скругления. Рассмотрите следующие варианты представления прямоугольника:

А) четыре отрезка и четыре дуги;

Б) сглаженная ломаная;

В) сглаженный полигон. В этом случае в список параметров можно будет добавить цвет внутренней области прямоугольника.

2) Добавьте возможность вывода текста в прямоугольник.

2. Пропорции и прогрессии

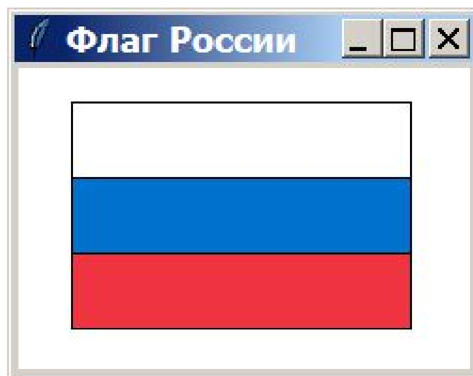
Добавим немного математики к программированию. Пусть перед нами стоит задача — нарисовать флаг России. Как известно он состоит из трёх полос одинакового размера — белой, синей и красной, а отношение ширины флага к его длине составляет 2:3. Федеральный закон о Государственном флаге Российской Федерации оттенки цветов не устанавливает, однако при изготовлении флага цвет полос должен выбираться из определённого стандартизированного набора. В цветовой гамме RGB эти оттенки задаются значениями компонент красного (red), зелёного (green), и синего (blue) цвета.

Стандартные цвета флага России: белая полоса — 255-255-255 (#ffffff), синяя полоса — 0-57-166 (#0039a6), красная полоса — 213-43-30 (#d52b1e).

Чтобы нарисовать флаг нам ещё нужно определить размеры составляющих его полос. Выберем в качестве параметра рисунка ширину одной полосы, которую обозначим через d. Тогда ширина всего флага равна 3*d, что составляет две части, т.е. одна часть равна 1.5*d, а три части (т.е., длина флага) равны 4.5*d.

Напишем программу рисования флага России.

```
from tkinter import *
root=Tk()
root.title('Флаг России')
canvas=Canvas(root,width=240,height=160,bg='white')
canvas.pack()

# -------- параметры ---------
(xs,ys)=(30,20); d=40; w=4.5*d
# ----------------------------
canvas.create_rectangle(xs,ys,xs+w,ys+d,fill='#ffffff')
ys+=d
canvas.create_rectangle(xs,ys,xs+w,ys+d,fill='#0072ce')
ys+=d
canvas.create_rectangle(xs,ys,xs+w,ys+d,fill='#ef3340')
```



(Цвет флага взят с сайта https://flagcolor.com/russia-flag-colors/)

В этой программе мы встречаем арифметическую прогрессию — переменная ys каждый раз увеличивается на одно и то же число. Последовательность значений можно получить используя формулу общего члена арифметической прогрессии, при этом ys выступает в роли начального значения: ys, ys+d, ys+2*d.

Получить арифметическую прогрессию можно с помощью встроенной функции range(start,stop,step) . В общем случае функция имеет три параметра:

start – начальное значение прогрессии;

stop — значение-ограничитель. Для возрастающей последовательности все значения прогрессии будут строго меньше значения stop. Для убывающей последовательности все значения будут строго больше ограничителя stop;

step – разность прогрессии. Если step>0 то получим возрастающую последовательность, если step<0 – то убывающую.

Один или два параметра функции может отсутствовать. Тогда они принимают значения по умолчанию.

range(start,stop) – тогда step = 1.

range(stop) – тогда start = 0, step = 1;

Примеры.

range(7) генерирует последовательность 0,1,2,3,4,5.6

range(3,7) генерирует последовательность 3,4,5,6

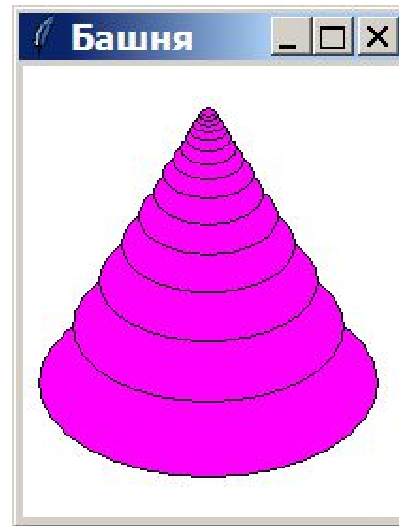
range(12,5,-2) генерирует последовательность 12, 10, 8, 6

Функция range чаще всего используется для организации цикла с параметром, т.е. step задаёт шаг изменения параметра цикла.

Пример. Нарисуем «мишень» - набор концентрированных окружностей:

```
from tkinter import *
root=Tk()
root.title('Мишень')
canvas=Canvas(root,width=200,height=200,bg='white')
canvas.pack()
# --------- параметры ---------
(xs,ys)=(100,100); R=90; dR=-20
# ---------------------------
for r in range(R,0,dR):
   canvas.create_oval(xs-r, ys-r, xs+r, ys+r,
                      width=10, outline='blue')
```

Будет нарисованы пять окружностей с радиусами 90, 70, 50, 30 и 10.



В геометрической прогрессии каждое следующее значение получается из предыдущего умножением его на некоторое число q — знаменатель прогрессии. Если q<1, то мы получим убывающую геометрическую прогрессию, если q>1, то мы получим возрастающую геометрическую прогрессию.

Пример. Построим «башню» из эллипсов, напоминающую башенки из гальки, которые дети выкладывают на берегу моря.

```
from tkinter import *
root=Tk()
root.title('Башня')
canvas=Canvas(root,width=200,height=240,bg='white')
canvas.pack()

# ---------------- параметры -------------
(xs,ys)=(100,170)  # центр нижнего эллипса
a=90; b=50; d=30   # полуоси эллипса и величина сдвига
q=0.8              # знаменатель прогрессии
# -----------------------------------------
while b>2:
   canvas.create_oval(xs-a, ys-b, xs+a, ys+b,
                      fill='magenta')
   ys-=d
   a*=q; b*=q; d*=q
```

Для дальнейшей работы напишем две функции. Первая из них получает координаты двух точек A и B плоскости и возвращает координаты точки Q, лежащей на отрезке, соединяющем эти точки. Положение результирующей точки будет определять параметр q, равный отношению длин отрезков AQ и AB. При условии 0<q<1 точка Q будет принадлежать отрезку AB.

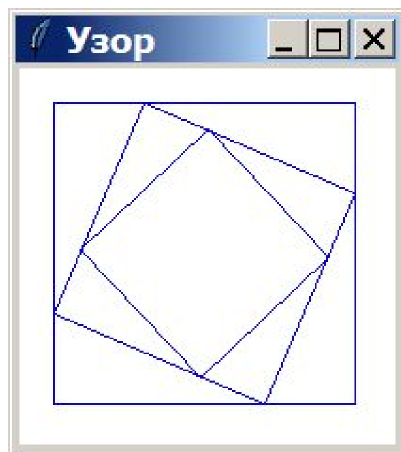
```
def inter(A,B,q)
    (xa,ya)=A
    (xb,yb)=B
    return (xa+q*(xb-xa), ya+q*(yb-ya))
```

Вторая функция получает координаты двух точек и целое число n – количество интервалов разбиения отрезка AB и возвращает список точек разбиения.

```
def partition(A,B,n):
    (xa,ya)=A
    (xb,yb)=B
    dx=(xb-xa)/n
    dy=(yb-ya)/n
    return [(xa+i*dx, ya+i*dy) for i in range(n+1)]
```

Используем эти функции при решении следующих задач.

Задача 1. Нарисовать узор, образованный последовательностью вписанных квадратов. Вершины каждого вписанного квадрата получаются в результате сдвига вершин предыдущего квадрата вдоль его сторон.



Решение.

```
from tkinter import *
root=Tk()
root.title('Узор')
canvas=Canvas(root, width=200, height=200, bg='white')
canvas.pack()

def inter(A,B,q):
    (xa,ya)=A
    (xb,yb)=B
    return (xa+q*(xb-xa), ya+q*(yb-ya))
```

```
L=[(20,20),(180, 20),(180,180),( 20,180)]
q=0.3; n=3
for i in range(n):
   canvas.create_line(L,L[0],fill='blue')
   L=[inter(L[i],L[(i+1)%4],q) for i in range(4)]
```

Пояснения. Сначала мы формируем список L вершин исходного квадрата. Затем задаём параметры рисунка:

q – безразмерный коэффициент сдвига;

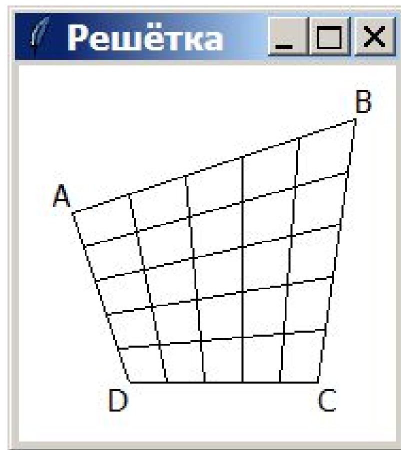
n – количество квадратов в узоре.

В цикле рисуем квадрат методом create_line. Чтобы квадрат получился замкнутым в параметрах метода указываем список L и повторяем начальную точку списка. После этого вычисляем координаты вершин следующего квадрата. Функция inter получает две последовательные вершины квадрата и коэффициент q и возвращает координаты вершины следующего квадрата. Отдельно следует пояснить выражение (i+1)%4, используемое в качестве индекса следующей вершины. Так как параметр цикла i изменяется от 0 до 3, то i+1 будет изменяться от 1 до 4, но вершины с индексом 4 не существует. На самом деле на последней итерации цикла нам нужно рассмотреть вершины L[3] и L[0], Таким образом, индекс второй вершины должен последовательно принимать значения 1, 2, 3, 0, т.е. должен равняться остатку от деления (i+1) на 4.

Изменяя значения q и n можно получить другие варианты узора. Можно также изменить координаты и число точек исходного списка L, практически не изменяя остальную часть программы. Исключение — выражения (i+1)%4 и range(4), так как они содержат значение 4 — число точек в списке L.

Чтобы избавиться от этого недостатка можно после создания списка L получить его длину с помощью функцию len: m = len(L), а затем вместо значения 4 использовать значение переменной m.

Задача 2. Дан выпуклый четырехугольник. Требуется разбить его на небольшие четырёхугольные элементы системой отрезков. Пример такого разбиения показан на рисунке.



Решение. Из рисунка понятно, что нужно соединить точки, лежащие на противоположных сторонах четырёхугольника. Чтобы получить точки разбиения сторон четырёхугольника, используем функцию partition.

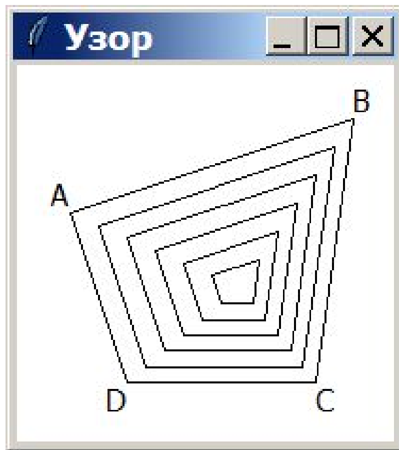
Приведём полный текст программы.

```
from tkinter import *
root=Tk()
root.title('Решётка')
canvas=Canvas(root, width=200, height=200, bg='white')
canvas.pack()

def partition(A,B,n):
    (xa,ya)=A
    (xb,yb)=B
    dx=(xb-xa)/n
    dy=(yb-ya)/n
    return [(xa+i*dx, ya+i*dy) for i in range(n+1)]

# Задаём координаты вершин четырёхугольника
# и отмечаем их на экране буквами A, B, C, D
A=( 30, 80)
B=(180, 30)
C=(160,170)
D=( 60,170)
canvas.create_text(A,text='A', anchor=SE)
canvas.create_text(B,text='B', anchor=SW)
canvas.create_text(C,text='C', anchor=NW)
canvas.create_text(D,text='D', anchor=NE)
n=5
# Разбиваем стороны AB и DC
AB=partition(A,B,n)
DC=partition(D,C,n)
```

```
# Соединяем пары точек линиями
for P,Q in zip(AB,DC):
   canvas.create_line(P,Q)
# Разбиваем стороны BC и AD
BC=partition(B,C,n)
AD=partition(A,D,n)
# Соединяем пары точек линиями
for P,Q in zip(BC,AD):
   canvas.create_line(P,Q)
```

Пояснение требует использование функции zip в заголовках циклов. Функция zip получает несколько кортежей или списков (в данном случае два списка) и возвращает особый объект — так называемый итератор, при каждом обращении к которому возвращается кортеж, составленный из элементов входных последовательностей с одним и тем же индексом. Например, в цикле for P,Q in zip(AB, DC) на первой итерации P=AB[0], Q=BC[0], при следующей P=AB[1], Q=BC[1] и т.д.

Задача 3. Дан произвольный многоугольник. Требуется заполнить его внутреннюю область меньшими многоугольниками, подобными исходному. Пример такого заполнения показан на рисунке.



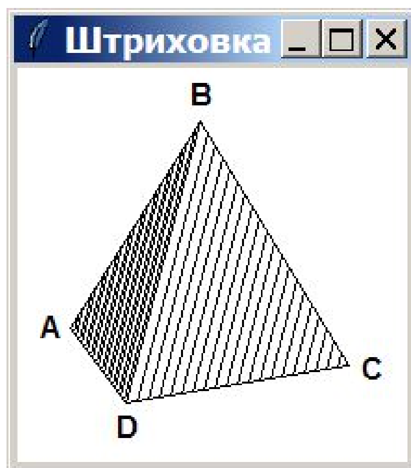
Решение. Выберем точку S внутри многоугольника в качестве центра гомотетии. Мысленно соединим её с вершинами исходного многоугольника. Каждый полученный отрезок разобьём на одинаковое число частей и соединим точки разбиения с одним и тем же индексом.

Реализуем этот алгоритм с использованием функции partition. Приведём только основную часть программы

```
# ----------- main ----------
A=( 30, 80)
B=(180, 30)
```

```
C=(160,170)
D=( 60,170)
canvas.create_text(A,text='A', anchor=SE)
canvas.create_text(B,text='B', anchor=SW)
canvas.create_text(C,text='C', anchor=NW)
canvas.create_text(D,text='D', anchor=NE)
S=(120,120)
n=6
AS=partition(A,S,n)
BS=partition(B,S,n)
CS=partition(C,S,n)
DS=partition(D,S,n)
for L in zip(AS,BS,CS,DS):
   canvas.create_polygon(L,outline='black',fill='')
```

Задача 4. Заштриховать четырёхугольник линиями, параллельными одной из диагоналей. Пример штриховки показан на рисунке ниже. Обратите внимание: наличие штриховки разной плотности создаёт объёмный эффект.



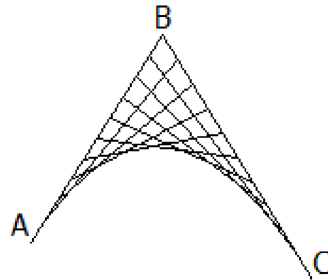
Решение. Задаём координаты вершин четырёхугольника ABCD. Создаём два списка. Один список объединяет точки сторон AB и BC, второй список объединяет точки разбиения сторон AD и DC. Для объединения (конкатенации) списков проще всего использовать знак операции сложения.

```
A=( 30,140)
B=(100, 30)
C=(180,160)
D=( 60,180)
canvas.create_line(A,B,C,D,A)
n=20
V1=partition(A,B,n)+partition(B,C,n)
V2=partition(A,D,n)+partition(D,C,n)
for P,Q in zip(V1,V2):
```

```
    canvas.create_line(P,Q)
```

Замечание. Разная плотность штриховки получилась за счёт разных размеров треугольников ABD и DBC. Однако плотность штриховки можно регулировать, задавая разные значения параметра n в этих треугольниках.

Задача 5. Заштриховать угол ABC перекрещивающимися отрезками. Пример:



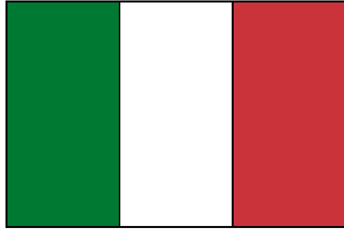
Решение

```
from tkinter import *
root=Tk()
root.title('Штриховка')
canvas=Canvas(root,width=210,height=200,bg='white')
canvas.pack()

def partition(A,B,n):
    (xa,ya)=A
    (xb,yb)=B
    dx=(xb-xa)/n
    dy=(yb-ya)/n
    return [(xa+i*dx, ya+i*dy) for i in range(n+1)]

A=( 30,140)
B=(100, 30)
C=(180,160)
n=10
V1=partition(A,B,n)
V2=partition(B,C,n)
for P,Q in zip(V1,V2):
    canvas.create_line(P,Q)
# вершины
canvas.create_text(A,text='A',anchor='se')
canvas.create_text(B,text='B',anchor='s')
canvas.create_text(C,text='C',anchor='sw')
```

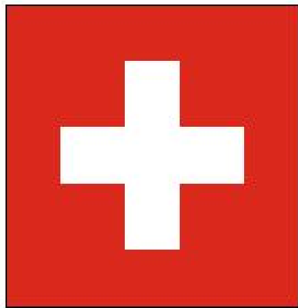
Задания для самостоятельной работы.

1. Нарисуйте флаги следующих стран:

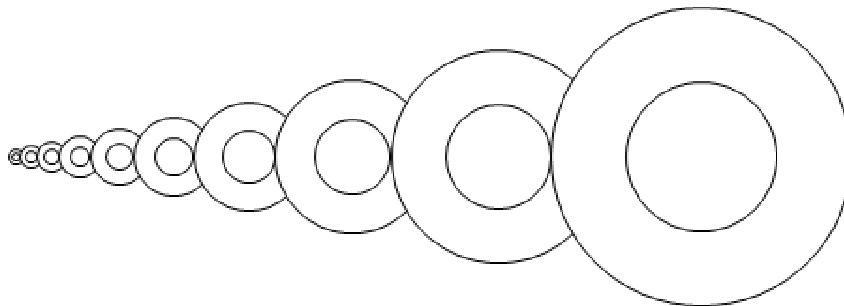
а) Италии. Указание: отношение длины к ширине флага 3:2, вертикальные полосы имеют одинаковую ширину; оттенки цветов: зелёный — #007a33, белый — чистый, красный — #cb333b .

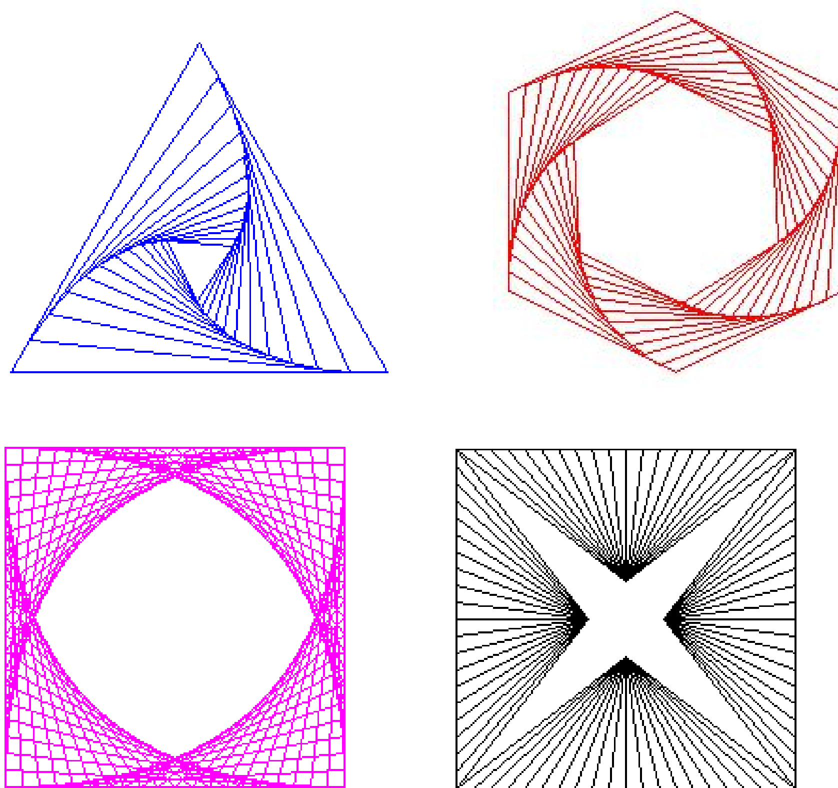


б) Швейцарии. Указание. Флаг Швейцарии имеет квадратную форму, ширина полос, образующих крест, равна 3/16, а длина — 5/8 от размера флага. Оттенки цветов: красный — #da291c, белый — чистый.



2. Написать программы для рисования следующих узоров:

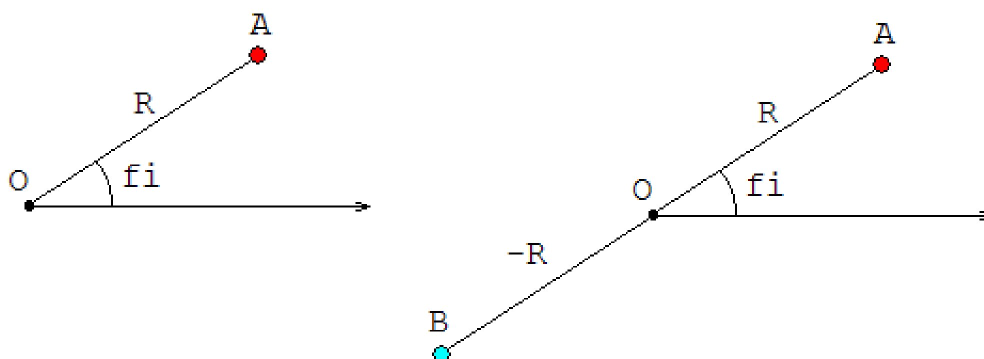




3. Полярная система координат

Экранная система координат является декартовой. с той лишь разницей, что ось Oy направлена вниз, а не вверх. Однако на плоскости можно ввести и другую систему координат, которая называется полярной системой координат.

В этой системе координат есть только одна ось – полярная ось. Обычно её отождествляют с положительной полуосью Ox. Любая точка A плоскости задаётся полярным углом fi и расстоянием R от начала координат – полярным радиусом.



В случае отрицательных значений R соответствующее расстояние откладывается в противоположном направлении (точка B на следующем рисунке). Заметим, что угол fi тоже может быть отрицательным. Отрицательные значения полярного угла откладываются от полярной оси по часовой стрелке, а положительные – против часовой стрелки.

В полярной системе очень просто реализовать операцию поворота на произвольный угол, поэтому её удобно использовать для рисования узоров, имеющих поворотную симметрию, т. е., узоров, переходящих в себя при поворотах на определённые углы. Так же просто реализуется и изменение размеров (масштабирование) рисунка. Сложности возникают при переходе из одной системы координат в другую.

Напишем поэтому функции перехода из полярной системы координат в экранную и наоборот. Обозначим:

(xs, ys) – начало полярной системы координат;

(x, y) – координаты некоторой точки. Заметим, что и точка (xs, ys) и точка (x, y) заданы в экранной системе координат;

R, fi – координаты той же точки в полярной системе координат с началом в точке (xs, ys). Для удобства угол fi будем задавать в градусах.

Воспользуемся известными математическими формулами для перехода из полярной системы координат в декартову, но сразу приспособим их для перехода в экранную систему координат. Учтём также, что аргумент любой тригонометрической функции должен задаваться в радианах:

x = xs + R*cos(pi*fi/180)

y = ys – R*sin(pi*fi/180)

Вместо явного преобразования угла в радианную меру можно воспользоваться функцией radians модуля math:

x = xs + R*cos(radians(fi))

y = ys – R*sin(radians(fi))

Придумаем название и напишем первый вариант функции перехода. Название не должно быть длинным, но в то же время должно отражать сущность функции и легко запоминаться. Обозначим полярную систему координат аббревиатурой ps, а декартову — ds, тогда функцию можно назвать ps2ds (цифра 2 заменяет английское to для большей выразительности имени функции).

```
def ps2ds(S,A):     # A - точка (кортеж)
    (xs,ys)=S
    (R,fi)=A
    return (xs + R*cos(pi*fi/180), ys - R*sin(pi*fi/180))
```

Функция возвращает кортеж с экранными координатами точки A.

Можно обобщить функцию для работы со списком точек

```
def ps2ds(S,L):     # L – список точек (кортежей)
   (xs,ys)=S
   return [(xs + R*cos(pi*fi/180),
            ys – R*sin(pi*fi/180))
            for (R,fi) in L]
```

однако тогда отдельную точку нужно будет передавать как список из одного кортежа, что может оказаться неудобным.

Объединим оба варианта в один. Чтобы распознать, какой объект получила функция — кортеж или список кортежей, воспользуемся функцией type(объект), которая возвращает тип объекта. Если это кортеж, то функция вернёт имя tuple, если список, то функция вернёт имя list.

```
# функция перехода к декартовым координатам
def ps2ds(S,L):
   """
   S - центр системы координат
   L - точка (кортеж) или список точек (кортежей)
       в полярной системе координат с центром в точке S
   результат - точка (кортеж) или список точек (кортежей)
       в экранной системе координат
   """
   (xs,ys)=S
   if type(L) == tuple:
      (R,fi)=L
      return (xs + R*cos(pi*fi/180),
              ys - R*sin(pi*fi/180))
   if type(L) == list:
      return [(xs + R*cos(pi*fi/180),
               ys – R*sin(pi*fi/180)) for (R,fi) in L]
```

Строки заключённые в тройные кавычки (одинарные или двойные) размещаются после заголовка функции и используются системой помощи для вывода информации о функции (УТОЧНИТЬ!).

Программу обратного перевода назовём по аналогии ds2ps.

\# функция перехода к полярным координатам

```
def ds2ps(S,L):
   """
   S - центр системы координат
   L - точка (кортеж)или список точек (кортежей)
       в экранной системе координат
   результат - точка (кортеж) или список точек (кортежей)
       в полярной системе координат с центром в точке S
   """
   (xs,ys)=S
```

```
    grd =180/pi
    if type(L) == tuple:
        (x,y)=L
        return (sqrt((x-xs)**2+(y-ys)**2),
                -atan2(y-ys,x-xs)*grd)
    if type(L) == list:
        return [(sqrt((x-xs)**2+(y-ys)**2),
                 -atan2(y-ys,x-xs)*grd) for (x,y) in L]
```

Функция возвращает координаты точки — кортеж (R,fi) или список кортежей того же вида. Значение R вычисляется по теореме Пифагора, а для вычисления угла fi используется встроенная функция арктангенса atan2(y,x). Она возвращает радианную меру угла точки (x,y), заданной в декартовой (математической) системе координат. Полученный результат умножается на коэффициент пересчета радиан в градусы. Знак минус обусловлен особенностью экранной системы координат (направлением оси Oy).

Рассмотрим несколько задач.

Задача 1. Нарисовать пятиугольную звезду.

Решение. Поместим начало полярной системы координат в центр звезды. Все вершины звезды будут находиться на одном и том же расстоянии от центра, а угол между направлениями на соседние вершины будет равен 360/5 = 72 градуса (в общем случае 360/n, где n – число вершин звездчатого многоугольника).

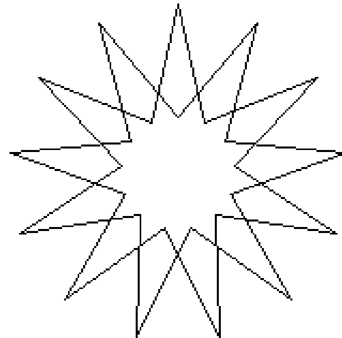
Пронумеруем вершины звезды числами от 0 до 4 и соединим в цикле точки с номерами 0 и 2, 1 и 3, 2 и 4, 3 и 0, 4 и 1. В общем случае будем соединять вершины с номерами i и j, где j вычисляется по формуле (i+k)%n.

Значения n (число вершин многоугольника) и k (количество дуг, стягиваемых одной стороной многоугольника) являются параметрами

рисунка и определяют его вид. В нашей задаче n=5, k=2. Приведём основную часть программы построения звезды (без дополнительных элементов):

```
# задаём параметры рисунка
n=5; k=2; R=80
# задаём начало координат
(xs,ys)= S = (100,105)
# задаём полярный угол «нулевой» вершины звезды
fi = 90
# задаём приращение угла
df = 360/n
# вычисляем полярные координаты всех вершин
V = [(R,fi+i*df) for i in range(n)]
# переводим их в экранные координаты
pts = ps2ds(S,V)
# рисуем звездчатый многоугольник
for i in range(n):
   j=(i+k)%n
   canvas.create_line(pts[i],pts[j],width=2)
```

Задача 2. Нарисовать узор, используя многократный поворот некоторой ломаной вокруг центра узора. Пример показан на рисунке ниже.



Решение. Проанализируем рисунок. Он состоит из 13 «шипов», каждый из которых есть ломаная из двух звеньев. Зададим один «шип» координатами его вершин в полярной системе, начало которой (точку S) разместим в центре многоугольника. Заметим, что координаты в полярной системе не зависят от координат точки S в экранной системе.

```
V = [(R1,90-df),(R2,90),(R1,90+df)]
```

Здесь R1 – радиус, на котором расположены «внутренние» вершины многоугольника, R2 – радиус, на котором расположены «внешние» вершины многоугольника, а угол df = 360/n, где n – количество «шипов» (в этом примере n=13).

Теперь вычислим экранные координаты вершин ломаной

```
   pts = ps2ds(S,V)
```

и нарисуем её

```
canvas.create_line(pts)
```

Выполним поворот ломаной на угол df. Поскольку координаты вершин ломаной мы задали в полярной системе координат, поворот сводится к прибавлению угла df к текущей угловой координате каждой вершины.

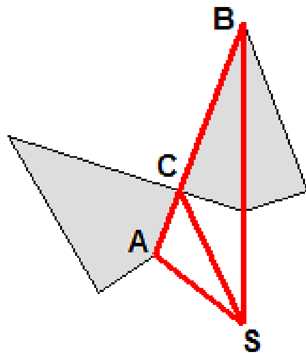
```
V = [(R,fi+df) for (R,fi) in V]
```

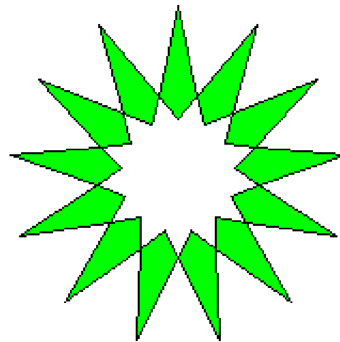
Осталось только повторить три последних оператора в цикле.

Приведём основную часть программы.

```
# задаём параметры многоугольника
n=13; R1=30; R2=80; df=360/n
# задаём начало координат
S = (100,100)
V = [(R1,90-df),(R2,90),(R1,90+df)]
for i in range(n):
   pts = ps2ds(S,V)
   canvas.create_line(pts)
   V = [(R,fi+df) for (R,fi) in V]
```

Есть и другой вариант расшифровки узора. Можно считать, что он состоит из неправильных четырёхугольников — полигонов. Этот полигон зададим четырьмя точками, расположенными на трёх радиусах. Внутренний радиус по-прежнему обозначим через R1, наружный — R2. Промежуточный радиус можно задать «на глазок», но лучше воспользоваться следующими соображениями. В треугольнике SAB отрезок SC является биссектрисой, SA=R1, SB=R2, угол ASB равен rd радиан, тогда длину SC можно определить по формуле SC = 2*R1*R2*cos(rd/2)/(R1+R2).





Приведём основную часть программы.

```
# задаём параметры рисунка
n=13; R1=30; R2=90
df=360/n; rd=pi*df/180
```

```
SC = 2*R1*R2*cos(rd/2)/(R1+R2)
# задаём начало координат
S = (100,100)
V = [(R1,90),(SC,90-df/2),(R2,90),(SC,90+df/2)]
for i in range(n):
   pts = ps2ds(S,V)
   canvas.create_polygon(pts,outline='black',fill='lime')
   V = [(R,fi+df) for (R,fi) in V]
```

Рассмотрим теперь ситуацию, когда вершины ломаной задаются в декартовой системе координат. В этом случае сначала координаты вершин ломаной нужно перевести в полярную систему, затем выполнить поворот ломаной и вернуться в декартову систему координат. Объединим эти операции в одну функцию и добавим её в модуль geom_sys.

```
def turn(S, L, df):
   """
   S - центр поворота
   L - точка (кортеж) или список точек (кортежей)
   df - угол поворота. При df>0 поворот против часовой
       стрелки, при df<0 - по часовой стрелке
   результат - точка (кортеж) или список точек (кортежей)
       после поворота
   """
   P = ds2ps(S,L)
   if type(P)==tuple:
      (R,fi)=P
      return ps2ds(S,(R,fi+df))
   if type(P)==list:
      W=[(R,fi+df) for R,fi in P]
      return ps2ds(S,W)
```

Приведём пример. Зададим шаблон ломаной. Координаты вершин зададим в условных единицах — «клетках».

```
V=[(1,1),(9,2),(5,0),(9,-2),(1,-1)]
```

Зададим размер клетки q и точку отсчета S:

```
q=10
(sx,sy)=S=(100,100)
```

тогда реальные координаты вершин ломаной вычисляются так:

```
L=[(sx+q*x,sy+q*y) for (x,y) in V]
```

Фактически мы применили к шаблону операции масштабирования и параллельного переноса ломаной.

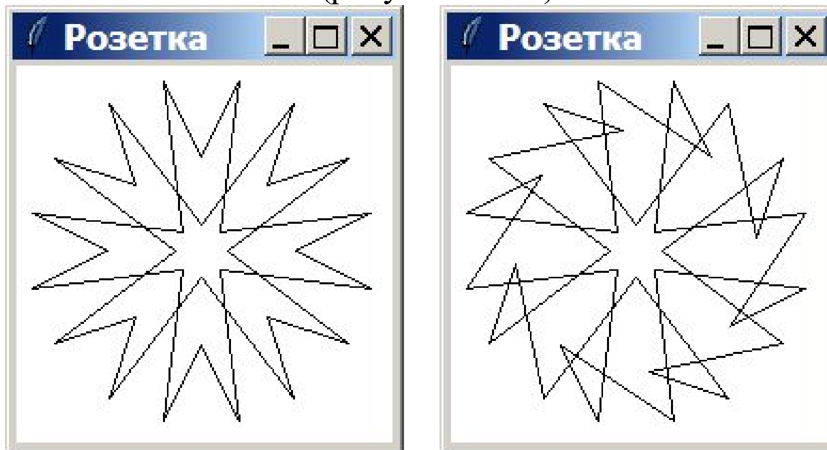
Осталось написать цикл, в котором реализовать вывод на экран и поворот ломаной на угол df=360/n:

```
for i in range(n):
   canvas.create_line(L)
   L=turn(S,L,df)
```

Приведём текст программы

```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Розетка')
canvas=Canvas(root,width=200, height=200,bg='white')
canvas.pack()

q=10; n=8; df=360/n
(sx,sy)=S=(100,100)
V=[(1,1),(9,2),(5,4),(9,-2),(1,-1)]
L=[(sx+q*x,sy+q*y) for (x,y) in V]
for i in range(n):
   canvas.create_line(L)
   L=turn(S,L,df)
```

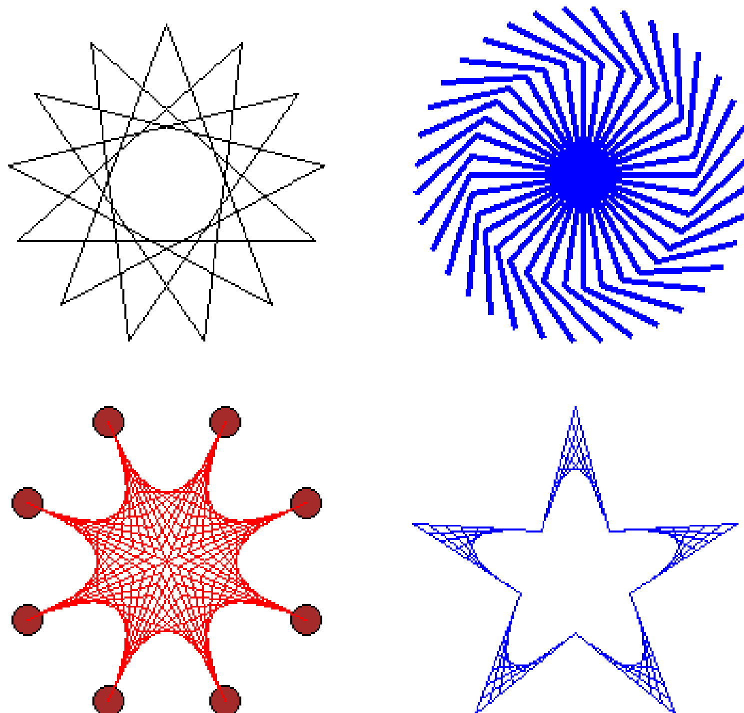
и результат её выполнения (рисунок слева).



Задания для самостоятельной работы

1) Измените в шаблоне координаты одной из точек так, чтобы получить узор, представленный на правом рисунке.

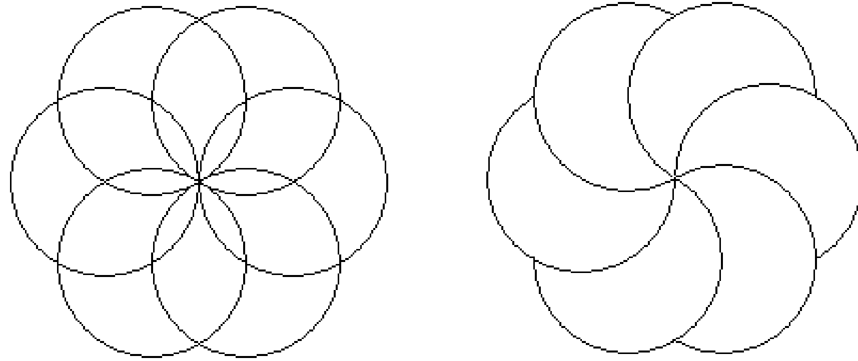
2) Напишите программы рисования следующих узоров;



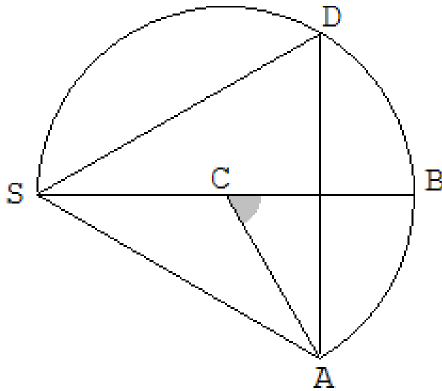
4. Узоры из окружностей и дуг

Простые узоры. Начнём с простых узоров. Пример: узор из окружностей:

```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Узор')
canvas=Canvas(root,width=240,height=240,bg='white')
canvas.pack()
R=50; n=6
df=360/n
S=(120,120)
fi=0
for k in range(n):
    (xc,yc)=ps2ds(S,(R,fi))
    canvas.create_oval(xc-R,yc-R,xc+R,yc+R)
    fi=fi+df
```



Чтобы получить узор на правом рисунке, нужно сначала разобраться с геометрией узора. Выделим элемент узора — дугу ABDS. Чтобы получить следующий элемент нужно эту дугу повернуть на угол ASD = df, при этом точка A совместится с точкой D. Положение дуги определяется углом fi, который диаметр SB образует с осью Ox. Соединим центр дуги — точку C с точкой A – начальной точкой дуги. Тогда угол ACB является центральным углом, опирающимся на дугу AB, а угол ASB – вписанным углом, опирающимся на ту же дугу. По известной теореме планиметрии, вписанный угол равен половине центрального угла, опирающегося на ту же дугу. Следовательно, угол ACB равен удвоенному углу ASB и равен df. Отсюда получим параметры дуги ABDS: начальный угол start = fi-df, протяженность дуги extent=180+df.



Внесём изменения в программу. Вместо окружностей создадим n дуг с вычисленными выше параметрами:

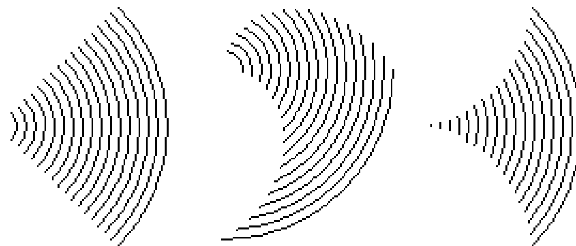
```
canvas.create_arc(xc-R,yc-R,xc+R,yc+R,style='arc',
                  start=fi-df,extent=180+df)
```

Дугами можно заштриховать область на экране. Следующая программа заштриховывает разные области. Для первой и второй области угловой

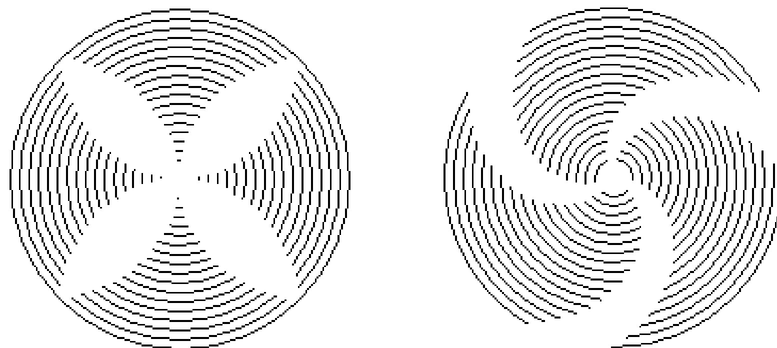
размер для всех дуг один и тот же — по умолчанию 90 градусов, но во второй области меняется начальный угол дуги. В третьей области изменяется как начальный угол, так и размер дуги.

```
(sx,sy)=(50,100);
for R in range(10,91,5):
   canvas.create_arc(sx-R,sy-R,sx+R,sy+R,
                  start=-45,style='arc')
(sx,sy)=(170,70);
for R in range(10,91,5):
   canvas.create_arc(sx-R,sy-R,sx+R,sy+R,
                  start=-R,style='arc')
(sx,sy)=(270,100);
for R in range(10,91,5):
   canvas.create_arc(sx-R,sy-R,sx+R,sy+R,
                  start=-R/2,extent=R,style='arc')
```

Результат:



Из заштрихованных областей, как из кирпичиков, легко сложить новые узоры:



Приведём программы рисования этих узоров. Рисунок слева:

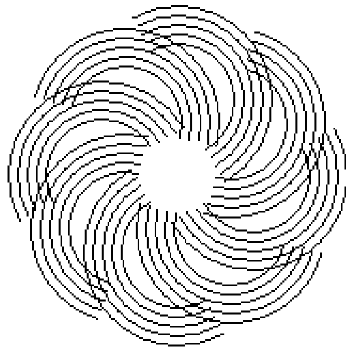
```
(sx,sy)=(120,120)
for k in range(4):
   for R in range(10,91,5):
      canvas.create_arc(sx-R,sy-R,sx+R,sy+R,
                  start=k*90-R/2,extent=R,style='arc')
```

Рисунок справа:

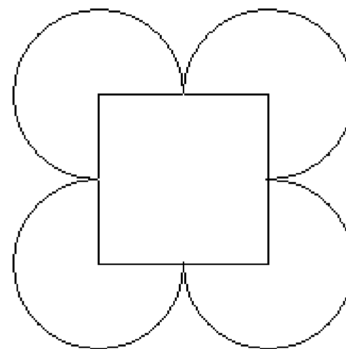
```
(sx,sy)=(120,120)
for k in range(3):
   for R in range(10,91,5):
      canvas.create_arc(sx-R,sy-R,sx+R,sy+R,
                        start=k*120-R,style='arc')
```

Задания для самостоятельной работы

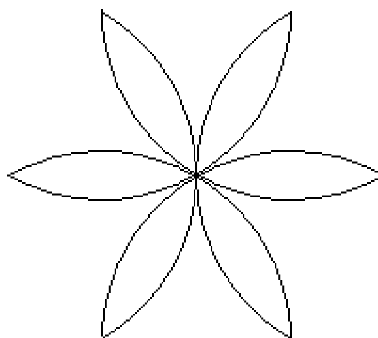
Написать программы рисования следующих узоров



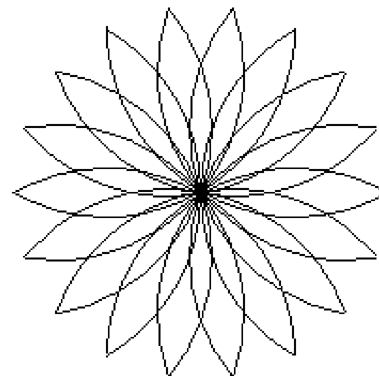
А)



Б)



В)



Г)

Разбиение дуги окружности. Аналогично отрезку дугу также можно разделить на равные части. Так как в модуле tkinter дуга задаётся углом начальной точки и угловой величиной, то промежуточные точки определяются "промежуточным" углом, который вычисляется по формуле

```
fi = u0 + k/n*du,  k = 0, 1, 2, …, n
```

где

- u0 – начальный угол дуги;
- du – угловой размер дуги;
- n – число отрезков, на которые разбивается дуга.

Для вычисления экранных координат полученной точки используем функцию `ps2ds`. Обозначим

S, R – координаты центра и радиус окружности;

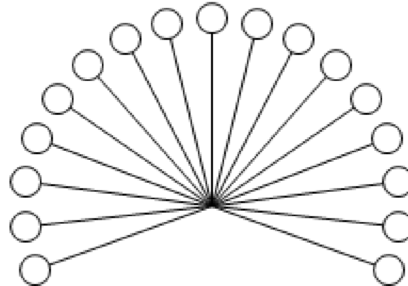
fi – полярный угол точки окружности;

x,y – декартовы координаты той же точки.

Тогда вызов функции ps2ds будет иметь следующий вид:

```
(x,y) = ps2ds(S,(R,fi));
```

Покажем на примере использование этой процедуры. Нарисуем «хвост павлина»



Для этого соединим центр окружности с точками, расположенными на дуге от -20 до 200 градусов, а потом нарисуем небольшие круги с центрами в этих же точках. Угловой размер дуги равен 220 градусов.

```
# задаём параметры разбиения
(xs,ys)=(120,160); R=100;
n=16; u1=-20; du=220
# рисуем узор
for k in range(n+1):
    fi=u1+k/n*du
    (x,y)=ps2ds((xs,ys),(R,fi))
    canvas.create_line(xs,ys,x,y)
    canvas.create_oval(x-8,y-8,x+8,y+8,fill='white')
```

Проводить линии к дуге можно из любой точки на экране. Приведём пример. Обозначим:

xs, ys, R – центр и радиус круга;

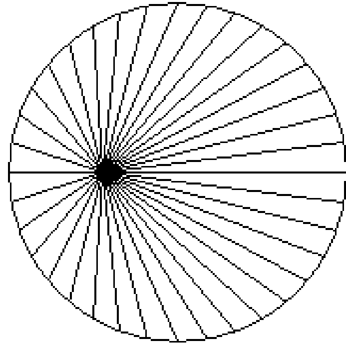
xa, ya – начальная точка всех лучей;

n – количество частей разбиения окружности

Напишем основную часть программы:

```
# задаём параметры узора
(xa,ya)=(80,120);        # точка выхода лучей
(xs,ys)=(120,120); R=90; # центр и радиус круга
n=36; u0=0; du=360
# рисуем узор
```

```
canvas.create_oval(xs-R,ys-R,xs+R,ys+R)
for k in range(n+1):
    fi=u0+k/n*du
    (x,y)=ps2ds((xs,ys),(R,fi))
    canvas.create_line(xa,ya,x,y)
```



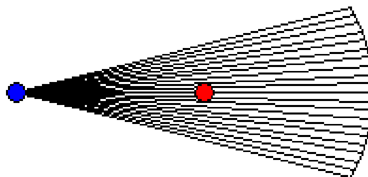
Вместо целой окружности можно взять её дугу, при этом точка выхода лучей и центр дуги могут не совпадать.

```
# задаём параметры разбиения
(xa,ya)=(20,120)          # точка выхода лучей
(xs,ys)=(120,120); R=90; # центр и радиус дуги
n=20; u0=-30; du=60
# рисуем узор
canvas.create_arc(xs-R,ys-R,xs+R,ys+R,
                  start=u0,extent=du,style=ARC)
for k in range(n+1):
    fi=u0+k/n*du
    (x,y)=ps2ds((xs,ys),(R,fi))
    canvas.create_line(xa,ya,x,y)
```

Для наглядности точку выхода лучей отметим синим, а центр дуги — красным цветом.

```
canvas.create_oval(xs-5,ys-5,xs+5,ys+5,fill='red')
canvas.create_oval(xa-5,ya-5,xa+5,ya+5,fill='blue')
```

Получим:



Из таких отдельных элементов можно составлять новые узоры. Для удобства напишем функцию sector, рисующую такой элемент.

Параметры функции sector:

obj – графический контекст — «холст»;

A – точка выхода лучей;
S, R – центр и радиус дуги;
n – количество частей разбиения дуги сектора;
u0 – начальный угол дуги;
du – размер дуги;
cv – цвет дуги;
w – толщина линий (по умолчанию равна 1).

```
def sector(obj, A, S, R, n, u0, du, cv, w=1):
    (xa,ya)=A; (xs,ys)=S
    obj.create_arc(xs-R,ys-R,xs+R,ys+R,
                   start=u0,extent=du,style=ARC,
                   outline=cv,width=w)
    for k in range(n+1):
       fi=u0+k/n*du
       (x,y)=ps2ds(S,(R,fi))
       obj.create_line(A,(x,y),fill=cv,width=w)
```

Из-за большого числа параметров пользоваться функцией не очень удобно. Как можно упростить вызов функции? Один прием мы уже использовали — параметр w – толщина линий — задан равным 1 по умолчанию. Это значит, что при вызове функции можно не указывать значение этого параметра, и тогда он будет равен единице.

Второй прием — при вызове функции именовать её параметры (имена параметров задаются при описании функции!). Такие параметры называются ключевыми, так как имеют вид «ключ=значение». В этом случае порядок появления параметров в списке несущественен. Однако нужно соблюдать правило: все параметры, не использующие при вызове своё название, должны идти перед ключевыми и в том порядке, в котором они заданы при описании функции. Приведем примеры.

В первом примере мы сохраним порядок параметров

```
sector(canvas, A=(100,100), S=(150,100),
               R=70, n=6, u0=-60, du=120, cv='brown',w=3)
```

Во втором — изменим произвольно их порядок.

```
sector(canvas, A=(50,100), R=70, n=6, u0=-60, du=120,
               S=(150,100), w=3, cv='brown')
```

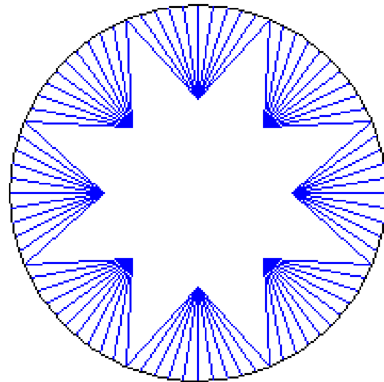
Результат и в первом и во втором примере будет одним и тем же. Заметим, что в этих примерах значение canvas нельзя переставить в другое место, так как мы не воспользовались именем obj этого параметра.

В заключение приведем пример использования функции sector.

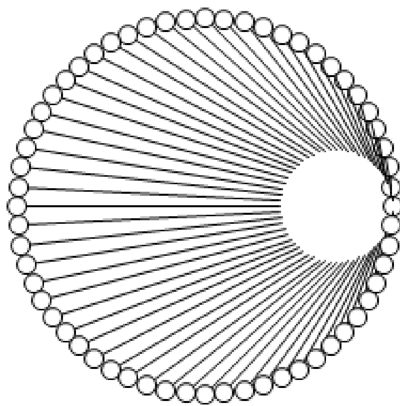
```
S=(120,120); R=100
```

```
m=8; df=180/m; fi=0
for k in range(m):
   A=ps2ds(S,(0.5*R,fi))
   sector(canvas,A,S,R,10,fi-df,2*df,cv='blue')
   fi=fi+2*df
```

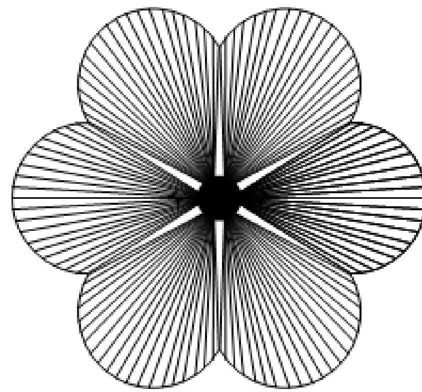
Результат :



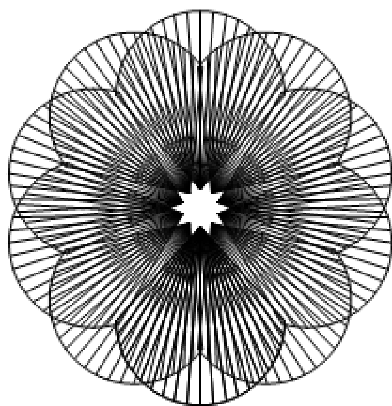
Задание. Нарисовать следующие узоры.

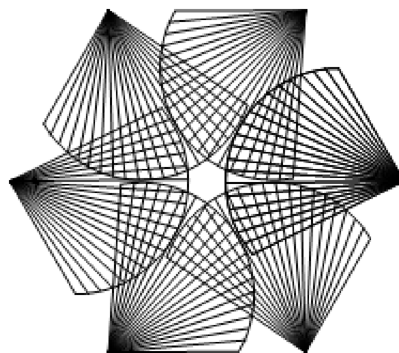


А)



Б)





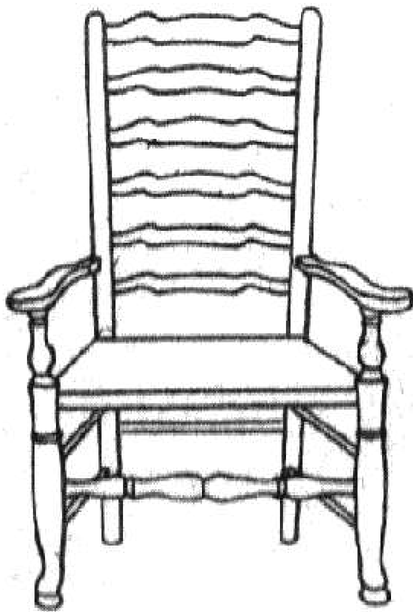
В) Г)

5. Зеркальная симметрия

В живой природе, предметах быта, одежде, архитектуре часто встречаются симметричные (обычно с некоторой погрешностью) объекты. Примеры:

Бабочка

Платок



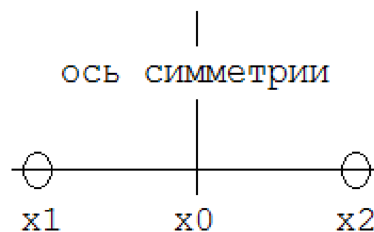
Кресло

Главное здание МГУ на Ленинских горах

На каждом из этих изображений можно провести прямую линию так, что если перегнуть рисунок по этой линии, то обе его половинки точно наложатся друг на друга. Эта линия называется осью симметрии. Расстояние от линии симметрии до одинаковых элементов в левой и правой частях изображения одинаково.

Рассмотрим случай, когда ось симметрии вертикальна. Пусть она проходит через точку x0, а точки x1 и x2 расположены симметрично относительно этой оси. Тогда длина отрезка [x0; x2] равна длине отрезка [x1; x0]. Если обозначить эти длины через d, то x1 и x2 будут вычисляться по следующим формулам:

```
x1 = x0 - d
x2 = x0 + d.
```



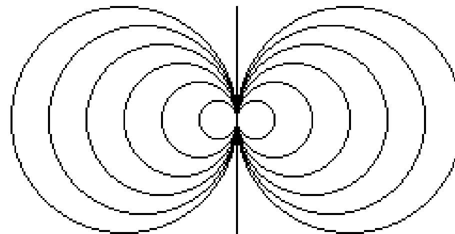
Если сложить обе формулы, получим, что x1 + x2 = 2*x0, откуда можно выразить x2 через x1: x2 = 2*x0 – x1. Y-координаты точек в этом случае не изменяются.

В случае, когда ось симметрии горизонтальна, можно вывести аналогичные формулы уже для вычисления y-координат точек узора:

```
y1 = y0 - d
y2 = y0 + d.
```

В этом случае также можно выразить y2 через y1: y2 = 2*y0 – y1. X-координаты точек в этом случае не изменяются.

Задача 3. Нарисовать следующий узор:



Решение. Окружностями удобнее задавать координатами центра и величиной радиуса. Напишем, поэтому, свою функцию для рисования кругов и окружностей. Функция create_circle будет получать:

obj – холст для рисования;

S – центр круга (окружности);

R – радиус круга (окружности);

params – набор именованных параметров для передачи во встроенную функцию create_oval. Фактически этот параметр является словарём, поэтому при объявлении функции этот параметр стоит последним и предваряется символами **. Такой подход позволяет написанной программистом функции пользоваться теми же возможностями, что и встроенная функция.

```
def create_circle(obj, S, R,**params):
  (xs,ys) = S
  return obj.create_oval(xs-R,ys-R,xs+R,ys+R,params)
```

Обозначим через x0, y0 точку, через которую проходит вертикальная ось симметрии, через R — радиус окружности, через dR – шаг изменения радиуса, через n – число кругов в одной половине рисунка.

Напишем программу.

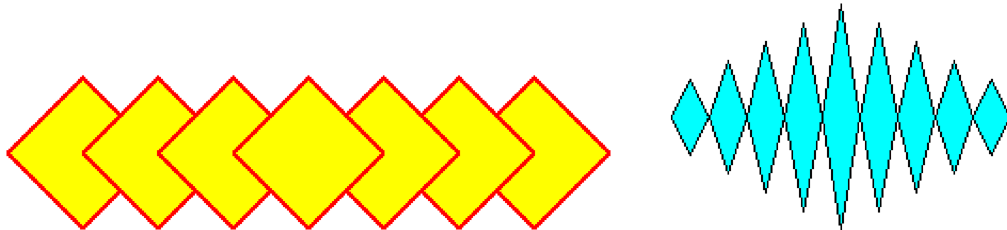
```
# импорт графической библиотеки Питона
from tkinter import *
```

```
# создание окна приложения с "холстом" для рисования
root=Tk()
root.title('Симметрия')
canvas=Canvas(root, width=300, height=200, bg='white')
canvas.pack()

(x0,y0)=(150,100); n=6; dR=10
for i in range(n,0,-1):
   R=i*dR
   create_circle(canvas, (x0-R,y0), R, fill='')
   create_circle(canvas, (x0+R,y0), R, fill='')
canvas.create_line(x0,y0-n*dR, x0,y0+n*dR)
```

Задания для самостоятельной работы

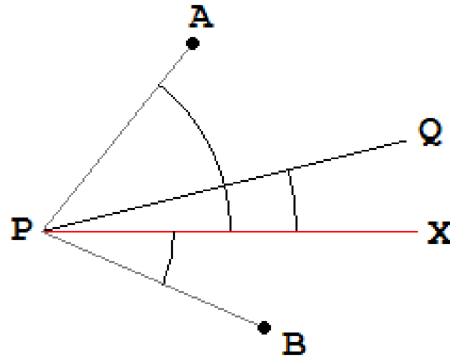
1) Напишите функцию create_romb(obj, S, a, b, **params), которая рисует ромб с диагоналями, параллельными осям экранной системы координат. Точка S есть точка пересечения диагоналей ромба, a и b – половины горизонтальной и вертикальной диагоналей.

2) Используя эту функцию и зеркальную симметрию, нарисуйте следующие узоры:



Общий случай. Рассмотрим на плоскости прямую, проходящую через точки P и Q. Вне прямой возьмём точку A. Найдём точку B, симметричную точке A относительно указанной прямой. Все точки задаются в экранной системе координат.

Введём полярную систему координат с началом в точке P. Обозначим (Rq, fq) – полярные координаты точки Q, (Ra, fa) – полярные координаты точки A в этой системе. Тогда угол APQ будет равен fa-fq, а угол APB – вдвое больше, т.е. 2*fa – 2*fq. Теперь легко получить полярный угол точки B. Он равен fa - (2*fa – 2*fq) = 2*fq – fa. Так как расстояние от точки P до точек A и B одинаково, то Rb = Ra.



Сформулируем порядок действий.

1) находим координаты точек Q и A в полярной системе координат с началом в точке P;

2) вычисляем полярные координаты точки B;

3) переводим полученные координаты в экранную систему координат.

```
(R1,fq)=ds2ps(P,Q)
(R2,fa)=ds2ps(P,A)
B=ps2ds(P,(R2,2*fq-fa))
```

Для проверки правильности выполним несколько тестовых расчетов для тех случаев, когда результат легко получить вручную.

а) симметрия относительно горизонтальной прямой:

Исходные данные: P=(100,100); Q=(200,100); A=(150,50)

Результат: B = (150,150)

б) симметрия относительно вертикальной прямой

Исходные данные: P=(100,100); Q=(100,200); A=(50,50)

Результат; B = (150,50)

б) симметрия относительно наклонной прямой

Исходные данные: P=(120,20); Q=(60,140); A=(120,120)

Результат; B = (40,80)

Напишем функцию, которая получает точку или список точек и вычисляет координаты симметричных относительно заданной прямой точек. Для удобства использования добавим её в модуль geom_sys.

```
def symmetry(P,Q,L):
    (rq,fq)=ds2ps(P,Q)
    if type(L)==tuple:
        (rw,fw)=ds2ps(P,L)
        return (ps2ds(P,(rw,2*fq-fw)))
    if type(L)==list:
        res=[]
        for W in L:
```

```
        (rw,fw)=ds2ps(P,W)
        res.append(ps2ds(P,(rw,2*fq-fw)))
    return res
```

Пользуясь этой функцией напишем программу, иллюстрирующую принцип работы калейдоскопа. Как известно, изображение в калейдоскопе получается в результате многократного отражения нескольких цветных стёклышек относительно зеркальных стенок калейдоскопа.

Стенки калейдоскопа образуют правильный треугольник. С достаточной для наших целей точностью сторона и высота равностороннего треугольника относятся как 15 к 13. Для получения нужного размера умножим эти числа на коэффициент 6, получим треугольник с основанием 90 пикселей и высотой 78 пикселей, Зададим в программе его вершины:

```
A=(55,178)
B=(145,178)
C=(100,100)
```

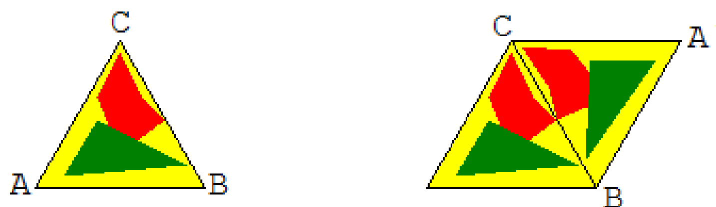
Поместим в калейдоскоп два «стёклышка»:

```
L1=[(88,130),(100,160),(124,142),(112,130),(100,106)]
L2=[(70,172),(136,166),(88,142)]
```

и отобразим треугольник и «стёклышки» на канве:

```
canvas.create_polygon(A,B,C,fill='yellow',
                      width=1,outline='black')
canvas.create_polygon(L1,fill='red')
canvas.create_polygon(L2,fill='green')
```

Получим один фрагмент (для наглядности добавлены обозначения точек):



Добавим к нему отражение треугольника CAB относительно стороны CB (рисунок справа). Следующий фрагмент получится отражение треугольника CBA’ относительно стороны CA’ и т.д. Следует учесть, что при каждом следующем отражении потребуется перевычислять координаты точек, задающие положение «зеркала».

Приведём всю программу и результат её выполнения.

```
from geom_sys import *
from tkinter import *
root=Tk()
```

```
root.title('Калейдоскоп')
canvas=Canvas(root,width=200, height=200,bg='white')
canvas.pack()

A=(55,178)
B=(145,178)
C=(100,100)
L1=[(88,130),(100,160),(124,142),(112,130),(100,106)]
L2=[(70,172),(136,166),(88,142)]
P=C; Q=B; R=A
for k in range(6):
   canvas.create_polygon(P,Q,R,fill='yellow',
                          width=1,outline='black')
   canvas.create_polygon(L1,fill='red')
   canvas.create_polygon(L2,fill='green')
   R=symmetry(P,Q,R)
   L1=symmetry(P,Q,L1)
   L2=symmetry(P,Q,L2)
   Q,R=R,Q
```

Результат:

Узор калейдоскопа можно продлить на любую область, т.к. равносторонними треугольниками можно заполнить всю плоскость. Если же уменьшить угол между зеркалами, то заполнить можно будет только ограниченную область. Для иллюстрации сказанного напишем программу построения узора «Салфетка» (рисунок справа).

```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Салфетка')
canvas=Canvas(root,width=200, height=200,bg='white')
```

```
canvas.pack()

S=(100,100)
n=24; df=360/n
A=ps2ds(S,(90,0))
B=ps2ds(S,(70,df))
V1=[(30,0),(40,df/2),(60,df/2),(40,0)]
V2=[(70,0),(50,df/2),(80,0)]
L1=ps2ds(S,V1)
L2=ps2ds(S,V2)
P=S; Q=B; R=A
for k in range(n):
   canvas.create_polygon(P,Q,R,fill='blue')
   canvas.create_polygon(L1,fill='white')
   canvas.create_polygon(L2,fill='white')
   R=symmetry(P,Q,R)
   L1=symmetry(P,Q,L1)
   L2=symmetry(P,Q,L2)
   Q,R=R,Q
```

Задания для самостоятельной работы.

1) Измените цвет и расположение стёклышек калейдоскопа
2) Добавьте ещё одно стёклышко в калейдоскоп
3) Продлите узор калейдоскопа ещё на несколько треугольников
4) Нарисуйте свою «салфетку»

6. Паркеты и мозаики

Мозаикой будем называть узор из повторяющихся элементов, имеющий поворотную симметрию. Для поворота фрагмента узора будем использовать функцию turn модуля geom_sys.

Пример мозаики - «салфетка» из предыдущего раздела.

Паркетом будем называть узор, в котором повторяющиеся элементы заполняют плоскость. В этом случае можно использовать параллельный перенос (сдвиг) одного и того же фрагмента узора. Обычно фрагменты паркета являются полигонами.

Добавим в модуль geom_sys функцию move для сдвига полигона L на заданный вектор V:

```
# сдвиг списка точек на заданный вектор
def move(V, L):
   """
   V - вектор слвига
   L - точка или список точек
   результат - точка или список точек после сдвига
```

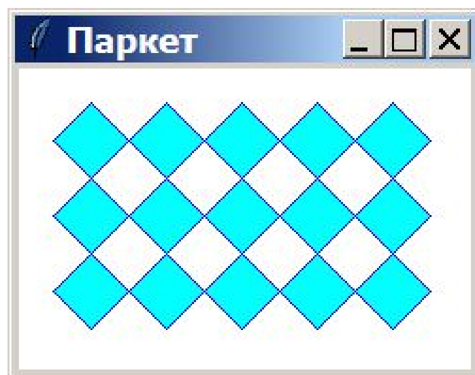
```
    """
    if type(L)==list:
       return [add(V,P) for P in L]
    else:
       return add(V,L)
```

где функция add используется для сложения векторов

```
# сложение векторов в экранной системе координат
def add(S,*L):
    """
    S - начальный вектор / начальная точка
    L - список векторов / точек
    xs, ys - координаты конечного вектора / конечной точки
    """
    (xs, ys)=S
    for (x,y) in L:
       xs=xs+x
       ys=ys+y
    return (xs, ys)
```

Самые простые паркеты состоят из одинаковых правильных многоугольников. Это могут быть треугольники, квадраты или шестиугольники.

Пример паркета.



Паркет образован чередующимися голубыми и белыми квадратами. Но мы будем рисовать только синие квадраты, так как белые квадраты образованы участками фона.

В основе построения паркета — параллельный перенос элемента узора. Для построения двумерного паркета нам понадобятся два вектора для сдвига списка вершин полигона: V1 – для сдвига в горизонтальном направлении и V2 – для сдвига в вертикальном направлении.

Для удобного масштабирования рисунка применим следующий прием. Зададим шаблон элемента рисунка, используя условные координаты вершин:

```
# шаблон элемента
SH=[(1,0),(0,1),(-1,0),(0,-1)]
```

В шаблоне центр квадрата совмещен с началом системы координат, поэтому некоторые координаты получились отрицательными. Чтобы теперь получить первый полигон, используем масштабирование и перенос шаблона.

Для масштабирования напишем функцию умножения списка векторов на число и добавим её в модуль geom_sys:

```
def mlt(L,k):
   """
   L - список векторов
   k - целое или вещественное число
   результат – список векторов
   """
   if type(L)==list:
      return [mlt(P,k) for P in L]
   else:
      (x,y)=L
      return (x*k, y*k)
```

Тогда масштабирование и перенос шаблона выполняется с помощью написанных функций так:

```
L=move(S,mlt(SH,q))
```

где точка S – координаты центра первого квадрата.

Вектора сдвигов можно масштабировать явно:

```
# масштабирование векторов сдвига
V1=(q*2,0)
V2=(0,q*2)
```

Напишем программу рисования паркета:

```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Паркет')
canvas=Canvas(root,width=240,height=160,bg='white')
canvas.pack()

# параметры узора:
S=(40,40); q=20; n=3; m=5
# S – центр первого квадрата
# q – коэффициент масштабирования
```

```
# n, m - количество рядов и столбцов паркета
# ------------------------------------------
# шаблон элемента
SH=[(1,0),(0,1),(-1,0),(0,-1)]
# масштабирование и перенос шаблона
L=move(S,mlt(SH,q))
# формирование векторов сдвига
V1=(q*2,0)
V2=(0,q*2)
# вложенный цикл рисования паркета
for i in range(n):
   LS=L
   for j in range(m):
      canvas.create_polygon(LS,width=1,outline='blue',
                                 fill='cyan')
      LS=move(V1,LS)
   L=move(V2,L)
```

Замечание. Продемонстрируем на этом примере и другой подход к построению паркетов. Напишем отдельно функцию, которая рисует один элемент паркета. Функция должна получить точку привязки и размер элемента. Тогда в главной программе останется написать вложенный цикл, который будет менять положение и/или размер элемента паркета.

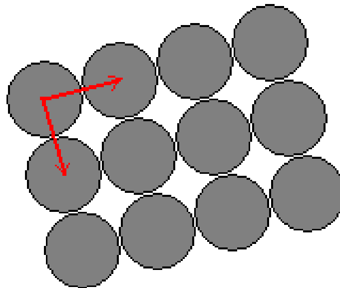
```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Паркет')
canvas=Canvas(root,width=240,height=160,bg='white')
canvas.pack()
# ------------------------------------------
def elem(S,q):
   # шаблон элемента
   SH=[(1,0),(0,1),(-1,0),(0,-1)]
   # масштабирование и перенос шаблона
   L=move(S,mlt(SH,q))
   # рисование элемента узора
   canvas.create_polygon(L,width=1,outline='blue',
                                 fill='cyan')
# ----------- main -----------
# параметры узора:
S=(40,40); q=20; n=3; m=5
# S - центр первого квадрата
# q - коэффициент масштабирования
# n, m - количество рядов и столбцов паркета
# формирование векторов сдвига
```

```
V1=(q*2,0)
V2=(0,q*2)
# вложенный цикл рисования паркета
for i in range(n):
   for j in range(m):
      w=add(S,mlt(V1,j),mlt(V2,i))
      elem(w,q)
```

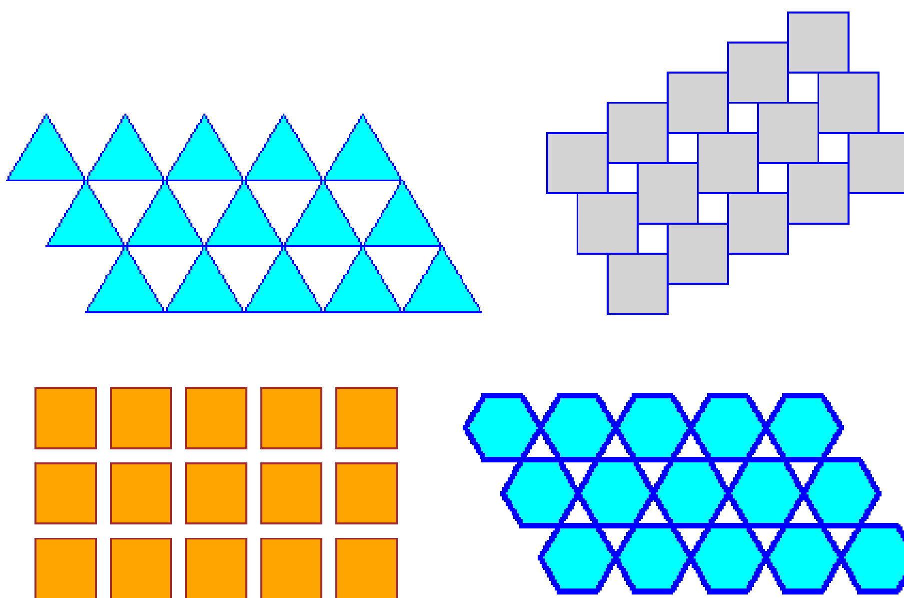
Вектора сдвигов могут быть произвольными. Покажем это на примере. Построим узор из кругов. Для построения элемента узора используем процедуру create_circle построения круга из модуля geom_sys.

```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Решетка')
canvas=Canvas(root,width=240,height=200,bg='white')
canvas.pack()
S =(50,75)
V1=(40,-10)
V2=(10, 40)
for i in range(3):
   for j in range(4):
      w=add(S,mlt(V1,j),mlt(V2,i))
      create_circle(canvas,w,20,fill='gray')
```

Результат (решетка из серых кругов) показан на рисунке. Красным цветом дополнительно показаны векторы сдвигов.

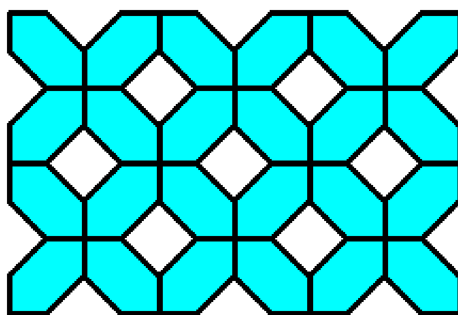


Задания для самостоятельной работы

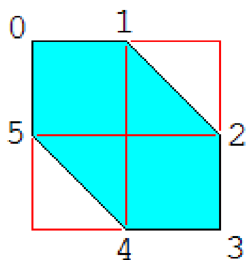


Более сложные паркеты строятся по тому же принципу, что и простые, но элементы паркета могут быть произвольной формы.

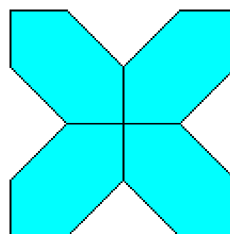
Пример сложного паркета.



Последовательность построения паркета: сначала задаём шаблон, на основе которого строим элемент узора:



Шаблон



Элемент узора

Точка с номером 0 соответствует центру элемента узора, сам элемент получается в результате масштабирования и последовательных поворотов на

90 градусов. Построенный элемент узора переносится в двух направлениях. Приведём полный текст программы.

Обозначения в программе:

SH – шаблон

S – центр элемента узора;

q – коэффициент масштабирования;

L – список вершин шаблона после масштабирования и переноса;

V1, V2 – векторы переносов.

Напишем программу

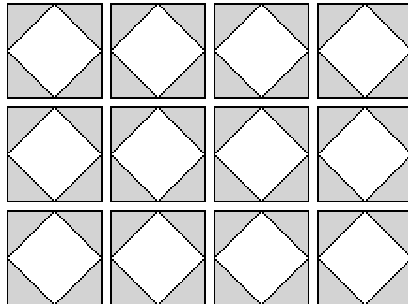
```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Паркет')
canvas=Canvas(root,width=280,height=200,bg='white')
canvas.pack()

def elem(S,q):
   #     0     1     2     3     4     5  - номер точки
   SH=[(0,0),(1,0),(2,1),(2,2),(1,2),(0,1)]
   # масштабирование и перенос шаблона
   L=move(S,mlt(SH,q))
   for k in range(4):
      canvas.create_polygon(L,width=3,
             outline='black',fill='cyan')
      L = turn(S,L,90)

# ----------- main -----------
S=(60,60); q=20
V1=(4*q,0)
V2=(0,4*q)
# рисуем двумерный узор
for i in range(2):
   for j in range(3):
      w=add(S,mlt(V1,j),mlt(V2,i))
      elem(w,q)
```

В случае, если повторяющийся фрагмент узора состоит из независимых частей, для некоторого упрощение программы можно использовать следующий прием. Соберем все вершины фрагмента в один список, а при рисовании отдельных частей фрагмента будем использовать выборки из списка.

Приведём пример.

```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Паркет')
canvas=Canvas(root,width=285,height=230,bg='white')
canvas.pack()

def elem(S,q):
   SH=[(-1,-1),(1,-1),(1,1),(-1,1),
       (-1,0),(0,-1),(1,0),(0,1)]
   L=move(S,mlt(SH,q))
   canvas.create_polygon(L[:4],outline='black',
                            fill='lightgray')
   canvas.create_polygon(L[4:],outline='black',
                            fill='white')

# ---------- main --------- #
S=(60,60); q=25
V1=(2*q+5,0)
V2=(0,2*q+5)
for i in range(3):
   for j in range(4):
      w=add(S,mlt(V1,j),mlt(V2,i))
      elem(w,q)
```

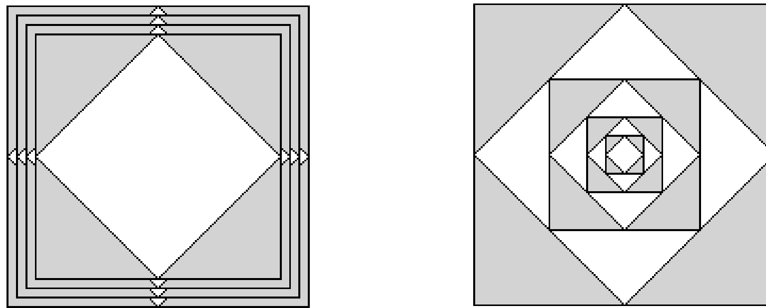
Результат:



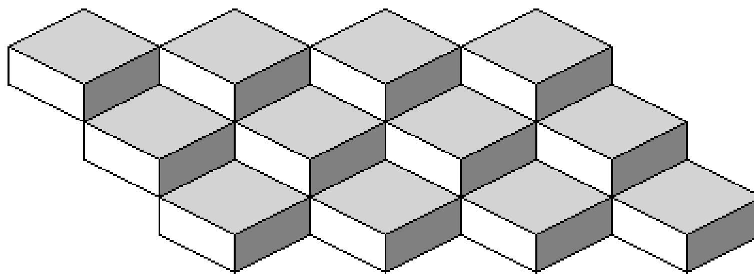
Попробуем изменить главную программу. Ниже приведены два варианта, в которых точка привязки остайтся постоянной, а изменяются только размеры элементов.

```
# ---------- main --------- #
S=(145,115)
q=80
for i in range(4):
   elem(S,q)                     # вариант
   q=q-5                         # q=q/2
```

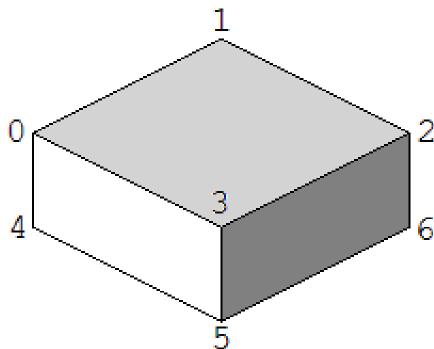
Результаты:

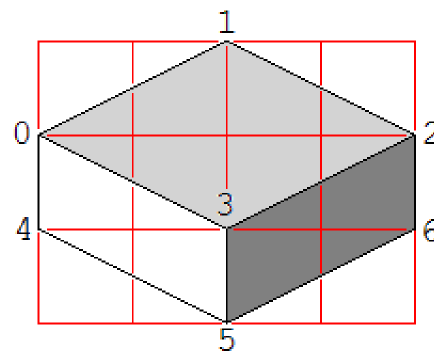


В следующем примере нарисуем «трехмерный» узор.



Выделим повторяющийся элемент и пронумеруем его вершины (рисунок слева)





Представим себе, что элемент нарисован на бумаге в клетку (рисунок справа), тогда можно записать координаты каждой отмеченной точки в условных единицах — «клетках».

```
#    0     1     2     3     4     5     6  - номер точки
SH=[(0,1),(2,0),(4,1),(2,2),(0,2),(2,3),(4,2)]
```

Напишем функцию рисования одного «кубика» по заданному шаблону.

```
def elem(S,q):
```

```
    #    0     1     2     3     4     5     6
    SH=[(0,1),(2,0),(4,1),(2,2),(0,2),(2,3),(4,2)]
    # масштабирование и перенос шаблона
    L=move(S,mlt(SH,q))
    up=[L[k] for k in (0,1,2,3)]
    lf=[L[k] for k in (0,3,5,4)]
    rt=[L[k] for k in (2,3,5,6)]
    canvas.create_polygon(up,outline='black',
                             fill='lightgray')
    canvas.create_polygon(lf,outline='black',
                             fill='white')
    canvas.create_polygon(rt,outline='black',
                             fill='gray')
```

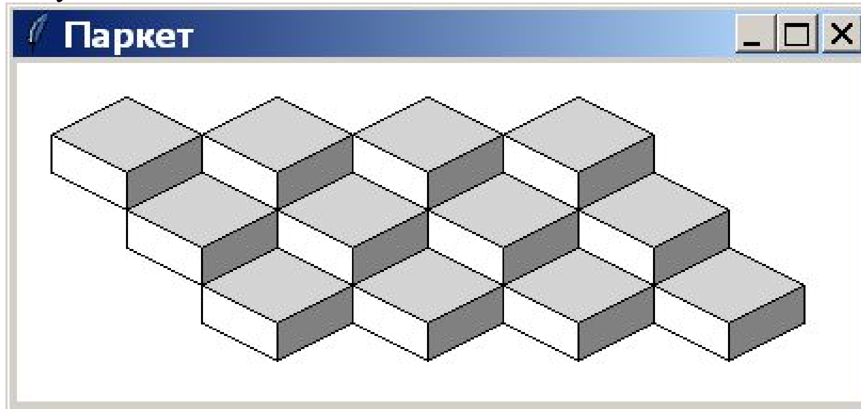
Здесь up, lf, rt – последовательности вершин полигонов для рисования верхней, левой и правой граней «кубика». В главной программе с помощью вложенного цикла рисуем двумерный узор.

Приведём программу полностью.

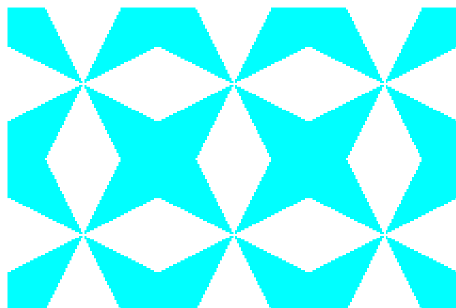
```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Паркет')
canvas=Canvas(root,width=450,height=180,bg='white')
canvas.pack()

def elem(S,q):
   #    0     1     2     3     4     5     6  - номер
точки
   SH=[(0,1),(2,0),(4,1),(2,2),(0,2),(2,3),(4,2)]
   # масштабирование и перенос шаблона
   L=move(S,mlt(SH,q))
   up=[L[k] for k in (0,1,2,3)]
   lf=[L[k] for k in (0,3,5,4)]
   rt=[L[k] for k in (2,3,5,6)]
   canvas.create_polygon(up,outline='black',
                            fill='lightgray')
   canvas.create_polygon(lf,outline='black',
                            fill='white')
   canvas.create_polygon(rt,outline='black',
                            fill='gray')
# ----------- main -----------
S=(20,20); q=20
V1=(4*q,0)
V2=(2*q,2*q)
n=3; m=4
```

```
# рисуем двумерный узор
for i in range(n):
    for j in range(m):
        w=add(S,mlt(V1,j),mlt(V2,i))
        elem(w,q)
```

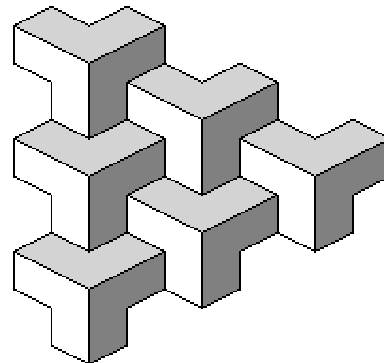
Результат:



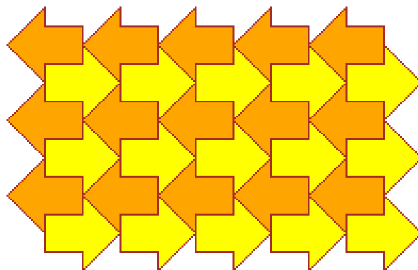
Задания для самостоятельной работы.
Нарисуйте следующие паркеты и мозаики



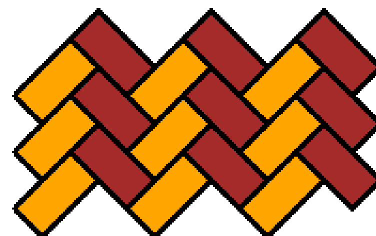
А)



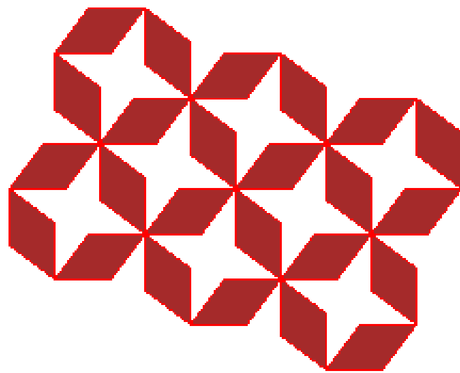
Б)



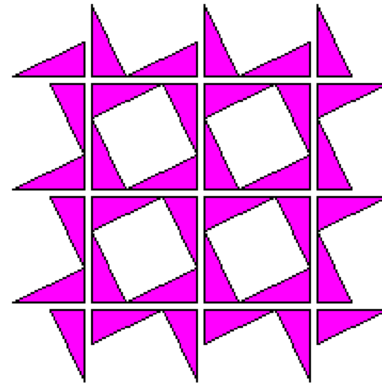
В)



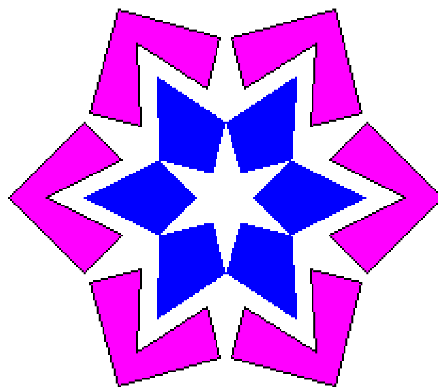
Г)



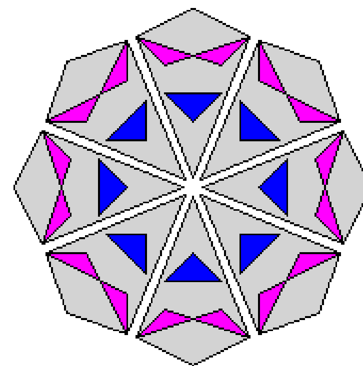
Д)



Е)



Ж)



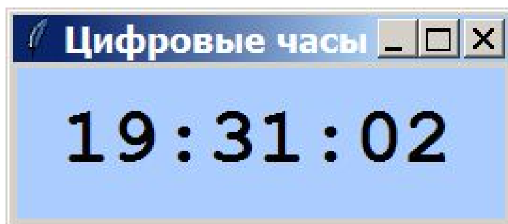
З)

7. Анимация

Слово «анимация» можно перевести как «оживление», «одушевление». Компьютерная анимация предполагает постоянное изменение изображения через определённые промежутки времени. С помощью компьютерной анимации можно отображать процессы и объекты, изменяющиеся во времени.

Для реализации временной задержки в программном коде можно использовать метод sleep(dt) модуля time. Параметр dt задаёт время задержки в секундах. При использовании метода sleep для корректной работы приложения необходимо принудительно обновлять экран методом update().

Проиллюстрируем использование этих методов на примере цифровых часов. Напишем программу, которая каждую секунду получает системное время и отображает его в окне приложения.



Для получения текущего времени в модуле time есть методы:

time() возвращает время в секундах, прошедшее после 1 января 1900 года;

ctime() возвращает текстовую строку, содержащую текущие значения даты и времени, Пример возвращаемого значения: Sun May 10 20:03:00 2020.

localtime() возвращает структуру типа struct_time, содержащую текущие значения даты и времени. Поля структуры:

tm_year – год;
tm_mon — номер месяца (1 соответствует январю);
tm_mday — день:
tm_hour — часы (в 24-часовом формате);
tm_min — минуты;
tm_sec — секунды;
tm_wday — номер дня недели (0 соответствует понедельнику);
tm_yday — номер дня в году (1 соответствует 1 января)
tm_isdst — признак даты по старому стилю (0 — дата по новому стилю).

Для вывода данных в окно приложения используем метод create_text, который должен получить текстовую строку, содержащую текущие часы, минуты и секунды. Следовательно, сначала нужно эту строку сформировать. Это можно сделать двумя способами:

1) извлечь данные из текстовой строки, возвращаемой методом ctime();

2) сформировать данные из полей tm_hour, tm_min, tm_sec.

Наиболее простым в нашем случае является первый способ. Разобьём строку на отдельные слова методом split() и извлечём из полученного списка нужное значение.

Пример. Пусть dt = 'Sun May 10 20:03:00 2020', тогда dt.split() вернёт список ['Sun', 'May', '10', '20:03:00', '2020']. Нужное нам значение есть элемент списка с индексом 3.

Приведём текст программы:

```
from time import *
from tkinter import *
root=Tk()
root.title('Цифровые часы')
canvas=Canvas(root, width=260, height=80, bg='#aaccff')
```

```
canvas.pack()

for k in range(20):
   res=ctime().split()[3]
   tm=canvas.create_text(130,40,text=res,
             font=('Courier New',32,'bold'))
   canvas.update()
   sleep(1)
   canvas.delete(tm)
canvas.create_text(130,40,text=res,
              font=('Courier New',32,'bold'))
```

Цифровые часы будут идти 20 секунд, а затем остановятся. На каждой итерации цикла определяется текущее время, создаётся графический объект (текст), который уничтожается через одну секунду. Если этого не сделать, то новый текст будет накладываться на уже существующий.

Напишем теперь программу для вывода времени с помощью стрелочных часов. При запуске программы нужно выставить на часах текущее время и обновлять результаты каждую секунду. Текущее время для данного случая удобно получить методом localtime(). Из данных, возвращаемых методом нам понадобятся tm_hour, tm_min и tm_sec.

Для решения задачи свяжем с циферблатом часов полярную систему координат, начало которой совпадает с центром циферблата. Наша задача — нарисовать часовую, минутную и секундную стрелку, а для этого нужно определить углы, образуемые стрелками с полярной осью в любой момент времени. Обновлять положение всех стрелок будем каждую секунду.

Начнём с секундной стрелки. Угол между двумя соседними положениями секундной стрелки равен 360/60 = 6 градусов. Значение tm_sec определяет угол sfi=6*tm_sec отклонения стрелки от направления на 12 часов. Количество минут tm_min определяет положение минутной стрелки на начало каждой минуты. Но к углу, задаваемому целым числом tm_min, нужно прибавить угол, задаваемый отношением sfi/360. Т.е. угол отклонения минутной стрелки mfi=6*(tm_min+sfi/360). Для часовой стрелки ситуация немного сложнее. Сначала нужно учесть, что за сутки часовая стрелка делает два оборота, следовательно вместо значения tm_hour нужно рассматривать значение h = tm_hour%12, т.е. остаток от деления на 12. К углу, задаваемому этим значением нужно прибавить угол, задаваемый отношением mfi/360. Так как один час — это пять минутных делений, то формула для угла отклонения часовой стрелки принимает вид hfi=30*(h+mfi/360).

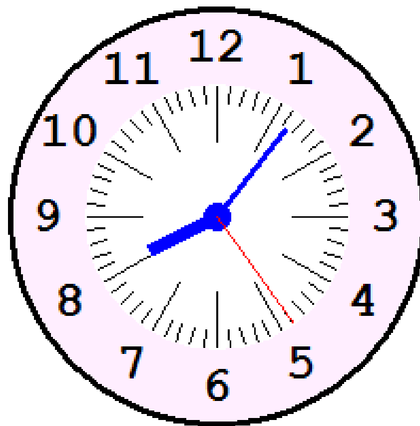
Остальное будет понятно из программы.

```
from geom_sys import *
from time import *
```

```
from tkinter import *
root=Tk()
root.title('Часы')
canvas=Canvas(root,width=260,height=260,bg='white')
canvas.pack()

# параметры-----------
R1= 40 # длина часовой стрелки
R2= 60 # длина минутной стрелки
R3= 70 # радиус внутреннего круга и секундной стрелки
R4= 90 # срединный радиус числовых отметок
R5=110 # внешний радиус часов
# --------------------
(xs,ys)=S=(130,130)
create_circle(canvas,S,R5,width=3,
              outline='black',fill='#ffeeff')
create_circle(canvas,S,R3,outline='',fill='white')
create_circle(canvas,S,7,outline='',fill='blue')
# отметки на циферблате
hour=0
for i in range(1,61):
   fi=90-6*i
   if i%5==0:
      hour+=1
      P=ps2ds(S,(R1,fi))
      Q=ps2ds(S,(R3,fi))
      T=ps2ds(S,(R4,fi))
      canvas.create_text(T,text=str(hour),
                         font=('Courier New',20,'bold'))
   else:
      P=ps2ds(S,(R2,fi))
      Q=ps2ds(S,(R3,fi))
   canvas.create_line(P,Q)

# движение стрелок
for k in range(30):
   tm=localtime()
   sfi=6*tm.tm_sec
   mfi=6*(tm.tm_min+sfi/360)
   h=tm.tm_hour%12
   hfi=30*(h+mfi/360)
   Ph=ps2ds(S,(R1,90-hfi))
   Pm=ps2ds(S,(R2,90-mfi))
   Ps=ps2ds(S,(R3,90-sfi))
   idh=canvas.create_line(S,Ph,width=7,fill='blue')
```

```
        idm=canvas.create_line(S,Pm,width=3,fill='blue')
        ids=canvas.create_line(S,Ps,width=1,fill='red')
        canvas.update()
        sleep(1)
        canvas.delete(ids,idm,idh)
    idh=canvas.create_line(S,Ph,width=7,fill='blue')
    idm=canvas.create_line(S,Pm,width=3,fill='blue')
    ids=canvas.create_line(S,Ps,width=1,fill='red')
```

Результат (без заголовка окна) представлен на рисунке ниже.



8. Анимация (продолжение)

Движение бильярдного шара. Рассмотрим модель бильярдного стола и шара — прямоугольник и круг. Шар (в нашей модели — круг) начинается двигаться в заданном направлении. Отражение от бортов бильярдного стола (в модели — от сторон прямоугольника) происходит по закону: угол падения равен углу отражения. В модели этот закон реализуется следующим образом: при отражении относительно горизонтальных сторон меняет знак Y-ая компонента вектора направления движения, а при отражении от вертикальных сторон меняет знак X-ая компонента вектора.

Программа использует методы канвы move – для перемещения объекта и coords – для получения координатных параметров объекта.

Синтаксис вызова:

move(идентификатор, компоненты_вектора_сдвига)

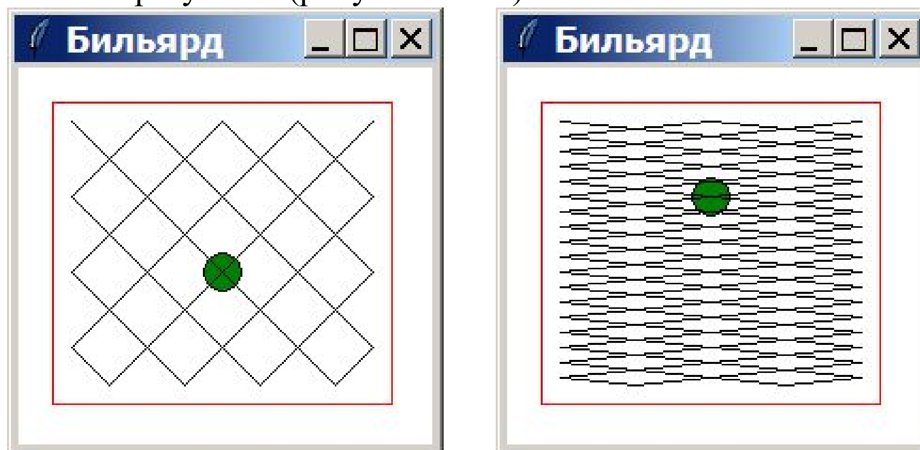
coords(идентификатор) возвращает координатные параметры объекта. Так, например, в программе возвращаются координаты двух вершин прямоугольника (а на самом деле квадрата), описанного возле круга.

Метод coords может также задавать новые координаты графическому объекту:

coords(идентификатор, новые_координаты)

```
from time import *
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Бильярд')
canvas=Canvas(root,width=220,height=200,bg='white')
canvas.pack()
# задаём параметры стола и шара
(x1,y1)=(20,20)
(x2,y2)=(200,180)
(x,y)=(30,30); R=10
(Vx,Vy)=(10,10)
# ------------------------------
canvas.create_rectangle(x1,y1,x2,y2,outline='red')
for k in range(121):
   if k==0:
      A=(x,y)
      ball=create_circle(canvas,A,R,fill='green')
   else:
      if x+Vx-R<x1 or x+Vx+R>x2: Vx=-Vx
      if y+Vy-R<y1 or y+Vy+R>y2: Vy=-Vy
      B=(x+Vx,y+Vy)
      canvas.create_line(A,B,)
      canvas.move(ball,Vx,Vy)
      (xL,yU,xR,yD)=canvas.coords(ball)
      x=(xL+xR)//2; y=(yU+yD)//2
      canvas.update()
      sleep(0.05)
      A=B
```

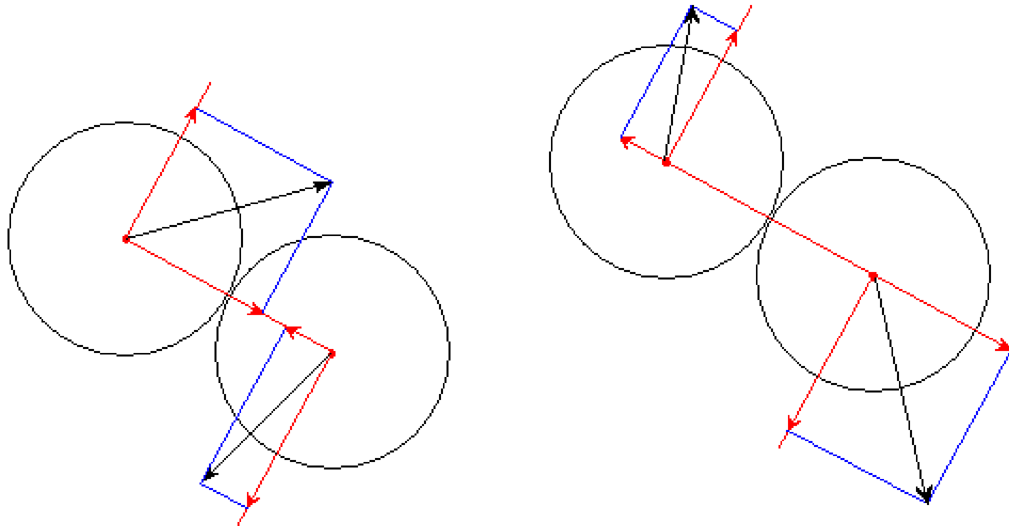
Конечный результат (рисунок слева):



Изменим начальный вектор движения шара (Vx,Vy)=(10,1) и увеличим количество итераций основного цикла: for k in range(601), получим рисунок справа.

Сталкивающиеся круги. Расположим на канве несколько одинаковых по размеру кругов и зададим направления их движения, а тем самым и их начальные скорости. Будем контролировать положение шаров через небольшие промежутки времени, чтобы «поймать» момент их столкновения. После столкновения изменятся и скорости и направления движения кругов.

На схеме ниже показан момент столкновения двух кругов. Спроектируем векторы скоростей (черные стрелки) на прямую, соединяющие центры кругов. При столкновении круги «обмениваются» этими компонентами скорости. Вторые компоненты — векторные проекции скоростей на перпендикулярные направления не изменяются.



Теперь, чтобы получить новые вектора скоростей, нужно сложить полученные компоненты по правилу сложения векторов (рисунок справа).

Чтобы реализовать в программе этот алгоритм нам понадобятся некоторые сведения из высшей математики, а именно формула для вычисления вектора проекции. Рассмотрим два вектора A = (Ax, Ay) и B = (Bx, By). Векторная проекция A на B равна некоторому коэффициенту q, умноженному на вектор B. Коэффициент q вычисляется по формуле (Ax*Bx + Ay*By)/(Bx**2 + By**2).

Векторная проекция скорости на перпендикулярное к вектору B направление вычисляется тем же методом. В качестве перпендикулярного к вектору B проще всего взять вектор (By, -Bx), тогда формула для вычисления

соответствующего коэффициента примет вид (Ax*By - Ay*Bx)/(Bx**2 + By**2).

Предусмотрим в программе три круга радиуса R = 15:

```
n=3; R=15
```

Данные всех кругов соберём в список S:

```
S=[(220,100,-2,-1),(120,150,2,-1),(100,50,0,-3)]
```

а их цвета — в список colors:

```
colors=['lightgreen','lightblue','pink'].
```

Элементы списка S являются кортежами вида (x,y,Vx,Vy), где x,y – координаты центра круга, а Vx,Vy – компоненты вектора скорости круга.

Сформируем список кругов:

```
balls=[]
for i in range(n):
   x,y,Vx,Vy=S[i]
   balls.append(canvas.create_oval(x-R,y-R,x+R,y+R,
                                   fill=colors[i]))
```

Основная часть программы содержит цикл, в котором:

1) все круги сдвигаются на свой вектор скорости, при этом проверятся достиг ли круг ограничивающей «стенки». Если достиг, то изменяется направление движения;

2) для каждой пары кругов вычисляется расстояние между их центрами. Если оно меньше, чем 2*R+1, то считается, что круги столкнулись. В этом случае для столкнувшихся кругов вычисляются новые вектора скоростей, а сами круги сдвигаются на этот вектор.

В заключение приведём код программы.

```
from time import *
from math import *
from tkinter import *
root = Tk()
w=250; h=200
canvas = Canvas(root,width=w,height=h,bg='white')
canvas.pack()
x1=10; x2=w-10
y1=10; y2=h-10
canvas.create_rectangle(x1,y1,x2,y2,fill='lightgray')

n=3; R=15
S=[(220,100,-2,-1),(120,150,2,-1),(100,50,0,-3)]
colors=['lightgreen','lightblue','pink']
balls=[]
```

```
for i in range(n):
   x,y,Vx,Vy=S[i]
   balls.append(canvas.create_oval(x-R,y-
R,x+R,y+R,fill=colors[i]))
canvas.update()
sleep(0.5)
for k in range(1000):
   for i in range(n):
      x,y,Vx,Vy=S[i]
      if x+Vx-R<x1 or x+Vx+R>x2: Vx=-Vx
      if y+Vy-R<y1 or y+Vy+R>y2: Vy=-Vy
      S[i]=(x+Vx,y+Vy,Vx,Vy)
      canvas.move(balls[i],Vx,Vy)
   canvas.update()
   sleep(0.02)
   for i in range(n):
      xi,yi,Vxi,Vyi=S[i]
      for j in range(i+1,n):
         xj,yj,Vxj,Vyj=S[j]
         dist=sqrt((xj-xi)**2+(yj-yi)**2)
         if dist<2*R+1:
            # столкновение!
            dx,dy=(xj-xi,yj-yi)
            # раскладываем вектор скорости i-ого круга
            q=(Vxi*dx+Vyi*dy)/(dx**2+dy**2)
            Lxi,Lyi=(q*dx,q*dy)
            q=(Vxi*dy-Vyi*dx)/(dx**2+dy**2)
            Nxi,Nyi=(q*dy,-q*dx)
            # раскладываем вектор скорости j-ого круга
            q=(Vxj*dx+Vyj*dy)/(dx**2+dy**2)
            Lxj,Lyj=(q*dx,q*dy)
            q=(Vxj*dy-Vyj*dx)/(dx**2+dy**2)
            Nxj,Nyj=(q*dy,-q*dx)
            # вычисляем новые вектора скоростей
            Vxi,Vyi=(Lxj+Nxi, Lyj+Nyi)
            Vxj,Vyj=(Lxi+Nxj, Lyi+Nyj)
            S[i]=(xi+Vxi,yi+Vyi,Vxi,Vyi)
            S[j]=(xj+Vxj,yj+Vyj,Vxj,Vyj)
            canvas.move(balls[i],Vxi,Vyi)
            canvas.move(balls[j],Vxj,Vyj)
```

Задания для самостоятельного выполнения.

1) Добавьте в программу ещё один круг

2*) Проведите на каждом круге радиус, направленный по вектору скорости круга и изменяйте его при каждом столкновении

Раздувающиеся круги. Расположим случайным образом круги на канве. Каждый круг будет увеличиваться в размерах пока не коснётся другого. Оба коснувшихся круга лопаются и снова начинают увеличиваться в размерах. Для изменения размера графического объекта используем метод канвы scale:

scale (tagOrId, xOrigin, yOrigin, xScale, yScale).

Параметры метода:

tagOrId – тэг или идентификатор графического объекта;

xOrigin, yOrigin – центр преобразования;

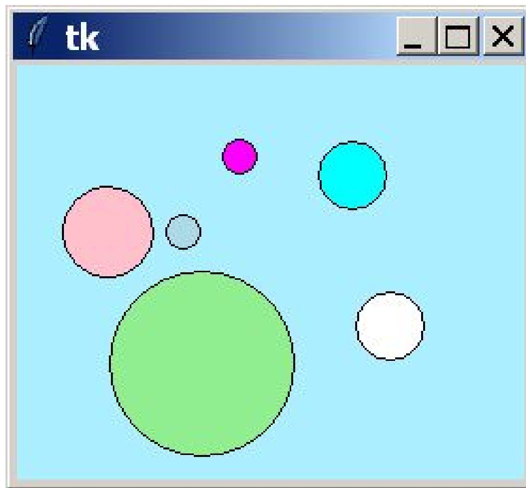
xScale, yScale – коэффициенты масштабирования по горизонтали и вертикали. Если эти коэффициенты равны, то при увеличении круга, он останется кругом.

Возьмем за основу предыдущую программу, только на этот раз соберём в список S координаты x,y центра круга и его радиус R. Столкновение кругов происходит, когда расстояние между центрами кругов равно сумме радиусов, но в силу возможных погрешностей будем проверять условие dist<Ri+Rj+1.

Начальный радиус кругов зададим одинаковым и равным двум пикселям. После столкновения круги «лопаются», а их радиус снова становится равным начальному.

```
from time import *
from math import *
from tkinter import *
root = Tk()
w=270; h=220
canvas = Canvas(root,width=w,height=h,bg='#aaeeff')
canvas.pack()

n=6; dR=1 # количество кругов и «скорость» их раздувания
S=[(100,160,2),(90,95,2),(50,90,2),
    (180,60,2),(120,50,2),(200,140,2)]
colors=['lightgreen','lightblue','pink','cyan','magenta',
'white']
balls=[]
for i in range(n):
   x,y,R=S[i]
   balls.append(canvas.create_oval(x-R,y-
R,x+R,y+R,fill=colors[i]))
canvas.update()
sleep(0.5)
for k in range(400):
   for i in range(n):
      x,y,R=S[i]
      canvas.scale(balls[i],x,y,1+dR/R,1+dR/R)
```

```
            S[i]=(x,y,R+dR)
        canvas.update()
        sleep(0.05)
        for i in range(n):
            xi,yi,Ri=S[i]
            for j in range(i+1,n):
                xj,yj,Rj=S[j]
                dist=sqrt((xj-xi)**2+(yj-yi)**2)
                if dist<Ri+Rj+1:
                    # столкновение!
                    canvas.scale(balls[i],xi,yi,2/Ri,2/Ri)
                    canvas.scale(balls[j],xj,yj,2/Rj,2/Rj)
                    S[i]=(xi,yi,2); S[j]=(xj,yj,2)
                    break
```



Разлетающиеся круги. Приведём ещё пример использования стандартных модулей Питона. Нарисуем что-то вроде разлетающихся «осколков». Чтобы картина «взрыва» была более интересной, примем, что «осколки» разлетаются от центра на расстояния, определённые с помощью датчика случайных чисел. В Питоне методы генерации случайных чисел собраны в модуле random.

В программе используется функция randint(a,b), которая при каждом обращении возвращает случайное целое число в диапазоне от a до b включительно. Функция задаёт случайную величину разлёта L – свою для каждого из «осколков». По мере перемещения от центра осколок увеличивается в размерах, что создаёт эффект приближения «осколков» к зрителю. Но в отличие от предыдущих примеров на каждой итерации цикла создаются новые круги, в результате каждый «осколок» оставляет след.

```
from time import sleep
from random import randint
```

```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Взрыв')
canvas=Canvas(root,width=240,height=240,bg='white')
canvas.pack()

S=(120,120)
n=12; df=360/n; q=0.15
SL=[randint(10,120) for k in range(n)]
fi=0; m=20
for i in range(m):
   t=i/m
   for L in SL:
      A=ps2ds(S,(t*L,fi))
      create_circle(canvas,A,t*L*q,fill='#ddaa66')
      fi+=df
   canvas.update()
   sleep(0.1)
```

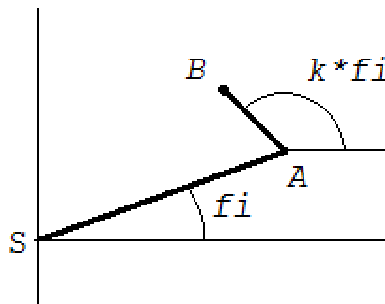
9. Моделирование механизмов

Модель спирографа. Спирограф — это устройство, состоящее из нескольких кругов с дырочками и кольца или прямоугольника с круглым отверстием. Круг вкладывается в отверстие, и обкатывается без проскальзывания внутри отверстия. При этом карандаш, вставленный в дырочку, чертит на бумаге замысловатую кривую.

Математическая модель спирографа представляет два шарнирно соединённых отрезка. Отрезок SA соединяет центр отверстия и центр круга.

Второй отрезок — отрезок AB соединяет центр круга и острие карандаша. Угловую скорость вращения отрезка SA можно считать одной и той же для всех вариантов, а вот угловая скорость вращения отрезка AB определяется отношением радиусов отверстия и круга. Заметим, что радиус круга и длина отрезка AB – это независимые величины, поэтому в математической модели мы можем задавать отношение радиусов независимо от длин отрезков SA и AB.

Для простоты примем, что при повороте отрезка SA на угол fi, отрезок AB поворачивается на угол k*fi, где k – целое число, причём число k может быть как положительным, так и отрицательным.



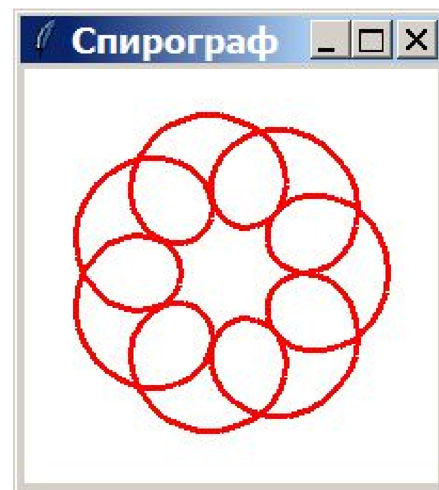
Напишем программу, моделирующую работу механизма.

```
from time import *
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Спирограф')
canvas=Canvas(root,width=220,height=220,bg='white')
canvas.pack()

# ----- параметры -------
R1=55  # длина отрезка SA
R2=30  # длина отрезка AB
k=8    # коэффициент k
S=(110,110)
id=create_circle(canvas,S,5,width=0,fill='gray')
for fi in range(0,361,2):
    A=ps2ds(S,(R1,fi))
    B=ps2ds(A,(R2,k*fi))
    canvas.create_line(S,A,width=3,fill='gray',
                       tags='SAB')
    canvas.create_line(A,B,width=3,fill='gray',
                       tags='SAB')
    create_circle(canvas,B,3,fill='red',tags='SAB')
    if fi>0:
        canvas.create_line(W,B,width=3,fill='red')
```

```
        W=B
        canvas.update()
        sleep(0.1)
        canvas.delete('SAB')
    canvas.delete(id)
```

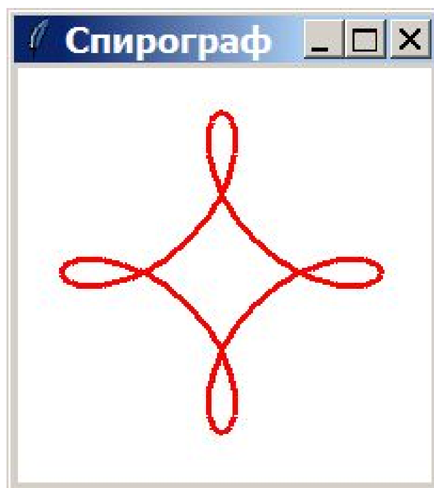
Звенья механизма рисуем серым цветом, след, оставляемый карандашом — красным. Звенья механизма и «карандаш» снабжены тэгом 'SAB' и удаляются методом delete(тэг). Центр механизма удаляется после завершения цикла методом delete(идентификатор).

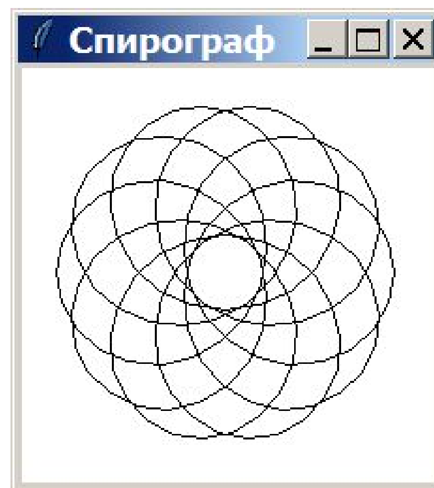
На рисунке слева показано состояние механизма в процессе рисования, а справа — полностью нарисованная кривая.

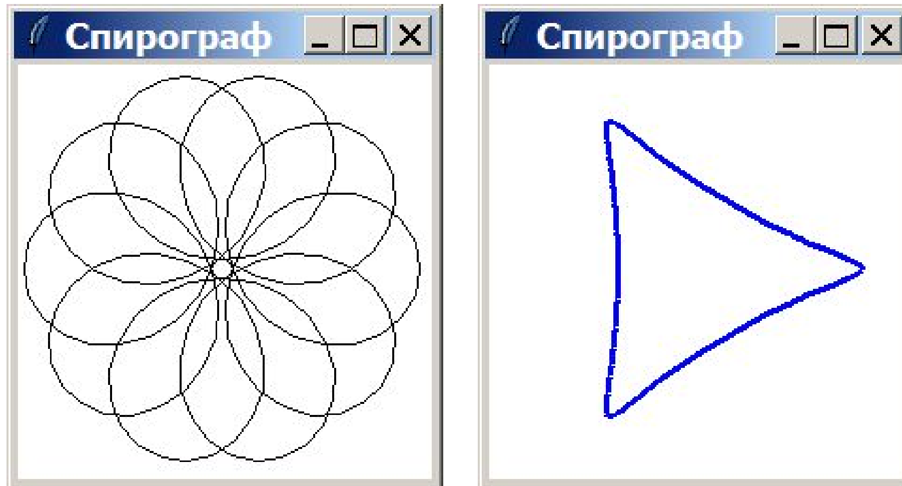


Задание для самостоятельной работы.

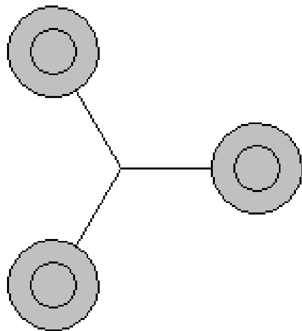
1. Подберите параметры механизма и нарисуйте следующие кривые:







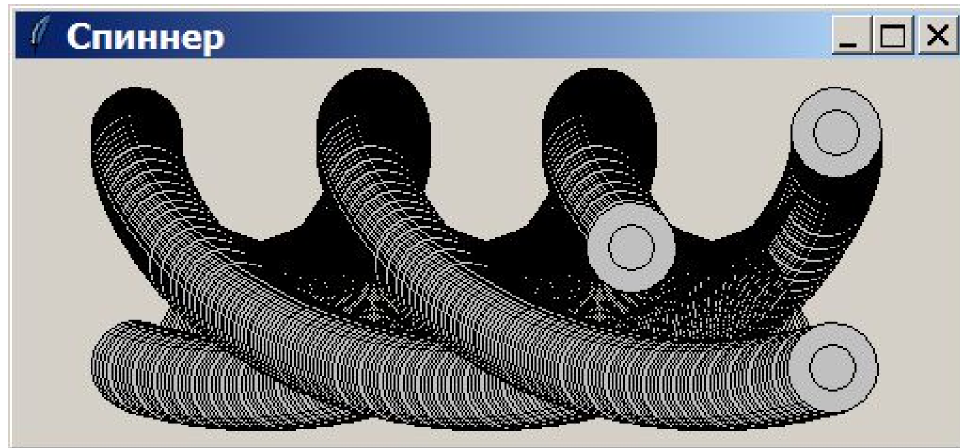
Спиннер. Нарисуем схематично некогда популярную игрушку — спиннер и смоделируем процесс его одновременного перемещения и вращения.



```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Спиннер')
canvas = Canvas(root,width=500,height=200)
canvas.pack()

R=24; size=3*R
def spinner(xs,ys,angle):
    for k in range(3):
        fi=k*120 + angle
        (x,y)=ps2ds((xs,ys),(size,fi))
        canvas.create_line(xs,ys,x,y)
        canvas.create_oval(x-R,y-R,x+R,y+R,fill='silver')
        canvas.create_oval(x-R//2,y-R//2,x+R//2,y+R//2)
```

```
for x in range(300):
   spinner(100+x,100,x)
```

Результат работы программы представлен на рисунке:



Чтобы увидеть процесс движения, подключим к программе модуль time и добавим в главную программу строчки

```
   sleep(0.01)
   root.update()
```

10. Управление окружностью

Анимацию можно реализовать, предоставив пользователю возможность управлять графическим объектом с помощью клавиатуры.

Чтобы приложение могло реагировать на действия пользователя, необходимо добавить в приложение функцию-обработчик событий. К событиям относятся нажатия/отпускания клавиш клавиатуры, кнопок мыши, перемещения указателя мыши и другие. Подробнее об этом мы будем говорить во второй части книги.

Событие всегда связано с некоторым объектом. Для событий мыши — это элемент управления или объект, над которым находится указатель мыши, для событий клавиатуры — это главное окно или окно верхнего уровня. Есть также возможность «привязать» событие к графическому объекту с заданным тэгом. Этот вариант мы рассмотрим позже.

Чтобы задать связь между событием, элементом управления и функцией-обработчиком этого события используем метод bind. Синтаксис вызова:

```
объект.bind(событие, функция)
```

Параметр «событие» передаётся методу как текстовая строка. В самом простом случае она имеет вид '<имя_события>', в частности для события нажатия клавиши клавиатуры — '<KeyPress>' или сокращенно '<Key>'.

Параметр «функция» является именем функции-обработчика без указания её параметров. Сама функция-обработчик обязательно имеет по крайней мере один параметр, через который функция получает информацию о событии. Традиционно этот параметр имеет имя event.

Информация о событии представляет собой структуру с несколькими полями. Для событий клавиатуры наиболее удобный вариант анализа нажатой клавиши — по её названию, содержащемуся в поле keysym. Менее удобно пользоваться числовыми кодами клавиш — полями keycode или keysym_num.

Изобразим схематично окружность, которой мы управляем, направления движения и значения keysym для соответствующих клавиш клавиатуры.

В главной программе зададим параметры, создадим окружность и присвоим ей идентификатор id:

```
# ----------------- параметры -----
ds=10                # шаг перемещения
R=12                 # радиус окружности
(xs,ys)=(110,100)    # начальное положение окружности
# -------------------------------
id=canvas.create_oval(xs-R,ys-R,xs+R,ys+R,
          width=5,outline='black',fill='')
```

Там же свяжем главное окно, событие KeyPress и ненаписанную пока функцию-обработчик события:

```
root.bind('<Key>',move)
```

Осталось написать обработчик нажатия клавиши. Для перемещения окружности используем метод канвы move. Метод канвы вызывается через экземпляр класса Canvas, что обеспечивает однозначность вызова, несмотря на формальное совпадение имён!

```
def move(event):
   (Vx,Vy)=(0,0)
   key=event.keysym
   if   key=='Left'  : (Vx,Vy)=(-ds, 0)
   elif key=='Right' : (Vx,Vy)=( ds, 0)
   elif key=='Up'    : (Vx,Vy)=( 0,-ds)
   elif key=='Down'  : (Vx,Vy)=( 0, ds)
   # -------------------------------------
   canvas.move(id,Vx,Vy)
```

Переменная canvas создаётся в «шапке» программы.

```
from tkinter import *
root=Tk()
root.title('Управление')
canvas=Canvas(root,width=220,height=200,bg='white')
canvas.pack()
```

поэтому функция-обработчик должна быть описана после неё. Но сама функция-обработчик должна быть описана до её использования в главной программе, поэтому порядок приведенных фрагментов программы определяется однозначно.

Задание для самостоятельной работы.

1) Соберите все написанные фрагменты, чтобы получить работающую программу.

2) Добавьте в обработчик реакцию приложения на нажатие клавиш > и <. При нажатии клавиши > радиус окружности увеличивается на 1, при нажатии клавиши < радиус окружности уменьшается на 1, однако он не может стать меньше 10 пикселей.

Подсказка. Для получения и изменения размеров окружности используйте метод канвы coords.

Продолжим разработку проекта. Пока у нас получилась довольно скучная программа. Превратим её в простейший графический редактор. Пусть при движении окружности за ней остаётся след — цветная полоска. Полоску составим из небольших квадратиков, которые будут «рождаться» внутри окружности.

Главная программа изменится незначительно. Добавим в неё процедуру создания квадрата с центром в точке xs,ys. Присваивать созданному квадрату

идентификатор нет смысла, так как создаваемые квадраты перемещаться не будут. Стороны всех квадратов сделаем равными величине, равной шагу перемещения окружности, тогда отдельные квадраты будут сливаться в одну линию.

```
# ---------------- параметры -----
ds=10                # шаг перемещения
R=12                 # радиус окружности
d=ds//2              # половина стороны квадрата
(xs,ys)=(110,100)    # начальное положение окружности
# -------------------------------
id=canvas.create_oval(xs-R,ys-R,xs+R,ys+R,
                      width=5,outline='black',fill='')
canvas.create_rectangle(xs-d,ys-d,xs+d,ys+d,
                      width=0,fill='red')
root.bind('<Key>',move)
```

Наибольшие изменения будут сделаны в функции-обработчике. При нажатии клавиш-стрелок кроме перемещения окружности, нам нужно создать квадрат на новом месте. Получить «точку привязки» — центр квадрата можно двумя способами. Первый вариант — использовать метод coords для получения координат окружности, второй вариант — отслеживать изменение координат xs,ys, т.е. изменять их при изменении положения окружности. Выберем второй вариант. Добавим в конце функции move(event) строчки

```
    xs+=Vx; ys+=Vy
    canvas.create_rectangle(xs-d,ys-d,xs+d,ys+d,
                          width=0,fill='red')
```

При запуске программы в окне появится окружность с красным квадратом внутри. Однако при нажатии клавиши-стрелки мы получим сообщение об ошибке «локальная переменная xs используется до присваивания ей значения»:

```
UnboundLocalError:
   local variable 'xs' referenced before assignment
```

Ошибка возникает в строке xs+=Vx; ys+=Vy, в которой изменяются значения переменных xs и ys. Но ведь эти имена уже есть в главной программе, т.е. соответствующие переменные уже существуют и имеют определённые значения! Этими переменными можно пользоваться внутри функции, но только при условии неизменяемости их значений. Чтобы можно было изменять значения таких переменных внутри функции их нужно объявить глобальными.

Следующая версия функции-обработчика вполне работоспособна.

```
def move(event):
   global xs,ys
   (Vx,Vy)=(0,0)
   key=event.keysym
   if   key=='Left'  : (Vx,Vy)=(-ds, 0)
   elif key=='Right' : (Vx,Vy)=( ds, 0)
   elif key=='Up'    : (Vx,Vy)=( 0,-ds)
   elif key=='Down'  : (Vx,Vy)=( 0, ds)
   # ------------------------------------
   canvas.move(id,Vx,Vy)
   xs+=Vx; ys+=Vy
   canvas.create_rectangle(xs-d,ys-d,xs+d,ys+d,
                       width=0,fill='red')
```

Один недостаток этой версии очевиден — все линии рисуются красным цветом. Второй не очевиден — в процессе рисования окружность скрывается за нарисованными квадратиками. Это происходит потому, что более поздние объекты закрывают объекты, созданные ранее. Чтобы «поднять» окружность наверх, используем метод канвы tag_raise(идентификатор_объекта). Добавим к функции move(event) строку canvas.tag_raise(id) и проблема будет решена.

Добавим теперь в функцию move(event) возможность выбора цвета. В главной программе создадим переменную cv со значением 'red', а внутри функции объявим её глобальной. Во вложенных elif продолжим анализ нажатой клавиши. Для клавиш с цифрами от 1 до 6 зададим цвета, а клавишу 0 будем использовать для пустого квадрата. Это позволит перемещать окружность без изменения рисунка.

```
   elif key=='0': cv=''
   elif key=='1': cv='red'
   elif key=='2': cv='green'
   elif key=='3': cv='blue'
   elif key=='4': cv='cyan'
   elif key=='5': cv='magenta'
   elif key=='6': cv='yellow'
```

Последний штрих: при нажатии клавиши Esc будем удалять весь рисунок и восстанавливать начальное состояние редактора. Для этого укажем в параметрах всех создаваемых квадратов один и тот же тэг, например, tags='Q'. Тогда удалить все квадраты сразу можно одним вызовом метода канвы delete('Q'). Дополнительный код можно добавить в функцию move(event), но чтобы проиллюстрировать возможности Tkinter, сделаем второй обработчик только для этой клавиши.

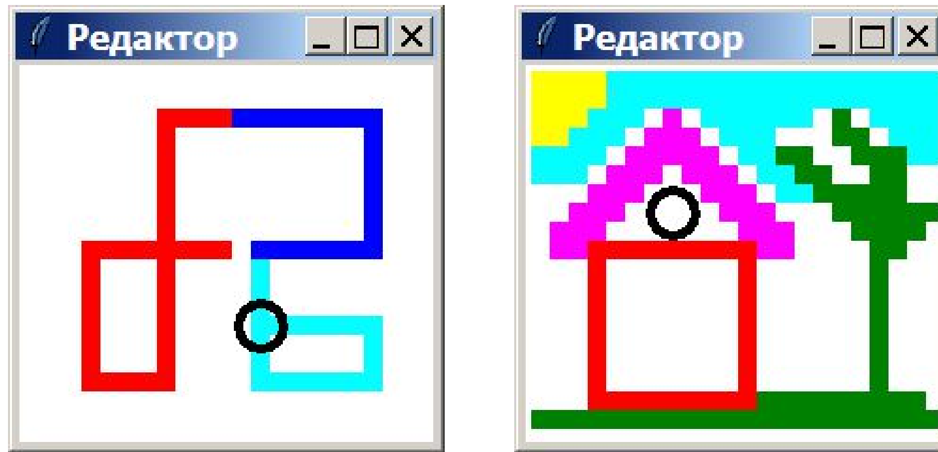
Приведём полный текст редактора.

```
from tkinter import *
root=Tk()
```

```
root.title('Редактор')
canvas=Canvas(root,width=220,height=200,bg='white')
canvas.pack()
def move(event):
   global xs,ys,cv
   (Vx,Vy)=(0,0)
   key=event.keysym
   # --------------------------------------
   if   key=='Left' : (Vx,Vy)=(-ds, 0)
   elif key=='Right': (Vx,Vy)=( ds, 0)
   elif key=='Up'   : (Vx,Vy)=( 0,-ds)
   elif key=='Down' : (Vx,Vy)=( 0, ds)
   # --------------------------------------
   elif key=='0': cv=''
   elif key=='1': cv='red'
   elif key=='2': cv='green'
   elif key=='3': cv='blue'
   elif key=='4': cv='cyan'
   elif key=='5': cv='magenta'
   elif key=='6': cv='yellow'
   # --------------------------------------
   canvas.move(id,Vx,Vy)
   xs+=Vx; ys+=Vy
   canvas.create_rectangle(xs-d,ys-d,xs+d,ys+d,
                     width=0,fill=cv,tags='Q')
   canvas.tag_raise(id)
def restore(event):
   global xs,ys,cv
   canvas.delete('Q')
   (xs,ys)=(110,100); cv='red'
   canvas.coords(id,xs-R,ys-R,xs+R,ys+R)
   canvas.create_rectangle(xs-d,ys-d,xs+d,ys+d,
                     width=0,fill=cv,tags='Q')
# ---------------- параметры -----
ds=10               # шаг перемещения
R=12                # радиус окружности
d=ds//2             # половина стороны квадрата
cv='red'            # начальный цвет квадрата
(xs,ys)=(110,100)   # начальное положение окружности
# ------------------------------
id=canvas.create_oval(xs-R,ys-R,xs+R,ys+R,
                     width=5,outline='black',fill='')
canvas.create_rectangle(xs-d,ys-d,xs+d,ys+d,
                     width=0,fill=cv,tags='Q')
root.bind('<Escape>',restore)
```

```
root.bind('<Key>',move)
```

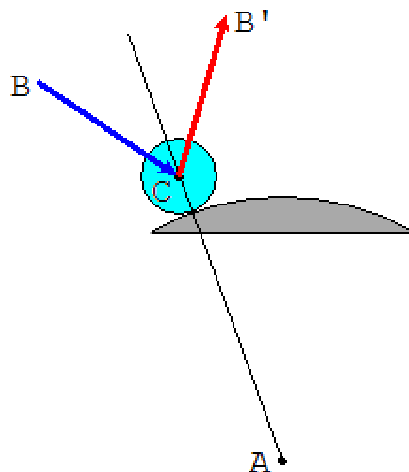
Примеры работы с программой-редактором приведены ниже.



11. Игра «Сквош»

Сквош — это спортивная игра на корте, окруженном со всех сторон стенами. Игроки «вооружены» ракетками, которыми бьют по мячу. После удара одного из игроков мяч отскакивает несколько раз от стен и должен быть отбит вторым игроком.

В нашем проекте оставим одного игрока и три стены. Управлять будем ракеткой стрелками влево и вправо. Если ракетка будет плоской, то игра будет не очень интересной, поэтому сделаем ракетку в виде сегмента.



На рисунке показан момент касания мяча ракетки. Точка A есть центр дуги, образующей поверхность ракетки, точка C есть центр мяча. Стрелка BC

есть вектор, по которому движется мяч до удара, а стрелка CB' - направление движения мяча после удара. При отражении мяча от ракетки выполняется правило: угол падения равен углу отражения, т.е., углы векторов с прямой, проходящей через точки A и C одинаковы.

Для получения вектора CB' отразим точку B относительно указанной прямой методом symmetry: B'= symmetry(A,C,B), тогда CB' = B' – C.

Условие касания мяча и ракетки: расстояние между центрами кругов равно сумме их радиусов. Поскольку движение мяча на экране дискретно, это условие фактически никогда не выполняется. Следовательно проверять придётся условие: расстояние между центрами кругов меньше суммы радиусов ракетки и мяча. Поскольку «внедрение» мяча в ракетку физически невозможно, будем вычислять правильное положение мяча методом линейной интерполяции.

Наиболее сложным моментом при реализации проекта оказалась необходимость одновременного движения мяча и управления ракеткой. Желательно не ограничивать время движения мяча, однако если мы захотим использовать бесконечный цикл для реализации движения мяча, то мы не сможем управлять ракеткой. Можно разнести движение мяча и управление ракеткой в разные потоки, однако эти возможности лежат за пределами тематики данной книги. Наиболее простой способ — использовать метод after для повторного вызова метода, реализующего перемещение мяча, через заданный промежуток времени.

Написание программы начнём с подключения библиотек и создания окна приложения:

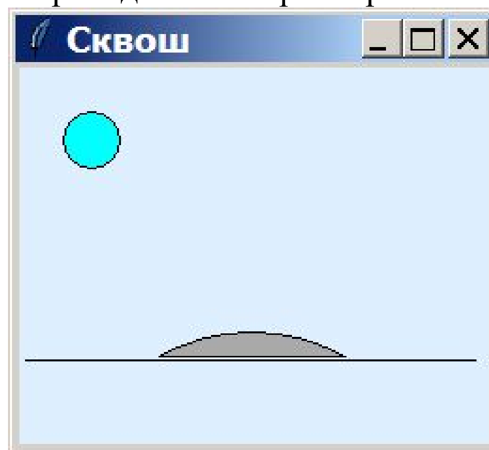
```
from time import *
from geom_sys import *
from tkinter import *
from tkinter.messagebox import *
root=Tk()
root.title('Сквош')
w=250; h=200
canvas=Canvas(root,width=w,height=h,bg='#ddeeff')
canvas.pack()
```

Зададим параметры мяча и ракетки. Зафиксируем также начальное состояние игры в «нулевых» переменных.

```
# ------- параметры мяча ---------
(xc0,yc0)=(40,40)
(dx0,dy0)=(6,8)
xc=xc0; yc=yc0  # координаты центра мяча
R=15            # радиус мяча
dx=dx0; dy=dy0  # направление движения
```

```
ball=canvas.create_oval(xc-R,yc-R,xc+R,yc+R,fill='cyan')

# ------- параметры ракетки ------
dr=50        # половина хорды ракетки
yrkt=h-3*R   # размещение ракетки по y-координате
RR=2*dr      # радиус дуги ракетки 60 градусов
# начальные координаты центра дуги
(xa0,ya0)=(w/2,yrkt+sqrt(3)*dr)
(xa,ya)=(xa0,ya0)
# угол и размер дуги ракетки
strt=60; exnt=60
raket=canvas.create_arc(xa-RR,ya-RR,xa+RR,ya+RR,
            start=strt,extent=exnt,fill='darkgray',
            style='chord')
# стоп-линия
line=canvas.create_line(5,yrkt+2,w-5,yrkt+2)
```

На следующем рисунке показано начальное положение игры, соответствующее приведенным параметрам.



Для вычисления расстояния между центрами ракетки и мяча напишем вспомогательную функцию dist:

```
def dist(xa,ya,xc,yc):
   return sqrt((xa-xc)**2+(ya-yc)**2)
```

Движение мяча реализуем в функции move_ball. Объявим глобальными переменные xc,yc – координаты центра мяча, xa,ya – координаты цента дуги, образующей «ударную» поверхность ракетки, dx,dy – вектор направления движения мяча, start – момент начала игры. При завершении игры будет вычислено время, в течении которого игрок смог удержать мяч от вылета за стоп-линию.

```
def move_ball():
```

```
    global xc,yc,xa,ya,dx,dy,start
```

Отражение от стены сводится к изменению знака у соответствующей компоненты вектора dx,dy

```
    if xc+dx<R or xc+dx>w-R: dx=-dx
    if yc+dy<R or yc+dy>h-R: dy=-dy
```

Расчёт следующего положения мяча. Сначала мы просто прибавляем компоненты вектора направления к текущим координатам центра мяча. Для нового положения мяча вычисляем значение выражения

```
Dnext = dist(xa,ya,xnext,ynext)-R-RR
```

Если значение переменной Dnext больше нуля, то мяч ещё не коснулся ракетки, если меньше нуля — то мяч уже пересекается с ракеткой. В последнем случае необходимо уменьшить величину перемещения мяча, чтобы получить положение, в котором мяч будет касаться поверхности ракетки. Используем для этого линейную интерполяцию между текущим и следующим положением мяча.

Вычислим то же самое выражение для текущего положения мяча:

```
Dpred=dist(xa,ya,xc,yc)-R-RR
```

тогда переместить мяч нужно только на вектор q*dx, q*dy, где коэффициент q вычисляется по формуле

```
q=Dpred/(Dpred-Dnext)
```

Итак, фрагмент функции, вычисляющий новое положение мяча:

```
    xnext=xc+dx; ynext=yc+dy
    Dnext = dist(xa,ya,xnext,ynext)-R-RR
    if  Dnext<0 and xa-dr<xnext<xa+dr:
       # корректировка положения мяча
       Dpred=dist(xa,ya,xc,yc)-R-RR
       q=Dpred/(Dpred-Dnext)
       xnext=xc+q*dx
       ynext=yc+q*dy
```

После установления контакта с ракеткой вычисляем новый вектор движения мяча:

```
       (wx,wy)=symmetry((xa,ya),(xnext,ynext),
                        (xnext-dx, ynext-dy))
       dx=wx-xnext; dy=wy-ynext
```

После выхода из оператора if (или его пропуска) можно переместить мяч в новое положение

```
    canvas.move(ball,xnext-xc,ynext-yc)
    xc=xnext; yc=ynext
```

Выполним анализ положения мяча относительно стоп-линии. Если мяч оказался за линией, то выводим сообщение о завершении игры с указанием времени игры (чем дольше игрок отбивает мяч, тем он круче!) и вопросом «Начать заново?». Если игрок нажимает «Да», то восстанавливается начальная позиция игры.

```
    if yc>yrkt:
       canvas.itemconfig(ball,fill='red')
       timeplay=time()-start
       yes=askyesno('Вы проиграли!','Время игры = '+
                    '{0:.1f}'.format(timeplay)+
                    ' сек.\nНачать заново?')
       if yes:
          # восстановление исходной позиции
          canvas.move(ball,xc0-xc,yc0-yc)
          xc=xc0; yc=yc0; dx=dx0; dy=dy0
          canvas.itemconfig(ball,fill='cyan')
          canvas.move(raket,xa0-xa,ya0-ya)
          (xa,ya)=(xa0,ya0)
       else:
          # конец игры
          root.destroy()
          return
```

После паузы в 20 миллисекунд функция move_ball вызывается повторно. Однако здесь нет рекурсии, так как этот вызов происходит *после* завершения предыдущего вызова.

```
    canvas.after(20,move_ball)
```

Замечание. При выводе сообщения число секунд округляется до одного знака после запятой с помощью метода format.

Осталось написать функцию, реализующую перемещение ракетки при нажатии стрелок влево или вправо:

```
def move_raket(event):
    global xa
    key=event.keysym
    if key=='Left':
       canvas.move(raket,-10,0)
       xa-=10
    if key=='Right':
       canvas.move(raket,10,0)
       xa+=10
```

и завершить программу:

```
start=time()
move_ball()
root.bind('<Key>',move_raket)
root.mainloop()
```

Задание для самостоятельного выполнения. Напишите программу, моделирующую игру двух игроков. Каждый игрок управляет своей ракеткой, которой отбивает движущийся мяч. В игре должен вестись подсчёт голов для каждого игрока. Выигрывает тот, кто первым забьёт третий гол.

Возможный вид приложения показан на рисунке.

Подсказка: используйте разные клавиши для управления левой и правой ракеткой.

12. Исполнитель «Черепашка»

Этого исполнителя придумал математик Сеймур Пейперт для обучения детей началам программирования. В основе лежит образ черепахи, которая ползает по песку, оставляя за собой след. Исполнитель "понимает" и может выполнить несколько простых команд: "вперед", "назад", "поверни влево", "поверни вправо" и некоторые другие. При движении вперёд или назад нужно указать, на какое расстояние перемещается Черепашка, при поворотах – на какой угол поворачивается Черепашка.

В Питоне есть модуль turtle, который независимо от библиотеки tkinter реализует исполнителя (turtle переводится с английского как «черепаха»). Но для иллюстрации возможностей tkinter реализуем исполнителя самостоятельно.

Более того, мы реализуем возможность создавать сразу нескольких исполнителей-черепашек, которые синхронно выполняют все команды.

Разработаем класс Turtle. Параметры черепашки сохраняются в переменных класса:

xp, yp – координаты черепашки;

fi – угол в градусах, задающий направление движения черепашки;

cv – цвет линии, которую рисует черепашка;

wl – толщина линии;

st – тип поведения: «послушная» черепашка (st=1) правильно выполняет команды поворота, а «непослушная» (st=-1) поворачивает в противоположную сторону.

При вызове конструктора можно изменить значения по умолчанию именованных параметров angle (задаёт значение переменной fi), color (задаёт значение переменной cv), width (задаёт значение переменной wl) и stype (задаёт значение переменной st).

```
class Turtle:
   def __init__(self,
               angle=0,color='black',width=1,stype=1):
      self.xp=xs
      self.yp=ys
      self.fi=angle*pi/180
      self.cv=color
      self.wl=width
      if stype==1: self.st=1
      else : self.st=-1
```

Конструктор размещает черепашку в центре канвы, координаты центра вычисляются отдельно и хранятся в глобальных переменных xs, ys.

Будем изображать черепашку в виде окружности и отрезка прямой, указывающей направление движения «вперёд». Размер черепашки зададим в глобальной переменной d.

```
      x=self.xp + 2*d*cos(self.fi)
      y=self.yp - 2*d*sin(self.fi);
      self.oval=canvas.create_oval(
                self.xp-d,self.yp-d,self.xp+d,self.yp+d,
                width=1,outline=self.cv)
      self.line=canvas.create_line(
                self.xp,self.yp,x,y,fill=self.cv)
```

Команды «вперёд» и «назад» реализуем с помощью методов класса. Метод move(self, len) перемещает Черепашку «вперёд» без рисования линии. Метод forw(self, len) перемещает Черепашку «вперёд» и рисует линию, которая соединяет начальное и конечное положение Черепашки. Метод back(self,len) делает то же самое, только Черепашка ползёт назад.

```
   def move(self,len):
      x=self.xp + len*cos(self.fi)
      y=self.yp - len*sin(self.fi);
      canvas.move(self.oval,x-self.xp,y-self.yp)
      canvas.move(self.line,x-self.xp,y-self.yp)
```

```
        self.xp=x; self.yp=y
    def forw(self, len):                # вперёд
        x=self.xp + len*cos(self.fi)
        y=self.yp - len*sin(self.fi);
        canvas.create_line(self.xp,self.yp,x,y,
                           fill=self.cv,width=self.wl)
        canvas.move(self.oval,x-self.xp,y-self.yp)
        canvas.move(self.line,x-self.xp,y-self.yp)
        self.xp=x; self.yp=y
    def back(self, len):                # назад
        x=self.xp-len*cos(self.fi)
        y=self.yp+len*sin(self.fi);
        canvas.create_line(self.xp,self.yp,x,y,
                           fill=self.cv,width=self.wl)
        canvas.move(self.oval,x-self.xp,y-self.yp)
        canvas.move(self.line,x-self.xp,y-self.yp)
        self.xp=x; self.yp=y
```

Повороты налево и направо выполняем методами left и right:

```
    def left(self, ug):
        self.fi+=self.st*ug*pi/180
        x=self.xp+2*d*cos(self.fi)
        y=self.yp-2*d*sin(self.fi);
        canvas.coords(self.line,self.xp,self.yp,x,y)
    def right(self, ug):
        self.fi-=self.st*ug*pi/180
        x=self.xp+2*d*cos(self.fi)
        y=self.yp-2*d*sin(self.fi);
        canvas.coords(self.line,self.xp,self.yp,x,y)
```

Класс Turtles реализует список Lst черепашек. Конструктор создаёт пустой список, метод addTurtle добавляет новую черепашку в список. Кроме того, класс содержит параметр state, который может приниать значения ‘draw’ – рисуй, и ‘undraw’ – не рисуй.

Методы класса forw, back, left и right позволяют отдавать одну и ту же команду сразу всем черепашкам из списка. Эти методы для каждой черепашки из списка Lst вызывают тот же метод класса Turtle. Особенность методов forw и back в том, что одноименные методы вызываются только если выполнено условие self.state==‘draw’, иначе вызывается метод move(len) при движении черепашки вперёд и метод move(-len) при движении назад.

```
class Turtles:
    def __init__(self):
        self.Lst=[]
        self.state='draw'
    def addTurtle(self,obj):
```

```
            self.Lst.append(obj)
        def move(self,len):
            for w in self.Lst: w.move(len)
        def forw(self,len):
            if self.state=='draw':
               for w in self.Lst: w.forw(len)
            else:
               for w in self.Lst: w.move(len)
        def back(self,len):
            if self.state=='draw':
                for w in self.Lst: w.back(len)
            else:
               for w in self.Lst: w.move(-len)
        def left(self, ug):
            for w in self.Lst: w.left(ug)
        def right(self, ug):
            for w in self.Lst: w.right(ug)
        def setColor(self,color):
            for w in self.Lst: w.cv=color
```

Наиболее сложным является метод reaction, который является обработчиком действий пользователя. Предусмотрены следующие управляющие клавиши:

'Up' — при нажатии стрелки вверх черепашка перемещается на расстояние 10 пикселей в направлении, заданном в параметрах черепашки;

'Down' — при нажатии стрелки вниз черепашка перемещается на расстояние 10 пикселей в противоположном направлении;

'Left' — при нажатии стрелки влево черепашка поворачивается на угол 15 градусов против часовой стрелки;

'Right' — при нажатии стрелки вправо черепашка поворачивается на угол 15 градусов по часовой стрелке;

'Next' — при нажатии клавиши PageDown устанавливается режим 'draw';

'Prior' — при нажатии клавиши PageUp устанавливается режим 'undraw';

Цифровые клавиши от '1' до '7' задают цвет линий.

```
        def reaction(self,event):
            key = event.keysym
            if key == 'Up'  : self.forw(10)
            if key == 'Down': self.back(10)
            if key == 'Left': self.left(15)
            if key == 'Right': self.right(15)
            if key == 'Next': self.setState('draw')
            if key == 'Prior': self.setState('undraw')
            if key == '1' : self.setColor('red')
            if key == '2' : self.setColor('green')
```

```
        if key == '3' : self.setColor('blue')
        if key == '4' : self.setColor('cyan')
        if key == '5' : self.setColor('magenta')
        if key == '6' : self.setColor('yellow')
        if key == '7' : self.setColor('black')
```

После этого изменяется цвет изображения черепашек:

```
        for w in self.Lst:
           canvas.itemconfig(w.oval,outline=w.cv)
           canvas.itemconfig(w.line,fill=w.cv)
```

Чтобы очистить рисунок от изображений черепашек нужно нажать на клавишу «пробел»:

```
        if key == 'space':
           for w in self.Lst:
              canvas.itemconfig(w.oval,outline='')
              canvas.itemconfig(w.line,fill='')
```

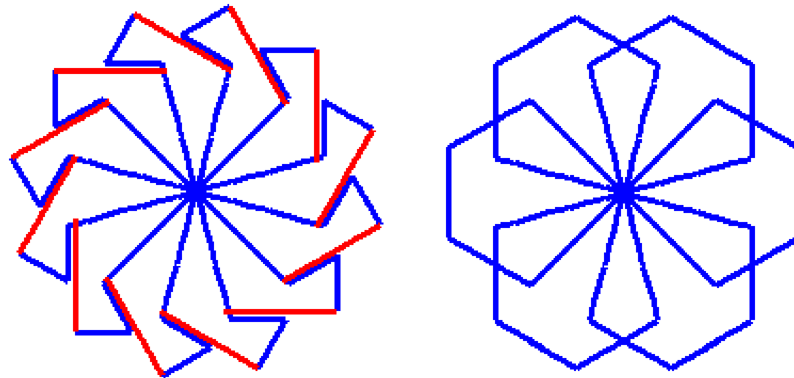
В начало модуля добавим метод create_win для подключения библиотек и создания окна приложения с "холстом" для рисования. Метод возвращает ссылку на созданное окно.

```
def create_win(w_title,c_width,c_height,color='white'):
   global root, canvas, xs, ys
   xs=c_width/2; ys=c_height/2
# создание окна приложения с "холстом" для рисования
   root=Tk()
   root.title(w_title)
   canvas=Canvas(root,
          width=c_width,height=c_height,bg=color)
   canvas.pack()
   return root           # необязательно
```

Теперь напишем программу, в которой создадим 12 «послушных» черепашек и нарисуем с их помощью какой-нибудь узор.

```
from turtles import *
win=create_win('Черепашки',250,250)
# -------- main -------
n=12; df=360/n
L=Turtles()
for i in range(n):
   L.addTurtle(Turtle(angle=i*df+df/2,color='blue',
                      width=3,stype=1))
win.bind('<Key>',L.reaction)
```

Начальное положение черепашек — в центре окна, направления движения черепашек равномерно распределены по кругу. В процессе управления черепашками мы несколько раз меняли направление движения и один раз цвет линий. Получился левый рисунок. Понятно, что при другой последовательности нажатия клавиш получится другой рисунок, но между всеми рисунками будет определённое сходство — все линии будут закручиваться в одну и ту же сторону (рисунок слева).



Модифицируем программу — будем чередовать послушных и непослушных черепашек. Для этого параметру stype присвоим значение выражения (-1)**i. Число -1, возведённое в чётную степень равно 1, следовательно все чётные черепашки будут послушными. Число -1, возведённое в нечётную степень равно -1, следовательно все нечётные черепашки будут непослушными.

```
from turtles import *
win=create_win('Черепашки',250,250)
# -------- main -------
n=12; df=360/n
L=Turtles()
for i in range(n):
   L.addTurtle(Turtle(angle=i*df+df/2,color='blue',
                      width=3,stype=(-1)**2))
win.bind('<Key>',L.reaction)
```

Результат работы изменённой программы приведен на рисунке справа.
Этот же рисунок можно получить, явно указав команды черепашки:

```
from turtles import *
win=create_win('Черепашки',250,250)
# -------- main -------
n=12; df=360/n
L=Turtles()
```

```
for i in range(n):
   L.addTurtle(Turtle(angle=i*df+df/2,color='blue',
                      width=3,stype=(-1)**i))
L.forw(70)
L.left(75)
L.forw(50)
L.left(60)
L.forw(25)
```

Однако в этом варианте на рисунке остаются изображения черепашек. Чтобы их убрать используем написанный нами метод reaction. Предварительно создадитм событие e, для которого e.keysym = ‘space’, и передадим его методу reaction:

```
e=Event()
e.keysym='space'
L.reaction(e)
```

Однако удобнее, на взгляд автора, добавить новый метод в класс Turtles:

```
   def execEvent(self, cmd):
      e=Event()
      e.keysym=cmd
      self.reaction(e)
```

тогда завергить предыдущую программу можно одним оператором

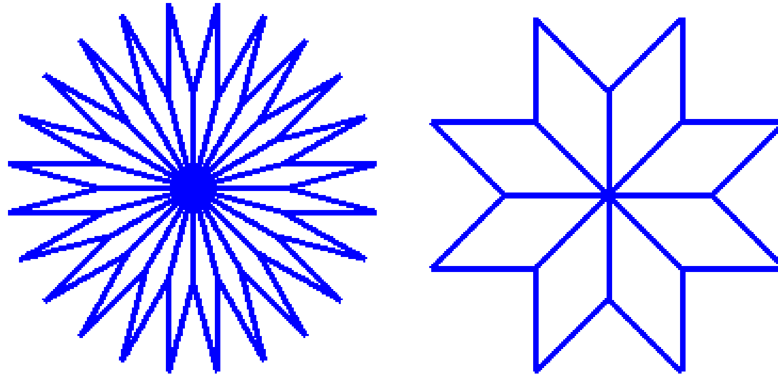
```
L.execEvent('space')
```

Следующая программа иллюстрирует использование переменных вместо числовых значений, что позволяет легко получать целый ряд узоров, изменяя их параметры.

Выполним программу

```
from turtles import *
win=create_win('Черепашки',250,250)
# -------- main -------
n=24; df=360/n
L=Turtles()
for i in range(n):
   L.addTurtle(Turtle(angle=i*df,color='blue',
                      width=3,stype=1))
   L.addTurtle(Turtle(angle=i*df,color='blue',
                      width=3,stype=-1))
len=50
L.forw(len)
L.left(df)
L.forw(len)
L.execEvent('space')
```

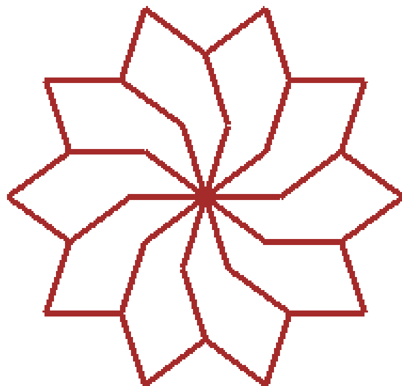
В результате получим левый рисунок.



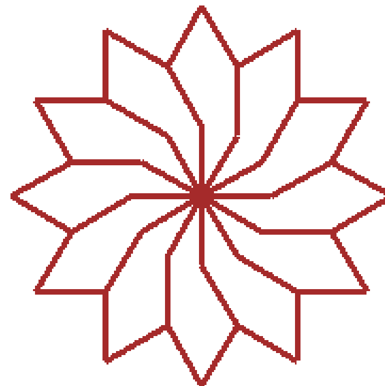
Изменим значение параметров: n = 8; len = 55 и мы получим правый рисунок.

Задания для самостоятельной работы.

1. Составьте параметризированную программу рисования черепашками следующих узоров:



А) n = 10, len =40



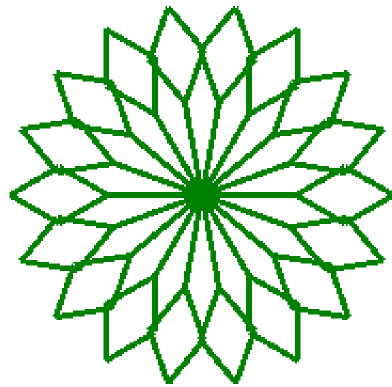
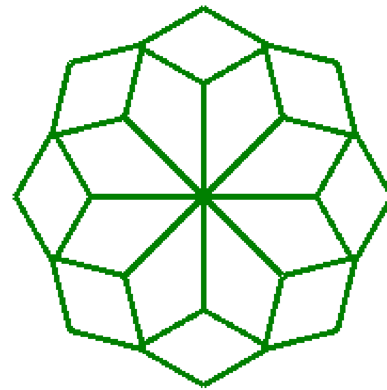
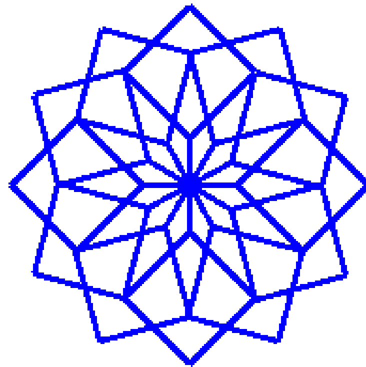
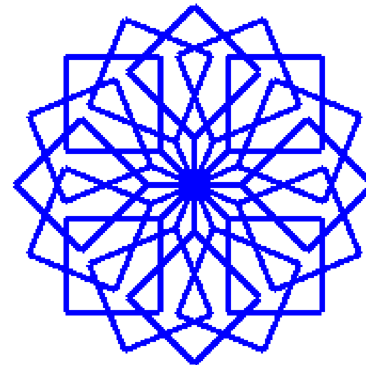
Б) n = 12, len = 37

2. Добавьте в программу процедуру рисования ромба. Параметрами ромба являются величина одного из углов fi и длина стороны len. В качестве подсказки приведём алгоритм рисования ромба в командах черепашки:

```
left(fi/2);   forw(len)
right(fi);    forw(len)
left(fi);     back(len)
right(fi/2);  back(len)
left(fi/2)
```

В результате черепашка нарисует перед собой ромб и вернётся в своё состояние (включая направление движения) которое у неё было перед началом рисования ромба.

Пользуясь этой процедурой, нарисуйте следующие узоры:

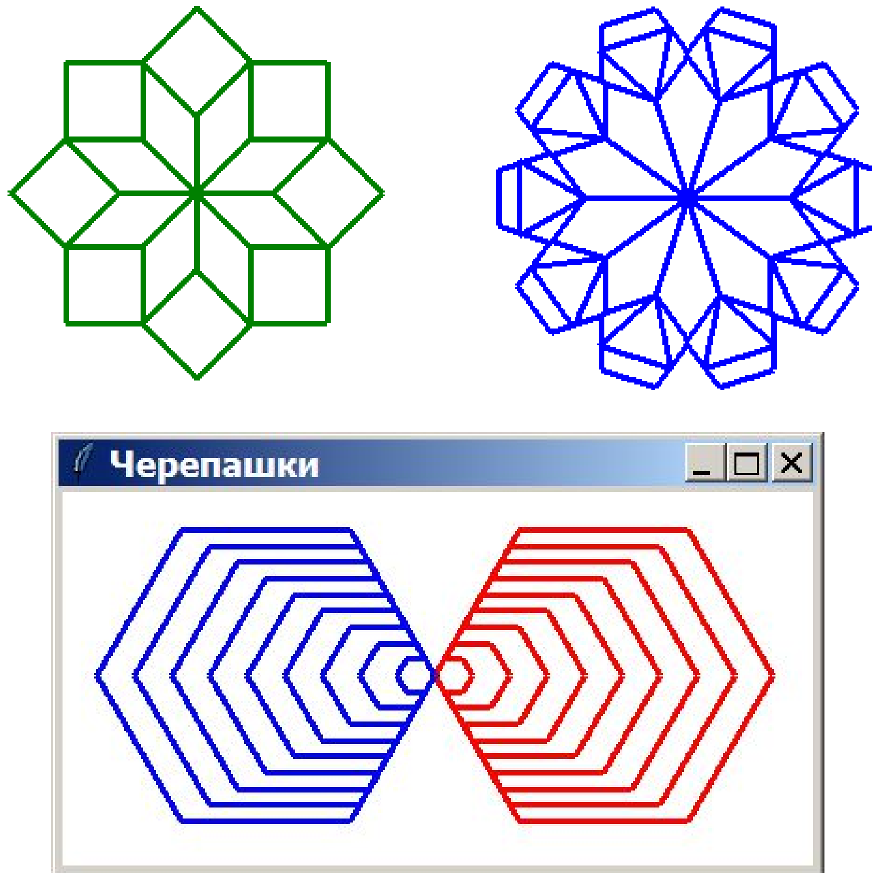





3. Добавьте в программу процедуру рисования правильного n-угольника. Параметрами процедуры является число сторон и длина len одной стороны. Черепашка должна рисовать n-угольник перед собой и возвращаться в своё начальное состояние.

Пользуясь этой процедурой, нарисуйте приведенные ниже узоры. Реализуйте два варианта программы:

А) с использованием одной черепашки;

Б) с использованием нескольких черепашек, каждая из которых рисует один фрагмент узора.



13. Исполнитель «Чертёжник»

Рассмотрим ещё одного исполнителя, который работает в декартовой системе координат и можнет чертить прямые линии. Напишем модуль drawers, в котором объяевими два класса: класс Drawer – отдельного исполнителя, и класс Drawers – множества совместно работающих исполнителей.

Для работы исполнителя создадим графическое окно методом create_win:

```
def create_win(w_title, c_width, c_height,
color='white'):
   global root, canvas, xs, ys
   xs=c_width/2; ys=c_height/2
# создание окна приложения с "холстом" для рисования
   root=Tk()
   root.title(w_title)
   canvas=Canvas(root, width=c_width, height=c_height,
                 bg=color)
   canvas.pack()
```

```
    return root
```

В классе Drawer реализуем методы отдельного исполнителя:

__init__(self, xp=0, yp=0, color='black', width=1) – конструктор. Параметры исполнителя:

xp, yp – текущее положение. Координаты исполнителя задаются в декартовой системе координат, причем ось y направлена вверх;

color – цвет линии, оставляемой исполнителем;

width – толщина линии, оставляемой исполнителем.

Методы:

moveto(self, x, y) – перемещение исполнителя в точку с координатами x,y без рисования линии;

lineto(self, x, y) – перемещение исполнителя в точку с координатами x,y, при этом проводится линия, соединяющая прежнее положение исполнителя с его новым положением.

moveon(self, dx, dy) – перемещение исполнителя на вектор dx,dy без рисования линии;

lineon(self, dx, dy) – перемещение исполнителя на вектор dx,dy, при этом проводится линия, соединяющая прежнее положение исполнителя с его новым положением.

Все перечисленные методы изменяют значения переменных xp, yp, в которых запоминается новое положение исполнителя.

```
class Drawer:
   def __init__(self,xp=0,yp=0,color='black',width=1):
      self.xp=xs+xp
      self.yp=ys-yp
      self.cv=color
      self.wl=width
   def moveto(self,x,y):
      self.xp=xs+x
      self.yp=ys-y
   def lineto(self,x,y):
      canvas.create_line(self.xp,self.yp,xs+x,ys-y,
                         fill=self.cv,width=self.wl)
      self.xp=xs+x
      self.yp=ys-y
   def moveon(self,dx,dy):
      self.xp+=dx
      self.yp-=dy
   def lineon(self,dx,dy):
      xp=self.xp+dx
      yp=self.yp-dy
      canvas.create_line(self.xp,self.yp,xp,yp,
```

```
                              fill=self.cv,width=self.wl)
        self.xp+=dx
        self.yp-=dy
```

Основной класс модуля = это класс Drawers, основой которого является список отдельных исполнителей-чертёжников. Он содержит методы moveto(…), lineto(…), moveon(…), lineon(…), которые вызывают одноимённые методы класса Drawer для каждого исполнителя.

```
    def moveto(self, D):
       for (w,d) in zip(self.Lst,D): w.moveto(*d)
    def lineto(self, D):
       for (w,d) in zip(self.Lst,D): w.lineto(*d)
    def moveon(self, D):
       for w in self.Lst:
          for d in D: w.moveon(*d)
    def lineon(self, D):
       for w in self.Lst:
          for d in D: w.lineon(*d)
```

Кроме того класс содержит и «свои» методы:

addDrawer(self, obj) – добавоение нового исполнителя в общий список:

axes(self,x,y) – задание начала координат среды исполнителя. Значения параметров x,y присваиваются глобальным переменным xs, ys, задающим начало координат среды исполнителя. Одновременно метод рисует и подписывает оси системы координат, в которой «работают» все созданные исполнители.

Приведём пример программы

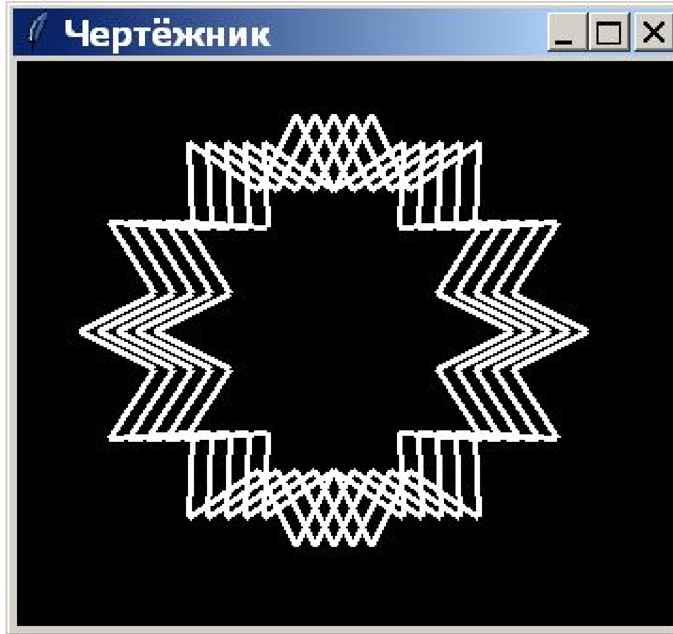
```
from drawers import *
root=create_win('Чертёжник',350,250,color='lightblue')
L=Drawers()
L.axes(40,220)
for i in range(5):
   for j in range(3):
      w=Drawer(50*i,50*(j+1),width=3,color='blue')
      L.addDrawer(w)
D=[(40,40),(40,-40),(-40,-40),(-40,40)]
L.lineon(D)
```

Результат:

Можно соединить возможности модулей drawers и geom._sys. Пример:

```
from drawers import *
from geom_sys import *
root=create_win('Чертёжник',350,300,color='black')
L=Drawers()
L.axes(150,70)
for i in range(5):
   w=Drawer(10*(i-2),0,width=3,color='white')
   L.addDrawer(w)
D=[(20,40),(20,-40)]
n=12
for k in range(n):
   L.lineon(D)
   D=turn((0,0),D, 360/n)
```

Результат представляет собой узор, состоящий из пяти наложенных друг на друга (с небольшим сдвигом) зубчатых многоугольников.



Однако основная цель модуля drawers – визуализация решения дифференциальных уравнений. Такие уравнения описывают поведение некоторой системы, в которой скорость изменения параметров системы можно выразить через её текущее состояние.

Для наглядности будем рассматривать системы с двумя параметрами.

Примеры.

1) Колебание маятника. Состояние маятника можно описать двумя значениями: угловым отклонением маятника от положения равновесия и его мгновенной угловой скоростью.

2) Система «хищник-жертва». Состояние системы определяется двумя параметрами: численностью жертв и численностью хищников. Взаимодействие между ними описываются системой двух дифференциальных уравнений.

Состояние системы изменяется со временем, поэтому её параметры есть функции времени x = x(t), y = y(t). В общем случае ситема уравнений имеет вид:

$$\frac{dx}{dt} = f(x, y)$$

$$\frac{dy}{dt} = g(x, y)$$

Конкретное решение системы определяется при задании начальных условий – начального состояния – обычно при t=0: $x(0)=x_0$, $y(0)=y_0$.

В ряде случаев можно получить решение в виде формулы, но более универсальным является численный метод решения системы. Для начала приведём систему уравнений к следующему виду:

$$dx = f(x, y) \cdot dt$$

$$dy = g(x, y) \cdot dt$$

С точки зрения высшей математики dt – это бесконечно малая величина, но для расчета мы заменим её на конечное приращение Δt, тогда получим приближенную систему уравнений

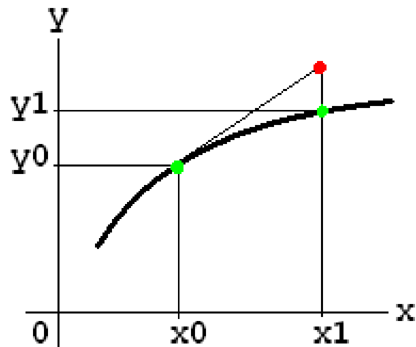
$$\Delta x = f(x, y) \cdot \Delta t$$

$$\Delta y = g(x, y) \cdot \Delta t$$

Теперь, зная текущие значения переменных x и y, легко найти их следующие значения. К сожалению, точность такого решения зависит от величины приращения Δt, чем оно меньше, тем решение точнее. Однако в таком случае и программа будет работать существенно дольше.

Математики давно нашли выход из этой ситуации. Они придумали алгоритмы, которые повышают точность расчета без уменьшения шага по времени.

Из системы уравнений следует, что пара значений f(x,y), g(x,y) есть компоненты вектора производной, который задаёт направление касательной к траектории. Математики называют графики зависимости y от x фазоывми кривыми/



На рисунке зеленым цветом отмечены две точки фазовой кривой, соответствующие моментам времени t0 и t1. Красным отмечена точка, полученная в результате использования численного расчета. Чтобы уменьшить погрешность, вместо «касательного» вектора возьмём полусумму касательных векторов в точках x0,y0 и x1,y1. Это позволит существенно уменьшить погрешность расчета.

Алгоритм

```
t=0; x=x0; y=y0
пока t<tend повторять
   xp,yp = [x+ Δt*f(x,y), y+ Δt*g(x,y)]
```

```
x,y = [x+*Δt*(f(x,y)+f(xp,yp))/2,
       y+*Δt*(g(x,y)+g(xp,yp))/2]
t = t+Δt
```

Для повышения точности можно подбирать каждый раз значение приращения Δt таким образом, чтобы расстояние между текущим и новым состоянием системы не превышало заданной величины.

Дополним модуль drawers методами

startPos(pos,col) – задаёт начальные точки и цвет фазовых кривых;

nextPos(pos, func) – метод получает набор текущих состояний (координат точек фазовой плоскости) и возвращает набор состояний через некоторый промежуток времени. Второй аргумент метода – функция расчета правых частей системы дифференциальных уравнений.

Рассмотрим этот метод подробнее.

```
def nextPos(self,pos,func):
   dt=0.5
   while True:
      V=[(dt*func(x,y)[0], dt*func(x,y)[1])
         for x,y in pos]
      if max(dx**2+dy**2 for dx, dy in V)<2: break
      else: dt=dt*0.9
   R=[(px+Vx, py+Vy)
      for (px,py), (Vx,Vy) in zip(pos,V)]
   W=[(dt*func(x,y)[0], dt*func(x,y)[1]) for x,y in R]
   self.t+=dt
   return [(px+(Vx+Wx)/2, py+(Vy+Wy)/2)
           for (px,py),(Vx,Vy),(Wx,Wy) in zip(pos,V,W)]
```

Метод подбирает величину приращения по времени, вычисляет (с использованием вышеописанного алгоритма) и возвращает новое состояние системы.

Теперь можно рассмотреть некоторые системы.

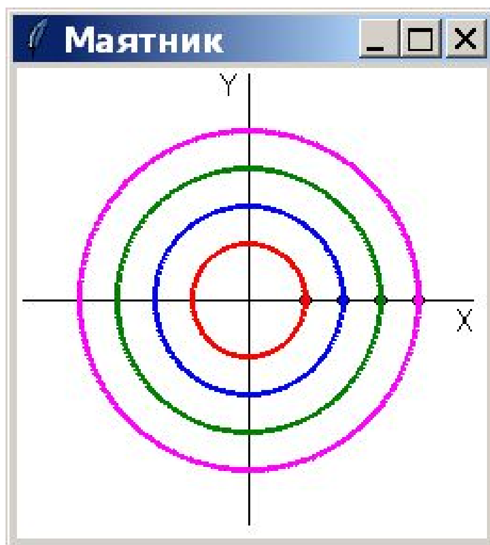
Математический маятник. Обозначим угол отклонения через x, а угловую скорость через y. При небольших углах отклонения система уравнений, описывающих движение маятника имеет вид

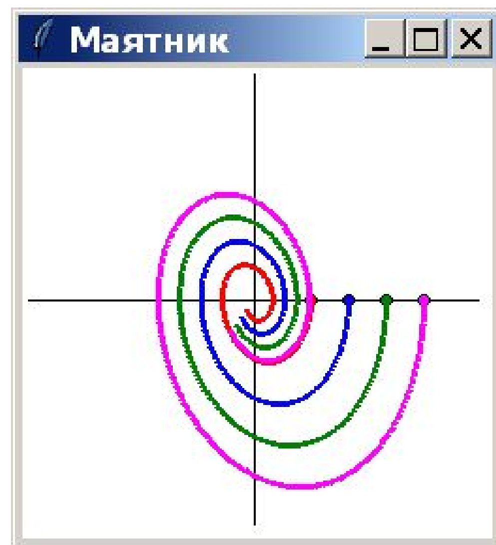
$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = -k \cdot x$$

```
from turtles import *
root=create_win('Маятник',250,250)
def func(x,y):
   return [y, -x]
```

```
# -------- main -------
pos=[(30,0),(50,0),(70,0),(90,0)]
col=['red','blue','green','magenta']
n=len(pos)
k=len(col)
L=Turtles()
for i in range(n):
   w=Turtle(color=col[i%k],width=3)
   L.addTurtle(w)
L.execEvent('space')
L.axes(125,125)
L.startPos(pos,col)
while L.t<2*pi:
   pos=L.nextPos(pos, func)
   L.lineto(pos)
   root.update()
```





Маятник с трением. Заменим функцию для расчета правой части уравнений на

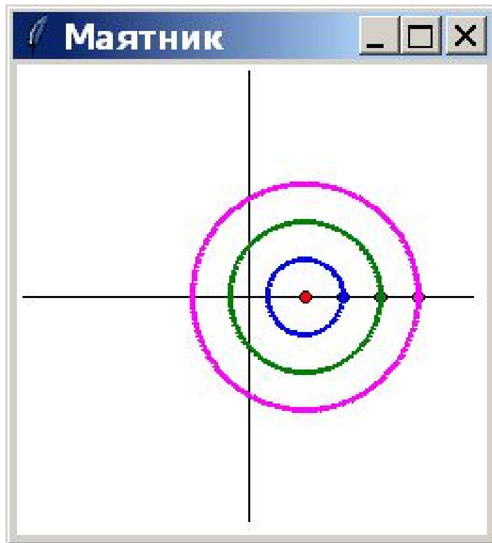
```
def func(x,y):
   return [y, -2*x-0.5*y]
```

Получим затухание колебаний маятника (рисунок справа).

Положение равновесия системы может не совпадать с началом координат (в точке равновесия правые части системы обращаются в 0). Заменим функцию func:

```
def func(x,y):
   return [y, -x+30]
```

Получим в результате точку равновесия (30, 0)



При больших углах система становится нелинейной и её поведение усложняется

$$\frac{dx}{dt} = y$$

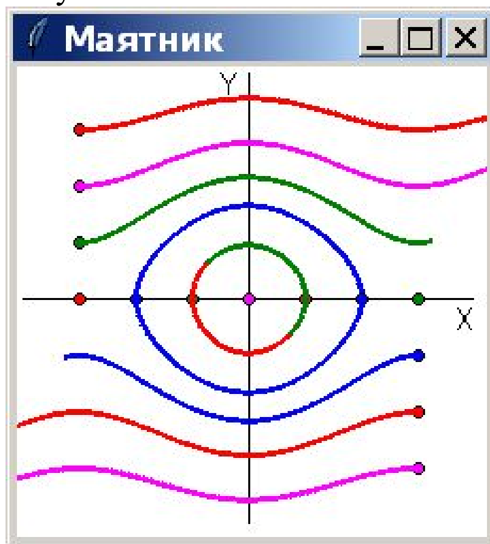
$$\frac{dy}{dt} = -k \cdot \sin(x)$$

Программа

```
from turtles import *
root=create_win('Маятник',250,250)
def func(x,y):
   q=pi/90
   return [y*q, -sin(x*q)]
# -------- main -------
pos=[]
for x in range(-90,91,30):
   pos.append((x,0))
for y in range(-90,-29,30):
   pos.append((90,y))
for y in range(30,91,30):
   pos.append((-
90,y))col=['red','blue','green','magenta']
n=len(pos)
k=len(col)
L=Turtles()
for i in range(n):
```

```
        w=Turtle(color=col[i%k],width=3)
        L.addTurtle(w)
    L.execEvent('space')
    L.axes(125,125)
    L.startPos(pos,col)
    while L.t<120:
        pos=L.nextPos(pos, func)
        L.lineto(pos)
        root.update()
```

Результат



X – угол отклонения. При небольших углах наблюдаются колебания маятника вокруг нижней точки равновесия. При максимальном отклонении скорость маятника равна нулю.

Y – мгновенная скорость. Если начальная скорость небольшая, то также наблюдаются колебания маятника вокруг нижней точки равновесия. При некоторой начальной скорости маятник доходит до верхней точки и возвращается назад. При превышении этой скорости маятник будет вращаться!

Система «хищник – жертва». Приближенно (при довольно серьёзных ограничениях) поведение системы описывается системой дифференциальных уравнений

$$\frac{dx}{dt} = Ax - Cxy$$

$$\frac{dy}{dt} = -By + Dxy$$

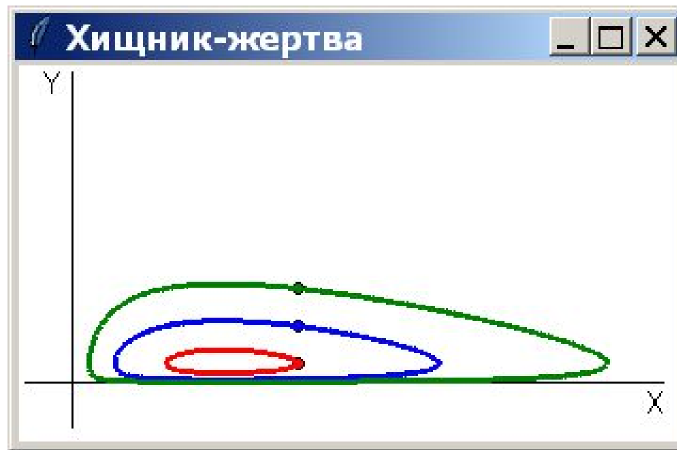
где x есть число жертв, y есть число хищников.

Параметры системы:
A – коэффициент размножения жертв при отсутствии хищников
B – коэффициент вымирания хищников при отсутствии жертв
C – коэффициент истребления жертв хищниками
D – коэффициент прироста хищников за счет истребления жертв.

```
from turtles import *
root=create_win('Хищник-жертва',350,200)
def func(x,y):
   return [A*x-C*x*y, -B*y+D*x*y]
# -------- параметры -------
A=…; C=…
B=…; D=…
pos=…
#----------------------
col=['red','blue','green','magenta']
n=len(pos)
k=len(col)
L=Turtles()
L.axes(30,170)
for i in range(n):
   w=Turtle(color=col[i%k],width=3)
   L.addTurtle(w)
L.execEvent('space')
L.startPos(pos,col)
while L.t<30:
   pos=L.nextPos(pos, func)
   L.lineto(pos)
   root.update()
```

Получим результаты для разных сочетаний коэффициентов.

```
A=0.2; C=0.02
B=0.4; D=0.005
pos=[(120,10),(120,30),(120,50)]
```



Мы видим циклический характер поведения системы. Сначала число хищников даже немного увеличивается, но количество жертв (пищи хищников) быстро уменьшается. Скоро хищникам перестаёт хватать жертв, и хищники начинают вымирать. В результате количество жертв начинает увеличиваться, и через какое-то время это приводит к росту числа хищников. При данном наборе коэффициентов система возвращается к прежнему состоянию, причем, чем больше хищников было вначале (при одном и том же числе жертв), тем больше времени требуется для восстановления системы.

Другой характер решения мы видим при следующих значениях коэффициентов:

```
A=0.1; C=0.04
B=0.2; D=0.01
pos=[(220,10),(220,30),(220,50)]
```

В этом случае за относительно небольшой промежуток времени хищники съедают всех жертв, а затем сами умирают от голода.

Для математиков представляет интерес поведение системы дифференциальных уравнений вблизи так называемых особых точек. В этих точках правые части системы обращаются в 0. Перечислим некоторые типы особых точек.

1) Особая точка – центр. Этот вариант иллюстрируется уравнением малых отклонений математического маятника. Вблизи такой точки фазовые кривые представляются собой окружности или эллипсы.

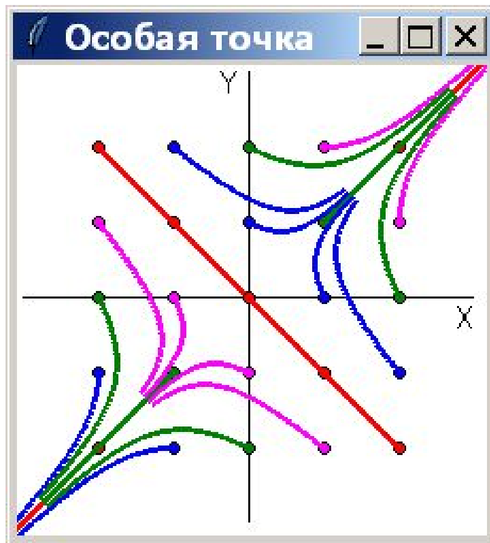
2) Особая точка – фокус. В этом случае фазовые кривые или стремятся к особой точке (устойчивый фокус) или «убегают» от неё (неустойчивый фокус).

```
from drawers import *
root=create_win('Особая точка',250,250)
def func(x,y):
   return [x+4*y, -4*x+y]
# -------- main -------
pos=[(10,10), (-10,10), (10,-10), (-10,-10)]
col=['red','blue','green','magenta']
n=len(pos)
k=len(col)
L=Drawers()
for i in range(n):
   w=Drawer(color=col[i%k],width=3)
   L.addDrawer(w)
L.axes(125,125)
L.startPos(pos,col)
while L.t<2:
   pos=L.nextPos(pos, func)
   L.lineto(pos)
   root.update()
```

Особая точка – фокус

3) Особая точка – седло. Заменим функцию func и зададим другие начальные точки.

```
from drawers import *
root=create_win('Особая точка',250,250)
def func(x,y):
   return [-x+2*y, 2*x-y]
# -------- main -------
pos=[]
for x in range(-80,81,40):
   for y in range(-80,81,40):
      pos.append((x,y))
col=['red','blue','green','magenta']
n=len(pos)
k=len(col)
L=Drawers()
for i in range(n):
   w=Drawer(color=col[i%k],width=3)
   L.addDrawer(w)
L.axes(125,125)
L.startPos(pos,col)
while L.t<1:
   pos=L.nextPos(pos, func)
   L.lineto(pos)
   root.update()
```



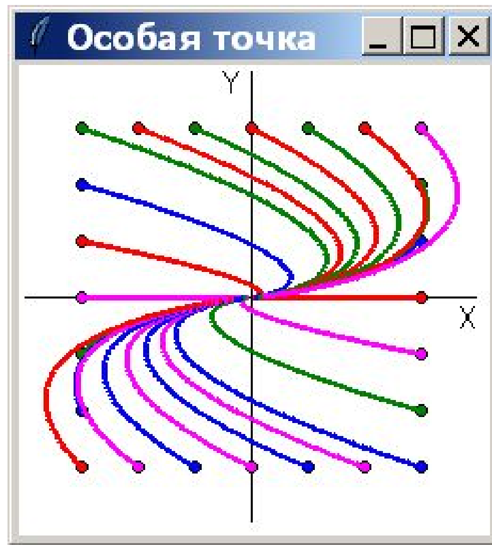
Особая точка – седло

Средствами высшей математики можно показать, что фазовые траектории в данном примере «прижимаются» к прямым x+y=0 и x – y=0.

4) Особая точка - узел

```
from drawers import *
from math import *
root=create_win('Особая точка',250,250)
def func(x,y):
   return [-x+2*y, -y]
# -------- main -------
pos=[]
for x in range(-90,91,30):
   for y in range(-90,91,30):
      if x==-90 or x==90 or y==-90 or y==90:
         pos.append((x,y))
col=['red','blue','green','magenta']
n=len(pos)
k=len(col)
L=Drawers()
for i in range(n):
   w=Drawer(color=col[i%k],width=3)
   L.addDrawer(w)
L.axes(125,125)
L.startPos(pos,col)
while L.t<5:
   pos=L.nextPos(pos, func)
   L.lineto(pos)
   root.update()
```

Результат



Особая точка – узел

14. Рекурсия в графике

Во всех современных языках программирования можно использовать рекурсивные процедуры и функции. Рекурсивной называется процедура/функция, которая вызывает сама себя. При каждом вызове создаётся новая локальная среда, которая содержит значения параметров функции и её локальные переменные. При возврате из рекурсии происходит возврат в старую локальную среду, при этом восстанавливаются значения параметров и локальных переменных функции.

Основная проблема рекурсии — её потенциальная бесконечность. Чтобы избежать зацикливания программы, нужно при каждом вызове рекурсивной функции передавать ей некоторую дополнительную информацию, которая позволяет определить момент выхода из рекурсии.

Пример. Нарисуем «мишень» из кругов. Мишень изобразим в виде набора концентрических окружностей. Параметры мишени:

x,y – координаты центра мишени:

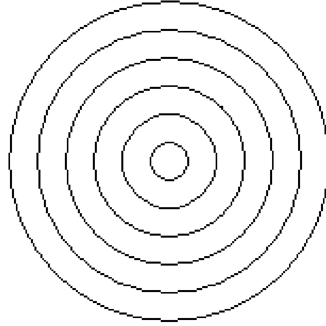
R – радиус внешней окружности;

dR – разность радиусов двух соседних окружностей.

```
from tkinter import *
root=Tk()
root.title('Узор')
canvas=Canvas(root,width=240,height=240,bg='white')
canvas.pack()
```

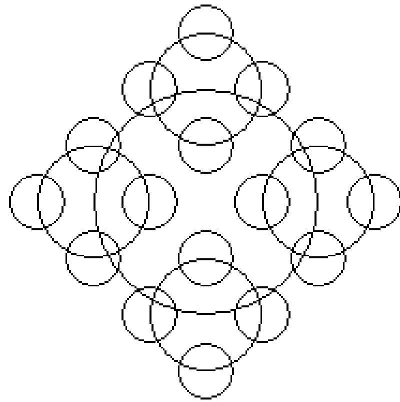
```
def target(x,y,R,dR):
   if R<=0: return
   canvas.create_oval(x-R,y-R,x+R,y+R,fill='')
   target(x,y,R-dR,dR)
target(120,120,85,15)
```

Результат:



Рекурсивный вызов функции target является последним оператором в функции target. Такая рекурсия называется хвостовой и легко заменяется циклом. Однако в более сложных случаях такая замена невозможна.

Рассмотрим следующий узор. Он образован большой окружностью и четырьмя нанизанными на неё фрагментами узора, причём каждый фрагмент устроен точно так же. Такая структура называется самоподобной и достаточно просто реализуется рекурсивной функцией.



```
def uzor(x,y,R):
   if R<=10: return
   canvas.create_oval(x-R,y-R,x+R,y+R,fill='')
   uzor(x-R,y,R//2)
   uzor(x+R,y,R//2)
   uzor(x,y-R,R//2)
```

```
    uzor(x,y+R,R//2)
uzor(120,120,60)
```

Часто для выхода из рекурсии вводят некоторый счётчик, который ограничивает глубину рекурсии. Перепишем функцию uzor в следующем виде:

```
def uzor(x,y,R,k):
    if k<0: return
    canvas.create_oval(x-R,y-R,x+R,y+R,fill='')
    uzor(x-R,y,R//2,k-1)
    uzor(x+R,y,R//2,k-1)
    uzor(x,y-R,R//2,k-1)
    uzor(x,y+R,R//2,k-1)
```

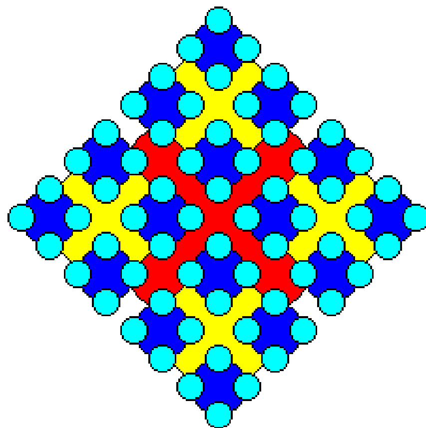
Тогда тот же рисунок получится, если ограничить глубину рекурсии значением 2:

```
uzor(120,120,60,2)
```

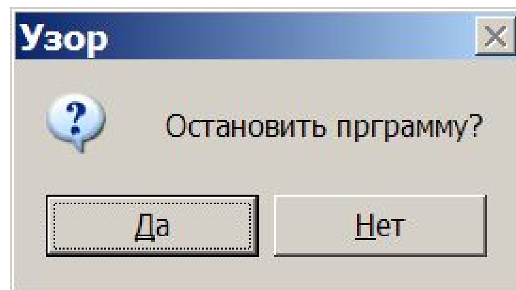
Увеличим глубину рекурсии до 3. Для наглядности на каждом уровне вместо окружностей будем рисовать цветные круги. Цвет круга будет зависеть от уровня рекурсии — параметра k.

```
colors=['cyan','blue','yellow','red']
def uzor(x,y,R,k):
    if k<0: return
    canvas.create_oval(x-R,y-R,x+R,y+R,fill=colors[k])
    uzor(x-R,y,R//2,k-1)
    uzor(x+R,y,R//2,k-1)
    uzor(x,y-R,R//2,k-1)
    uzor(x,y+R,R//2,k-1)
uzor(120,120,60,3)
```

Получим результат:



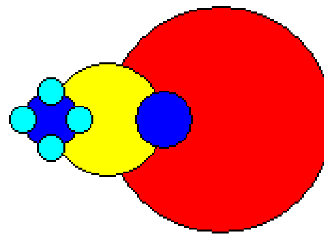
Чтобы увидеть «по шагам» процесс формирования узора используем диалоговое окно, которое будет появляться на экране после вывода каждого круга. Если пользователь нажмёт кнопку «Да», по программа завершит свою работу и на экране останется незаконченный узор. Если пользователь нажмёт кнопку «Нет», то будет выведен следующий круг и снова появится окно сообщения.



Чтобы реализовать этот механизм добавим в программу глобальную переменную flag. В главной программе flag = False и пока значение переменной не станет True рекурсивные вызовы будут выполняться. Как только flag получит значение True, будет осуществляться выход из каждого вызванного экземпляра функции. Заметим, что объявление flag глобальной переменной имеет принципиальное значение!

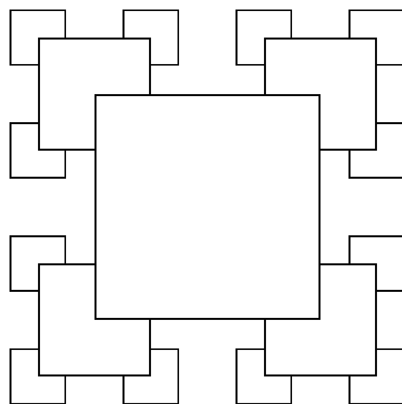
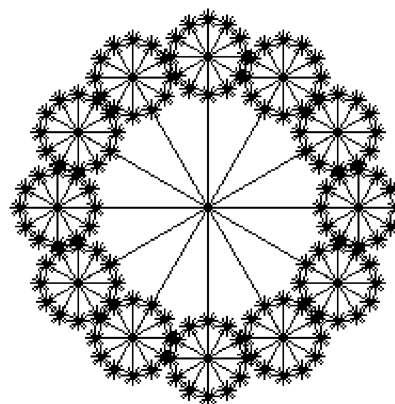
```
flag = False
def uzor(x,y,R,k):
   global flag
   if flag: return
   if k<0: return
   canvas.create_oval(x-R,y-R,x+R,y+R,fill=colors[k])
   yes=askyesno('Узор','Остановить прграмму?')
   if yes: flag=True
   uzor(x-R,y,R//2,k-1)
   uzor(x+R,y,R//2,k-1)
   uzor(x,y-R,R//2,k-1)
   uzor(x,y+R,R//2,k-1)
```

На следующем рисунке показана часть узора, которая иллюстрирует последовательность вызовов. На каждом уровне рекурсии сначала рисуется левая часть, потом правая часть узора, затем верхняя и нижняя части. Этот порядок обусловлен порядком рекурсивных вызовов в теле функции uzor.



Задания для самостоятельной работы.

1. Используя рекурсию напишите программы рисования следующих узоров:

2. Напишите рекурсивную программу, которая выводит на канву случайно расположенные круги случайного цвета и размера. На каждом уровне рекурсии выводится n кругов (n задаётся в программе) одинакового размера. Размер кругов при каждом рекурсивном вызове уменьшается. Реализуйте следующие варианты выхода из рекурсии:

а) при достижении определённого уровня рекурсии;

б) при достижении определённого размера кругов.

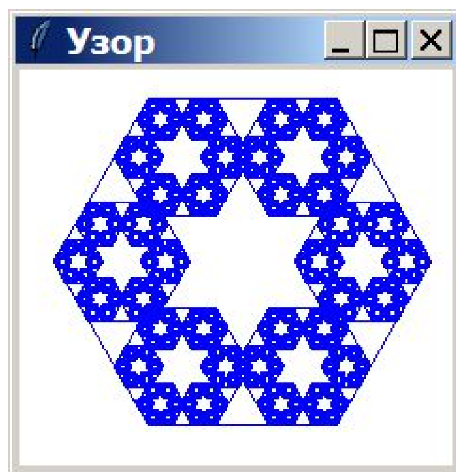
Напишем теперь рекурсивную функцию для черепашки. В основе узора лежит шестиугольник. Он будет нарисован на нулевом уровне рекурсии. С увеличением глубины рекурсии на каждой стороне шестиугольника будет нарисована уменьшенная копия узора с меньшей глубиной рекурсии.

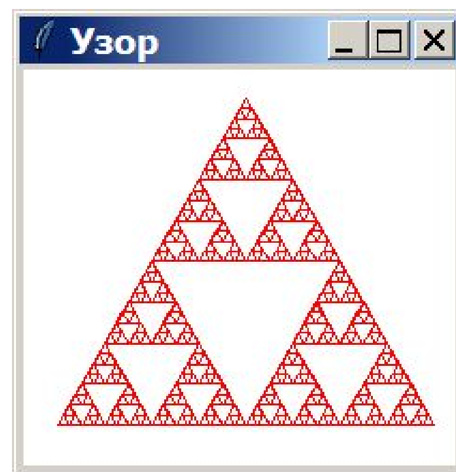
Приведём программу полностью.

```
from turtles import *
win=create_win('Узор',230,210,'white')

def uzor(A,len,k):
   if k<0: return
```

```
    for i in range(6):
        A.forw(len)
        A.left(60)
        uzor(A,0.36*len,k-1)
# -------- main -------
A=Turtle(angle=0,color='blue',width=1)
A.move(70,190)
uzor(A,100,5)
A.setState('hidden')
```

Результат (левый рисунок:

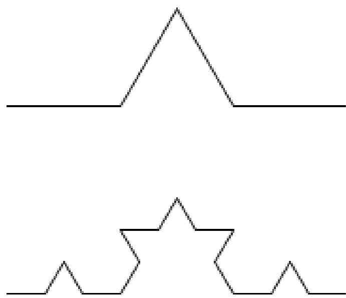




Задание. Модифицируйте программу, чтобы получить треугольник Серпинского. (правый рисунок)

Построение фрактальных кривых. Большой класс рекурсивных объектов составляют фрактальные кривые. Это тоже самоподобные объекты, так как каждый участок такой кривой в случае бесконечной рекурсии есть уменьшенная копия всей кривой. На практике глубина рекурсии всегда ограничена, поэтому это правило выполняется лишь приблизительно.

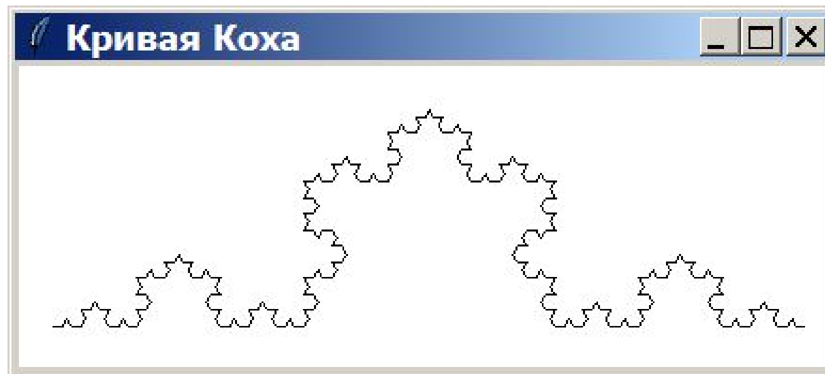
Начнём с рассмотрения кривых Коха. На нулевом шаге это просто отрезок прямой. На первом уровне рекурсии на средней трети отрезка строится излом, на каждом следующем на средней трети каждого отрезка строится подобный излом и т.д.



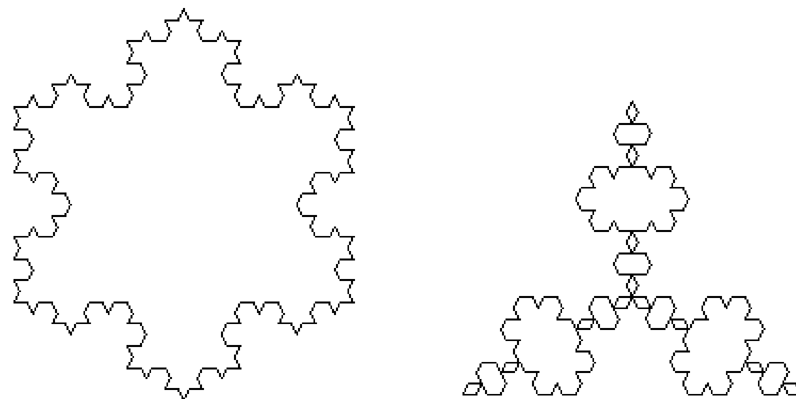
Поскольку процесс построения кривой Коха рекурсивный по своей природе, то и функция её построения будет рекурсивной. Параметры функции:

A – черепашка (экземпляр класса Turtle);
len – длина начального отрезка;
k – глубина рекурсии.

```
from turtles import *
win=create_win('Кривая Коха',430,160,'white')
def koch(A, len, k):
   if k==0:
      A.forw(len)
      return
   d = len/3
   koch(A,d,k-1)
   A.left(60)
   koch(A,d,k-1)
   A.right(120)
   koch(A,d,k-1)
   A.left(60)
   koch(A,d,k-1)
# -------- main -------
A=Turtle(angle=0,color='black',width=1)
A.move(20,140)
koch(A,400,4)
A.setState('hidden'
```



Расположив три кривые Коха на сторонах равностороннего треугольника мы получим снежинку Коха (слева) или треугольник Коха (справа):



Программный код снежинки Коха (основная часть):

```
A=Turtle(angle=0,color='black',width=1)
A.move(30,80)
for i in range(3):
   koch(A,180,3)
   A.right(120)
A.setState('hidden')
```

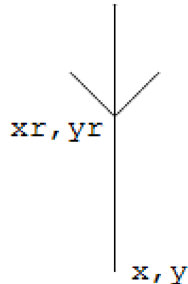
Программный код треугольника Коха (основная часть):

```
A=Turtle(angle=0,color='black',width=1)
A.move(30,180)
for i in range(3):
   koch(A,180,3)
   A.left(120)
A.setState('hidden')
```

Существует ещё много других фрактальных кривых. Их описание можно найти в литературе.

Моделирование древовидных структур. Рекурсия часто используется для построения ветвящихся узоров, напоминающих соцветия трав или кроны деревьев. При этом каждая веточка получается подобной всему «дереву».

Рассмотрим такую «веточку»



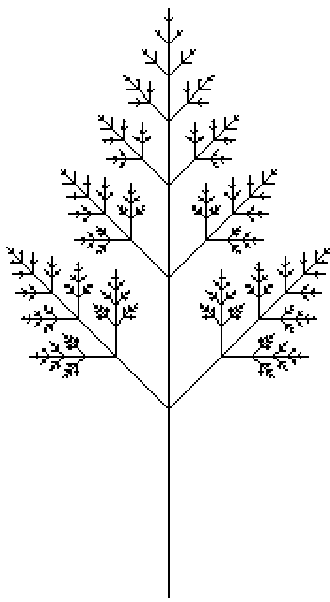
Основа рисунка – отрезок прямой, длиной d проведённый из точки x,y в общем случае под углом u к горизонту. Угол u отсчитывается против часовой стрелки. В этом случае координаты конечной точки вычисляются по формуле

```
(xr,yr) = ps2ds((x,y),(d,u))
```

Точка `xr,yr` служит начальной точкой для построения трёх веточек. Две из них растут влево и вправо от "ствола" дерева, а третья растёт вверх. Зададим коэффициенты уменьшения: 0.7 для центральной веточки и 0.4 для боковых. В теле функции нарисуем основной отрезок и сделаем три рекурсивных вызова с нужными нам параметрами.

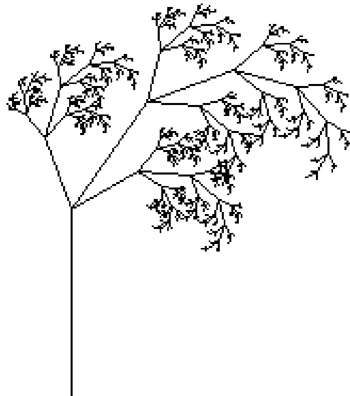
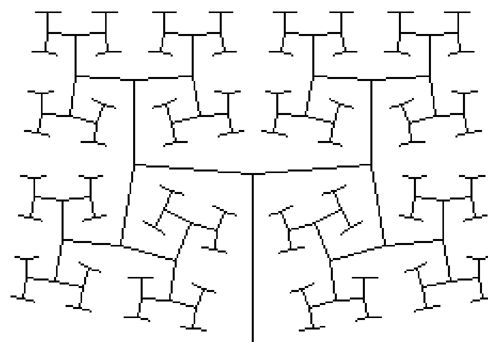
```
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Соцветие')
canvas=Canvas(root,width=200,height=350,bg='white')
canvas.pack()

def tree(x,y,d,u,k):
   if k<0: return
   (xr, yr) = ps2ds((x,y),(d,u))
   canvas.create_line(x,y,xr,yr)
   tree(xr,yr,0.7*d,u,k-1)
   tree(xr,yr,0.4*d,u-45,k-1)
   tree(xr,yr,0.4*d,u+45,k-1)

tree(100,340,100,90,7)
```

Результат на левом рисунке, а на правом — соцветие травы полевицы.



Изменим в функции tree углы наклона ветвей и сразу на рисунке появится ветер.

```
def tree(x,y,d,u,k):
   if k<0: return
   (xr, yr) = ps2ds((x,y),(d,u))
   canvas.create_line(x,y,xr,yr)
   tree(xr,yr,0.7*d,u-35,k-1)
   tree(xr,yr,0.4*d,u-60,k-1)
   tree(xr,yr,0.4*d,u+20,k-1)
```

Модифицируя функцию можно получить необычные узоры. На правом рисунке мы видим 16 пляшущих букв Н. Код программы:

```
from geom_sys import *
from tkinter import *
```

```
root=Tk()
root.title('16 букв H')
canvas=Canvas(root,width=300,height=250,bg='white')
canvas.pack()

def tree(x,y,d,u,k):
   if k<0: return
   (xr, yr) = ps2ds((x,y),(d,u))
   canvas.create_line(x,y,xr,yr,width=1)
   tree(xr,yr,0.7*d,u-85,k-1)
   tree(xr,yr,0.7*d,u+85,k-1)

tree(150,220,90,90,7)
```

15. Рисование мышкой

Рисовать стрелками довольно утомительно. Заменим клавиатуру мышкой. Воспользуемся тем, что при движении мышки через равные промежутки времени генерируется событие <Motion>. Если при этом нажата левая клавиша мыши, то генерируется событие <B1-Motion>, если правая клавиша — событие <B3-Motion>. Событие <B2-Motion> соответствует средней клавише, которую в современных мышах заменяет колёсико.

В программе отслеживаются два события: нажатие на левую клавишу мыши (начало рисования), и движение мыши с нажатой клавишей. В обработчике первого события мы должны запомнить точку, в которой произошло это событие:

```
def start(event):
   global x,y
```

```
    x=event.x
    y=event.y
```

При изменении положения курсора мыши будут изменяться значения переменных x, y, поэтому координаты этой точки сделаем глобальными переменными. В обработчике второго события эти координаты соответствуют предыдущему положению курсора мыши. Соединяя текущее и предыдущее положение курсора мыши получим траекторию движения курсора.

```
def motion(event):
    global x,y
    x1=event.x
    y1=event.y
    canvas.create_line(x,y,x1,y1)
    x=x1; y=y1
```

При отпускании клавиши мыши построение кривой заканчивается. Теперь можно переместить курсор мышки без рисования. Это позволяет получить рисунок из нескольких кривых линий.

Осталось написать «шапку» программы и установить связи между событиями и их обработчиками:

```
# рисование мышкой
from tkinter import *
root=Tk()
root.title('Рисунок')
canvas=Canvas(root,width=200,height=200,bg='white')
canvas.pack()
# обработчики событий
...
canvas.bind('<ButtonPress-1>',start)
canvas.bind('<B1-Motion>',motion)
```

Другой вариант простейшего графического редактора — рисовать разноцветными «пятнами». Каждое пятно представляет собой закрашенный полигон. Мышкой мы будем рисовать контур полигона, а закрашивать его в момент отпускания левой кнопки мыши. Цвет закраски в обработчике события можно выбирать случайным образом или с помощью диалога выбора цвета.

Реализуем сначала более простой способ закраски — закраску случайным цветом. Напишем функцию rndcolor() и обработчики событий:

```
def rndcolor():
    r=randint(0,255)
    g=randint(0,255)
    b=randint(0,255)
```

```
    return '#{0:02x}{1:02x}{2:02x}'.format(r,g,b)

# <ButtonPress-1>
def start(event):
    global L
    # запоминаем в списке начальную точку кривой
    L=[(event.x,event.y)]

# <B1-Motion>
def motion(event):
    global L
    # добавляем в список текущее положение курсора мыши
    L.append((event.x,event.y))
    # соединяем две последние точки списка
    canvas.create_line(L[-2],L[-1])

# <ButtonRelease-1>
def finish(event):
    global L
    # соединяем первую и последнюю точки списка
    canvas.create_line(L[1],L[-1])
    # создаём полигон случайного цвета
    color=rndcolor()
    canvas.create_polygon(L,fill=color,outline='black')
```

Связываем события и их обработчики:

```
canvas.bind('<ButtonPress-1>',start)
canvas.bind('<B1-Motion>',motion)
canvas.bind('<ButtonRelease-1>',finish)
```

Заменим в обработчике события <ButtonRelease-1> вызов функции rndcolor() на вызов диалогового окна выбора цвета:

```
    _,color=colorchooser.askcolor(title='Выбор цвета')
```

Метод askcolor(...) возвращает два значения: во-первых, кортеж компонентов RGB-цвета, во-вторых, текстовую константу цвета в шестнадцатеричном виде. Первое значение нам не понадобится, поэтому его можно записать в анонимную переменную, представляемую символом подчеркивания. Заметим, что фактически символ подчёркивания и есть имя переменной, т.е. его можно использовать так же, как и обычную переменную. Поэтому использование символа подчеркивания есть просто удобный способ указать, что значение этой переменной далее не используется.

Второе замечание касается применения имени colorchooser для доступа к методу askcolor. Если его убрать, то нам нужно будет добавить импорт метода askcolor в «шапку» программы:

```
from tkinter.colorchooser import askcolor
```

Запустив модифицированную программу мы обнаружим, что появляющееся окно выбора цвета полностью закрывает рисунок. В результате пользователь будет вынужден «вслепую» выбирать цвет закрашивания очередного элемента. Хотелось бы разместить диалог выбора цвета рядом с окном приложения. Для этого нам придётся создать новое окно, разместить его справа от окна приложения и «прикрепить» диалог выбора цвета к новому окну.

Новое окно — это экземпляр класса Toplevel, который практически идентичен окну приложения. Поэтому перечисленные ниже методы будут применимы к окнам обоих типов.

Метод geometry задаёт размеры и положение окна на экране. Метод получает текстовую строку вида

```
'ширина_окнаxвысота_окна+x+y'
```

где x,y – координаты левого верхнего угла окна относительно левого верхнего угла экрана монитора.

Заметим, что любой знак «плюс» можно заменить на знак «минус». Если методу передать строку вида

```
'ширина_окнаxвысота_окна-x-y'
```

то значения -x,-y будут рассматриваться как координаты правого нижнего угла окна приложения в системе координат с началом в правом нижнем углу экрана монитора. Комбинации +x-y и -x+y соответствуют двум оставшимся углам приложения и экрана.

Заметим, что можно задать только часть данных: размеры окна или его расположение.

Метод resizable(...) позволяет разрешить (поведение по умолчанию) или запретить изменение окна пользователем по ширине или по высоте или по обоим этим параметрам. Например,

```
root.resizable(False, False)
```

запрещает любые изменения размеров окна. В этом случае становится неактивной и кнопка разворачивания окна.

Метод title(строка) задаёт строку — заголовок окна. Остальное оформление заголовка определяется операционной системой.

Метод withdraw() позволяет скрыть окно. Чтобы вернуть спрятанное окно на экран нужно воспользоваться методом deiconify() или iconify().

Иногда необходимо получить окно без заголовка, в этом случае вызывается метод окна overrideredirect с параметром True. Закрыть такое окно можно только комбинацией клавиш Alt+F4.

Кроме перечисленных можно использовать и методы, общие для всех элементов управления:

w.update() — принудительное обновление w;
w.focus_set() — передача фокуса элементу w;
w.destroy() — удаление элемента w;
и многие другие.

Приведём полный текст программы с необходимыми комментариями.

```
# рисование мышкой
from tkinter import *
root=Tk()
root.title('Рисунок')
root.geometry('+100+100')
root.resizable(False,False)
canvas=Canvas(root,width=200,height=200,bg='white')
canvas.pack()
# создаём новое окно нулевого размера
# и размещаем его справа от окна приложения
w=Toplevel()
w.geometry('0x0+310+100')
# убираем заголовок нового окна
w.overrideredirect(True)
# передаём фокус окну приложения
root.focus_set()

def start(event):
   global L
   # запоминаем в списке начальную точку кривой
   L=[(event.x,event.y)]

def motion(event):
   global L
   # добавляем в список текущее положение курсора мыши
   L.append((event.x,event.y))
   # соединяем две последние точки списка
   canvas.create_line(L[-2],L[-1])

def finish(event):
   global L
   # соединяем первую и последнюю точки списка
   canvas.create_line(L[1],L[-1])
   # выбираем цвет полигона
   _,color=colorchooser.askcolor(parent=w,
                         title='Выбор цвета')
```

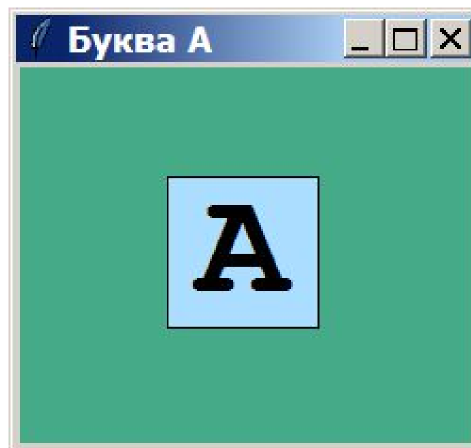
```
        root.focus_set()
        # создаём полигон выбранного цвета
        canvas.create_polygon(L,fill=color,outline='black')

    # связываем события и их обработчики:
    canvas.bind('<ButtonPress-1>',start)
    canvas.bind('<B1-Motion>',motion)
    canvas.bind('<ButtonRelease-1>',finish)
    root.mainloop()
```

Полужирным шрифтом в теле программы выделен именованный параметр parent=w, который и прикрепляет стандартный диалог выбора цвета к окну w.

На рисунке ниже показан результат работы программы и автора этой книги.

16. Учебный проект «Буква А».

Поместим в окно приложения холст, на который выведем квадрат и латинскую букву А.



Приложение должно реагировать на действия пользователя. При нажатии стрелок влево, вправо, вверх или вниз квадрат с буквой А перемещается в заданном направлении на 10 пикселей. При щелчке левой кнопкой мыши на квадрате будет случайным образом меняться цвет квадрата. Аналогично при щелчке на канве меняться будет цвет канвы.

```
from tkinter import *
root=Tk()
root.title('Буква A')
canvas=Canvas(root,width=240,height=200,bg='#44aa88')
canvas.pack()
# создаём шрифт, квадрат и текст
ds=60
xfont=font.Font(family='Courier New',
                size=ds, weight='bold')
rect=canvas.create_rectangle(80,60,160,140,tag='A',
                             fill='#aaddff')
text=canvas.create_text(120,100,text='A',
                             font=xfont,tag='A')
```

Гарнитуру шрифта, размер и начертание буквы А зададим с помощью конструктора Font объекта font модуля tkinter. Конструктор Font получает набор именованных параметров. Основными являются:

family – гарнитура шрифта. Гарнитура определяет семейство шрифтов одинакового вида, отличающихся размером и начертанием;

size – размер шрифта. Если size>0, то размер шрифта задан в пунктах, если size<0, то размер шрифта задан в пикселях и равен абсолютной величине параметра. Следует понимать, что и в том и в другом случае size определяет размер области, в которой будет размещаться символ, а не размер самого символа;

weight – толщина линий символов. Варианты значений:

weight='normal' — нормальная толщина;

weight='bold' — увеличенная толщина («полужирный» шрифт);

slant – наклон символов. Варианты:

slant='roman' — прямой шрифт;

slant='italic' — наклонный шрифт («курсив»).

Квадрат и буква получают индивидуальные идентификаторы rect и text, и общий тэг 'A'. Это позволит нам иметь доступ как к отдельным частям, так и ко всему составному объекту квадрат+буква.

Напишем обработчик событий клавиатуры.

```
def reaction(event):
    key=event.keysym
```

```
    if key=='Left':  canvas.move('A',-10,0)
    if key=='Right': canvas.move('A', 10,0)
    if key=='Up':    canvas.move('A',0,-10)
    if key=='Down':  canvas.move('A',0, 10)
```

Метод move холста перемещает группу объектов с тэгом 'A' на заданный вектор при нажатии на одну из клавиш управления курсором.

Теперь рассмотрим событие мыши — щелчок левой кнопкой. Примем, что реакция на это событие — изменение цвета квадрата или фона окна в зависимости от точки щелчка. Цвет будем меняться случайным образом. Для случайного выбора цвета используем написанную ранее функцию rndcolor().

В обработчике события мыши проверим попадание точки щелчка в квадрат. Для этого получим координаты двух вершин, которые его задают, с помощью метода холста coords.

```
def change(event):
    x=event.x
    y=event.y
    color=rndcolor()
    (x1,y1,x2,y2)=canvas.coords(rect)
    if x1<x<x2 and y1<y<y2:
       canvas.itemconfig(rect, fill=color)
    else:
       canvas['bg']=color
```

Изменить цвет объекта можно, задав новое значение параметрам fill для квадрата, background для холста. Для изменения цвета квадрата мы использовали метод itemconfig, для изменения цвета фона — альтернативный вариант — непосредственное задание нового значения соответствующему элементу словаря параметров объекта.

Связываем объект и событие с его обработчиком.

```
root.bind('<Key>',reaction)
canvas.bind('<Button-1>',change)
root.mainloop()
```

Заметим, что события клавиатуры получает окно (объект root), а события мыши — холст (объект canvas).

Итак, мы можем перемещать клавишами клавиатуры квадрат с нарисованной на нём буквой А и менять цвета щелчком кнопки мыши. Что можно к этому добавить? Перечислим возможные варианты:

- изменить букву;
- изменить начертание буквы;
- изменить размер буквы (вместе с размером квадрата).

Для изменения буквы задействуем клавиши клавиатуры, для изменения начертания добавим в приложение меню, для изменения размера используем командные кнопки.

Как изменить букву? Понятно, что перечислить все клавиши в обработчике событий невозможно, но можно определить, что нажата клавиши с буквой или цифрой с помощью методов строки isalpha() и isdigit(). Добавим в обработчик событий клавиатуры строки

```
if key.isalpha() or key.isdigit()::
   canvas.itemconfig(text, text=key)
```

и протестируем программу.

Пока мы нажимаем латинские буквы или цифры, программа работает правильно. Однако при нажатии клавиш со служебными знаками, мы увидим на экране их названия, а это не то, что нам нужно.

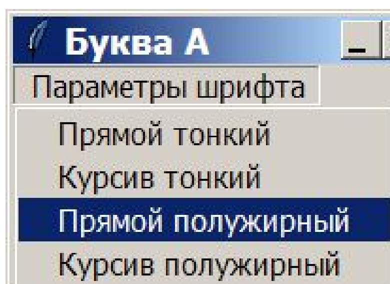
Используем для проверки ввода буквы не символы, а коды клавиш, которые должны попадать в диапазон от 65 до 90 включительно (на этих клавишах находятся латинские буквы), а для определения цифр оставим метод isdigit().

```
code=event.keycode
if 65<=code<=90 or key.isdigit():
   canvas.itemconfig(text, text=key)
```

Убедимся, что всё работает. Заметим, что один и тот же код клавиши соответствует и строчной и прописной букве, т.е. код клавиши не зависит от того, нажата ли клавиша Shift или включен режим CapsLock.

17. Меню. Элементы управления

Изменяем начертание символов. Добавим простое меню в наше приложение. Сделаем его раскрывающимся (иногда называют его выпадающим).



В основное меню поместим пункт «Параметры шрифта», при выборе которого будет появляться подменю с четырьмя пунктами, задающими

конкретные параметры шрифта. При щелчке на выбранном пункте должна вызываться функция-обработчик события. Поэтому сначала напишем все четыре обработчика.

```
def normal():
   xfont.config(weight='normal', slant='roman')
def italic():
   xfont.config(weight='normal', slant='italic')
def bold():
   xfont.config(weight='bold', slant='roman')
def bolditalic():
   xfont.config(weight='bold', slant='italic')
```

Каждый из них использует метод config для изменения параметров weight и slant созданного ранее шрифта xfont.

Теперь создадим само меню. Основное меню «привязываем» к приложению:

```
menubar = Menu(root)
root.config(menu=menubar)
```

Подменю «привязываем» к основному меню:

```
submenu = Menu(menubar, tearoff=0)
```

Значение 0 параметра tearoff (в переводе с английского — отрывать) запрещает переносить подменю на другое место, т.е. отрывать его от основного меню.

Создадим теперь пункты подменю с помощью метода add_command. Метод получает два именованных параметра: label – текст пункта меню и command – имя функции-обработчика.

```
submenu.add_command(label='Прямой тонкий',command=normal)
submenu.add_command(label='Курсив тонкий',command=italic)
submenu.add_command(label='Прямой
полужирный',command=bold)
submenu.add_command(label='Курсив полужирный',
                    command=bolditalic)
```

Вместо текста пункт меню может содержать рисунок, тогда вместо label нужно задать значение параметру image или bitmap.

Последний шаг — добавляем созданное подменю к основному меню:

```
menubar.add_cascade(label='Параметры шрифта',
menu=submenu)
```

Изменяем размер квадрата и символа. Создадим две командные кнопки и разместим их друг возле друга, но после канвы. Менеджер размещения pack в этом случае получает именованные параметры side (указывает сторону окна,

к которой «прижимается» элемент управления) и padx (отступ от края по оси x).

На самих кнопках напишем текст «Увеличить» и «Уменьшить» и свяжем их с функциями increase() и dscrease(). Заметим, что параметр width задаёт ширину кнопок «в символах».

```
Inc=Button(root, width=10, text='Увеличить',
                           command=increase)
Dec=Button(root, width=10, text='Уменьшить',
                           command=decrease)
Inc.pack(side='left', padx=10)
Dec.pack(side='right', padx=10)
```

Приведём текст функций, вызываемых при нажатии кнопок (но в программе эти функции должны располагаться до создания командных кнопок!),

Определяем функцию-обработчик для кнопки «Увеличить».

```
def increase():
   # объявляем размер шрифта глобальной переменной
   # и увеличиваем его на 12 пунктов
   global ds; ds+=12
   xfont.config(size=ds)
   # включаем кнопку «Уменьшить»
   Dec['state']='normal'
   # определяем текущие координаты квадрата
   (x1,y1,x2,y2)=canvas.coords(rect)
   # увеличиваем квадрат
   canvas.coords(rect,x1-10,y1-10,x2+10,y2+10)
   # и символ
   canvas.itemconfig(text, font=xfont)
   # при достижении максимального размера
   if ds>=120:
      # отключаем кнопку «Увеличить»
      Inc['state']='disabled'
```

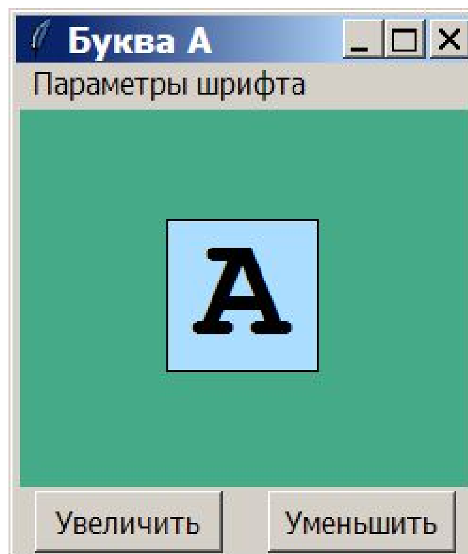
Аналогично определяем функцию-обработчик для кнопки «Уменьшить».

```
def decrease():
   # объявляем размер шрифта глобальной переменной
   # и уменьшаем его на 12 пунктов
   global ds
   ds-=12
   xfont.config(size=ds)
   # включаем кнопку «Увеличить»
   Inc['state']='normal'
   # определяем текущие координаты квадрата
```

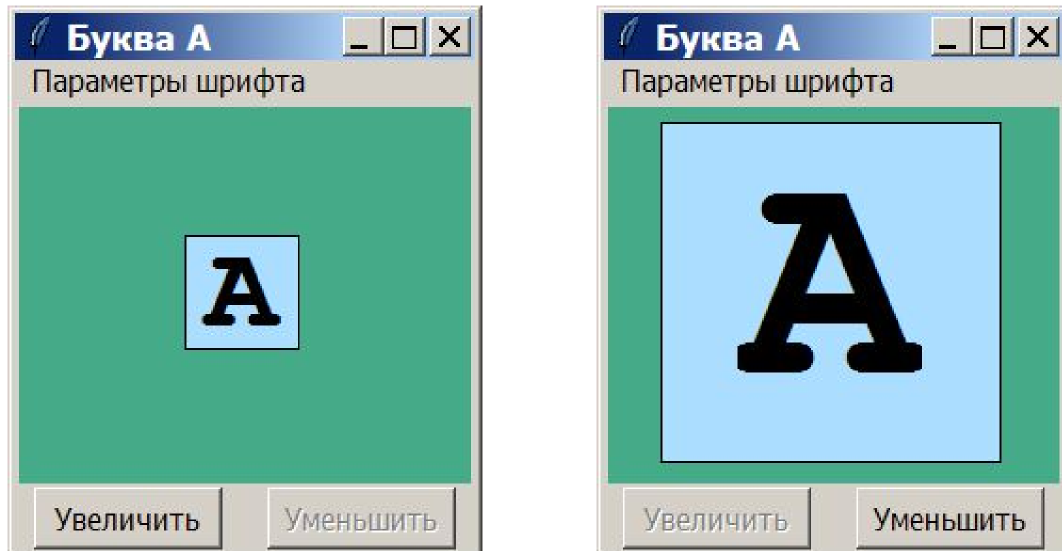
```
(x1,y1,x2,y2)=canvas.coords(rect)
# уменьшаем квадрат
canvas.coords(rect,x1+10,y1+10,x2-10,y2-10)
# и символ
canvas.itemconfig(text,font=xfont)
# при достижении минимального размера
if ds<=48:
   # отключаем кнопку «Уменьшить»
   Dec['state']='disabled'
```

Параметр state определяет, является ли кнопка активной, т.е. воспринимает ли она действия пользователя. Для активной кнопки этот параметр имеет значение 'normal', для неактивной кнопки — 'disabled' . При отключении кнопки меняется её внешний вид и она перестаёт нажиматься.

Внешний вид работающего приложения показан ниже.



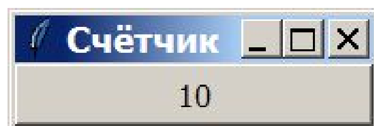
Предельные случаи: слева квадрат с буквой имеет минимальный размер и кнопка «Уменьшить» неактивна, справа — квадрат с буквой имеет максимальный размер и неактивна кнопка «Увеличить».



Элемент управления «Кнопка». В предыдущем разделе мы познакомились с двумя элементами управления — меню и командная кнопка, причём в наиболее стандартном варианте. На самом деле каждый элемент управления имеет гораздо больше параметров, позволяющих задать внешний вид и функциональность элемента.

Рассмотрим более подробно командную кнопку.

Пример. Счётчик нажатий. Разместим в окне кнопку, большинство параметров которой заданы по умолчанию. Первоначально на кнопку помещается значение 0. При каждом нажатии на кнопку число на кнопке будет увеличиваться на единицу.



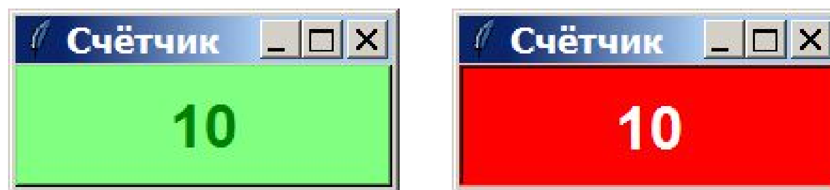

```
from tkinter import *
root=Tk()
root.title('Счётчик')

k=0
def counter():
   global k
   k+=1
   button.config(text=str(k))
   # или button['text']=str(k)
button=Button(root, width=20, text=str(k),
command=counter)
```

```
button.pack()
```

Зададим другие значения следующим параметрам кнопки.
bg или background – цвет фона кнопки;
fg или foreground – цвет переднего плана (цвет текста) кнопки;
font=(гарнитура, размер, начертание) – шрифт текста.

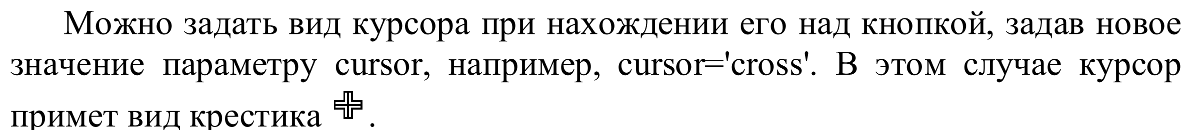
```
button=Button(root, width=10, text=str(k),
              bg='#80ff80', fg='green',
              font=('Arial',24,'bold'),
              command=counter)
```

Результат приведен на рисунке слева.



Можно также изменить цвет нажатой кнопки. Для этого следует задать значения параметрам activebackground — цвет фона нажатой кнопки, и activeforeground — цвет переднего плана (текста) нажатой кнопки. На рисунке показано состояние нажатой кнопки при значениях этих параметров

```
              activebackground='red',
              activeforeground='white',
```

Можно задать вид курсора при нахождении его над кнопкой, задав новое значение параметру cursor, например, cursor='cross'. В этом случае курсор примет вид крестика .

Следует, однако, понимать, что чаще всего элементы управления, в том числе и командные кнопки, используются с параметрами по умолчанию.

18. Игра «Таблица чисел»

Идея игры: на экране появляется таблица, в которой в случайном порядке размещены натуральные числа без повторений. Пользователь должен последовательно отметить мышкой все числа в порядке возрастания.

Реализацию идеи начнём с самого простого варианта — таблицы фиксированного размера. Определим переменные:

row – число строк в таблице, row=3;
col – число столбцов в таблице, col=5.
nums – список чисел;
btns – список кнопок;

Сформируем список чисел. Вариант А: в список помещаем натуральные числа от 1 до row*col, а затем «перемешиваем» их:

```
nums=[]; w=0
for i in range(row*col):
   w=w+1
   nums.append(w)
# меняем случайным образом порядок чисел
shuffle(nums)
```

Вариант Б: в список помещаем случайные, не обязательно последовательные натуральные числа. Чтобы среди них не было одинаковых каждое следующее число «отодвигаем» от предыдущего на случайную величину из небольшого диапазона, например, из диапазона 2..7, а затем «перемешиваем» их:

```
nums=[]; w=0
for i in range(row*col):
   w=w+randint(2,7)
   nums.append(w)
# меняем случайным образом порядок чисел
shuffle(nums)
```

Пояснения.

1. Метод списка append(значение) добавляет в конец списка указанное значение.

2. Функция shuffle(список) «перемешивает», т.е. меняет местами значения списка. В результате получается случайная последовательность. Чтобы использовать эту функцию, нужно подключить модуль random.

Итак, список значений подготовлен. Теперь создадим список командных кнопок и разместим их в окне приложения. Используем менеджер размещения grid для формирования прямоугольной таблицы объектов.

Синтаксис вызова (указаны только основные параметры):

объект.grid(row=номер_строки, column=номер_столбца, ...)

Нумерация строк и столбцов начинается с нуля.

```
btns=[]
for i in range(row*col):
   b=Button(root, width=3, height=1,
            text=str(nums[i]),
            font=('Georgia',18,'bold'),
            command=lambda x=i: hide(x))
   btns.append(b)
   b.grid(row=i//col, column=i%col)
```

Пояснения.

1. Параметры кнопки:

width – ширина кнопки в символах;

height – высота кнопки в символах;

text – текст на кнопке, в данном случае текст есть очередное число из списка nums;

font – шрифт, которым выводится текст. Параметр представляет собой кортеж, содержащий гарнитуру, размер и начертание символов текста на кнопке;

command – ссылка на функцию-обработчик события «щелчок левой кнопкой мыши». В этом примере мы первый раз сталкиваемся с анонимной функцией или лямбда-функцией.

Синтаксис объявления лямбда-функции:

```
lambda <список аргументов> <выражение>
```

Список аргументов по своим возможностям не отличается от списка аргументов обычной функции, но тело лямбда-функции состоит только из одного выражения. Лямбда-функции чаще всего используются, если нужно передать некоторую функцию в другую функцию. Так в данном примере в конструктор командной кнопки передаётся функция-обработчик события в виде лямбда-функции.

Функция-обработчик убирает число с правильно нажатой кнопки, следовательно она (функция) должна знать, какая кнопка нажата, т.е. получать её индекс в списке btns. Но если мы напишем command=hide(i), то получим сообщение об ошибке, так как hide(i) есть результат вызова функции hide с параметром i. Наличие ключевого слова lambda сообщает интерпретатору, что мы создаём функцию с параметром x, имеющим по умолчанию значение i. Именно эта функция будет вызываться при щелчке левой кнопкой мыши.

В функции-обработчике нужно проверить, нажата ли «правильная» кнопка. Для этого определяем минимальное значение в списке nums и сравниваем его с числом на кнопке. Если эти значения совпадают, то мы стираем это значение и делаем кнопку неактивной. Само же число удаляем из списка nums.

```
    w=min(nums)
    if btns[x]['text']==str(w):
       btns[x]['text']=''
       btns[x]['state']='disable'
       nums.remove(w)
```

Заметим, что возможность передать значение функции-обработчику позволяет нам иметь одну функцию для всех кнопок!

Итак соберём все фрагменты в одну программу:

```
from random import *
```

```
from tkinter import *
root=Tk()
root.title('Таблица чисел')

def hide(x):
   w=min(nums)
   if btns[x]['text']==str(w):
      btns[x]['text']=''
      btns[x]['state']='disable'
      nums.remove(w)

row=3; col=5
nums=[]; w=0
for i in range(row*col):
   w=w+1
   nums.append(w)
# меняем случайным образом порядок чисел
shuffle(nums)

btns=[]
for i in range(row*col):
   b=Button(root, width=3, height=1,
            text=str(nums[i]),
            font=('Georgia',16,'bold'),
            command=lambda x=i: hide(x))
   btns.append(b)
   b.grid(row=i//col, column=i%col)
```

Ниже приведены два вида экрана: случайное начальное состояние всех кнопок (А) и вид таблицы во время игры (Б).

А) Начальное состояние кнопок

Б) Вид таблицы во время игры.

Более трудный (и более интересный) вариант игры получается, когда случайные числа генерируются произвольно, о чём уже упоминалось выше. Чтобы пользователь сразу получал подтверждение правильности своего

выбора будем в процессе игры подсвечивать нажатую клавишу светло-зелёным или розовым цветом.

Для этого перепишем функцию hide:

```
def hide(x):
    w=min(nums)
    if btns[x]['text']==str(w):
        btns[x]['text']=''
        btns[x]['state']='disable'
        btns[x]['bg']='lightgreen'
        nums.remove(w)
    else:
        btns[x]['bg']='pink'
```

На рисунке приведен пример неверного хода. Выбрано число 55, хотя следовало нажать на кнопку с числом 49, наименьшим из оставшихся в таблице чисел.

Займёмся дальнейшим усовершенствованием интерфейса игры.

Рассмотрим следующие возможности:

- изменение размеров игрового поля;
- подсчёт числа ошибочных нажатий;
- сообщение об окончании игры;
- изменение алгоритма выбора кнопки — минимальное или максимальное значение из оставшихся в таблице чисел;

Изменение размеров игрового поля сделаем с помощью меню.

Добавим в приложение меню с пунктом «Параметры», а в раскрывающееся подменю несколько вариантов размеров игрового поля. При выборе варианта вызывается функция-обработчик, поэтому нам придётся переделать часть программы, перенеся код создания игрового поля в функцию newgame(row, col).

Приведём сначала код меню:

```
menubar = Menu(root)
```

```
root.config(menu=menubar)
submenu = Menu(menubar, tearoff=0)
submenu.add_command(label='3 строки, 4 столбца',
            command=lambda n=3,m=4: newgame(n,m))
submenu.add_command(label='3 строки, 5 столбцов',
            command=lambda n=3,m=5: newgame(n,m))
submenu.add_command(label='4 строки, 6 столбцов',
            command=lambda n=4,m=6: newgame(n,m))
menubar.add_cascade(label='Параметры', menu=submenu)
```

Теперь займёмся функцией newgame. Поскольку создаваемые внутри этой функции переменные nums и btns должны быть доступны для изменения в теле функции hide, эти переменные и в функции newgame и в функции hide должны быть объявлены как глобальные.

Внесём необходимые изменения:

```
def hide(x):
   global nums, btns
   ...

def newgame(n, m):
   global nums, btns
   ...
```

При тестировании игры оказалось, что при переходе к меньшему размеру игровое поле не уменьшалось, так как на нём оставались кнопки предыдущего варианта, причём они уже не входили в список btns. Поэтому при нажатии на такую кнопку генерировалось сообщение об ошибке «индекс вне диапазона»:

```
IndexError: list index out of range
```

Очевидное решение — перед созданием нового варианта удалять уже существующие кнопки. Однако как убедиться, что эти кнопки уже были созданы? Воспользуемся тем, что все переменные Python хранит в виде пар 'имя' : значение. Совокупность таких пар образует так называемый словарь. Доступ к словарю локальных переменных реализует функция locals(), соответственно доступ к словарю глобальных переменных — функция globals().

Поэтому перед созданием новых кнопок проверяем наличие имени 'btns' в словаре глобальных переменных и при положительном результате удаляем все кнопки из списка 'btns' методом destroy():

```
   if 'btns' in globals():
      for b in btns: b.destroy()
```

Для подсчёта числа ошибочных нажатий введём новую переменную err. При создании новой игры обнулим эту переменную, а при завершении игры — выведем окно с сообщением о количестве ошибочных нажатий. В этом же окне предложим пользователю сделать выбор: начать новую игру или закрыть приложение.

Чтобы отследить завершение игры в переменной count будем хранить количество правильно выбранных кнопок с числами. В функции newgame инициализируем эту переменную значением 0, а при каждом успешном нажатии будем увеличивать её на единицу и сравнивать со значением row*col. Естественно, переменные count, err, а также размеры игрового поля row и col должны быть глобальными переменными.

Приведём сразу полный текст модифицированной программы.

```
from random import *
from tkinter import *
root=Tk()
root.title('Таблица чисел')

menubar = Menu(root)
root.config(menu=menubar)
submenu = Menu(menubar, tearoff=0)
submenu.add_command(label='3 строки, 4 столбца',
            command=lambda n=3,m=4: newgame(n,m))
submenu.add_command(label='3 строки, 5 столбцов',
            command=lambda n=3,m=5: newgame(n,m))
submenu.add_command(label='4 строки, 6 столбцов',
            command=lambda n=4,m=6: newgame(n,m))
menubar.add_cascade(label='Параметры', menu=submenu)

def hide(x):
   global nums, btns, count, err, row, col
   w=min(nums)
   if btns[x]['text']==str(w):
      btns[x]['text']=''
      btns[x]['state']='disable'
      btns[x]['bg']='lightgreen'
      nums.remove(w)
      count+=1
      if count==row*col:
         yes=messagebox.askyesno('Игра окончена',
      'Сделано '+str(err)+' ошибки(ок,а). Начать новую?')
         if yes: newgame(row,col)
         else: root.destroy()
   else:
      btns[x]['bg']='pink'
```

```
        err+=1

def newgame(n, m):
   global nums, btns, count, err, row, col
   # инициализация глобальных переменных
   row=n; col=m
   count=0; err=0
   nums=[]; w=0
   for i in range(row*col):
      w=w+randint(2,7)
      nums.append(w)
      # меняем случайным образом порядок чисел
      shuffle(nums)
   # удаление имеющихся кнопок
   if 'btns' in globals():
      for b in btns: b.destroy()
   # создание новых кнопок
   btns=[]
   for i in range(row*col):
      b=Button(root, width=3, height=1,
               text=str(nums[i]),
               font=('Georgia',16,'bold'),
               command=lambda x=i: hide(x))
      btns.append(b)
      b.grid(row=i//col, column=i%col)
# ------ главная программа ------
newgame(3,5)
root.mainloop()
```

Изменение алгоритма выбора кнопки. Собственно говоря, есть только два варианта выбора очередного числа: выбирать минимальное или максимальное значение из оставшихся в таблице чисел. Для выбора одного из нескольких вариантов удобно использовать меню, пункты которого функционируют как радиокнопки. Т.е. при выборе любого пункта меню отменяется предыдущий вариант.

Добавление пунктов-радиокнопок в меню происходит по той же схеме, что и обычных пунктов, однако вместо метода add_command используется метод add_radiobutton.

Основные параметры:

label='текст' – текст пункта меню;

variable=имя – переменная, связанная с элементом управления;

value=значение – значение, присваиваемое связанной переменной.

В одну группу объединяются те пункты меню, для которых указана одна и та же связанная переменная. А вот значения связанной переменной должны

быть разными для разных пунктов меню. При выборе пункта меню это значение передаётся переменной, связанной с этой группой.

В отличие от обычных переменных, связанные переменные создаются явно. Для целочисленных переменных используется конструктор IntVar(), для строковых переменных — StringVar(). При создании связанной переменной она по умолчанию равна нулю (для целочисленных переменных) или пустой строке (для строковых переменных).

При необходимости можно изменить значение связанной переменной методом set(значение), а получить его — методом get().

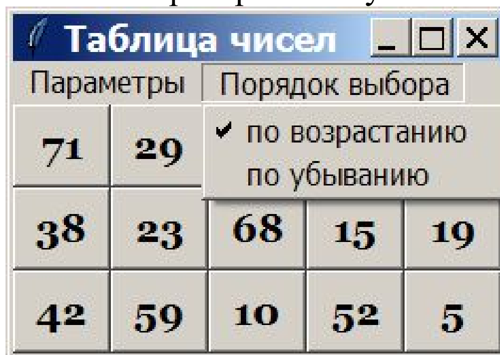
Добавим дополнительное меню из двух радиокнопок:

```
v=IntVar()   # по умолчанию 0
dopmenu = Menu(menubar, tearoff=0)
dopmenu.add_radiobutton(label='по возрастанию',
                        variable=v, value=0)
dopmenu.add_radiobutton(label='по убыванию',
                        variable=v, value=1)
menubar.add_cascade(label='Порядок выбора', menu=dopmenu)
```

Значение переменной v должно определять порядок выбора следующего числа в таблице. Добавим поэтому в функцию hide проверку значения этой переменной перед вычисление w:

```
    if v.get()==0: w=min(nums)
    else: w=max(nums)
```

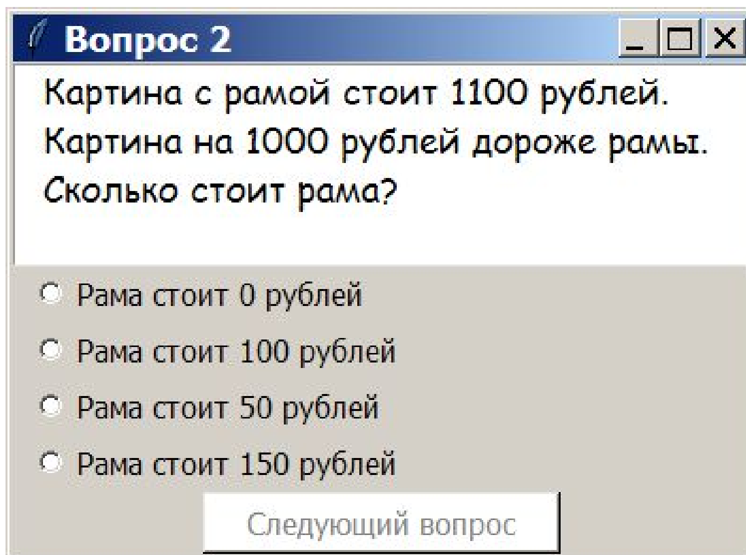
Теперь пользователь может установить желаемый порядок прохода по таблице, выбрав соответствующий пункт меню. На рисунке показан вид приложения с раскрытым пунктом «Порядок выбора».



19. Проект «Тест по математике»

Радиокнопки можно использовать и как обычные элементы управления. Разработаем небольшой проект, в котором набор радиокнопок применяется для выбора правильного ответа из набора возможных.

На рисунке приведен «снимок» работающего приложения. После ответа на первый вопрос была нажата кнопка «Следующий вопрос» и обновилось содержание как самого вопроса (элемента управления Text), так и набора ответов (четырёх радиокнопок Radiobutton). Заметим, что пока пользователь не выберет один из предлагаемых ответов, кнопка перехода к следующему вопросу будет неактивна.



Начнём с создания элементов управления. Для вывода вопроса используем «виджет» Text, который позволяет выводить многострочный текст. Этот элемент обладает возможностями текстового редактора, чем мы воспользуемся позже. Сейчас мы рассмотрим следующие параметры:

width – ширина элемента в символах;

height – количество текстовых строк;

wrap – способ переноса строк. По умолчанию строка переносится «по символу» (wrap=CHAR), т.е. слово, попавшее в конец строки, будет разорвано, и часть слова окажется на следующей строке. Если wrap=WORD, то слова при переносе строк не разрываются. Третий вариант — wrap=NONE, тогда строки не переносятся, а для их просмотра включается горизонтальная прокрутка текста.

font – шрифт для вывода текста, задаёт гарнитуру, размер (кегль) и начертание символов;

padx, pady – отступы (в пикселях) текста от границ элемента;

state – определяет возможность редактирования текста. По умолчанию state=NORMAL (редактирование разрешено). Для нашего проекта естественно после вывода текста вопроса запретить его редактирование, для этого необходимо будет установить state=DISABLED.

Для вставки текста используется метод insert(позиция, текст). Позиция в тексте задаётся в виде строки (или вещественного числа) вида 'строка.позиция'. Строки нумеруются начиная с единицы, а позиции — с нуля.

Для удаления текста используется метод delete(позиция_от, позиция_до). В этом случае удаляются все символы, начиная с символа в позиции_от и не включая символ в позиции_до. Для удобства работы с текстом введены некоторые обозначения, в частности позиция после последнего символа текста имеет имя END. Следовательно, вызов delete('1.0',END) удалит весь текст из элемента управления.

Программный код для создания элементов управления:

```
quest=Text(root,width=30,height=4,bg='white',wrap=WORD,
           font=('Comic Sans MS',14),padx=15)
quest.grid(row=0)
```

Добавляем четыре радиокнопки, выровненные по левой стороне окна:

```
rb1=Radiobutton(root,variable=v,value=1,padx=10)
rb2=Radiobutton(root,variable=v,value=2,padx=10)
rb3=Radiobutton(root,variable=v,value=3,padx=10)
rb4=Radiobutton(root,variable=v,value=4,padx=10)
rb1.grid(row=1,sticky=W)
rb2.grid(row=2,sticky=W)
rb3.grid(row=3,sticky=W)
rb4.grid(row=4,sticky=W)
```

Для всех радиокнопок определена одна и та же связанная переменная v. В этом случае реализуется их правильная совместная работа.

Последний элемент управления — командная кнопка.

```
button=Button(root,width=20,text='Следующий вопрос',
              bg='white',command=check)
button.grid(row=5)
```

Обработчик нажатия кнопки напишем позже.

При выборе любой радиокнопки будем «включать» командную кнопку:

```
def reaction(event):
   button.config(state='normal')
```

Свяжем функцию reaction с радиокнопками:

```
rb1.bind('<Button-1>',reaction)
rb2.bind('<Button-1>',reaction)
rb3.bind('<Button-1>',reaction)
```

```
rb4.bind('<Button-1>',reaction)
```

Вопросы теста и все варианты ответов соберём в текстовый файл. Структура файла:

строка 1: номер вопроса 1
строка 2: текст вопроса 1
строка 3: номер правильного ответа
строки 4-7: варианты ответов
строка 8: номер вопроса 2
строка 9: текст вопроса 2
и т.д.

Завершаем файл строкой с символом #.

Для визуального выделения номера правильного ответа его можно записать со знаком +

Пример файла:

```
1
Кирпич весит 1кг и ещё полкирпича. Сколько весит кирпич?
+2
Кирпич весит 1кг
Кирпич весит 2кг
Кирпич весит 3кг
Кирпич весит 4кг
...
#
```

Напишем функцию newQuestion, которая будет читать из файла очередную порцию данных и отображать их на экране. При обнаружении конца файла функция должна вывести окно с результатами теста.

Обозначим:

num – общее количество вопросов;
res – количество правильных ответов;
k – номер правильного ответа.

Переменные num и res объявим в главной программе, чтобы они были доступны функции newQuestion, а переменную k объявим глобальной, чтобы использовать её при проверке правильности выбранного ответа.

```
def newQuestion():
   global k      # номер правильного ответа
   s=f.readline()
   if s[0]=='#': # проверка конца файла
      showinfo('Результат',
      'Правильных ответов: '+str(res)+' из '+str(num))
      f.close()
```

```
        root.destroy()
        return
```

Последовательность дальнейших действий:

1) отключаем командную кнопку

```
button.config(state='disabled')
```

2) выводим номер вопроса в заголовок окна (при этом убираем символ конца строки)

```
root.title('Вопрос '+s.rstrip())
```

3) читаем из файла строку с вопросом

```
vopros=f.readline().rstrip()
```

4) разрешаем редактирование текста в элементе quest

```
quest.config(state='normal')
```

5) удаляем текст предыдущего вопроса

```
quest.delete('1.0',END)
```

6) вставляем текст нового вопроса

```
quest.insert('1.0',vopros)
```

7) запрещаем редактирование текста в элементе quest

```
quest.config(state='disabled')
```

8) читаем из файла номер правильного ответа

```
k=int(f.readline())   # номер правильного ответа
```

9) обнуляем значение связанной переменной. Теперь её значение не совпадает со значением value ни в какой радиокнопке, следовательно, ни одна радиокнопка не будет выделенной.

```
v.set(0)
```

10) читаем из файла и выводим на экран варианты ответов

```
txt1=f.readline().rstrip(); rb1.config(text=txt1)
txt2=f.readline().rstrip(); rb2.config(text=txt2)
txt3=f.readline().rstrip(); rb3.config(text=txt3)
txt4=f.readline().rstrip(); rb4.config(text=txt4)
```

Осталось написать обработчик нажатия командной кнопки, «шапку» программы:

```
from tkinter import *
from tkinter.messagebox import *
root=Tk()
```

```
v=IntVar()
res=0
num=0

def check():
   global num,res,k
   num+=1
   if v.get()==k:
      res+=1
   newQuestion()
```

и добавить в конце

```
f=open('voprosy_testa.txt')
newQuestion()
```

Полный текст программы приведён в Приложении.

Задание для самостоятельной работы. Придумайте свои задания и дополните ими тест.

20. Игра «ним»

Игра «ним» — одна из игр с кучками однородных предметов. В неё можно играть на отдыхе, используя в качестве предметов камешки. Описание этой и родственных ей игр можно найти и в литературе [7] и в сети Интернет.

В классическом варианте камешки раскладывают в три кучки. При своём ходе каждый из двух игроков может взять любое положительное количество камешков из какой-то одной кучки и даже забрать всю кучку. Выигрывает тот, кто сделает последний ход, т.е. заберёт последний камешек.

Для игры «ним» разработана исчерпывающая теория, которая определяет, является ли текущая позиция игры выигрышной или проигрышной для игрока, делающего ход в этой позиции. В выигрышной позиции существует по крайне мере один ход, который переводит её в проигрышную позицию для второго игрока. При этом любой (!) ход второго игрока переводит позицию в выигрышную для первого игрока. Таким образом результат игры известен до её начала! Однако найти выигрышный ход в произвольной позиции очень не просто.

Пример. Рассмотрим позицию с количеством камешков 1, 2 и 4. Из всех возможных ходов выигрышным является только один — взять один камешек из третьей кучки. В этом случае противнику достанется проигрышная позиция из 1, 2 и 3 камешков. Возникает вопрос — а как найти этот единственный ход? В простых случаях возможно проанализировать все варианты, однако с увеличением количества камешков в кучках этот подход становится нереальным.

Ниже излагается известный алгоритм анализа позиции и поиска выигрышного хода, но в виде, удобном для последующего программирования.

Анализ позиции:

1) переводим количество камней в каждой кучке в двоичную систему;

2) полученные двоичные числа складываем поразрядно по модулю 2;

3) если во всех разрядах результата получатся нули, то данная позиция проигрышная, если хотя бы в одном разряде есть единица, то позиция выигрышная.

Пример. Проанализируем позицию 2, 8, 13 :

$2 = 0010_2$

$8 = 1000_2$

$13 = 1101_2$

Результат: 0111_2, следовательно, эта позиция выигрышная.

Алгоритм поиска хода в выигрышной позиции основан на результате проведённого выше анализа позиции:

1) полученное на этапе анализа двоичное число (оно называется ним-суммой позиции) складываем поразрядно по модулю 2 с каждым исходным значением;

2) выбираем из результатов то значение, которое оказалось меньше исходного числа (или любое из них, если таких значений несколько);

3) разность между исходным и полученным значением есть количество камней, которые необходимо убрать из кучки с исходным количеством камней.

Продолжим пример.

$2 = 0010_2 \Rightarrow 0101_2 = 5 > 2$

$8 = 1000_2 \Rightarrow 1111_2 = 15 > 8$

$13 = 1101_2 \Rightarrow 1010_2 = 10 < 13$

Следовательно, нужно забрать три камня из кучки с 13-ю камнями, чтобы передать противнику ход в проигрышной позиции.

Проверим полученную позицию.

$2 = 0010_2$

$8 = 1000_2$

$10 = 1010_2$

Результат: 0000_2, следовательно, эта позиция проигрышная.

Реализуем описанный алгоритм на Питоне. Основная операция — поразрядное сложение двоичных чисел по модулю 2 — это логическая операция XOR (поразрядное исключающее ИЛИ). Знак этой операции в Питоне — символ ^ («крышечка»). Таблица истинности операции:

0 ^ 0 = 0

0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0

Легко видеть, что эта операция эквивалентна сложению двоичных разрядов по модулю 2. Самое интересное, что перевод чисел в двоичную систему происходит автоматически при выполнении этой операции, что упрощает реализацию алгоритма.

Пусть A есть список числа камней в текущей позиции. Напишем программный код поиска выигрышного хода в позиции A.

```
# вычисляем ним-сумму
res=0
for a in A: res^=a
# вычисляем вспомогательные значения
B=[a^res for a in A]
# и находим разности между исходными
# и вспомогательными значениями
C=[a-b for (a,b) in zip(A,B)]
# определяем номер кучки, из которой следует забрать
камни
id=C.index(max(C))
# получаем и выводим следующую позицию
A[id]=B[id]
print(A)
```

Если задать A=[2,8,13], то будет напечатана позиция [2, 8, 10].

Приступим собственно к выполнению проекта. Поскольку ним — игра для двух игроков, то роль второго игрока поручим компьютеру. Компьютер будет играть по описанному алгоритму, но начальную позицию будем задавать выигрышной для человека. Ему же предоставим право первого хода.

Разместим на форме три надписи для вывода текущей позиции (элементы управления Label), три поля для ввода количества убираемых камешков (элементы управления Entry) и кнопку (элемент управления Button).

Основные параметры для Label (Надпись):

anchor («якорь») — параметр определяет размещение текста внутри надписи. По умолчанию anchor = ‘center’, т.е. текст размещается в центре поля надписи.

width — ширина поля надписи в символах;
height — высота поля надписи в символах;
text — текст надписи;
font — параметры шрифта;
bg или background — цвет фона;
fg или foreground — цвет символов текста;
relief — вид рамки вокруг надписи;
bd or borderwidth — ширина рамки;

Элемент управления Entry, в отличие от рассмотренного ранее элемента Text, предназначен для ввода только одной строки текста.

Основные параметры для Entry:

width — ширина поля надписи в символах;
textvariable — задаёт связанную с полем ввода переменную;
font — параметры шрифта;
justify — выравнивание текста. По умолчанию justify=’left’;
bg или background — цвет фона;
fg или foreground — цвет символов текста;
relief — вид рамки вокруг поля ввода;
bd or borderwidth — ширина рамки;

Для всех элементов управления большинство параметров заданы по умолчанию, поэтому при создании элемента управления нужно задавать только «нестандартные» значения.

В этом проекте нам нужно создать несколько одинаковых по оформлению элементов управления. Можно задавать все параметры отдельно для каждого такого элемента. А можно собрать эти значения в словарь и использовать его при вызове конструктора.

Сравним эти два подхода. Создадим три надписи первым способом.

```
label1=Label(root,width=3,bg='#eeaaff',bd=5,relief='ridge
',
             font=('Courier New',36,'bold'),
             textvariable=v1)
label2=Label(root,width=3,bg='#eeaaff',bd=5,relief='ridge
',
             font=('Courier New',36,'bold'),
             textvariable=v2)
```

```
label3=Label(root,width=3,bg='#eeaaff',bd=5,relief='ridge
',
             font=('Courier New',36,'bold'),
             textvariable=v3)
```

Мы видим, что все параметры, кроме textvariable, принимают одни и те же значения. Создадим словарь, в который включим все эти значения:

```
label_params={'width':3,'bg':'#eeaaff','bd':5,
       'relief':'ridge','font':('Courier New',36,'bold')}
```

Тогда вызов конструктора примет следующий вид:

```
label1=Label(root,label_params,textvariable=v1)
label2=Label(root,label_params,textvariable=v2)
label3=Label(root,label_params,textvariable=v3)
```

Преимущество второго подхода очевидно. Кроме более компактной записи мы получаем возможность изменив значения параметра в одном месте, получить одинаковые изменения во всех трёх надписях.

Аналогично поступим и с полями ввода:

```
entry_params={'width':7,'justify':'center',
              'font':('Courier New',18,'bold')}
entry1=Entry(root,entry_params,textvariable=d1)
entry2=Entry(root,entry_params,textvariable=d2)
entry3=Entry(root,entry_params,textvariable=d3)
```

Переменные v1, v2, v3 содержат число камешков в кучках, а переменные d1, d2, d3 – параметры хода игрока. Создадим их с помощью конструктора IntVar():

```
v1=IntVar(); v2=IntVar(); v3=IntVar()
d1=IntVar(); d2=IntVar(); d3=IntVar()
```

По умолчанию все значения равны нулю.

Напишем функцию создания новой игры. Первый ход отдадим игроку-человеку. Чтобы игрок мог выиграть зададим выигрышную для него начальную позицию. Число камней в каждой кучке выбираем случайным образом в диапазоне от 2 до 15. Определяем, является ли выигрышной полученная позиция, в противном случае генерируем заново начальную позицию.

```
def newgame():
   res=0
   while res==0:
      A=[randint(1,15) for i in range(3)]
      res=0
      for a in A: res^=a
```

```
v1.set(A[0]); v2.set(A[1]); v3.set(A[2])
```

В конце «включаем» командную кнопку (она будет отключаться при ходе компьютера).

```
button.config(state='normal')
```

Игрок вводит в поле ввода, соответствующее выбранной кучке, количество забираемых камней и нажимает кнопку «Сделать ход». В обработчике события нам нужно:

1) проверить корректность хода игрока;

2) выполнить ход;

3) выдержать паузу, при этом кнопка «Сделать ход» должна быть отключена;

4) определить, является ли позиция, полученная после хода игрока, заключительной (в этом случае выиграл игрок) или выигрышной или проигрышной уже для компьютера.

В случае выигрыша игрока программа выводит окно сообщения и предлагает начать новую игру. Если пользователь выбирает «Да» вызывается метод newgame(). Если пользователь выбирает «Нет» программа завершает свою работу.

5) сделать ход компьютера. В случае выигрышной позиции компьютер определяет выигрышный ход по приведенному выше алгоритму. В случае проигрышной позиции компьютер делает случайный ход — забирает случайное число камней из произвольной кучки. В программе выбирается кучка с максимальным количеством камней.

6) после этого компьютер проверяет, является ли полученная позиция заключительной (в этом случае выиграл компьютер).

В случае выигрыша компьютера программа выводит окно сообщения и предлагает начать новую игру. Если пользователь выбирает «Да» вызывается метод newgame(). Если пользователь выбирает «Нет» программа завершает свою работу.

Если позиция не является заключительной, то ход передаётся игроку, для чего включается кнопка «Сделать ход»

Приведём текст функции без дальнейших пояснений.

```
def move():
    num=0
    if d1.get()!=0: num+=1
    if d2.get()!=0: num+=1
    if d3.get()!=0: num+=1
    if d1.get()<0 or d2.get()<0 or d3.get()<0\
    or d1.get()>v1.get() or d2.get()>v2.get()\
    or d3.get()>v3.get() or num != 1:
        showinfo('Ошибка','Некорректный ход!')
```

```
        return
    else:
        v1.set(v1.get()-d1.get());
        v2.set(v2.get()-d2.get())
        v3.set(v3.get()-d3.get())
    d1.set(0); d2.set(0); d3.set(0)
    root.update()
    # ход компьютера
    button.config(state='disabled')
    root.update()
    sleep(2)
    A=[v1.get(),v2.get(),v3.get()]
    if A==[0,0,0]:
        yes=askyesno('Результат игры',
                     'Вы выиграли!\nНачать новую игру?')
        if yes:
            newgame()
            return
        else:
            root.destroy()
            return
    res=0
    for a in A: res^=a
    if res>0:
        B=[a^res for a in A]
        C=[a-b for (a,b) in zip(A,B)]
        id=C.index(max(C))
        A[id]=B[id]
        v1.set(A[0])
        v2.set(A[1])
        v3.set(A[2])
        if A==[0,0,0]:
            yes=askyesno('Результат игры',
                         'Вы проиграли!\nНачать новую
игру?')
            if yes :
                newgame()
                return
            else:
                root.destroy()
                return
    else:
        m=max(A)
        id=A.index(m)
        r=randint(1,m)
```

```
        if id==0: v1.set(v1.get()-r)
        if id==1: v2.set(v2.get()-r)
        if id==2: v3.set(v3.get()-r)
    button.config(state='normal')
    root.update()
```

Задания для самостоятельного выполнения.

1) Завершите разработку проекта.

2) Усложните игру — добавьте ещё одну кучку камней.

Зададимся вопросом — можно ли модифицировать программу так, чтобы число кучек камней можно было задавать как параметр игры? И нужно ли это?

Понятно, что число кучек не должно быть большим, так как игра становится громоздкой и пользователь потеряет к ней интерес. Оптимальным, по мнению автора, является классический вариант с тремя кучками. Для двух кучек алгоритм выигрыша проще некуда: первым ходом следует уравнять количество камней в кучках, а затем повторять ходы противника. Следовательно, если в исходной позиции число камней разное, то начинающий игру выиграет, если одинаковое — проиграет.

Однако с точки зрения программиста такое обобщение игры позволит освоить работу со списками объектов и переменных, а значит глубже изучить возможности языка программирования Python.

Обозначим через n число кучек и заменим переменные d1, d2, d3, v1, v2 и v3 списками:

```
d=[IntVar() for i in range(n)]
v=[IntVar() for i in range(n)]
```

Таким образом, d[0], d[1], d[2], ... соответствуют переменным d1, d2, d3, ... Аналогично, v[0], v[1], v[2], ... соответствуют переменным v1, v2, v3, ...

Создадим два списка элементов управления: label — надписи и entry — поля ввода, и разместим их в окне приложения:

```
label=[]
entry=[]
label_params={'width':3,'bg':'#eeaaff','bd':5,
       'relief':'ridge','font':('Courier New',36,'bold')}
entry_params={'width':7,'justify':'center',
              'font':('Courier New',18,'bold')}
for i in range(n):

label.append(Label(root,label_params,textvariable=v[i]))

entry.append(Entry(root,entry_params,textvariable=d[i]))
```

```
    label[i].grid(row=0,column=i)
    entry[i].grid(row=1,column=i)
button=Button(root,width=17,text='Сделать ход',
              command=move)
button.grid(row=2,columnspan=n)
```

Изменения в функции newgame(). Вместо присваиваний

```
v1.set(A[0]); v2.set(A[1]); v3.set(A[2])
```

естественно написать цикл с параметром

```
for i in range(n): v[i].set(A[i])
```

Изменений в функции move() будет существенно больше, но их принцип в большинстве случаев останется тем же — набор присваиваний заменяется циклом!

Проверка корректности хода:

```
num=0
for i in range(n):
   if d[i].get()!=0: num+=1
A=[0<=d[i].get()<=v[i].get() for i in range(n)]
if not all(A) or num != 1:
   showinfo('Ошибка','Некорректный ход!')
   return
```

Для корректности хода необходимо, чтобы все значения в списке d было в диапазоне от нуля до числа камней в этой кучке и в этом же списке было только одно ненулевое значение.

Создадим список логических значений: True, если для переменной d[i] выполняется сформулированное выше условие корректности. Ход является корректным, если все значения в списке равны True. Для этой цели применим к списку предикат all(список). Он возвращает True, если все значения в списке равны True. Заметим, что в языке есть и предикат any(список), который равен True, есть в списке есть хотя бы одно значение True.

Проверка завершения игры: сравниваем позицию игры со списком из n нулей

```
    A=[v[i].get() for i in range(n)]
    if A==n*[0]:
       yes=askyesno('Результат игры',
                    'Вы выиграли!\nНачать новую игру?')
...
```

Выбор хода для компьютера в проигрышной позиции упрощается:

```
m=max(A)
id=A.index(m)
```

```
        r=randint(1,m)
        v[id].set(v[id].get()-r)
```

Остальные изменения реализуйте самостоятельно.

21. Проект «Спирограф»

Следующий проект основан на программе, разработанной в первой части книги. Программа моделировала работу двухзвенного механизма и рисовала узоры на плоскости. Недостаток этой программы — отсутствие интерфейса пользователя. Чтобы изменить какой-либо параметр приходилось изменять код программы и заново его выполнять. Наличие интерфейса позволит нам намного проще исследовать поведение механизма при разных значениях его параметров.

Определим набор параметров механизма. Напомним, что их было три:

R1 – длина первого звена;

R2 – длина второго звена;

k – отношение угловой скорости вращения второго звена к угловой скорости вращения первого звена.

Неудобство реализации состоит в том, что радиус узора не будет постоянным — он определяются суммой R1+R2 длин звеньев механизма. Примем поэтому, что R1+R2 = 100 пикселей, тогда можно ограничиться одним параметром. Положим R1 = R, тогда R2 = 100 – R.

Для ввода параметров k и R используем элементы управления Label и Entry.

Элемент управления Label (Метка) используется для вывода поясняющих надписей, а Entry (Ввод) — для ввода редактируемой строки текста. Нам ещё понадобится кнопка Start для запуска процедуры построения кривой.

Конструктор Label первым параметром получает ссылку на объект, в котором метка будет размещена. При отсутствии этой ссылки используется окно приложения. Аналогично для создания поля ввода используется конструктор Entry, а для создания кнопки — конструктор Button.

Последующий набор именованных параметров определяет вид элемента управления и его функциональность. Значительная часть параметров одинакова для различных элементов управления. Рассмотрим, например, упомянутые выше элементы управления.

Некоторые общие для всех трёх элементов управления параметры:

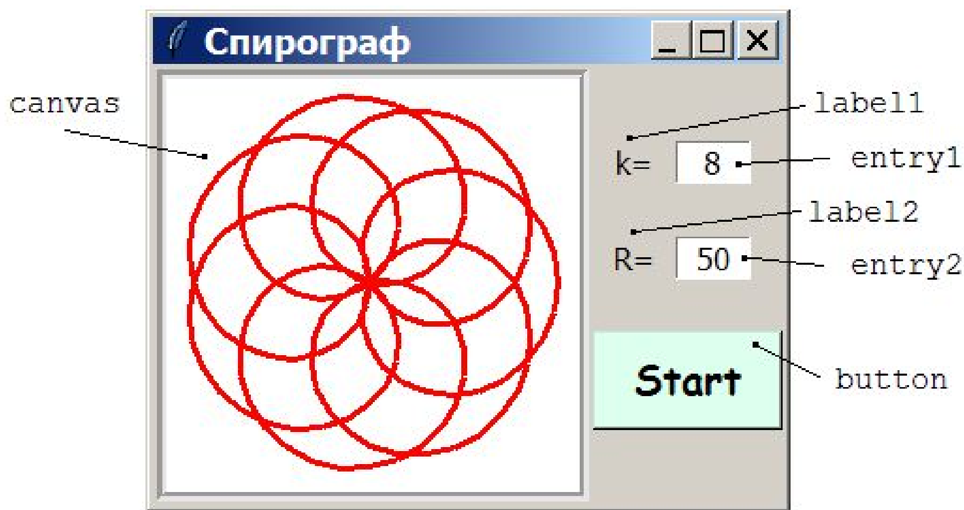
- font – гарнитура, размер и начертание символов;
- fg or foreground — цвет переднего плана (цвет текста);
- bg или background — цвет фона элемента управления;
- justify — выравнивание многострочного текста внутри элемента. Значения параметра:

LEFT или 'left' — выравнивание по левому краю;
RIGHT или 'right' — выравнивание по левому краю;
CENTER или 'center' — выравнивание по центру.

Заметим, что для метки и кнопки многострочный текст можно получить, вставив в текст символ перевода строки \n. По умолчанию текст для этих элементов управления центрируется. А вот текст в поле ввода по умолчанию выравнивается влево!

- cursor – вид курсора при нахождении над элементом
- relief – трёхмерный вид элемента управления:
  FLAT или 'flat' — плоский;
  RAISED или 'raised' — приподнятый;
  SUNKEN или 'sunken' — утопленный;
  RIDGE или 'ridge' — приподнятая рамка;
  GROOVE или 'groove' — утопленная рамка;
- bd или borderwidth — ширина рамки вокруг элемента управления (в пикселях);
- textvariable — строковая переменная, связанная с элементом. В проекте «Спирограф» она будет использоваться для изменения значений полей ввода. Для элементов метка и кнопка удобнее пользоваться параметром text.
- width – ширина элемента в знакоместах. На метку или кнопку можно, используя параметр image, вывести изображение, тогда width задаётся в пикселях.
- height – для метки и кнопки — высота элемента в знакоместах (для текста) или в пикселях (для изображения);

Приступим к реализации проекта. Спроектируем сначала интерфейс приложения «на бумаге».



Напишем отдельные фрагменты, которые потом соберём в единую программу. Создадим элементы управления, предполагая, что окно приложения root уже создано:

```
canvas=Canvas(root,width=220,height=220,
              bg='white',bd=5,relief='groove')
label1=Label(text='k=')
label2=Label(text='R=')
entry1=Entry(root, width=4,textvariable=kv)
entry2=Entry(root, width=4,textvariable=rv)
button=Button(text='Start',width=7,bg='#ddffee',
              font=('Comic Sans MS',16,'bold'),
              command=draw)
```

Поскольку при создании полей ввода мы ссылаемся на переменные kv и rv, то их нужно предварительно объявить:

```
kv=StringVar()
rv=StringVar()
```

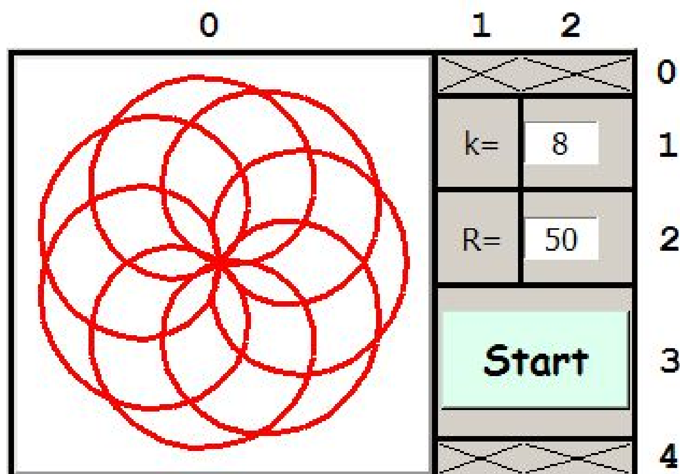
То же самое относится к функции draw – обработчику события «нажатие кнопки».

```
def draw():
    # определяем «точку крепления» механизма
    S=(int(canvas['width'])//2 +int(canvas['bd']),
       int(canvas['height'])//2+int(canvas['bd']))
    # получаем значения параметров
    k=int(kv.get())
    R1=int(rv.get())
    R2=100-R1
    # удаляем все графические объекты с холста
    canvas.delete(ALL)
    # создаём узор
    id=create_circle(canvas,S,5,width=0,fill='gray')
    for fi in range(0,361,2):
        A=ps2ds(S,(R1,fi))
        B=ps2ds(A,(R2,k*fi))
        canvas.create_line(S,A,width=3,fill='gray',
                           tags='SAB')
        canvas.create_line(A,B,width=3,fill='gray',
                           tags='SAB')
        create_circle(canvas,B,3,fill='red',tags='SAB')
        if fi>0:
            canvas.create_line(W,B,width=3,fill='red')
        W=B
```

```
            canvas.update()
            sleep(0.02)
            canvas.delete('SAB')
        canvas.delete(id)
```

Разместим теперь элементы управления в окне приложения, используя тот же менеджер размещения grid, что и в предыдущем проекте. Но сейчас задача у нас сложнее, так как сетка для размещения элементов не является регулярной.

Начертим «на бумаге» окно приложения и разобьём его на ячейки. Получится таблица из пяти строк и трёх столбцов, в которой некоторые ячейки объединены в одну. Перечеркнутые ячейки для размещения элементов управления не используются. Они добавлены, чтобы улучшить внешний вид приложения.



Мы можем указать менеджеру grid, какое количество строк и столбцов занимает объединённая ячейка. Для этого служат именованные параметры rowspan=число_строк и columnspan=число_столбцов.

```
canvas.grid(row=0,column=0,rowspan=5)
label1.grid(row=1,column=1)
entry1.grid(row=1,column=2,sticky=W)
label2.grid(row=2,column=1)
entry2.grid(row=2,column=2,sticky=W)
button.grid(row=3,column=1,columnspan=2)
```

Ещё один новый параметр — sticky. В переводе с английского — прилипающий. Этот параметр указывает, к какой стороне ячейки будет «прилеплен» элемент управления при выводе на экран. Стороны определяются по географическому принципу:

W (от англ. West – запад) – середина левой стороны;

N (от англ. North – север) – середина верхней стороны;

E (от англ. East – восток) – середина правой стороны;

S (от англ. South – юг) – середина нижней стороны;

Можно поместить элемент управления в угол ячейки, если указать сразу две стороны: NW, NE, SW, SE, растянуть по вертикали (N+S), горизонтали (E+W) или заполнить всю ячейку (N+E+S+W).

И, наконец, зададим начальные значения связанным переменным

```
kv.set('8')     # коэффициент k
rv.set('50')    # длина первого звена
```

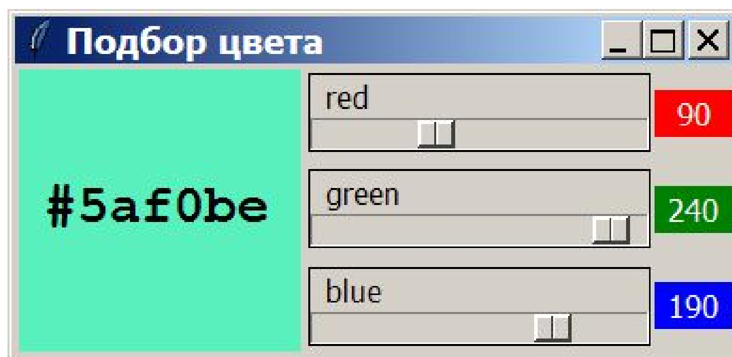
При сборке программы не забудем подключить нужные модули и создать главное окно приложения:

```
from time import *
from geom_sys import *
from tkinter import *
root=Tk()
root.title('Спирограф')
```

Полный текст программы приведён в приложении 2

22. Проект «Подбор цвета»

Разработаем небольшой проект, использующий элемент управления Scale (Шкала). Внешний вид приложения показан на рисунке.



Окно приложения содержит холст, три шкалы с диапазоном изменения от 0 до 255 и три метки, которые красиво отображают значения, установленные на шкалах.

Шкала имеет много параметров. Основные из них:

from_ - нижняя граница диапазона (символ подчеркивания нужен, так как ключевое слово from используется в директиве подключения модулей);

to – верхняя граница диапазона;

orient — ориентация шкалы:

HORIZONTAL или 'horizontal' — горизонтальное расположение;

VERTICAL или 'vertical' — вертикальное расположение.

length – длина шкалы, по умолчанию равна 100 пикселей;

sliderlength — длина ползунка (слайдера), по умолчанию 30 пикселей;

width – ширина шкалы, по умолчанию равна 15 пикселей;

relief – тип рамки вокруг шкалы. Кроме рассмотренных ранее возможен вариант SOLID (или 'solid'), как показано на рисунке.

label – надпись на шкале;

variable – переменная, в которой содержится значение, установленное на шкале;

showvalue – признак вывода установленного на шкале значения (при showvalue=0 значение не выводится);

command – функция-обработчик перемещения ползунка.

tickinterval — разность значений двух соседних делений шкалы. По умолчанию tickinterval=0, в этом случае шкала не содержит делений.

resolution — разрешение шкалы. Фактически задаёт дискретный набор значений, которые можно установить на шкале. В случае целочисленных значений разрешение шкалы равно единице.

Для «грубой» настройки значения шкалы используется перемещение ползунка при движении мыши с нажатой левой кнопкой. Но в этом случае трудно точно выставить определённое значение. Для «тонкой» настройки необходимо щёлкнуть левой кнопкой мыши левее/выше или правее/ниже ползунка. в этом случае значение шкалы изменяется на минимально возможное значение.

Напишем программу.

```
# импорт графической библиотеки Питона
from tkinter import *
# создание окна приложения с "холстом" для рисования
root=Tk()
root.title('Подбор цвета')

r=IntVar(); r.set(90)
g=IntVar(); g.set(240)
b=IntVar(); b.set(190)

def changecolor(event):
   txt='#{:02x}{:02x}{:02x}'.format(
                          r.get(),g.get(),b.get())
   canvas['bg']=txt
   canvas.itemconfig(color,text=txt)

canvas=Canvas(root,width=150,height=150,bg='white')
color=canvas.create_text(75,75,
```

```
        font=('Courier New',22,'bold'),
        text='#{0:02x}{1:02x}{2:02x}'.format(
               r.get(),g.get(),b.get()))
rscale=Scale(root,from_=0,to=255,orient='horizontal',
             length=180,sliderlength=20,showvalue=0,
             label='red',relief='solid',
             command=changecolor,
             variable=r)
gscale=Scale(root,from_=0,to=255,orient='horizontal',
             length=180,sliderlength=20,showvalue=0,
             label='green',relief='solid',
             command=changecolor,
             variable=g)
bscale=Scale(root,from_=0,to=255,orient='horizontal',
             length=180,sliderlength=20,showvalue=0,
             label='blue',relief='solid',
             command=changecolor,
             variable=b)
rvalue=Label(root,width=4,bg='red',  fg='white',
             textvariable=r)
gvalue=Label(root,width=4,bg='green',fg='white',
             textvariable=g)
bvalue=Label(root,width=4,bg='blue', fg='white',
             textvariable=b)

canvas.grid(row=0,column=0,rowspan=3)
rscale.grid(row=0,column=1)
gscale.grid(row=1,column=1)
bscale.grid(row=2,column=1)
rvalue.grid(row=0,column=2)
gvalue.grid(row=1,column=2)
bvalue.grid(row=2,column=2)
```

Задание для самостоятельной работы

Измените расположение и параметры шкал так, чтобы получить следующий вид приложения:

23. Игра «Полоски»

Разработаем проект для иллюстрации «геометрических» возможностей модуля tkinter. Проект представляет собой модель известной игры со спичками: на ровную поверхность насыпается горка спичек, требуется снимать их по одной так, чтобы не сдвинуть остальные. В компьютерной модели «спичками» будут полоски-прямоугольники, которые накладываются друг на друга. Убрать можно только ту полоску, которая не перекрывается никакой другой, – достаточно щелкнуть по ней мышкой. Игра считается оконченной, когда игрок уберёт все полоски.

Для повышения интереса к игре добавим ограничение по времени. Ограничим, например, время игры в секундах, равным начальному количеству полосок. Если пользователю удастся за это время убрать все полоски, то он выиграл, если не удастся — то проиграл. В любом случае пользователь имеет возможность запустить игру заново.

Начнём писать проект с подключения модулей. Из модуля time импортируем функцию time для определения текущего времени. Из модуля random – функцию randint для генерации случайных целых чисел из заданного диапазона. Из tkiner и tkinter.messagedox импортируем все функции.

```
from time import time
from random import randint
# импорт графической библиотеки Питона
from tkinter import *
from tkinter.messagebox import *
```

Создадим окно приложения с "холстом" для рисования

```
root=Tk()
root.title('Полоски')
canvas=Canvas(root,width=240,height=240,bg='white')
```

```
canvas.pack()
```

Инициализируем переменную count = число полосок:

```
count=15  # число "полосок"
```

Чтобы полоски были разноцветными, напишем функцию, возвращающую случайный цвет:

```
def rndcolor():
   r=randint(0,255)
   g=randint(0,255)
   b=randint(0,255)
   return '#{0:02x}{1:02x}{2:02x}'.format(r,g,b)
```

Напишем функцию newgame() создания новой игры. Основа функции — цикл, в котором генерируются прямоугольники случайного положения, случайного размера и случайного цвета. Каждый прямоугольник получает целочисленный идентификатор, фактически это его персональный номер. Прямоугольники нумеруются по возрастанию последовательными целыми числами, начиная с единицы. Заметим, что чем позже создаётся прямоугольник, тем «ближе» он к пользователю, т.е. он может перекрывать ранее созданные прямоугольники. К каждому прямоугольнику «привязываем» одну и ту же функцию reaction – обработчик щелчка левой кнопкой мыши.

```
rect=canvas.create_rectangle(x,y,x+w,y+h,
                             fill=rndcolor())
canvas.tag_bind(rect,'<Button-1>',reaction)
```

В функции newgame объявляем глобальные переменные n=count – начальное количество полосок, и tm_beg=time() – время начала сеанса игры. Ещё одно, но не очевидное действие — удаление всех графических объектов из канвы перед генерацией нового варианта. Дело в том, что в случае проигрыша на холсте останется несколько прямоугольников, которые будут видны после генерации новой игры.

```
def newgame():
   global n,tm_beg
   n=count
   tm_beg=time()
   canvas.delete(ALL)
   for i in range(n):
      x=randint(10,180)
      y=randint(10,180)
      w=randint(10,210-x)
      h=randint(10,210-y)
      rect=canvas.create_rectangle(x,y,x+w,y+h,
```

```
                            fill=rndcolor())
        canvas.tag_bind(rect,'<Button-1>',reaction)
```

Функционал игры реализуем в функции-обработчике reaction. Нам понадобятся следующие методы канвы:

1) метод bbox(идентификатор/тэг) — определение координат области, которую занимает объект с указанным идентификатором или тэгом. Метод возвращает координаты левого верхнего и правого нижнего угла области окна приложения в которой размещен этот объект. Если вызвать метод с ключевым словом CURRENT, то мы получим координаты области, которую занимает текущий объект, т.е. тот объект, на котором произошло событие;

2) метод find_overlapping(x1,y1,x2,y2) — определение идентификаторов объектов, которые имеют общие точки с заданным прямоугольником;

3) метод find_withtag(CURRENT) — определение идентификатора текущего объекта. Метод всегда возвращает список идентификаторов, так как в качестве параметра может получить тэг, одинаковый для нескольких графических объектов.

4) метод delete(идентификатор/тэг) — удаление объекта с указанным идентификатором или тэгом. Если метод delete вызвать с ключевым словом CURRENT, то будет удален текущий объект, а если вызвать с ключевым словом ALL, то будут удалены все объекты.

Теперь можем написать обработчик события <Button-1>. В обработчике прежде всего вычисляется время, прошедшее с начала игры:

```
tm_len=time()-tm_beg
```

Если интервал времени tm_len меньше заданного, то игра продолжается. В этом случае определяем координаты прямоугольника, по которому пользователь щелкнул мышкой, и его идентификатор.

```
R=canvas.bbox(CURRENT)
id=canvas.find_withtag(CURRENT)
```

Затем получаем список идентификаторов объектов, с которыми пересекается текущий прямоугольник:

```
objects=canvas.find_overlapping(*R)
```

Осталось проверить, лежит ли наш прямоугольник выше всех тех, с какими он пересекается. Эта задача облегчается тем, что в списке объектов objects все идентификаторы располагаются в порядке возрастания, т.е. самый верхний прямоугольник находится в конце списка.

Если id[0]==objects[-1], то это значит, что текущий прямоугольник является самым верхним в списке objects, и его можно удалить. Указание индекса 0 необходимо, так как метод find_withtag возвращает список, хотя в данном случае этот список состоит из одного элемента.

После удаления последнего прямоугольника на экране появляется сообщение «Вы выиграли» с выводом затраченного времени и вопросом «Играть ещё?». При нажатии на кнопку «Да» вызывается функция newgame() и начинается новая игра. Если пользователь за отведенное время не успел удалить все прямоугольники, то на экране появляется сообщение «Вы проиграли» с указанием количества оставшихся объектов и с тем же вопросом «Играть ещё?». И в этом случае при нажатии кнопки «Да» вызывается метод newgame(). Если же пользователь нажимает кнопку «Нет», то вызывается метод приложения destroy, который удаляет окно и завершает работу приложения.

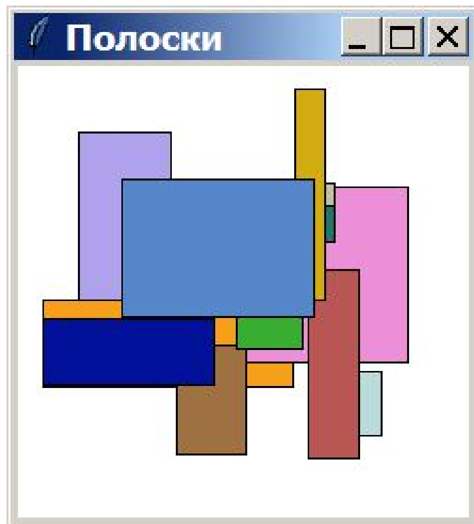
Приведём полный текст обработчика:

```
def reaction(event):
   global n,tm_beg
   tm_len=time()-tm_beg
   if tm_len<count:
      R=canvas.bbox(CURRENT)
      id=canvas.find_withtag(CURRENT)
      objects=canvas.find_overlapping(*R)
      if id[0]==objects[-1]:
         canvas.delete(CURRENT); n-=1
         if n==0:
            yes=askyesno('Вы выиграли!',
                'Затраченное время - '+
                +str(int(tm_len))+
                ' секунд.\n Играть ещё?')
            if yes: newgame()
            else: root.destroy()
   else:
      yes=askyesno('Вы проиграли!',
                   'Осталось прямоугольников - '
                   +str(n)+
                   '.\nИграть ещё?')
      if yes: newgame()
      else: root.destroy()
```

Запускаем игру:

```
newgame()
root.mainloop()
```

В заключение приведём вид приложения со случайным расположением «полосок» при запуске.

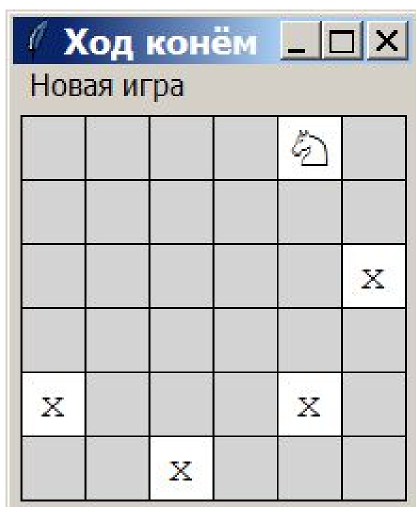


Задание для самостоятельной работы. Разработайте интерфейс приложения, позволяющий задавать число полосок и время игры. Предусмотрите кнопку Start, которая запускает игру. Модифицируйте программу, в частности замените метод askyesno(...) на метод showinfo(...), который только информирует пользователя о результате игры.

24. Проект «Ход конём»

Широко известна задача: обойти шахматную доску ходом коня так, чтобы конь побывал на каждом поле ровно один раз. Этой задачей увлекались многие математики, придумано несколько алгоритмов для нахождения маршрута коня, рассмотрены варианты досок другого размера и формы... Наша задача существенно скромнее — реализовать приложение, которое позволяло бы пользователю самому проложить маршрут коня по доске.

На рисунке представлен вид окна приложения во время игры. Начальное положение коня выбирается случайным образом. Чтобы сделать ход, пользователь щёлкает мышкой по по полю, на которое должен стать конь. Пройденные поля отмечаются крестиком.



Игра заканчивается выигрышем, когда пользователь заполнит крестиками все пустые поля доски, или проигрышем, когда у коня не будет возможности сделать ход при наличии на доске не посещённых полей.

Создадим информационную модель шахматной доски. Представим доску как двумерный массив полей. Каждое поле есть отдельный графический объект, внешний вид которого зависит от его внутреннего состояния. Для хранения параметров поля и его состояния нам понадобится некоторая структура данных. Зададим структуру данных с помощью класса. Класс — это обобщение типа данных, а экземпляр класса — это переменная, тип которой определяет класс.

Синтаксис описания класса:

```
class Имя_класса:
   # конструктор класса
   def __init__(self, список_параметров):
      ...
   # методы класса
   def имя_метода(self, список_параметров):
      ...
```

Имя конструктора класса состоит из ключевого слова init с двумя подчеркиваниями перед и после init. Для создания экземпляра класса используется имя класса и список параметров конструктора, который вызывается неявно. Ключевое слово self играет роль указателя на создаваемый объект — экземпляр класса.

Кроме методов класс может содержать данные, которые могут создаваться как в конструкторе класса, так и в любом из методов. Элементы данных также описываются с использованием ключевого слова self.

Однако, кроме данных, индивидуальных для каждого поля, есть параметры приложения, значения которых одинаковы для всех полей. Эти

параметры имеет смысл объявить отдельно и использовать при создании полей:

n=6 – размер игровой доски «в клетках»;

size=34 — размер поля в пикселях;

p=4 – сдвиг изображения игровой доски в окне приложения.

Разработаем теперь класс Cell – поле игровой доски. Какие данные нам понадобятся для идентификации и отображения поля? Во-первых, расположение поля на доске — номер строки и номер столбца в двумерном массиве полей. Во-вторых, фон поля и символ, отображаемый на поле. В-третьих, переменная, отвечающая за состояние поля.

Обозначим:

self.i – номер строки, в которой расположено поле;

self.j – номер столбца, в котором расположено поле;

self.fon – фон поля;

self.txt – символ поля;

self.s – состояние (статус) поля:

-1 для посещенного поля;

0 для ещё не посещенного поля;

1 для поля, на котором находится конь.

Параметры конструктора класса:

i – номер строки, в которой расположено поле;

j – номер столбца, в котором расположено поле.

В конструкторе создаётся пустой квадрат без символа, а конкретный вид поля определяется в методе setStatus, который устанавливает статус поля и изменяет параметры поля в соответствии с его статусом.

```
class Cell:
   # конструктор
   def __init__(self,i,j):
      self.i=i
      self.j=j
      x=p+self.j*size
      y=p+self.i*size
      self.fon=canvas.create_rectangle(
               (x,y),(x+size,y+size),outline='black')
      self.txt=canvas.create_text(
               (x+size//2,y+size//2),
               font=('Courier New',size//2))
   # изменение статуса поля
   def setStatus(self,s):
      self.s=s
      if s==-1:
         canvas.itemconfig(self.fon,fill='white')
```

```
            canvas.itemconfig(self.txt,text='x')
        if s==0:
            canvas.itemconfig(self.fon,fill='lightgray')
            canvas.itemconfig(self.txt,text='')
        if s==1:
            canvas.itemconfig(self.fon,fill='white')
            canvas.itemconfig(self.txt,text='\u2658')
```

В главной программе создадим двумерный массив полей:

```
#------- main ------
cells=[]
for i in range(n):
    w=[]
    for j in range(n):
        w.append(Cell(i,j))
    cells.append(w)
```

Двумерный массив представляется в виде списка списков, поэтому для его создания нам понадобился вспомогательный список w, который соответствует строке двумерного массива. Список w заполняется экземплярами класса Cell, а затем добавляется к основному списку cells.

Осталось определить поле, на котором будем стоять конь и обнулить счётчик посещённых полей. Обозначим

count – счётчик:

ns, np – номер строки и номер столбца поля, на котором расположен конь.

и напишем функцию newgame:

```
def newgame():
    global count, ns, np
    count=0
    ns=randint(0,n-1)
    np=randint(0,n-1)
    for i in range(n):
        for j in range(n):
            if i==ns and j==np: s=1
            else: s=0
            cells[i][j].setStatus (s)
```

Добавим в приложение меню и холст:

```
menubar=Menu()
root.config(menu=menubar)
menubar.add_command(label='Новая игра',command=newgame)
canvas=Canvas(root,width=p+n*size,height=p+n*size)
canvas.pack()
```

Напишем функцию reaction – обработчик щелчка мыши.

```
def reaction(event):
   global count, ns, np
   # вычисляем «координаты» поля
   i=(event.y-p)//size
   j=(event.x-p)//size
   # проверяем корректность хода
   if cells[i][j].s==0 and (i-ns)**2+(j-np)**2==5:
      # переставляем коня на поле (i,j)
      cells[i][j].setStatus (1)
      cells[ns][np].setStatus (-1)
      ns=i; np=j
      # проверкв завершения игры
      count+=1
      if count==n*n-1:
         showinfo('Игра окончена!','Вы выиграли!')
      else:
         if not nextstep(ns,np):
            showinfo('Игра окончена!','Вы проиграли!')
```

Функция nextstep получает координаты поля, на котором расположен конь, и определяет, возможен ли следующий ход. Если все поля, на которые мог бы пойти конь уже посещены, то функция возвращает логическое значение False, иначе возвращает значение True.

```
def nextstep(ns,np):
   step=[(-2,-1),(-2,1),(-1,-2),(-1,2),
         ( 1,-2),( 1,2),( 2,-1),( 2,1)]
   res=False
   for (ds,dp) in step:
      i=ns+ds; j=np+dp
      if 0<=i<n and 0<=j<n and cells[i][j].s==0:
         res=True
   return res
```

Задание для самостоятельной работы. Соберите все фрагменты в единую программу. Не забудьте подключить необходимые модули, свяжите событие <Button-1> (щелчок левой кнопкой мыши) с его обработчиком. Попробуйте найти решение задачи.

Оказалось, что найти решение задачи совсем не просто! Чаще всего остаётся одно-два не посещённых поля. Хотелось бы иметь возможность отмены сделанных ходов в обратном порядке. Для реализации такой возможности добавим в программу историю ходов — список history.

В начале игры в список history помещаем кортеж (ns,np) с начальными координатами коня. При каждом ходе в конец списка добавляем новое положение коня. Чтобы вернуть коня на предыдущую позицию нужно изменить состояние двух полей. Для отмены хода нужны два последних элемента списка.

Извлекаем из списка последний элемент:

```
(ns,np)=history.pop()
```

Метод pop() возвращает значение последнего элемента с одновременным удалением его из списка. Фактически список history выступает в роли стека — структуры данных, работающей по принципу: последний вошёл — первый вышел.

Делаем поле пустым:

```
cells[ns][np].setStatus (0)
```

Получаем координаты «предыдущего» поля (но оставляем их в списке):

```
(ns,np)=history[-1]
```

Ставим коня на это поле:

```
cells[ns][np].setStatus (1)
```

Уменьшаем на единицу количество посещённых полей.

```
count-=1
```

Описанные действия повторяем при каждом нажатии клавиши Escape, поэтому коня можно вернуть даже в начальную позицию. При этом в списке history останется только первый элемент.

Приведём текст функции-обработчика события <Escape>

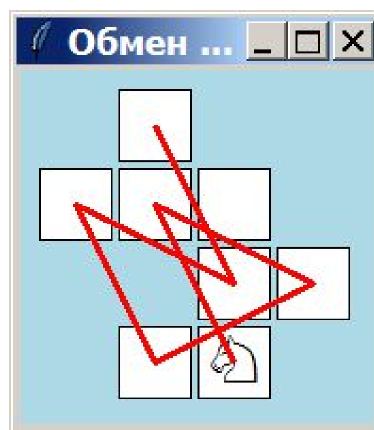
```
def cancel(event):
   global count, ns, np, history
   if len(history)>1:
      (ns,np)=history.pop()
      cells[ns][np].setStatus (0)
      (ns,np)=history[-1]
      cells[ns][np].setStatus (1)
      count-=1
```

Задание для самостоятельной работы. Модифицируйте программу «Ход конём», добавив в неё описанную возможность отмены ходов вплоть до начальной позиции.

25. Игра «Обмен коней»

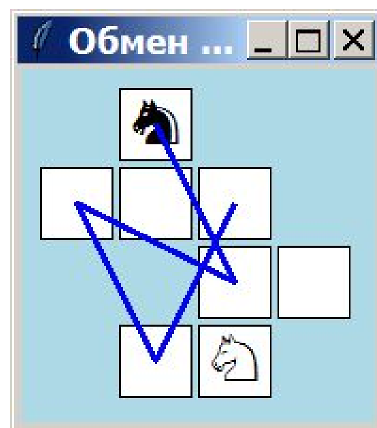
В этой игре необходимо обменять местами несколько шахматных фигур, придерживаясь шахматных правил. При этом игровая доска может иметь сложную форму, что затрудняет поиск решения.

Рассмотрим вариант обмена местами двух коней. Для наглядности сделаем одного коня белым, а другого чёрным. Белый конь есть символ Unicode '\u2658', а чёрный — символ Unicode '\u265e'.



Необходимо белого коня поставить на место чёрного, а чёрного — на место белого. Для решения задачи уберём с доски чёрного коня и проложим маршрут для белого коня. Он единственный при условии отсутствия возвратов на уже пройденные поля.

Этот маршрут оставляет незатронутым единственное поле, на которое нужно будет поставить чёрного коня, чтобы он не мешал пройти белому. На левом рисунке показан маршрут черного коня на это поле.



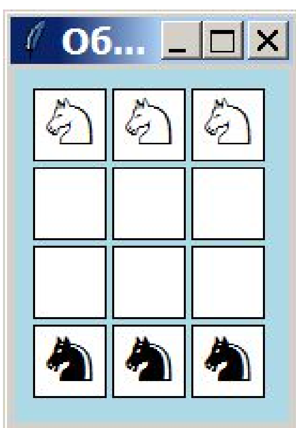
Теперь белый конь проходит по указанному выше маршруту, а после этого чёрный конь занимает место белого коня (рисунок справа).

Таким образом сначала чёрный конь освобождает дорогу белому, затем белый конь переходит на конечное поле, и, наконец, черный конь заканчивает свой маршрут.

Задания для самостоятельной работы.

1. Напишите приложение «Обмен коней», в котором реализуйте функционал игры для данного примера. При достижении требуемой позиции программа должна сообщить «Задача решена за ... ходов!»

2. Доработайте программу (если не предусмотрели эту возможность сразу) так, чтобы позиция игры включала не одну а несколько пар коней. Проверьте работу программы на следующих задачах:



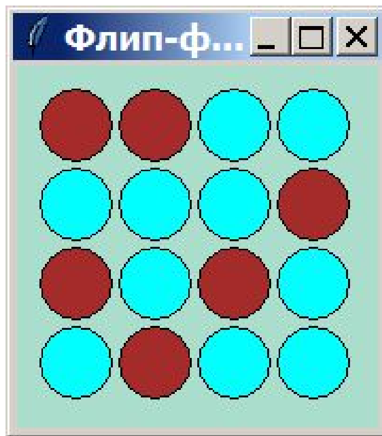
3. Добавьте заставку с правилами игры, пункт меню «Загрузка игры» и стандартный диалог выбора файла с начальной позицией игры.

4. Придумайте свой вариант начальной позиции.

26. Головоломка «Флип-флоп»

На игровой доске в виде квадрата nxn размещены двусторонние двухцветные фишки. При щелчке по фишке сама фишка и её соседи по горизонтали и вертикали переворачивается, в результате эти фишки меняют цвет. Требуется добиться, чтобы все фишки стали одного и того же цвета.

На рисунке представлен вариант начальной позиции на доске размером 4х4. Для доски такого размера найти решение несложно. Так, для указанной позиции задача решается всего за 5 ходов.



Однако не для каждой начальной позиции решение существует. Если цвета фишек выбирать случайным образом, то задача достаточно часто не будет иметь решения. Например, в том случае, когда на доске только одна фишка отличается от других по цвету. Чтобы получить позицию, которая имеет решение, поступим следующим образом: пусть сначала все фишки будут одного и того же цвета, а затем смоделируем несколько «ходов» игрока. В результате полученная позиция обязательно будет иметь решение.

В программу заложим два класса: Cell – класс фишки и Game – класс игры.

Конструктор Cell получает положение фишки и индекс её цвета в глобальном массиве colors, метод changeState «переворачивает» фишку.

```
class Cell():
   def __init__(self,i,j,s):
      x=p+j*size
      y=p+i*size
      self.s=s
      self.img=canvas.create_oval(x+2,y+2,
               x+size-2,y+size-2,fill=colors[self.s])
   def changeState(self):
      self.s=1-self.s
      canvas.itemconfig(self.img,fill=colors[self.s])
```

Конструктор Game создаёт двумерный массив фишек «нулевого» цвета». Метод newgame выполняет несколько «ходов», чтобы сгенерировать правильную позицию. При этом он вызывает метод reaction, который фактически выполняет один ход игрока. Этот же метод мы потом вызовем и в обработчике события мыши, т.е., одним методом мы «убиваем двух зайцев»!

```
class Game():
   def __init__(self):
```

```
        self.cells=[[Cell(i,j,0)
                    for j in range(n)] for i in range(n)]
    def newgame(self):
        for k in range(5):
            i=randint(0,n-1)
            j=randint(0,n-1)
            self.reaction(i,j)
    def reaction(self,i,j):
        if 0<=i<n and 0<=j<n:
            for (di,dj) in [(0,0),(0,1),(0,-1),(1,0),(-
1,0)]:
                if 0<=i+di<n and 0<=j+dj<n:
                    self.cells[i+di][j+dj].changeState()
```

Добавим в программу проверку завершения игры — метод check:

```
    def check(self):
        w=self.cells[0][0].s
        for i in range(n):
            for j in range(n):
                if self.cells[i][j].s!=w:
                    return False
        return True
```

тогда можно будет дополнить метод reaction проверкой окончания игры:

```
        if self.check():
            yes=askyesno('Задача решена!',
                         'Начать новую игру?')
            if yes: self.newgame()
            else: root.destroy()
```

Осталось написать обработчик событий мыши и главную программу:

```
def onClick(event):
    x=event.x
    y=event.y
    i=(y-p)//size
    j=(x-p)//size
    g.reaction(i,j)

root=Tk()
root.title('Флип-флоп')
canvas=Canvas(root,
width=2*p+n*size,height=2*p+n*size,bg='#aaddcc')
canvas.bind('<Button-1>',onClick)
canvas.pack()
g=Game()
```

```
g.newgame()
root.mainloop()
```

И не забудем «шапку» программы:

```
# ---- параметры ----
n=4; size=42; p=12
colors=['brown','cyan']
# -------------------
from random import *
from tkinter import *
from tkinter.messagebox import *
```

Задания для самостоятельной работы.

1. Соберите все фрагменты в единую программу, протестируйте её работу.

2. Измените цвета фишек и размер игровой доски. С увеличением размера доски возможно придётся увеличить число ходов для получения начальной позиции.

Вернёмся к задаче получения начальной позиции. Альтернативный вариант получения правильной позиции даёт метод event_generate(...), который имитирует событие без его реального выполнения. Т.е., можно имитировать нажатие клавиши на клавиатуре, щелчок или перемещение мыши. В результате программа будет выполнять действия игрока. Такая возможность может оказаться полезной, например, для создания демонстрационной версии приложения.

При вызове метода event_generate(...) мы должны указать, какое событие генерируется методом, и задать при необходимости значения ключевых параметров события.

Синтаксис метода: event_generate(событие, параметр=значение, ...).

Примеры событий.

1) нажатие клавиши «стрелка вверх»:

event_generate('<Key>', keysym='Up')

или

event_generate('<Key-Up>')

или совсем коротко

event_generate('<Up>')

2) щелчок левой кнопкой мыши в точке с координатами (120,70):

event_generate('<Button-1>', x=120, y=70)

Несмотря на кажущуюся простоту использования метода, его применение требует понимания событийной модели модуля tkinter. В следующем примере в программе создаётся главное окно root, в него добавляется холст

canvas с текстом. Затем выполнение программы приостанавливается на 5 секунд. Всё это время окно приложения на экране не отображается!

```
from tkinter import *
from time import sleep
root=Tk()
canvas=Canvas(root,width=160,height=160,bg='#aaddff')
txt=canvas.create_text(80,80,text='?',
           font=('Arial',100,'bold'))
canvas.pack()
sleep(5)
```

После завершения программы все события, связанные с созданием и отображением объектов, обрабатываются интерпретатором и окно с холстом и большим вопросительным знаком появляется на экране. Заметим, что обработка событий в конце статической части программы происходит и при отсутствии вызова метода root.mainloop().

Продолжим программу. Добавим к ней обработчик нажатия клавиши на клавиатуре, сгенерируем событие нажатия клавиши с цифрой 1

```
def onKey(event):
   canvas.itemconfig(txt,text=event.keysym)

root.bind('<Key>',onKey)
root.event_generate('<Key>',keysym='1')
```

и снова запустим программу на выполнение. Никаких изменений не произойдёт! Но если после завершения программы нажать клавишу с цифрой 1, то она тут же появится в окне приложения! Но почему? В чём разница? Отличие очевидно: в этот момент окно уже создано и оно реагирует на события клавиатуры.

Решение проблемы: нужно выполнить события-запросы на создание всех объектов как можно раньше. Используем для этого метод update, при вызове которого выполнятся все события очереди задач:

```
def onKey(event):
   canvas.itemconfig(txt,text=event.keysym)

root.update()
sleep(1)
root.bind('<Key>',onKey)
root.event_generate('<Key>',keysym='1')
```

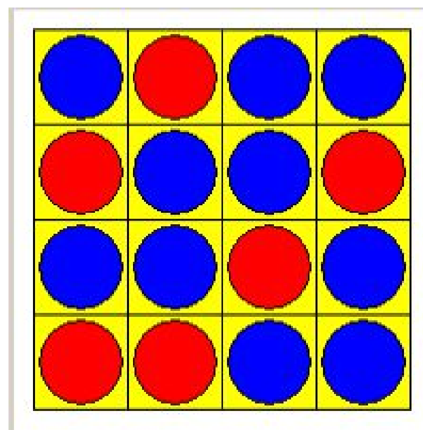
После обновления окна сделаем задержку на одну секунду, чтобы увидеть, как вопросительный знак меняется на единицу. Решение проблемы получено!

Задание для самостоятельного выполнения.
Реализуйте второй вариант получения правильной начальной позиции.

27. Головоломка «Открой холодильник»

Идея проекта позаимствована из детективной игры "Братья Пилоты. По следам полосатого слона". В этой игре в одном из эпизодов нужно открыть холодильник, для чего требуется поставить шестнадцать переключателей на дверце холодильника в одно и то же положение. Переключатели расположены друг под другом в четыре ряда, и при повороте одного из переключателей меняют своё состояние все переключатели, расположенные в том же ряду и в том же столбце, что и исходный.

На рисунке слева — холодильник с переключателями, на рисунке справа - модель игры. Каждый переключатель заменён кругом, причём положение переключателя задаётся цветом круга: горизонтальное положение — синий круг, вертикальное положение — красный круг.



В отличие от головоломки «Флип-флоп» эта головоломка имеет решение для любой начальной позиции, поэтому начальную позицию можно выбирать случайным образом.

Основание для такого вывода следующее. Существует алгоритм, который позволяет изменить положение только одного переключателя (изменить цвет только одного круга в модели задачи). Отметим мысленно этот переключатель и изменим на противоположное его состояние и состояние ещё шести переключателей, которые находятся в том же ряду или в том же столбце, что и отмеченный — и мы получим нужный результат.

Пользоваться этим алгоритмом нужно в конце решения головоломки, когда останется привести в нужное состояние один, два или три переключателя. В начале решения можно воспользоваться следующим

приёмом: можно изменить положение четырёх переключателей, которые находятся в вершинах воображаемого прямоугольника, при этом остальные переключатели вернутся в исходное состояние. Этот приём работает и для вырожденного случая, когда прямоугольник вырождается в одну линию.

Задание для самостоятельного выполнения.

Взяв за основу реализацию головоломки «Флип-флоп» разработайте приложение «Открой холодильник».

28. Проект «Тест памяти»

На экран выводится прямоугольная таблица, в некоторых клетках которой расположены кружки. После небольшой задержки кружки исчезают. Требуется указать (щелчком мыши) клетки таблицы, в которых находились кружки.

Если место кружка указано верно, в таблице появляется жёлтый кружок. Если все кружки указаны верно, то игрок получает сообщение «Вы выиграли» и предложение сыграть ещё одну игру. Если место кружка указано неверно, то в этой клетке появляется красный кружок и сообщение «Вы проиграли» и то же самое предложение.

29. Игра «Найди пару»

Обычно в этой популярной игре требуется искать пары одинаковых предметов. Оставим этот вариант для самостоятельного выполнения, а здесь реализуем его в более простом виде. Вместо пар картинок используем пары одинаковых чисел.

На игровом поле размером n строк на 2*n столбцов случайным образом размещены числа от 1 до n^2., причём каждое значение встречается два раза. При запуске игры все ячейки игрового поля закрыты квадратиками. При

щелчке на квадратике открывается спрятанное под ним число. После этого пользователь должен открыть другое число. Если оба значения одинаковы, то они удаляются с холста. Если значения разные, то эти числа снова закрываются. На рисунке показана позиция в середине игры. Две пары одинаковых чисел уже найдены и удалены с холста, два открытых числа разные, поэтому через секунду они закроются.

Чтобы реализовать этот алгоритм используем программирование с выделением состояний игры. Введём глобальную переменную state. В начале игры state = 0. Первый щелчок открывает одно из чисел, а переменной state присваивается значение 1. Следующий щелчок открывает второе число. Поскольку каждый щелчок обрабатывается одним и тем же обработчиком только знание состояния игры позволяет обработчику выполнить нужные действия.

Для реализации игры напишем класс для работы с «фишками» - элементами игрового поля. Графическое изображение фишки — квадрат с написанным на нём числом. Данные класса:

i – номер строки игрового поля;

j – номер столбца игрового поля;

k – число на фишке.

Кроме этих данных в классе используется переменная s – состояние ячейки:

s = 0 – закрытое число;

s = 1 – открытое число;

s = -1 – удалённое число.

```
class Cell:
    def __init__(self,i,j,k):
        self.i=i
        self.j=j
        self.k=k
```

```
        x=p+self.j*size
        y=p+self.i*size
        self.fon=canvas.create_rectangle(
           (x+3,y+3),(x+size-3,y+size-3),
           outline='', fill='#cccccc')
        self.num = canvas.create_text(
           (x+size//2,y+size//2),
           font=('Courier New',22,'bold'))
```

В конструкторе класса Cell создаются два геометрических объекта — квадрат с идентификатором fon и текстовая строка — число с идентификатором num. Фактически это заготовка для элементов игрового поля. Превращение заготовки в квадрат с числом происходит в методе change(self,s), где s – состояние ячейки.

```
    def change(self,s):
        self.s=s
        if self.s==0:
           canvas.itemconfig(self.fon,
              outline='black', fill='#88cc88')
           canvas.itemconfig(self.num, text='')
        if self.s==1:
           canvas.itemconfig(self.fon,
              outline='black', fill='yellow')
           canvas.itemconfig(self.num, text=str(self.k))
        if self.s==-1:
           canvas.itemconfig(self.fon,
              outline='', fill='#cccccc')
           canvas.itemconfig(self.num, text='')
```

Создание игры — метод newgame(). Каждое число появляется на игровом поле сначала в открытом виде, а через полсекунды число закрывается.

```
def newgame():
    global state, count
    count=2*n*n
    state=0
    L=[k+1 for k in range(n*n)]
    L.extend(L); shuffle(L)
    k=0
    for i in range(n):
       for j in range(2*n):
          mcell[i][j].k=L[k]
          mcell[i][j].change(1)
          canvas.update()
          sleep(0.2)
          mcell[i][j].change(0)
```

```
            k+=1
```

Обработчик событий мыши.

Расположение первого открытого числа запомним в глобальных переменных ns (номер строки) и np (номер столбца). Расположение второго открытого числа — в переменных i и j.

Если числа в открытых ячейках совпали, то обе «фишки» удаляются, т.е., переводятся в состояние s=-1. Количество фишек уменьшается на 2, это фиксируется в глобальной переменной count.

После удаления всех фишек, появляется диалоговое окно с предложением сыграть ещё одну игру. Если пользователь нажимает кнопку «Да», то программа вызывает метод newgame, который генерирует новую случайную расстановку фишек. При этом каждая фишка на короткое время появляется в открытом виде.

```
def reaction(event):
   global state, ns, np, count
   i=(event.y-p)//size
   j=(event.x-p)//size
   if 0<=i<n and 0<=j<2*n:
      if state==0 and mcell[i][j].s==0:
      # обработка первого щелчка
         mcell[i][j].change(1)
         ns=i; np=j
         state=1
      elif state==1 and mcell[i][j].s==0:
      # обработка второго щелчка
         mcell[i][j].change(1)
         canvas.update()
         #root.after(1000,None)
         sleep(1)
         if mcell[i][j].k==mcell[ns][np].k:
            mcell[i][j].change(-1)
            mcell[ns][np].change(-1)
            state=0
            count-=2
            if count==0:
               res=askyesno('Найди пару','Играть ещё?')
               if res: newgame()
               else: root.destroy()
         else:
            mcell[i][j].change(0)
            mcell[ns][np].change(0)
            state=0
```

После второго щелчка нужна задержка выполнения программы, чтобы пользователь успел увидеть открытое им число. Используем метод sleep(число_секунд) из модуля time. Перед выполнением задержки нужно обновить окно приложения (или объект “холст») вызвав метод update(). В противном случае все изменения появятся на экране только в конце работы функции.

Осталось добавить «шапку»

```
#--- параметры ---
n=3; size=50; p=12
#-----------------
from time import *
from random import *
# импорт графической библиотеки Питона
from tkinter import *
from tkinter.messagebox import *
# создание окна приложения с "холстом" для рисования
root=Tk()
root.title('Найди пару')
canvas=Canvas(root, width=2*p+2*n*size,
                    height=2*p+n*size,bg='lightgray')
canvas.pack()
```

и главную программу:

```
#------- main ------
mcell=[]
for i in range(n):
   w=[]
   for j in range(2*n):
      w.append(Cell(i,j,-1))
   mcell.append(w)

newgame()
canvas.bind('<Button-1>',reaction)
root.mainloop()
```

Задания для самостоятельной работы.

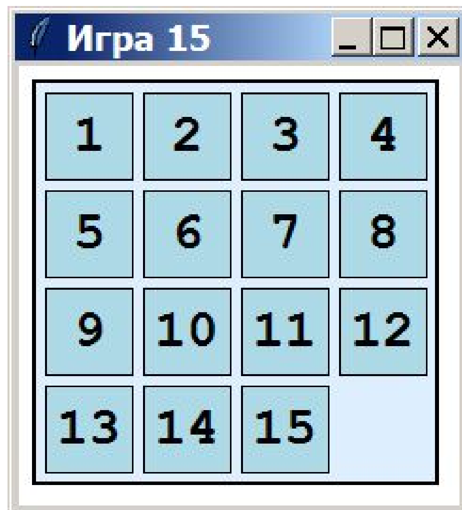
1. Вместо набора последовательных чисел используйте набор произвольных (но не повторяющихся!) значений.

2. Добавьте в программу меню выбора значения параметра n, например, из значений 3,4 или 5.

3. Добавьте возможность вычисления времени игры и выведите результат в конце игры.

30. Игра «15»

Игра 15 — игра, придуманная Самюэлем Ллойдом. Требуется, передвигая фишки, уложенные в случайном порядке в коробку, получить вот такое их расположение.



Передвинуть фишку можно только на соседнее свободное место. Вынимать фишки из коробочки нельзя.

Для реализации игры напишем класс для работы с фишками. Графическое изображение — квадрат с написанным на нём числом. Элементы данных:

i – номер строки игрового поля;

j – номер столбца игрового поля;

k – число на фишке.

Параметры i,j определяют место фишки на игровом поле. Для получения места фишки в окне приложения нам понадобится параметр размера size.

В конструкторе создадим два геометрических объекта и запомним их дескрипторы. Это даст нам возможность, используя методы канвы, перемещать фишки.

Перемещение фишек обеспечивает метод класса move.

```
#--- параметры ---
n=4; size=52; p=size//4
#-----------------
class Cell:
   # конструктор
   def __init__(self,i,j,k):
      self.i=i
      self.j=j
      self.k=k
```

```
        x=p+self.j*size
        y=p+self.i*size
        if k<n*n:
           self.fon = canvas.create_rectangle(
              (x+3,y+3),(x+size-3,y+size-3),
              outline='black',fill='lightblue')
           self.num = canvas.create_text(
              (x+size//2,y+size//2),
              font=('Courier New',22,'bold'),text=str(k))
     # перемещение фишки
     def move(self,key):
        global ns,np
        if key=='Left':
           canvas.move(self.fon,-size,0)
           canvas.move(self.num,-size,0)
           np+=1; self.j-=1
        if key=='Right':
           canvas.move(self.fon,size,0)
           canvas.move(self.num,size,0)
           np-=1; self.j+=1
        if key=='Up':
           canvas.move(self.fon,0,-size)
           canvas.move(self.num,0,-size)
           ns+=1; self.i-=1
        if key=='Down':
           canvas.move(self.fon,0,size)
           canvas.move(self.num,0,size)
           ns-=1; self.i+=1
```

В основной программе создадим набор фишек в конечной расстановке. Этот набор представляет собой двумерный массив

```
#------- main ------
mcell=[]; k=0
for i in range(n):
   w=[]
   for j in range(n):
     k+=1
     w.append(Cell(i,j,k))
   mcell.append(w)
ns=n-1; np=n-1
newgame()
root.bind('<Key>',reaction)
root.mainloop()
```

Процедура newgame «перемешивает» фишки, имитируя получение кодов клавиатуры. Перемешивание происходит по правилам игры, т.е. перемещениями фишек на соседнее свободное поле. Это сделано для того, чтобы решение задачи существовало.

Наиболее сложное — это обработка событий клавиатуры и реализация перемещений. Результат должен отражаться в массиве mcell – следует обменять местами ссылки на переставляемые экземпляры класса, и на экране — следует переместить изображение фишки на свободное место.

В главной программе свяжем событие «нажатие клавиши» и его обработчик с помощью метода bind:

```
root.bind('<Key>',reaction)
```

Функция reaction извлекает из переменной event (эта переменная хранит параметры события) символьное значение нажатой клавиши и вызывает функцию handleevent, которой и передаёт это значение:

```
def handleevent(key):
   global ns,np
   if key=='Left' and np<n-1 :
      mcell[ns][np+1],mcell[ns][np]=\
                      mcell[ns][np],mcell[ns][np+1]
      mcell[ns][np].move('Left')
   if key=='Right' and np>0:
      mcell[ns][np-1],mcell[ns][np]=\
                      mcell[ns][np],mcell[ns][np-1]
      mcell[ns][np].move ('Right')
   if key=='Up' and ns<n-1 :
      mcell[ns+1][np],mcell[ns][np]=\
                      mcell[ns][np],mcell[ns+1][np]
      mcell[ns][np].move('Up')
   if key=='Down' and ns>0:
      mcell[ns-1][np],mcell[ns][np]=\
                      mcell[ns][np],mcell[ns-1][np]
      mcell[ns][np].move('Down')
```

Замечание. Чтобы перенести на следующую строку часть длинного выражения, Python использует обратный слеш (символ \).

Вернёмся теперь к обработчику reaction:

```
def reaction(event):
   key=event.keysym
   handleevent(key)
```

Кроме вызова функции handleevent(…) обработчик проверяет, получено ли решение задачи.

```
count=0
for i in range(n):
   for j in range(n):
      if mcell[i][j].k==n*i+j+1:
         count+=1
if count==n*n:
   res=askyesno('Вы выиграли!','Играть ещё?')
   if res: newgame()
   else: root.destroy()
```

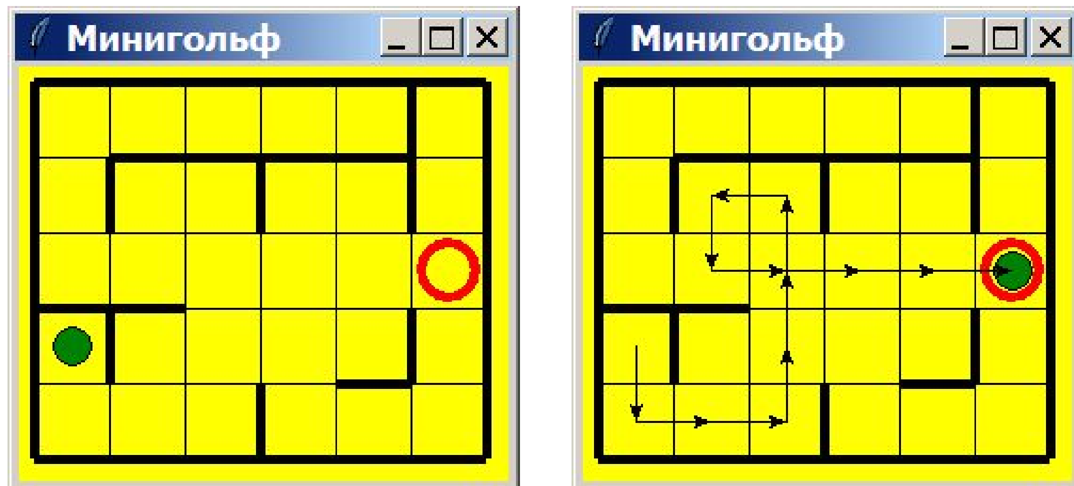
Если получена конечная позиция, то выводится системное окно с вопросом «Играть ещё?» При нажатии кнопки «Да» вызывается функция newgame(), при нажатии копки «Нет» окно закрывается и программа прекращает работу.

Задание для самостоятельной работы. Соберите вместе все фрагменты и проверьте работу программы.

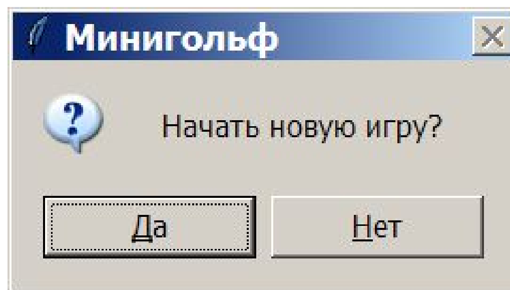
31. Игра «Минигольф»

Игра с несколько условным названием «Минигольф» предлагает пользователю загнать шарик в лунку. По шарику можно наносить удары только в четырёх фиксированных направлениях: влево, вправо, вверх и вниз. После удара шарик катится, пока не встретит препятствие на игровой доске.

На рисунке слева показана начальная позиция одного из вариантов расположения шарика и лунки. Чтобы загнать шарик в лунку нужно нанести следующую последовательность ударов: вниз, вправо, вверх, влево, вниз, вправо. Траектория движения шарика и конечная позиция игры показана на рисунке справа.



После того, как шарик попадает в лунку, выводится диалоговое окно



При нажатии клавиши Enter или кнопки «Да» меняется положение шарика и лунки на игровой доске и можно начинать новую игру. При нажатии кнопки «Нет» приложение закрывается.

Итак, приступим к разработке проекта. Спроектируем три класса.

Класс Cell – поле игровой доски;

класс Ball – шарик;

класс Target – лунка.

Любое поле игровой доски может содержать ограничивающие элементы, которые будут препятствовать движению шара. Закодируем наличие ограничений строкой из четырёх символов. Отсутствие ограничения обозначим символом '0', а его наличие — символом '1'. Зафиксируем последовательность просмотра границ поле: левая, верхняя, правая и нижняя границы. Тогда структуру полей игровой доски можно задать двумерным списком

```
walls=[('1100','0101','0101','0101','0111','1110'),
       ('1010','1100','0110','1100','0110','1010'),
       ('1001','0001','0000','0000','0000','0010'),
       ('1110','1100','0000','0000','0011','1010'),
```

```
    ('1001','0001','0011','1001','0101','0011')]
```

Заметим, что при кодировке полей мы соблюдали принцип взаимности: граница между двумя полями должна быть обоюдной.

Параметры поля:

i, j – номер строки и столбца - «координаты» поля на игровой доске;
sw – текстовая строка.ю определяющая границы поля.

Геометрически каждое поле представляет собой квадрат со сторонами — отрезками разной толщины. Толщина границы определяется по формуле

```
w=1+4*int(sw[k]) k= 0,1,2,3
```

и равна 1 пиксель при отсутствии ограничения и 5 пикселей при наличии ограничения.

```
class Cell:
   def __init__(self,i,j,sw):
      x=p+j*size
      y=p+i*size
      w=1+4*int(sw[0])           # левая граница клетки
      canvas.create_line(x,y,x,y+size,width=w)
      w=1+4*int(sw[1])           # верхняя граница клетки
      canvas.create_line(x,y,x+size,y,width=w)
      w=1+4*int(sw[2])           # правая граница клетки
      canvas.create_line(x+size,y,x+size,y+size,width=w)
      w=1+4*int(sw[3])           # нижняя граница клетки
      canvas.create_line(x,y+size,x+size,y+size,width=w)
```

В этом проекте, в отличие от предыдущего, мы не будем вводить состояние поля, а создадим два новых класса, которые описывают объекты: шарик и лунка.

```
class Ball:                  # шарик
   # конструктор
   def __init__(self,i=0,j=0):
      self.i=i
      self.j=j
      x=p+j*size
      y=p+i*size
      self.obj=canvas.create_oval(x+10,y+10,
               x+size-10,y+size-10,fill='green')
   # перемещение шарика в заданном направлении
   def move(self,di,dj):
      canvas.move(self.obj,dj*size,di*size)
      self.i+=di
```

```
        self.j+=dj
    # перемещение шарика на заданную позицию
    def setpos(self,i,j):
        canvas.move(self.obj,(j-self.j)*size,(i-
self.i)*size)
        self.i=i
        self.j=j

class Target:                # лунка
    # конструктор
    def __init__(self,i=0,j=0):
        self.i=i
        self.j=j
        x=p+j*size
        y=p+i*size
        self.obj=canvas.create_oval(x+5,y+5,
                 x+size-6,y+size-6,width=5,outline='red')
    # перемещение лунки на заданную позицию
    def setpos(self,i,j):
        canvas.move(self.obj,(j-self.j)*size,(i-
self.i)*size)
        self.i=i
        self.j=j
```

Оба конструктора содержат параметры со значениями по умолчанию, Это значит, что их можно вызывать без параметров. Создадим набор полей игровой доски из n строк и m столбцов, шарик и лунку:

```
cells=[]
for i in range(n):
   w=[]
   for j in range(m):
      w.append(Cell(i,j,walls[i][j]))
   cells.append(w)
ball=Ball()
targ=Target()
num=0
newgame()
root.bind('<Key>',reaction)
root.mainloop()
```

Так как конструкторы Ball и Target вызывались без параметров, то были использованы значения по умолчанию i=0, j=0. На самом деле нам важен сам факт создания объектов, так как реальное их положение на доске определяется в функции newgame:

```
def newgame():
```

```
    global num,ball,targ
    if num==0:
       ball.setpos(3,0)
       targ.setpos(2,5)
       num=1
    elif num==1:
       ball.setpos(2,5)
       targ.setpos(3,0)
       num=2
    elif num==2:
       ball.setpos(1,5)
       targ.setpos(0,4)
       num=0
```

При каждом обращении к функции на доске по очереди возникает одна из трёх определённых в теле функции позиций.

Напишем теперь обработчик действий игрока. Предположим, что игрок нажал стрелку влево. Положение шарика определяется его координатами ball.i, ball.j, поэтому наличие или отсутствие ограничения на левой стороне поля определяется значением символа walls[ball.i][ball.j][0]. Если он равен '0', то шарик должен переместиться на соседнее слева поле. Для этого мы вызываем метод ball.move(0,-1). При этом переменные ball.i, ball.j изменяют свои значения. К новому положения шарика применяем тот же алгоритм. Очевидно, что шарик будет двигаться влево, пока значение walls[ball.i][ball.j][0] остаётся равным '0'.

Аналогичные рассуждения справедливы и для остальных направлений.

```
def reaction(event):
    key=event.keysym
    if key=='Left':
       while walls[ball.i][ball.j][0]=='0': ball.move(0,-
1)
    if key=='Up':
       while walls[ball.i][ball.j][1]=='0': ball.move(-
1,0)
    if key=='Right':
       while walls[ball.i][ball.j][2]=='0': ball.move(0,1)
    if key=='Down':
       while walls[ball.i][ball.j][3]=='0': ball.move(1,0)
    # проверка окончания игры
    if check():
       yes=askyesno('Минигольф','Начать новую игру?')
       if yes: newgame()
       else: root.destroy()
```

После каждого перемещения шарика нужно проверить достижение конечной позиции. Для этого вызывается функция check:

```
def check():
   if ball.i==targ.i and ball.j==targ.j:
      return True
   else:
      return False
```

Осталось только написать «шапку» программы

```
from tkinter import *
from tkinter.messagebox import *
root=Tk()
root.title('Минигольф')

n=5; m=6; size=40; p=10
canvas=Canvas(root, height=2*p+n*size,
              width=2*p+m*size, bg='yellow')
canvas.pack()
```

и собрать воедино все фрагменты.

Задание для самостоятельного выполнения.

1. Соберите программу и проверьте её работу.
2. Добавьте новые варианты начальной позиции в метод newgame().
3. Измените расположение препятствий на игровой доске.

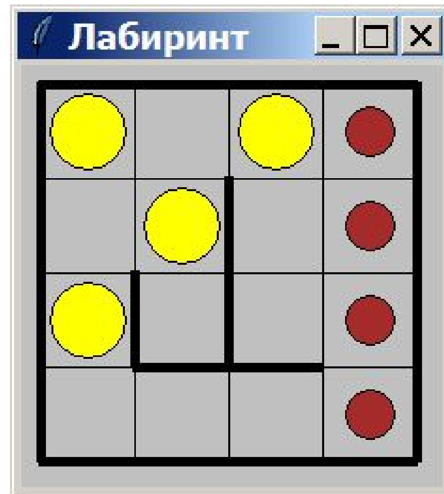
32. Проект "Шарики в лабиринте"

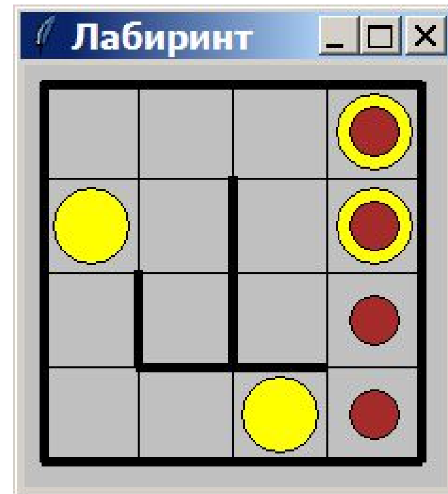
Этот проект предназначен для самостоятельного выполнения. В основу его реализации можно положить проект «Минигольф». Как и в проекте «Минигольф» игровое поле представляет собой лабиринт – прямоугольник из клеток, по границам которых расположены стены. Часть клеток занята шариками, которые могут перемещаться по лабиринту. Направление перемещения выбирается нажатием стрелки на клавиатуре. После этого все шарики начинают двигаться в заданном направлении (кроме тех, которые упираются в стенку лабиринта). Теперь можно рассмотреть два варианта дальнейших действий: более простой — после перемещения на одну клетку шарики останавливаются и на этом обработка события заканчивается), или более сложный — каждый из шариков продолжает двигаться до тех пор, пока не упрётся в в препятствие — стенку лабиринта.

Требуется поставить все шарики на отмеченные на игровом поле клетки.

Программа должна выводить сообщение «Задача решена», когда все шарики окажутся одновременно на отмеченных полях.

На следующем рисунке показана начальная позиция игры и возможная позиция в середине игры. Заметим, что в этом примере обработка нажатий стрелок была реализована по первому варианту.





Начальная позиция Позиция во время игры,

Особенности реализации. Чтобы при попадании шарика на конечное поле метка поля оставалась сверху, нужно «вытащить» метки на передний план. Для этого используется метод канвы tag_raise(...).

Задание для самостоятельного выполнения.
Напишите программу «Шарики в лабиринте».

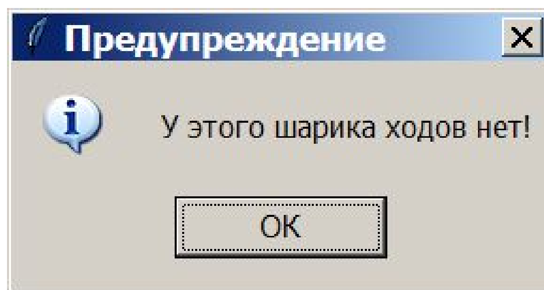
33. Игра «Столбики»

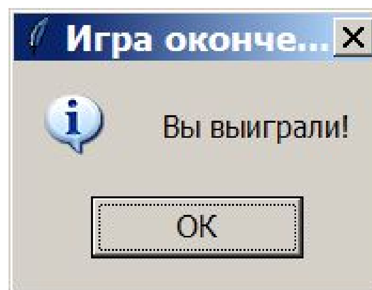
Игра «Столбики» есть клон известной игры ”Sort it”, в которой нужно рассортировать шарики по цвету. Особенность данной игры состоит в том, что шарики помещены в узкие стаканчики, так что вынуть из стаканчика можно только верхний шарик и положить его в другой такой же стакан. Значит здесь реализован основной принцип стека: последний вошёл — первый вышел, т.е. такой стаканчик с шариками есть модель стека!

Приведём внешний вид приложения:

и займёмся его разработкой.

Уточним правила игры. При щелчке левой кнопкой мыши по одному из стеков выделяется верхний элемент стека. Вторым щелчком пользователь указывает стек, куда нужно переместить шарик. Чтобы игра была не слишком простой примем, что выделенный шарик можно положить на шарик только того же цвета. Если у выделенного шарика нет ни одного хода, выводится окно с сообщением 'У этого шарика ходов нет!', после чего выделение с шарика снимается и игра продолжается.





Когда шарики рассортированы, то игра заканчивается выигрышем игрока. При этом порядок цветов в стеках и расположение пустого стека может быть произвольным.

Если создалась позиция, в которой ни у одного шарика нет ходов, то пользователь может начать игру сначала, выбрав соответствующий пункт меню.

Чтобы был понятен дальнейший код программы приведём сразу набор параметров приложения:

R – половина ширины «стаканчика»-стека;

size – ширина места для размещения стека;

n – количество стеков;

m – ёмкость стека (количество помещающихся в него шариков);
colors – список цветов шариков;
colmap – двумерный массив номеров цветов шариков (карта игры).

```
R=20; size=2.5*R
n=5; m=4
colors=['red','yellow','cyan','green']
colmap=[[ 3,-1,-1,-1, 1],
        [ 2, 3,-1, 2, 0],
        [ 0, 2, 0, 1, 3],
        [ 1, 1, 2, 3, 0]]
```

Каждый шарик есть экземпляр класса Cell.
Элементы данных:
cv – номер цвета шарика;
oval – графический объект (круг), моделирующий шарик.

Круг создаётся в конструкторе класса, который получает условные координаты круга:
i – номер строки;
j – номер столбца, на которые разделена игровая доска. Фактически это номер стека.

```
class Cell:
   def __init__(self,i,j,cv):
      self.cv=cv
      x=(j+1)*size
      y=(i+1)*2*R
      self.oval=canvas.create_oval(x-R+3,y-R+3,x+R-3,y+R-
3,

width=1,outline='black',fill=colors[cv])
```

Метод setWidth устанавливает толщину s контура. Для выделенного шарика s=5, для обычного s=1.

```
   def setWidth(self,s):
      canvas.itemconfig(self.oval,width=s)
```

Метод delete удаляет графический объект oval. Он понадобится при восстановлении начальной позиции игры.

```
   def delete(self):
      canvas.delete(self.oval)
```

Разработаем класс Stack. В практике программирования стек обычно реализуется как класс с набором следующих методов:
push(значение) — поместить значение в стек;
top() — прочесть значение верхнего элемента стека;

pop() — удалить верхний элемент стека и вернуть значение элемента;

isEmpty() — метод возвращает True, если стек пуст.

Значения элементов могут быть любыми — числами, строками, списками, экземплярами других классов и т.п. В нашем проекте все значения являются экземплярами класса Cell.

Кроме этих методов напишем следующие:

notEmpty() — метод возвращает True, если стек не пуст.

notFull() — метод возвращает True, если стек пуст или не полностью заполнен.

select() — выделяет верхний элемент стека;

unselect() — снимает выделение с верхнего элемента стека;

clear() — удаляет все элементы стека;

place() — вычисляет координаты прямоугольной области нового положения шарика.

Основой стека является список data. В конструкторе стека создаётся пустой список, метод push добавляет значение в конец списка, pop удаляет последний элемент списка, top реализует доступ к последнему элементу. Эти методы используют встроенные возможности для работы со списками.

```
class Stack:
   def __init__(self,j):
      self.j=j
      self.data=[]
      x=(j+1)*size
      self.fon=canvas.create_rectangle(x-
R,R,x+R,(2*m+1)*R,
               fill='#da80ff',width=1)
# проверки: список пустой, непустой, незаполненный
   def isEmpty(self): return len(self.data)==0
   def notEmpty(self): return len(self.data)>0
   def notFull(self): return len(self.data)<m
# основные операции со стеком
   def push(self,cell):
      if self.notFull(): self.data.append(cell)
      else: raise Exception('Переполнение стека.')
   def pop(self):
      if self.notEmpty(): return self.data.pop()
      else: raise Exception('Выборка из пустого стека.')
   def top(self):
      if self.notEmpty(): return self.data[-1]
      else: raise Exception('Выборка из пустого стека.')
# отметка и снятие отметки с шарика
   def select(self): self.top().setWidth(5)
   def unselect(self): self.top().setWidth(1)
```

```
# удаление элементов стека
   def clear(self):
      while self.notEmpty(): self.pop().delete()
# вычисление места нового положения шарика
   def place(self):
      k=len(self.data)
      x=(self.j+1)*size
      y=(m-k)*2*R
      return (x-R+3,y-R+3,x+R-3,y+R-3)
# ================ end Stack ==================
```

При извлечении из стека или добавлении в стек элемента могут возникнуть следующие ошибочные ситуации:

А) попытка извлечь элемент из пустого стека;

Б) попытка добавить элемент в заполненный стек ограниченного размера.

Python в этом случае прекращает выполнение программы и выводит информацию о том в каком месте и какого типа ошибка произошла. Однако в этой информации не так просто разобраться, поэтому программист может воспользоваться механизмом исключений, чтобы вывести своё сообщение.

Итак, необходимые для создания приложения классы уже есть. Можно писать логику игры. Но мы попробуем создать класс Game, который инкапсулирует всю функциональность приложения.

Данные класса:

game – список стеков;

state – состояние игры:

state = 0 – нет выделенного шарика;

state = 1 – есть выделенный шарик.

Конструктор получает параметр colmap – карту игры, создаёт столбики-стеки и заполняет их шариками — экземплярами класса Cell в соответствии с этой картой.

```
class Game:
   def __init__(self,colmap):
      self.game=[]
      for j in range(n):
         S=Stack(j)
         self.game.append(S)
      for j in range(n):
         for i in range(m-1,-1,-1):
            cv=colmap[i][j]
            if cv>=0: self.game[j].push(Cell(i,j,cv))
      self.state=0
      canvas.bind('<Button-1>',self.reaction)
```

Кроме того конструктор обнуляет переменную класса state — состояние игры и связывает событие нажатия левой кнопки мыши с его обработчиком, который является методом класса Game. В этом случае функция-обработчик получает два параметра:

self – ссылка на экземпляр класса;

event – информация о событии.

Первое, что должна сделать функция-обработчик — выяснить, по какому стеку был сделан щелчок. Для этого нужно получить координаты прямоугольной области, занятой каждым из стеком и проверить условие попадания точки щелчка в прямоугольник.

Второе — если состояние игры state==0, то выделить верхний элемент выбранного стека (если, конечно он не пуст). Если у выделенного шарика есть ход, то извлекаем его из стека и устанавливаем состояние игры state=-1, иначе выводим предупреждение и снимаем выделение с шарика.

Третье — если состояние игры state==-1, то это значит, что пользователь щёлкнул по стеку, куда нужно поместить выделенный шарик. Это можно сделать, или если этот стек пуст или выполняется следующий набор условий: в стеке есть свободное место и цвет верхнего элемента стека совпадает с цветом выделенного шарика.

Четвёртое — добавить шарик в стек и отобразить это изменение на экране. После этого установить состояние игры state=0.

Пятое — проверить условие окончания игры и, если игра окончена, вывести соответствующее сообщение.

Уже из этого перечня понятно, что функция будет большой и сложной. Чтобы упростить написание функции вынесем некоторые фрагменты кода в отдельные методы класса:

canmove – возвращает True, если у выделенного шарика есть ход;

check – возвращает True если все шарики удалось рассортировать по цветам.

```
def canmove(self,S):
   color=S.top().cv
   for w in self.game:
      if w!=S and (w.isEmpty() or
                  w.notFull() and w.top().cv==color):
         return True
   return False
# ------------------
def check(self):
   w=[]
   for S in self.game:
      w.append(len(S.data))
   if w.count(m)!=n-1: return False
```

```
        for S in self.game:
           if S.notEmpty():
              color=S.top().cv
              for cell in S.data:
                 if cell.cv!=color:
                    return False
        return True
```

Приведём текст функции-обработчика. Особенность этой функции в том, что при выполнении ОДНОГО хода она вызывается дважды. При первом вызове определяется выделенный шарик, при втором — он перемещается на новое место. Следовательно, информация о том, какой шарик является главным действующим лицом должна сохраняться МЕЖДУ вызовами. Кстати, то же самое относится и к переменной state – состоянию игры. Но поскольку переменная state есть переменная класса, то она сохраняется в экземпляре класса, который существует всё время. Для переменной cell, в которой хранится информация о шарике, применим другой подход — объявим её глобальной переменной. Глобальная переменная, в отличие от локальной, существует всё время работы приложения, поэтому будет сохранять своё значение между вызовами.

```
     def reaction(self,event):
        global cell
        x=event.x
        y=event.y
        for S in self.game:
           (x1,y1,x2,y2)=canvas.coords(S.fon)
           if x1<x and x<x2 and y1<y and y<y2 :
              if self.state==0 and S.notEmpty() :
                 S.select()
                 if self.canmove(S) :
                    cell=S.pop()
                    self.state=-1
                 else:
                    showinfo('Предупреждение',
                             'У этого шарика ходов нет!')
                    S.unselect()
                    return
              elif self.state==-1 and (S.isEmpty()
                   or S.notFull() and cell.cv ==
  S.top().cv):
                 new=S.place()
                 canvas.coords(cell.oval,new)
                 S.push(cell)
                 S.unselect()
```

```
                self.state=0
                if self.check():
                   showinfo('Игра окончена','Вы
выиграли!')
                   return
```

Последний метод класса — восстановление начальной позиции.

```
   def restore(self,colmap):
      for j in range(n):
         self.game[j].clear()
         for i in range(m-1,-1,-1):
            cv=colmap[i][j]
            if cv>=0: self.game[j].push(Cell(i,j,cv))
```

Осталось добавить в приложение меню и главную программу

```
menubar=Menu()
root.config(menu=menubar)
menubar.add_command(label='Начать сначала',
                    command=lambda x=colmap:
g.restore(x))
...
#------ main -----
g=Game(colmap)
root.mainloop()
```

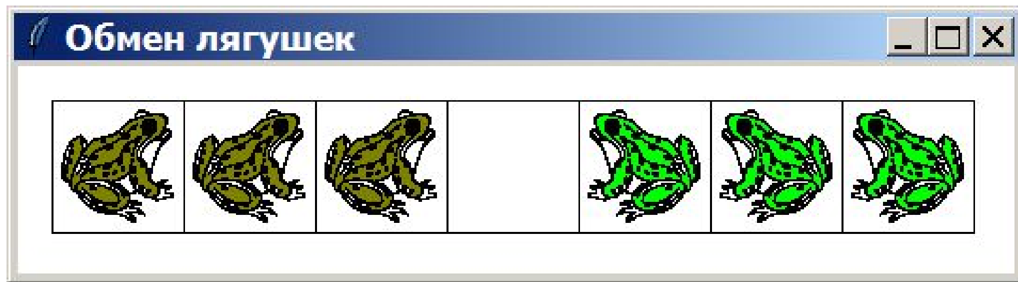
Задания для самостоятельного выполнения.

1. Соберите все фрагменты кода и решите предложенную задачу.
2. Придумайте новую задачу такого же уровня сложности и предложите решить её своим друзьям.

34. Игра «Обменяй лягушек».

Класс Canvas модуля Tkinter может работать с изображениями. Правда, из популярных форматов поддерживается только формат GIF.

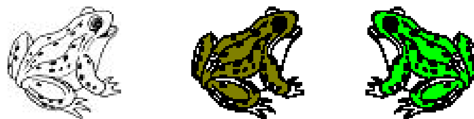
Реализуем на Питоне известную игру, в которой требуется поменять местами коричневых и зелёных лягушек.



Любая лягушка может передвигаться только вперёд, т.е. коричневые лягушки — только вправо, а зелёные — только влево. Лягушка может прыгнуть на свободное поле, расположенное на одно или два поля впереди. Требуется поменять местами коричневых и зелёных лягушек.

Вряд ли у пользователя получится решить задачу с первого раза, поэтому стоит предусмотреть возможность восстановления начальной позиции игры без её перезагрузки.

Итак, сначала получим два изображения лягушек. Проще всего взять черно-белый рисунок, в графическом редакторе сделать зеркальный вариант и раскрасить полученные рисунки в разные цвета.



Сохраним коричневую и зелёную лягушку в файлах frog-1.gif и frog-2.gif. Поместим эти файлы в каталог с программой на Питоне. В этом случае для доступа к файлу достаточно будет указать только его имя.

```
img1=PhotoImage(file='frog-1.gif')
img2=PhotoImage(file='frog-2.gif')
```

Разместим лягушек на игровой доске. Исходное и изменяющееся в процессе решения размещение поместим в переменную state, а требуемое конечное — в переменную result:

```
state  = [1,1,1,0,-1,-1,-1]
result = [-1,-1,-1,0,1,1,1]
```

Зададим параметры, определяющие внешний вид игровой доски:

n – количество полей в строке;
size – размер поля;
margin – величина отступа от рамки окна приложения.

```
n=7; size=70; margin=20
```

Заполним массив mcell

```
mcell=[]
for j in range(n):
   x=margin+j*size; y=margin
   canvas.create_rectangle(x,y,x+size,y+size,fill='')
   if state[j]==1:
      frog = canvas.create_image(x+5,y+5,
                                 anchor='nw',image=img1)
   if state[j]==-1:
      frog = canvas.create_image(x+5,y+5,
                                 anchor='nw',image=img2)
   if state[j]==0:
      frog = canvas.create_rectangle(
                    x,y,x+size,y+size,fill='')
   mcell.append((j,frog))
```

Каждый элемент массива содержит номер поля и идентификатор созданного графического примитива. Номер поля необходим для восстановления начального состояния игры, идентификатор — для вывода изображения. Для упрощения алгоритма работы пустая клетка тоже содержит идентификатор frog, связанный с прямоугольной рамкой вокруг поля.

Для перемещения лягушки достаточно щелкнуть по ней левой кнопкой мышки. В обработчике события проверяем, есть ли рядом с лягушкой или через поле от неё свободное место. Направление проверки (влево или вправо от лягушки) зависит от статуса лягушки — от значения элемента массива state.

Напишем обработчик щелчка мыши. Сначала определяем поле щелчка и отсеиваем случаи щелчка вне игровой доски.

```
def reaction(event):
   j=(event.x-margin)//size
   if j<0 or j>n-1: return
```

Извлекаем идентификатор лягушки из массива mcell

```
   frog=mcell[j][1]
```

Определяем, есть ли в пределах доступности свободное поле, и если есть, переставляем на него лягушку. После этого проверяем, получено ли требуемое расположение лягушек:

```
   k=-1
   if state[j]==-1:
      if j-1>=0 and state[j-1]==0: k=j-1
      elif j-2>=0 and state[j-2]==0: k=j-2
      if k!=-1:
         # переставляем лягушку
```

```
            canvas.move(frog,(k-j)*size,0)
            mcell[j], mcell[k] = mcell[k], mcell[j]
            state[j], state[k] = state[k], state[j]
            if state==result:
               showinfo('Обмен лягушек','Задача решена!')
         return
      if state[j]==1:
         if j+1<n and state[j+1]==0: k=j+1
         elif j+2<n and state[j+2]==0: k=j+2
         if k!=-1:
            # переставляем лягушку
            canvas.move(frog,(k-j)*size,0)
            mcell[j], mcell[k] = mcell[k], mcell[j]
            state[j], state[k] = state[k], state[j]
            if state==result:
               showinfo('Обмен лягушек','Задача решена!')
         return
```

Изменение положение лягушки на экране выполняет метод move. Чтобы изменить данные в памяти, обмениваем содержимое элементов с индексами j и k в массивах mcell и state.

Осталось написать обработчик клавиши Escape:

```
def newgame(event):
   global state, mcell
   state  = [1,1,1,0,-1,-1,-1]
   reset = n*[None]
   for k in range(n):
      j=mcell[k][0]
      canvas.move(mcell[k][1], (j-k)*size,0)
      reset[j]=mcell[k]
   mcell=reset
```

Массивы state и mcell объявлены глобальными, чтобы иметь возможность изменить их элементы. Для восстановления используется вспомогательный массив reset.

Алгоритм восстановления^

1) Из каждого элемента массива mcell извлекаем первоначальный номер поля: j=mcell[k][0]

2) Возвращаем картинку на место:

```
canvas.move(mcell[k][1], (j-k)*size,0)
```

3) Копируем данные в «правильный» элемент массива reset:

```
reset[j]=mcell[k]
```

4) Вспомогательный массив reset переписываем в рабочий массив mcell:

```
mcell=reset
```

Полный текст программы

```
#--- параметры ---
n=7; m=1; size=70; margin=20
#-----------------
# импорт графической библиотеки Питона
from tkinter import *
from tkinter.messagebox import *
# создание окна приложения с "холстом" для рисования
root=Tk()
root.title('Обмен лягушек')
canvas=Canvas(root, width=2*margin+n*size,
height=2*margin+m*size, bg='white')
canvas.pack()

def reaction(event):
   j=(event.x-margin)//size
   if j<0 or j>n-1: return
   frog=mcell[j][1]
   k=-1
   if state[j]==-1:
      if j-1>=0 and state[j-1]==0: k=j-1
      elif j-2>=0 and state[j-2]==0: k=j-2

      if k!=-1:
         # переставляем лягушку
         canvas.move(frog,(k-j)*size,0)
         mcell[j], mcell[k] = mcell[k], mcell[j]
         state[j], state[k] = state[k], state[j]
         if state==result:
            showinfo('Обмен лягушек','Задача решена!')
      return
   if state[j]==1:
      if j+1<n and state[j+1]==0: k=j+1
      elif j+2<n and state[j+2]==0: k=j+2
      if k!=-1:
         # переставляем лягушку
         canvas.move(frog,(k-j)*size,0)
         mcell[j], mcell[k] = mcell[k], mcell[j]
         state[j], state[k] = state[k], state[j]
         if state==result:
            showinfo('Обмен лягушек','Задача решена!')
```

```
        return

def newgame(event):
   global state, mcell
   state  = [1,1,1,0,-1,-1,-1]
   reset = n*[None]
   for k in range(n):
      j=mcell[k][0]
      canvas.move(mcell[k][1], (j-k)*size,0)
      reset[j]=mcell[k]
   mcell=reset

img1=PhotoImage(file='frog-1.gif')
img2=PhotoImage(file='frog-2.gif')
state  = [1,1,1,0,-1,-1,-1]
result = [-1,-1,-1,0,1,1,1]

mcell=[]
for j in range(n):
   x=margin+j*size; y=margin
   canvas.create_rectangle(x,y,x+size,y+size,fill='')
   if state[j]==1:
      frog =
canvas.create_image(x+5,y+5,anchor='nw',image=img1)
   if state[j]==-1:
      frog =
canvas.create_image(x+5,y+5,anchor='nw',image=img2)
   if state[j]==0:
      frog =
canvas.create_rectangle(x,y,x+size,y+size,fill='')
   mcell.append((j,frog))

canvas.bind('<Button-1>',reaction)
root.bind('<Escape>',newgame)
root.mainloop()
```

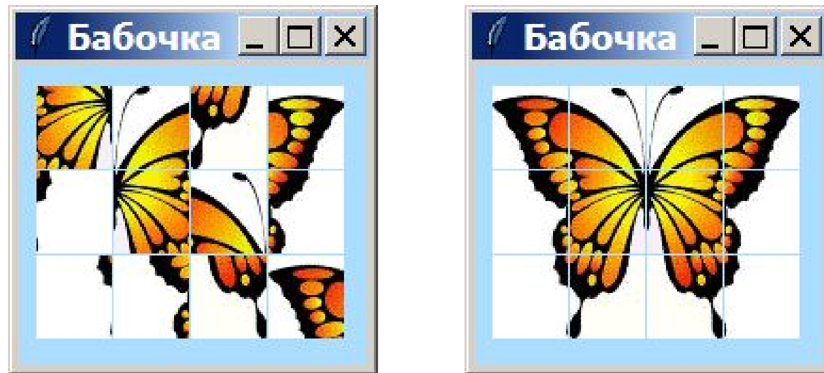
35. Игра «Собери картинку»

Суть игры — собрать картинку из отдельных фрагментов, расположенных в беспорядке на канве. Для простоты исходная картинка разбивается на одинаковые прямоугольные части, которые образуют на канве прямоугольную сетку. Пользователь может обменять два любых фрагмента, выделив щелчком мыщки один и указав также щелчком второй из них.

Создание проекта потребует некоторой подготовительной работы. Нужно подобрать картинку, «разрезать» её на части и каждую часть сохранить в отдельный файл (в интернете есть соответствующие сервисы).

Для вывода изображения на холст используем класс PhotoImage. Он позволяет загрузить рисунок в формате .gif из файла, а метод канвы create_image – разместить его на холсте.

На рисунке ниже приведен пример работы программы: слева — случайное начальное положение фрагментов рисунка, справа — конечное положение.



В этом примере 12 фрагментов рисунка находятся в файлах bf_01.jpg, bf_02.jpg, ..., bf_12.jpg в каталоге D:/BF. Размеры каждого фрагмента: 40 пикселей по горизонтали на 44 пикселя по вертикали.

Первая проблема, которую мы должны решить — определить структуру данных для хранения информации о фрагментах картинки. В предыдущем проекте мы объявляли класс и создавали двумерный массив экземпляров класса. В этом проекте продемонстрируем более простой подход — хранение данных в двумерном массиве списков. Так как двумерный массив есть список списков, то, строго говоря, наша структура является списком списков списков, но для большей ясности будем говорить о двумерном массиве списков.

Список данных каждого фрагмента:

num – номер фрагмента. Фрагменты нумеруются сквозной нумерацией по строкам, начиная с 1. Номер фрагмента совпадает с числом в имени файла;

part – указатель на изображение. Указатель не может быть локальной переменной. В нашем случае он будет создаваться в главной программе, что автоматически сделает его глобальной переменной;

img – идентификатор изображения (графического объекта), возвращаемый функцией create_image;

rect – идентификатор рамки для выделения фрагмента, возвращаемый функцией create_rectangle.

Глобальные параметры:

n – количество фрагментов по горизонтали;

m – количество фрагментов по вертикали;

xsize, ysize – ширина и высота места для размещения фрагмента — ячейки сетки. Для визуального разделения фрагментов эти параметры на один пиксель больше соответствующих размеров самого фрагмента;

xp – отступ от границ рабочей области по горизонтали;

yp – отступ от границ рабочей области по вертикали.

Разработку проекта начнём с написания главной программы.

1) получаем случайную последовательность kmap номеров фрагментов изображения:

```
kmap=list(range(1,n*m+1))
shuffle(kmap)
```

2) создаём структуру данных cells:

```
cells=[]; k=0
for i in range(m):
   w=[]
   y=yp+i*ysize
   for j in range(n):
      num=kmap[k]
      x=xp+j*xsize
      name = 'D:/BF/bf_{:02}'.format(num)+'.gif'
      part = PhotoImage(file=name)
      img=canvas.create_image(x+1,y+1,
                           anchor=NW, image=part)
      rect=canvas.create_rectangle(x+3,y+3,
                            x+xsize-3,y+ysize-3,fill='',
                            outline='red',width=0)
      w.append([num,part,img,rect])
      k+=1
   cells.append(w)
```

3) обнуляем состояние игры

```
state=0
```

4) связываем событие мыши с обработчиком

```
canvas.bind('<Button-1>',reaction)
```

Заметим, что для переносимости программы, которая обращается к файлам данных, её нужно поместить в тот же каталог, что и файлы данных, тогда в программе можно указать только имя файла. В нашем проекте в таком случае достаточно будет написать

```
    name = 'bf_{:02}'.format(num)+'.gif'
```

Доступ к данных структуры cells для фрагмента, который находится в i-ой строке и j-ом столбце будет осуществляться с помощью выражений:

`cells[i][j][0]` – номер фрагмента num;

`cells[i][j][1]` – указатель part на фрагмент изображения;

`cells[i][j][2]` – идентификатор img фрагмента изображения;

`cells[i][j][3]` – идентификатор rect прямоугольной рамки.

Заметим, что если первые два индекса имеют очевидный геометрический смысл, то значения третьего индекса не имеет логической связи с перечисленными данными. Чтобы такая связь появилась определим именованные константы

```
NUM=0; PART=1; IMG=2; RECT=3
```

и используем их вместо числовых значений, т.е.,

`cells[i][j][NUM]` – номер фрагмента num;

`cells[i][j][PART]` – указатель part на фрагмент изображения и т.д.

Следующим напишем обработчик события мыши — функцию reaction:

```
def reaction(event):
   global state,ns,np
   x=event.x
   y=event.y
   if state==0:
   # выделяем фрагмент картинки
      np=(x-xp)//xsize
      ns=(y-yp)//ysize
      if 0<=np<n and 0<=ns<m:
         select(cells[ns][np])
         state=-1
   else:
   # меняем местами два фрагмента
      j=(x-xp)//xsize
      i=(y-yp)//ysize
      if 0<=j<n and 0<=i<m:
         unselect(cells[ns][np])
         change(cells[ns][np],cells[i][j])
   # и проверяем окончание игры
         if check():
            showinfo('Игра окончена','Задание
выполнено!')
         state=0
```

Прокомментируем порядок действий более подробно.

1) Определяем координаты указателя мыши
2) Если состояние игры state == 0 то
    определяем расположение фрагмента на сетке,
    сохраняем его в глобальных переменных ns и np,
    выделяем фрагмент,
    устанавливаем состояние игры state = -1
3) Если состояние игры state == -1 то
    определяем расположение фрагмента на сетке,
    сохраняем его в локальных переменных i и j,
    снимаем выделение,
    меняем местами два фрагмента,
    проверяем окончание игры,
    устанавливаем состояние игры state = 0

Заметим, что в обработчике присутствуют вызовы пока не написанных функций. Их назначение:

функция select(A) выделяет фрагмент A;
функция unselect(A) снимает выделение с фрагмента A;
функция change(A, B) – обменивает местами фрагменты A и B
функция check() проверяет окончание игры.

Напишем первые две функции:

```
def select(A):
   canvas.itemconfig(A[RECT],width=5)
def unselect(A):
   canvas.itemconfig(A[RECT],width=0)
```

Напомним, что A[RECT] есть элемент списка A с индексом 3 – идентификатор прямоугольной рамки. Функция select устанавливает ширину этой рамки в 5 пикселей, а функция unselect – в 0 пикселей.

Для обмена фрагментов необходимо обменять значения параметров num и part этих фрагментов. Кроме того нужно отобразить эти изменения на экране.

```
def change(A,B):
# обмен параметров фрагментов
   A[NUM], B[NUM]= B[NUM], A[NUM]
   A[PART],B[PART]=B[PART],A[PART]
# отображение изменений на экране
   canvas.itemconfig(A[IMG],image=A[PART])
   canvas.itemconfig(B[IMG],image=B[PART])
```

Проверка окончания игры. Функция check() сравнивает номер фрагмента num с занимаемым им местом. Фрагмент находится на своём месте, если выполняется равенство num==n*i+j+1. Если хотя бы для одного фрагмента

это равенство не выполнено, функция возвращает значение False, иначе функция возвращает значение True.

```
def check():
   for i in range(m):
      for j in range(n):
         if cells[i][j][NUM]!=n*i+j+1: return False
   return True
```

Полностью текст программы приведен в приложении 4.

36. Игра «Сложи узор»

В этом проекте мы научимся перемещать мышкой графические объекты с использованием мыши. Начнём с небольшого примера.

Поместим в центр канвы небольшой кружок. При нажатии на левую кнопку мыши выделим его, например, увеличив толщину контура. При движении мыши с нажатой левой кнопкой будем перемещать круг вслед за указателем мыши. При отпускании кнопки мыши будем снимать выделение.

Таким образом, нам нужно написать обработчики следующих событий:

<ButtonPress-1> – нажатие левой кнопки мыши. Так как это наиболее часто встречающееся при работе с приложением событие, оно имеет несколько синонимов, в том числе <Button-1> и даже просто <1>,

<B1-Motion> - движение мыши с нажатой левой кнопкой.

<ButtonRelease-1> – отпускание левой кнопки мыши.

Для реального перемещения объекта можно использовать два разных метода канвы:

move(объект, перемещение_по_X, перемещение_по_Y)

coords(объект, новые_координаты_объекта)

Однако метод coords неприменим к объектам сложной структуры, например, к полигонам, поэтому воспользуемся методом move. Здесь могут возникнуть сложности с определением величины перемещений, так как для этого нам нужно знать предыдущее положение указателя мыши.

Введем глобальные переменные xm, ym, в которых запомним координаты указателя мыши в момент нажатия левой кнопки мыши. Напишем обработчик события <ButtonPress-1>.

```
def select(event):
   global xm,ym
   xm=event.x
   ym=event.y
   canvas.itemconfig(oval,width=3)
```

При движении мыши с определённой частотой возникает событие <B1-Motion>. В обработчике этого события определяем текущие координаты

мыши и находим их разность с предыдущим положением — значениями глобальных переменных xm,ym. После этого в глобальных переменных запоминаем текущие координаты указателя мыши.

```
def move_oval(event):
   global xm,ym
   x=event.x
   y=event.y
   canvas.move(oval,x-xm,y-ym)
   xm=x; ym=y
```

Выделение снимем в обработчике события <ButtonRelease-1>:

```
def unselect(event):
   canvas.itemconfig(oval,width=1)
```

Чтобы привязать события к конкретному графическому объекту, используем метод tag_bind.

```
canvas.tag_bind(oval,'<Button-1>',select)
canvas.tag_bind(oval,'<B1-Motion>',move_oval)
canvas.tag_bind(oval,'<ButtonRelease-1>',unselect)
```

Здесь oval – это идентификатор объекта, который возвращает метод create_oval.

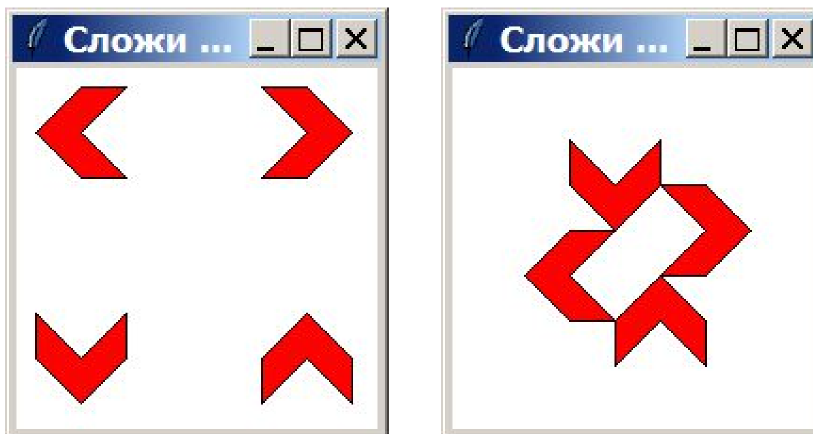
Приведём текст программы полностью.

```
from tkinter import *
# Создаём окно
root = Tk()
# Создаём и размещаем холст
canvas = Canvas(root, width=250, height=180, bg='white')
canvas.pack()

oval = canvas.create_oval(50, 50, 100, 100, fill='red')

def move_oval(event):
   global xm,ym
   x=event.x
   y=event.y
   canvas.move(oval, x-xm, y-ym)
   xm=x; ym=y

def select(event):
   global xm,ym
   xm=event.x
   ym=event.y
   canvas.itemconfig(oval,width=3)
```

```
def unselect(event):
   canvas.itemconfig(oval,width=1)

# Связываем объект на холсте с событием и действием
canvas.tag_bind(oval, '<Button-1>', select)
canvas.tag_bind(oval, '<B1-Motion>', move_oval)
canvas.tag_bind(oval, '<ButtonRelease-1>', unselect)

# Запускаем цикл исполнения
root.mainloop()
```

Перейдём теперь к разработке игры «Сложи узор».

Создадим в приложении четыре элемента узора и выведем их на экран (рисунок слева). Перемещая полигоны мышкой, пользователь может сложить из этих элементов узор «по своему вкусу», например такой, как на рисунке справа.



Для перемещения элемента используем тот же подход, что и в тестовом примере. Однако, вместо одного графического объекта у нас их четыре. Как же узнать, какой из объектов мы перемещаем? Оказывается, для указания на этот объект нужно передать методу move значение CURRENT (или 'current').

Итак, напишем метод select, в котором запомним положение указателя мыши в момент щелчка и выделим полигон:

```
def select(event):
   global xm,ym
   xm=event.x
   ym=event.y
   canvas.itemconfig(CURRENT,width=3)
```

Метод move_obj получает текущие координаты мыши и сдвигает текущий графический объект — выбранный элемент узора. После этого запоминаем новое положение мыши:

```
def move_obj(event):
   global xm,ym
   x=event.x
   y=event.y
   canvas.move(CURRENT,x-xm,y-ym)
   xm=x; ym=y
```

При отпускании кнопки мыши вызывается обработчик события ButtonRelease-1. Можно просто снять выделение, как это делалось в тестовом примере, однако мы решим более сложную задачу. Чтобы повысить точность позиционирования элементов, применим следующий приём. Представим, что вся поверхность холста равномерно покрыта «магнитными» точками, к которым будет прилипать начальная вершина полигона (а значит, и сам полигон) после перемещения.

Последовательность действий:

1) Получаем текущие координаты выделенного полигона. Заметим, что при перемещении объекта его координаты изменяются так, чтобы соответствовать реальному положению объекта.

```
coords=canvas.coords(CURRENT)
```

2) Метод coords возвращает список координат без разделения на пары x,y, поэтому извлекаем из списка первые два значения:

```
(x,y)=coords[0:2]
```

3) Определяем значения сдвигов dx,dy к ближайшей «магнитной» точке. Для этого нам потребуется знать шаг «магнитной» сетки, который будем хранить в глобальной переменной unit.

```
dx=round(x/unit)*unit-x
dy=round(y/unit)*unit-y
```

Для этой же цели можно воспользоваться методами канвы canvasx и canvasy. Эти методы получают x-овую / y-овую координату точки канвы и возвращают координату ближайшей к ней точки сетки. Шаг сетки (параметр gridspacing) задаётся отдельно для каждой функции.

```
dx=canvas.canvasx(x,unit)-x
dy=canvas.canvasy(y,unit)-y
```

Первый подход требует от программиста умения применять простые средства для решения задачи, второй — знания возможностей библиотеки tkinter. Оба подхода имеют свои плюсы и минусы и выбор одного из них будет определяться совокупностью знаний и умений программиста.

4) Корректируем положение выделенного объекта и снимаем выделение:

```
canvas.move(CURRENT,dx,dy)
canvas.itemconfig(CURRENT,width=1)
```

Заметим, что выделять объект необязательно, так как он и так выделен наличием на нём указателя мыши.

Приведём полностью обработчик:

```
def unselect(event):
   coords=canvas.coords(CURRENT)
   (x,y)=coords[0:2]
   dx=round(x/unit)*unit-x
   dy=round(y/unit)*unit-y
   canvas.move(CURRENT,dx,dy)
   canvas.itemconfig(CURRENT,width=1)
```

Напишем основную часть программы и прокомментируем основные моменты.

1) Создаём шаблон узора — список вершин W и сразу его увеличиваем, умножая на коэффициент масштабирования size:

```
W=[(0,0),(1,1),(0,1),(-1,0),(0,-1),(1,-1)]
W=mlt(W,size)
```

2) Определяем точки привязки элементов узора на холсте:

```
SL=[(1.5*size,1.5*size),(1.5*size,6.5*size),
    (6.5*size,1.5*size),(6.5*size,6.5*size)]
```

3) Используя процедуры переноса и поворота списка точек из модуля geom_sys создаём четыре полигона — элементы узора. Каждый элемент «привязываем» к точке, указанной в списке SL. Одновременно с каждым созданным элементом связываем обработчики событий мыши. Заметим, что идентификаторы создаваемых полигонов сразу же используются в методе tag_bind, поэтому их можно передавать через одну и ту же переменную poly.

```
for S in SL:
   elem=move(S,W)
   poly=canvas.create_polygon(elem,fill='red',
                              outline='black')
   canvas.tag_bind(poly,'<Button-1>',select)
   canvas.tag_bind(poly,'<B1-Motion>',move_obj)
   canvas.tag_bind(poly,'<ButtonRelease-1>',unselect)
   W=turn((0,0),W,90)
```

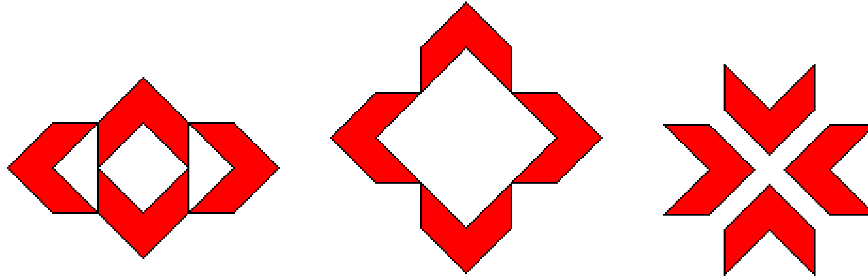
Осталось привести «шапку» приложения, в которой, кроме стандартных действий зададим коэффициент size, определяющий размер элементов узора и значение unit – шаг сетки «магнитных» точек. Такой подход позволяет пользователю установить наиболее комфортные для него параметры приложения.

```
from geom_sys import *
# импорт графической библиотеки Питона
```

```
from tkinter import *
# создание окна приложения с "холстом" для рисования
root=Tk()
root.title('Сложи узор')
size=24; unit=8
canvas=Canvas(root,width=8*size,height=8*size,bg='white')
canvas.pack()
```

Задания для самостоятельной работы.

1. Соберите вместе фрагменты программы и сложите следующие узоры:



2. Добавьте к набору элементов круг другого цвета и сложите новые узоры.

37. Проект «График функции»

Добавим немного математики в наши логические игры. Одним из преимуществ графического представления функций является наглядность. Особенно это касается случая параметрического задания функции. В этом случае вводится параметр, от которого зависят x- и y-координаты точек графика:

```
x=X(t)
y=Y(t)
```

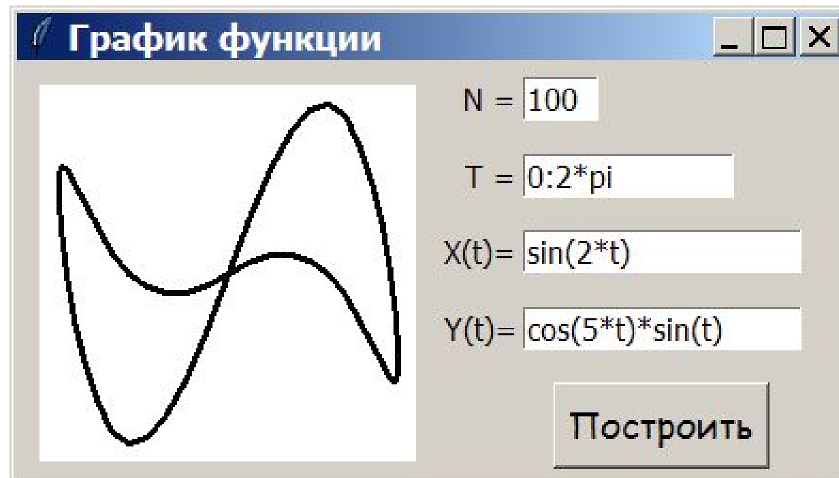
В эту схему можно уложить и явное задание функции. Соотношение y=F(x) можно задать системой

```
x=t
y=Y(t)
```

Для функций, заданных в полярной системе координат параметром фактически является полярный угол. Пусть функция r=R(t), где t есть полярный угол. Тогда, переходя к декартовой системе координат, получим

```
x=R(t)*cos(t)
y=R(t)*sin(t)
```

Разработаем интерфейс приложения, взяв за образец проект «Спирограф».



Здесь N задаёт число интервалов разбиения диапазона изменения переменной t. Сам диапазон задаётся текстовой строкой вида «выражение1 : выражение2». Выражение1 определяет левую, а выражение2 — правую границу диапазона T. В полях ввода для X(t) и Y(t) задаются формулы для вычисления x- и y-координат точек графика.

Новым в этом проекте будет использование функции eval (от английского evaluate – вычислить). Функция получает строку, содержащую некоторое математическое выражение и возвращает число — результат вычисления этого выражения. При этом выражение может содержать константы, переменные, знаки операций, вызовы математических функций и скобки, т.е. оно является полноценным математическим выражением.

Второе, что нам понадобится — это обработка ошибок пользователя, которые могут быть различного типа. Основные типы ошибок в математическом выражении:

SyntaxError – ошибка синтаксиса, например, несоответствие скобок;

NameError – использование несуществующей переменной;

ZeroDivisionError – деление на ноль.

Однако нельзя исключить появление и других ошибок, поэтому нужно предусмотреть универсальную обработку ошибок.

Если при работе программы возникает ошибка, то Python генерирует исключение. Это исключение можно перехватить и тогда программист может добавить в программу код для обработки этой ситуации.

Синтаксис обработки исключения:

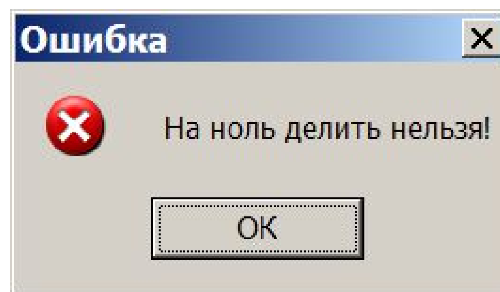
```
try:
  защищаемый блок кода
except тип_исключения:
  обработка исключения
else:
```

```
  выполняется, если исключение не возникло
finally:
  выполняется в любом случае.
```

Блоки else и finally не являются обязательными.
Пример.

```
x=0
try:
   res = 1/x
except ZeroDivisionError:
   showerror('Ошибка','На ноль делить нельзя!'
```

Результат:



Точно так же можно перехватить и другие ошибки, добавив блоки except для их обработки. Но выше уже отмечалось, что предусмотреть все возможные ошибки пользователя невозможно, поэтому нам нужен другой подход.

Чтобы перехватить любую ошибку, используем общий тип Exception, а чтобы конкретизировать исключение, определим его тип и значения параметров. Для изучения ситуации заменим блок обработки исключения в предыдущем примере на

```
except Exception as e:
   print(type(e))
   print(e.args)
```

и посмотрим на результат:

```
<class 'ZeroDivisionError'>
('division by zero',)
```

Разберём первую строку результата. Оператор print(type(e)) выводит строку, включающую название типа исключения. Уберём из неё лишнее. Для этого разделим строку на части, используя в качестве разделителя символ апострофа и присвоим среднюю часть переменной name:

```
    (_,name,_)=str(type(e)).split("'")
```

Необходимые пояснения.

1. Если оператор print получает объект (т.е. экземпляр класса), то он незаметно для нас использует функцию str(...), которая возвращает текстовую строку с описанием объекта. В других случаях необходимо её явное использование.

2. Метод split получает в качестве разделителя текстовый литерал — строку символов, заключенных в апострофы или кавычки. Наличие двух вариантов помогает в случаях, когда текстовая строка содержит символ апострофа. Тогда в качестве ограничителей используются кавычки.

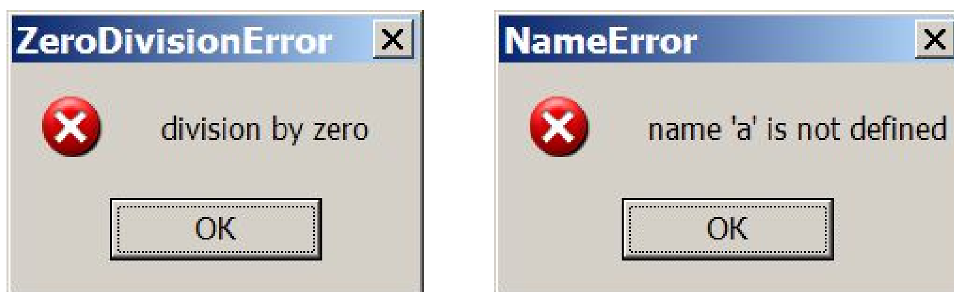
3. В левой части оператора присваивания стоит кортеж из трёх элементов, два из которых являются символами подчеркивания. Это означает, что значения, попадающие в эти позиции, игнорируются.

Разберём вторую строку результата. Оператор print вернул кортеж со значениями параметров исключения. В данном примере кортеж содержит одну текстовую строку с сообщением о характере ошибки (заметим, что в других случаях кортеж может содержать строки с дополнительной информацией об ошибке). Для доступа к основному параметру используем выражение e.args[0].

Используя метод showerror выведем полученную информацию на экран:

```
from tkinter.messagebox import showerror
x=0
try:
    res = 1/x
except Exception as e:
    (_,name,_)=str(type(e)).split("'")
    showerror(name,e.args[0])
```

Результат — окно сообщения об ошибке (рисунок слева):



Заменим оператор в блоке try на res = 1/a, тогда получим сообщение о другой ошибке (рисунок справа).

Таким образом мы убедились, что наш подход работает!

Вернёмся к разработке приложения. Начнём с импорта модулей и создания окна приложения

```
from math import *
from tkinter import *
from tkinter.messagebox import *
root=Tk()
root.title('График функции')
```

Задаём параметры, которые понадобятся для определения размеров окна и холста:

```
w=200; h=200; p=20
root.geometry(str(2*p+2*w)+'x'+str(p+h))
canvas=Canvas(root,width=w,height=h,bg='white')
```

Метод geometry получает текстовую строку, которая определяет размеры, а в общем случае и положение окна на экране дисплея. Например, geometry('400x200+500+100') устанавливает ширину окна 400 пикселей, высоту 200 пикселей, а левый верхний угол окна будет «привязан» к точке экрана с координатами x=500, y=100.

Объявим переменные, связанные с полями ввода и зададим им начальные значения:

```
nV=StringVar(); nV.set(100)
tV=StringVar(); tV.set('0:2*pi')
xV=StringVar(); xV.set('sin(2*t)')
yV=StringVar(); yV.set('cos(5*t)*sin(t)')
```

Основная часть проекта — функция-обработчик нажатия кнопки «Построить». Алгоритм работы функции:

1) извлечь из полей ввода необходимые данные:

- число интервалов разбиения в виде целого числа:

```
n=int(nV.get())
```

- границы диапазона (и вычислить шаг изменения) параметра:

```
(start,finish)=map(eval,tV.get().split(':'))
step=(finish-start)/n
```

Системная функция map получает два аргумента. Первый — это некоторая функция (в данном случае — функция eval), второй — список значений, к которым эта функция применяется. Список значений в этом примере есть список из двух строк ['0', '2*pi'], полученный методом split(':'). В общем случае разделитель представляет собой строку символов.

2) получить последовательность значений параметра:

```
T=[step*k for k in range(n+1)]
```

3) вычислить координаты точек графика

```
X=[eval(xV.get()) for t in T]
```

```
        Y=[eval(yV.get()) for t in T]
```

4) определить границы изменения координат точек графика

```
xmin=min(X)
xmax=max(X)
ymin=min(Y)
ymax=max(Y)
```

5) очистить холст

```
canvas.delete(ALL)
```

6) сформировать список вершин полигона — графика функции:

```
polygon=[]
for (x,y) in zip(X,Y):
   xw=12+(w-20)*(x-xmin)/(xmax-xmin)
   yh=12+(h-20)*(y-ymax)/(ymin-ymax)
   polygon.append((xw,yh))
```

7) нарисовать график

```
canvas.create_line(polygon,width=3)
```

Пункты 1), 2) и 3) необходимо включить в защищаемый блок кода

```
def plot():
   try:
      n=int(nV.get())
      (start,finish)=map(eval,tV.get().split(':'))
      step=(finish-start)/n
      T=[step*k for k in range(n+1)]
      X=[eval(xV.get()) for t in T]
      Y=[eval(yV.get()) for t in T]
   except Exception as e:
      argStr=''
      for arg in e.args: argStr+='\n'+str(arg)
      showerror('Ошибка!',str(type(e))+argStr)
      return
   xmin=min(X)
   xmax=max(X)
   ymin=min(Y)
   ymax=max(Y)
   canvas.delete(ALL)
   polygon=[]
   for (x,y) in zip(X,Y):
      xw=12+(w-20)*(x-xmin)/(xmax-xmin)
      yh=12+(h-20)*(y-ymax)/(ymin-ymax)
      polygon.append((xw,yh))
```

```
    canvas.create_line(polygon,width=3)

labelN=Label(root, text='  N =')
entryN=Entry(root, width=4,textvariable=nV)
labelT=Label(root, text='  T =')
entryT=Entry(root, width=12,textvariable=tV)
labelX=Label(root, text='X(t)=')
entryX=Entry(root, width=16,textvariable=xV)
labelY=Label(root, text='Y(t)=')
entryY=Entry(root, width=16,textvariable=yV)
button=Button(root, text='Построить',
              font=('Comic Sans MS',14), command=plot)
canvas.grid(row=0,column=0,rowspan=5,padx=10,pady=10)
labelN.grid(row=0,column=1)
entryN.grid(row=0,column=2,sticky=W)
labelT.grid(row=1,column=1)
entryT.grid(row=1,column=2,sticky=W)
labelX.grid(row=2,column=1)
entryX.grid(row=2,column=2)
labelY.grid(row=3,column=1)
entryY.grid(row=3,column=2)
button.grid(row=4,column=2)
```

Задания для самостоятельной работы.

1) Получить графики следующих параметрически заданных функций:

А) $\begin{cases} x(t) = \sin 2t \\ y(t) = \cos 3t \end{cases}$ $0 <= t <= 2\pi$

Б) $\begin{cases} x(t) = \sin 2t \\ y(t) = 2\sin t + \cos 2t \end{cases}$ $0 <= t <= 2\pi$

В) $\begin{cases} x(t) = t \sin t \\ y(t) = t \cos t \end{cases}$ $0 <= t <= 8\pi$

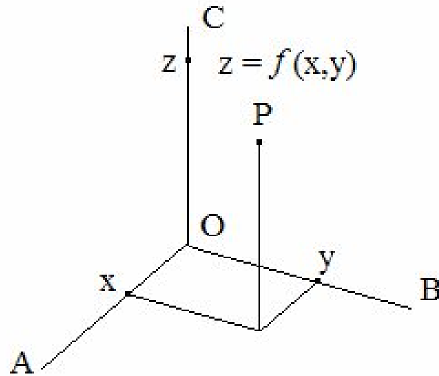
2) Добавьте в проект возможность построения графика при нажатии на клавишу Enter.

38. Проект «Построение поверхности»

Разработаем проект, в котором проиллюстрируем использование перегрузки операций для работы с точками и векторами на плоскости. Проект предназначен для построения графиков функций двух переменных вида z = *f*(x, y). График представляет собой некоторую поверхность в трехмерном пространстве. Для изображения поверхности на плоском экране

построим два семейства линий: одно состоит из сечений поверхности плоскостями x = const, другое – из сечений поверхности плоскостями y = const.

Наша первая задача – отобразить на экране значение функции *f*(x, y) при заданных значениях аргументов.



Система координат для построения графика

Построим на плоскости изображение пространственной декартовой системы координат, причём будем считать, что отрезок OA соответствует диапазону изменения переменной x: x0<=x<=x1; отрезок OB соответствует диапазону изменения переменной y: y0<=y<=y1; а отрезок OC определяет масштабирование графика по высоте: точка O соответствует значению z0, а точка C – значению z1. В качестве z0 можно выбрать минимальное, а в качестве z1 – максимальное значение функции *f*(x, y) на рассматриваемой области аргументов (x, y), но для простоты будем подбирать эти значения «вручную».

Как следует из рисунка, получить экранные координаты точки P можно, последовательно откладывая от точки O вектора Ox, Oy и Oz. В результате координаты точки P будут вычисляться по формуле:

P = O + Ox + Oy + Oz.

Учитывая, что вектор Ox есть некоторая часть вектора OA, вектор Oy есть некоторая часть вектора OB, а вектор Oz пропорционален вектору OC, приведенную формулу можно записать в следующем виде:

P = O + *a**OA + *b**OB + *c**OC,

где *a*, *b*, *c* определяются как:

$$a=\frac{x-x0}{x1-x0};\quad b=\frac{y-y0}{y1-y0};\quad c=\frac{f(x,y)-z0}{z1-z0}$$

Перейдём от теории к практике – напишем функцию `screen(f, x, y)`, которая должна вернуть кортеж из двух целых чисел – экранных координат точки, определяемой выбранной четвёркой опорных точек O, A, B, C, диапазонами изменения переменных *x*, *y* и их конкретными значениями, и самой функцией *f*(x, y).

Чтобы сохранить в программе внешний вид формул и, соответственно, облегчить её понимание, разработаем класс `Point`, в котором перегрузим необходимые нам операции. Объект класса в качестве данных содержит два числа с именами x и y, которые могут интерпретироваться либо как координаты точки, либо как координаты вектора. Перегрузим в классе `Point` операции сложения и вычитания точек-векторов и операцию умножения вещественного числа на экземпляр класса.

Когда интерпретатор находит в формуле операцию сложения, то он вызывает специальный метод с именем `__add__(...)` того класса, экземпляры которого складываются. Если мы складываем два числа, то вызывается метод встроенного класса `Numbers`, если мы «складываем» две строки, то вызывается метод встроенного класса `String`, если мы «складываем» два списка, то вызывается метод класса `List` и т.д. Этот же принцип применим к операции сложения экземпляров разрабатываемого нами класса `Point`.

Для перегрузки операции сложения добавим в класс метод с именем `__add__`, для перегрузки операции вычитания — добавить в класс метод `__sub__`, для перегрузки операции умножения — метод `__mul__` или `__rmull__`.

```
class Point:
   def __init__(self, x=0, y=0):
      self.x=x
      self.y=y
   def __add__(self, B):
      return Point(self.x+B.x, self.y+B.y)
   def __sub__(self, B):
      return Point(self.x-B.x, self.y-B.y)
   def __rmul__(self, B):
      return Point(self.x*B,self.y*B)
```

Заметим, что для реализации умножения вещественного числа на вектор (экземпляр класса) мы перегрузили операцию «правого» умножения

(функцию `__rmul__()` – от английского right multiply), так как экземпляр класса в приведенных формулах стоит *справа* от знака операции умножения.

Теперь создадим четвёрку опорных точек:

```
O=Point(200,150)
A=Point( 20,200)
B=Point(300,200)
C=Point(200,50)
```

зададим функцию *f*(x, y), границы изменения её аргументов и параметры масштабирования:

```
def f(x,y):
   return 20-x**2*cos(y)-2*y**2*sin(x)
n=12; m=12
x0=-2; x1=4; dx=(x1-x0)/n
y0=-2; y1=4; dy=(y1-y0)/m
z0=-20; z1=20
```

и напишем, наконец, функцию `screen`:

```
def screen(f,x,y):
    a=(x-x0)/(x1-x0)
    b=(y-y0)/(y1-y0)
    c=(f(x,y)-z0)/(z1-z0)
    W = O + a*(A-O) + b*(B-O) + c*(C-O)
    return (W.x,W.y)
```

Дискретизируем диапазоны изменения аргументов функции *f*(x, y):

```
X=[x0+i*dx for i in range(n+1)]
Y=[y0+j*dy for j in range(m+1)]
```

вычислим и сохраним экранные координаты значений функции *f*(x, y) в двух словарях. В первом из них в качестве ключей примем значения x_i, а в качестве значений словаря – списки экранных координат точек сечения поверхности плоскостью $x = x_i$:

```
Fx={}
for x in X:
    L=[screen(f,x,y) for y in Y]
    Fx[x]=L
```

Во втором словаре в качестве ключей используем значения y_j, а в качестве значений словаря – списки экранных координат точек сечения поверхности плоскостью $y = y_j$:

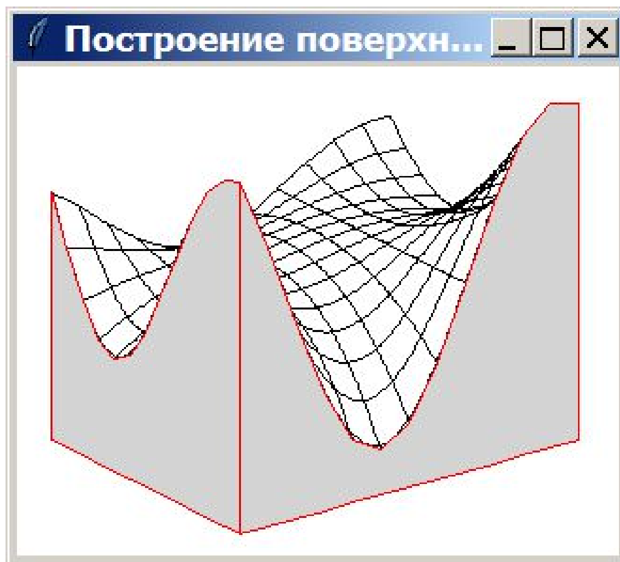
```
Fy={}
for y in Y:
```

```
    L=[screen(f,x,y) for x in X]
    Fy[y]=L
```

Осталось построить поверхность:

```
root=Tk()
root.title('Построение поверхности')
canvas=Canvas(root, width=420, height=350,
              bg='white')
canvas.pack()
for x in Fx:
    canvas.create_line(Fx[x], smooth=True)
for y in Fy:
    canvas.create_line(Fy[y], smooth=True)
```

Оформим боковые «стенки» графика функции в виде закрашенных полигонов. Первый полигон ограничим снизу прямой линией, соответствующей значению $x = x_n$, а сверху – точками сечения графика функции при том же значении x. Второй полигон ограничим снизу прямой линией, соответствующей значению $y = y_m$, а сверху – точками сечения графика функции при том же значении y.



Изображение поверхности на экране

Заметим, что точки нижних линий полигонов должны перечисляться в обратном (по отношению к точкам верхних линий) порядке

```
f0= lambda x,y : z0
Lx=[screen(f0, X[n], y) for y in reversed(Y)]
Ly=[screen(f0, x, Y[m]) for x in reversed(X)]
```

```
canvas.create_polygon(Lx+Fx[X[n]],
             fill='lightgray', outline='red')
canvas.create_polygon(Ly+Fy[Y[m]],
             fill='lightgray', outline='red')
```

Чтобы завершить программу, добавим в её начало импорт необходимых модулей

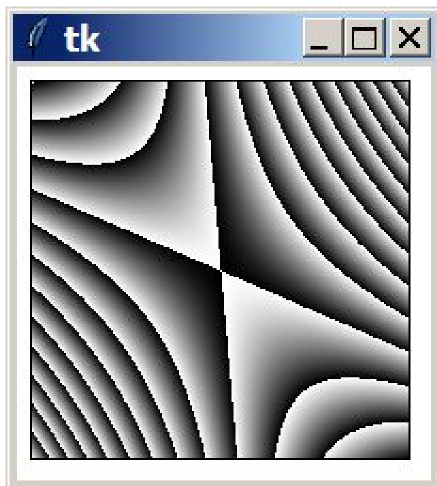
```
from math import *
from tkinter import *
```

Задание для самостоятельного выполнения. Разработайте интерфейс программы, который обеспечит ввод параметров графика.

39. Проект «Красота математических функций»

Еще один способ визуализации математической функции — это построение её линий уровня. Каждый, кто рассматривал физическую географическую карту, знает, что нанесенные на ней линии уровня и закраска областей, ими ограниченных позволяет определить высоту места над уровнем моря.

Однако построение линий уровня с вычислительной точки зрения является непростой задачей, а результат может оказаться не очень впечатляющим. Используем поэтому другой подход, при котором линии уровня формируются автоматически простым алгоритмом, при этом промежуток между линиями уровня закрашивается градиентным серым цветом. Следующий рисунок иллюстрирует сказанное.



На нем изображены линии уровня функции $f(x,y) = 5x^2 - 12xy + y^2$ в области -2<=x, y<=2.

Алгоритм построения:

1) задаем параметры: функцию f(x,y), границы области построения и число разбиений диапазона изменения функции;

2) определяем минимальное и максимальное значение функции f(x,y) в заданной области и шаг между уровнями;

2) для каждого пикселя области экрана, предназначенной для вывода графика определяем значение функции в точке (x,y), соответствующей пикселю экрана;

3) определяем цвет пикселя в зависимости от значения функции f в полученной точке (x,y)

4) выводим на экран квадрат, состоящий из одной точки

Приведем программу полностью.

Обозначим:

size – сторона области построения графика;
marg – ширина поля вокруг графика;
(x0,xn) – диапазон изменения x;
(y0,yn) – диапазон изменения y;
kvalue – число разбиений диапазона изменения функции;
dz – шаг разбиения;

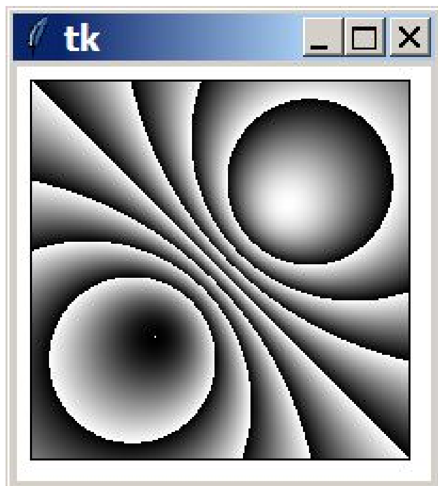
```
from math import *
from tkinter import *
root=Tk()
size=200; marg=10
canvas=Canvas(root, width=size+2*marg,
                    height=size+2*marg, bg='white')
canvas.pack()
canvas.create_rectangle((marg-1,marg-1),
                        (marg+size+1,marg+size+1),
                        fill='')
# задаём параметры графика
def f(x,y):
   return 5*x*x - 12*x*y + y*y
x0=-2; y0=-2; xn=2; yn=2
kvalue=12
# вычисляем минимальное и максимальное значение функции
# на заданной области
n=200
dx=(xn-x0)/n
dy=(yn-y0)/n
zmin=zmax=f(x0,y0)
for j in range(n+1):
   for i in range(n+1):
      w=(f(x0+i*dx,y0+j*dy))
```

```
        if w<zmin: zmin=w
        if w>zmax: zmax=w
dz=(zmax-zmin)/kvalue
# строим график
for sx in range(size):
   x=x0+(xn-x0)*sx/size
   for sy in range(size):
      y=y0+(yn-y0)*sy/size
      z=f(x,y)
      for k in range(kvalue):
         if k*dz<(z-zmin)<=(k+1)*dz:
            cv=int(((z-zmin)/dz-k)*255)
            color=
            '#{0:02x}{1:02x}{2:02x}'.format(cv,cv,cv)
            canvas.create_rectangle((marg+sx,marg+sy),
                                    (marg+sx+2,marg+sy+2),
                                     width=0,fill=color)
```

Варьируя параметры графика, можно получить большое количество красивых картин. Приведём лишь некоторые их них с указанием параметров построенных графиков.

А)

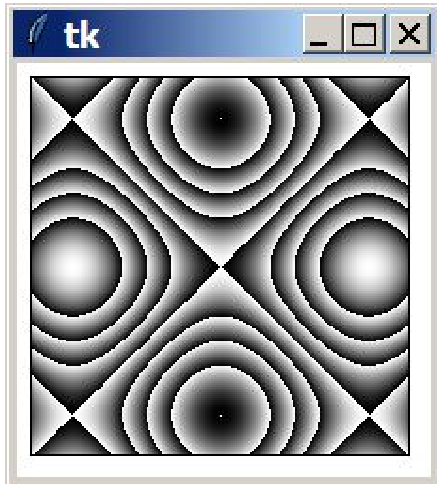
```
# задаём параметры графика
def f(x,y):
   return (x-y)/(1+x*x+y*y)
x0=-2; y0=-2; xn=2; yn=2
kvalue=8
```



Б)

```
# задаём параметры графика
```

```
def f(x,y):
   return (sin(x-y)*sin(x+y)
x0=-2; y0=-2; xn=2; yn=2
kvalue=8
```



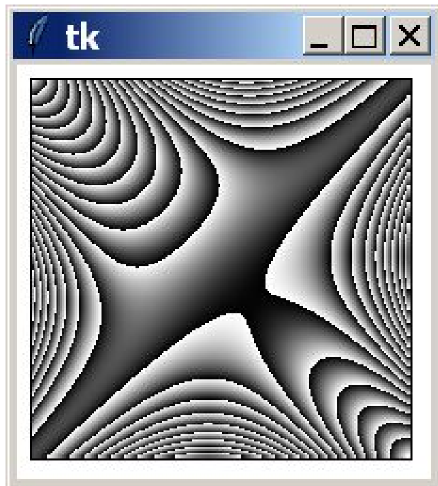
В)

```
# задаём параметры графика
def f(x,y):
   return y*cos(x)+x*cos(y)
x0= 0; y0= 0; xn= 8; yn= 8
kvalue=8
```

Г)

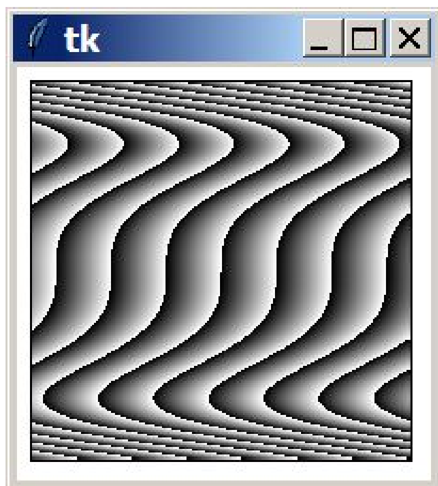
```
# задаём параметры графика
def f(x,y):
```

```
    return 2*(1-x-y)**2-(y*y-x*x)**2
x0=-3; y0=-3
xn= 3; yn= 3
kvalue=21
```



Д)

```
# задаём параметры графика
def f(x,y):
    return x+y*sin(y**2)
x0=-2; y0=-2
xn= 2; yn= 2
kvalue=12
```



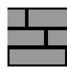
Задание для самостоятельного выполнения. Разработайте интерфейс программы, который обеспечит ввод параметров графика.

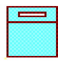
40. Игра «Сокобан»

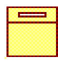
Слово «сокобан» переводится с японского как «кладовщик». Игра представляет собой план склада со множеством помещений, перегородок и коридоров. На плане отмечены положения ящиков и места, на которые их следует установить. Возможности кладовщика сильно ограничены — он может только толкать перед собой один ящик. Ни отодвинуть его в сторону, ни тащить за собой ящик невозможно.

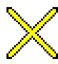
В Интернете можно найти сайты со множеством заданий для кладовщика, но мы, для иллюстрации возможностей библиотеки Tkinter напишем свою программу.

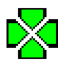
Начнём с плана склада. Чтобы задать расположение элементов игрового поля используем символьные обозначения. Свободное место отметим символом «точка»(.). Подготовим изображения элементов игрового поля и сопоставим их латинским буквам, указанным в скобках.

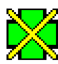
 - часть стены или перегородки склада (w);

 - изображение ящика, который ещё не занял нужное место (b);

 - изображение ящика, который стоит на отмеченном месте (p);

 - отметка места (x);

 - условное обозначение кладовщика (s);

 - условное обозначение кладовщика на отмеченном месте (z).

Разница в изображениях на отмеченных и не отмеченных местах нужна только для наглядности, т.к. ящик всё равно остаётся ящиком, а кладовщик — кладовщиком.

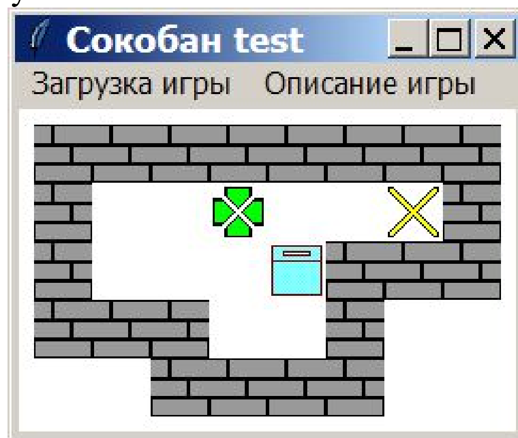
Планы складов для игры будем хранить в отдельных текстовых файлах. Чтобы отличить файлы игры от других текстовых файлов, используем расширение .sbn. Пример: файл test.sbn:

```
test
wwwwwwww
w..s..xw
```

```
w...bwww
www..w..
..wwww..
#
```

Первая строка файла содержит название игры, которое будет выводиться в заголовок окна. Следующие строки содержат план склада. Каждый символ строки определяет тип поля прямоугольной игровой доски. Последняя строка содержит в первой позиции символ решётки, который обозначает конец файла.

На рисунке ниже показан склад, соответствующий приведенному выше файлу.



Для работы с файлами используем стандартный диалог открытия файла, который необходимо импортировать из модуля tkinter.filedialog

```
from tkinter.filedialog import askopenfilename
```

Чтобы в диалоговом окне показывались только файлы с расширением .sbn необходимо параметру filetypes функции присвоить список, состоящий из единственного кортежа.

```
filetypes=[('Файлы игры "Сокобан"','*.sbn')]
```

Диалог возвращает полный путь и имя выбранного файла (если файл не был выбран, возвращается пустая строка). Метод open по умолчанию открывает файл на чтение, причём файл считается текстовым, поэтому open(fname) эквивалентно open(fname,mode=’rt’).

```
    fname = askopenfilename(
            filetypes=[('Файлы игры "Сокобан"','*.sbn')])
    if fname!='': f = open(fname)
    else: return
```

Метод open возвращает файловый объект f, методы которого предназначены для чтения информации из файла. В частности, метод

readlines() возвращает список всех строк текстового файла, а метод readline() — только одну, очередную строку.

Отметим, что возвращаемая строка содержит символ новой строки \n. Чтобы избавиться от него, а заодно и возможных пробелов в конце строки, применим к ней метод rstrip():

```
title=f.readline().rstrip()  # чтение строки-заголовка
```

Из оставшихся строк (за исключением последней) сформируем список data, который и есть план склада.

```
data=[]
while True:
   s=f.readline().rstrip()
   if s[0]=='#': break
   else: data.append(s)
f.close()
```

В соответствии с планом склада создадим двумерный список экземпляров класса Cell. Предварительно опишем класс Cell. Элементы данных:

self.i, self.j – координаты ячейки;

self.state – состояние ячейки;

self.img – идентификатор изображения.

Координаты и состояние ячейки задаются в конструкторе класса, изображение создаётся и выводится на канву в методе draw.

```
# глобальные параметры
size=31  # размер изображений
p=10     # размер отступов
# класс "Ячейка"
class Cell:
   def __init__(self,i,j,state):
      self.i=i
      self.j=j
      self.s=state

   def draw(self):
      ''' Условные обозначения:
       w - стена
       b - ящик
       x - место назначения
       p - ящик на отмеченной клетке
       s - кладовщик
       z - кладовщик на отмеченной клетке
       . - пустая клетка
      '''
      x=p+self.j*size
```

```
        y=p+self.i*size
        if self.s=='w':
           self.img=canvas.create_image(x,y,
                                        anchor=NW,
                                        image=wall)
        if self.s=='b':
           self.img=canvas.create_image(x,y,
                                        anchor=NW,
                                        image=box)
        if self.s=='p':
           self.img=canvas.create_image(x,y,
                                        anchor=NW,
                                        image=boxoplace)
        if self.s=='x':
           self.img=canvas.create_image(x,y,
                                        anchor=NW,
                                        image=xplace)
        if self.s=='.':
           self.img=canvas.create_image(x,y,
                                        anchor=NW,
                                        image=fon)
        if self.s=='s':
           self.img=canvas.create_image(x,y,
                                        anchor=NW,
                                        image=sokoban)
        if self.s=='z':
           self.img=canvas.create_image(x,y,
                                        anchor=NW,
                                        image=sokoplace)
```

Все изображения загружаются из соответствующих файлов в главной программе:

```
wall=PhotoImage(file='wall.GIF')
box=PhotoImage(file='box.GIF')
xplace=PhotoImage(file='place.GIF')
boxoplace=PhotoImage(file='boxoplace.GIF')
fon=PhotoImage(file='fon.GIF')
sokoban=PhotoImage(file='soko.GIF')
sokoplace=PhotoImage(file='sokoplace.GIF')
```

Вернёмся к созданию игры. После формирования двумерного плана склада мы можем определить размеры склада «в клетках» и установить параметры канвы.

Здесь:

n – количество клеток по вертикали;

m – количество клеток по горизонтали;
w, h – ширина и высота канвы в пикселях.

```
n=len(data)
m=len(data[0])
w=2*p+m*size
h=2*p+n*size
canvas['width']=w
canvas['height']=h
canvas['bg']='white'
```

Размещаем канву в окне приложения и удаляем её старое содержимое:

```
canvas.pack()
canvas.delete(ALL)
```

Создаём объекты новой игры:

```
root.title('Сокобан '+title)
cells=[]
for i in range(n):
   w=[]
   for j in range(m):
      cell=Cell(i,j,data[i][j])
      cell.draw()
      if data[i][j]=='s' or data[i][j]=='z':
         sbi=i; sbj=j
      w.append(cell)
   cells.append(w)
```

Переменные sbi, sbj – координаты кладовщика на плане склада.

Объединим написанные фрагменты в функцию newgame(), при этом переменные n,m,sbi,sbj,cells, которые нам понадобятся в дальнейшем, объявим глобальными:

```
def newgame():
   global n,m,sbi,sbj,cells
   fname = askopenfilename(
           filetypes=[('Файлы игры "Сокобан"','*.sbn')])
   if fname!='': f = open(fname)
   else: return
   title=f.readline().rstrip()  # чтение строки-заголовка
   data=[]
   while True:
      s=f.readline().rstrip()
      if s[0]=='#': break
      else: data.append(s)
   f.close()
```

```
    n=len(data)
    m=len(data[0])
    w=2*p+m*size
    h=2*p+n*size
    canvas['width']=w
    canvas['height']=h
    canvas['bg']='white'
    canvas.pack()
    canvas.delete(ALL)

    root.title('Сокобан '+title)
    cells=[]
    for i in range(n):
       w=[]
       for j in range(m):
          cell=Cell(i,j,data[i][j])
          cell.draw()
          if data[i][j]=='s' or data[i][j]=='z':
             sbi=i; sbj=j
          w.append(cell)
       cells.append(w)
```

Напишем теперь обработчик действий пользователя — функцию move(event). Функция «знает» положение кладовщика — его координаты на плане склада передаётся в функцию через глобальные переменные sbi, sbj.

В зависимости от нажатой стрелки функция-обработчик вычисляет вектор действия кладовщика — значения относительных сдвигов di, dj

```
def move(event):
   global sbi, sbj
   key=event.keysym
   if   key=='Left' : di=0; dj=-1
   elif key=='Right': di=0; dj= 1
   elif key=='Up'   : di=-1; dj=0
   elif key=='Down' : di= 1; dj=0
   else: return
```

Получаем статус ячейки перед кладовщиком (с учётом вектора действия кладовщика):

```
   s1 = cells[sbi+di][sbj+dj].s
```

Если перед кладовщиком находится ящик, а перед ящиком свободное место, то передвигаем ящик на это место.

```
   if s1=='b' or s1=='p':
   #перемещаем ящик
      s2 = cells[sbi+2*di][sbj+2*dj].s
```

```
        if s2=='.':
           cells[sbi+2*di][sbj+2*dj].s = 'b'
        if s2=='x':
           cells[sbi+2*di][sbj+2*dj].s = 'p'
        if s1=='b':
           cells[sbi+di][sbj+dj].s = '.'
        if s1=='p':
           cells[sbi+di][sbj+dj].s = 'x'
        # перерисовывем клетку с ящиком
        canvas.delete(cells[sbi+2*di][sbj+2*dj].img)
        cells[sbi+2*di][sbj+2*dj].draw()
```

Перемещение ящика сводится к изменению статуса двух ячеек и перерисовке клетки, на которую становится ящик.

После перемещения ящика кладовщик ставится на освободившееся место:

```
     s1 = cells[sbi+di][sbj+dj].s
     if s1=='.' or s1=='x':
     #перемещаем кладовщика
        s0 = cells[sbi][sbj].s
        if s1=='.':
           cells[sbi+di][sbj+dj].s = 's'
        if s1=='x':
           cells[sbi+di][sbj+dj].s = 'z'
        if s0=='s':
           cells[sbi][sbj].s = '.'
        if s0=='z':
           cells[sbi][sbj].s = 'x'
        # перерисовывем клетку с кладовщиком
        canvas.delete(cells[sbi+di][sbj+dj].img)
        cells[sbi+di][sbj+dj].draw()
        # перерисовывем освободившуюся клетку
        canvas.delete(cells[sbi][sbj].img)
        cells[sbi][sbj].draw()
        # вычисляем новое положение кладовщика
        sbi+=di; sbj+=dj
```

Заметим, что если перед кладовщиком нет ящика, то сразу будет выполняться код, приведенный выше.

Осталось проверить, решили мы задачу или нет:

```
     if check():
        yes=askyesno('Задача решена','Играть ещё?')
        if yes : newgame()
        else: root.destroy()
```

Функция check() проверяет состояние всех клеток. Если хотя бы одна имеет статус 'b', то не все ящики стоят на отмеченных местах. В этом случае функция возвращает логическое значение False. Если таких клеток нет, то функция вернёт True.

```
def check():
   for i in range(n):
      for j in range(m):
         if cells[i][j].s=='b':
            return False
   return True
```

Задания для самостоятельной работы.

1. Соберите написанные фрагменты в единую программу. Программа должна вызывать функцию newgame() и реагировать на действия пользователя. Не забудьте добавить в главную программу строчку

```
root.bind('<Key>',move)
```

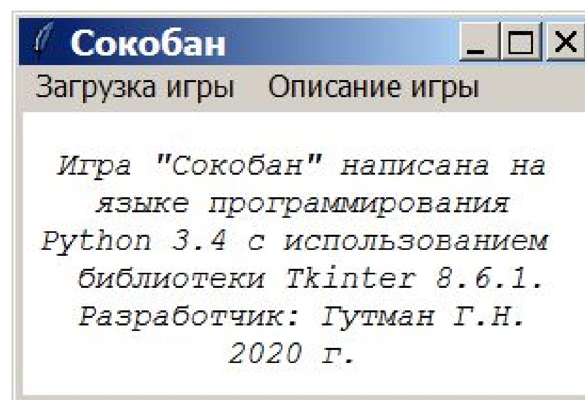
2. Добавьте в приложение строку меню из двух пунктов: «Загрузка игры» и «Описание игры». При выборе пункта «Загрузка игры» вызывается функция newgame(), при выборе пункта «Описание игры» вызывается функция helpgame(). Поскольку эта функция ещё не написана, используйте так называемую «заглушку»:

```
def helpgame():
   pass
```

Оператор pass ничего не делает, но создаёт синтаксически правильное описание функции.

41. Игра «Сокобан» (продолжение)

Займёмся усовершенствованием интерфейса программы. При запуске программы окно приложения пусто. Давайте добавим в него заставку:



Для этого при создании окна приложения вставим в него канву и добавим на неё желаемый текст. Особенность состоит в том, что текст будет вставлен точно в том же виде, в котором он приведен в программе, даже если он занимает несколько строк. Для этого достаточно использовать тройные кавычки в начале и конце текста.

```
# создание окна приложения
root=Tk()
root.title('Сокобан')
canvas=Canvas(root,width=300,height=150,bg='white')
canvas.pack()
# заставка
info='''
  Игра "Сокобан" написана на
    языке программирования
 Python 3.4 с использованием
   библиотеки Tkinter 8.6.1.
   Разработчик: Гутман Г.Н.
           2020 г. '''
canvas.create_text(0,0,anchor=NW,text=info,
                   font=('Courier New',13,'italic'))
```

Второе, что мы сделаем, — добавим в программу описание игры. Описание должно вызываться в отдельном окне при выборе соответствующего пункта меню.

Дополнительное окно создадим как экземпляр класса Toplevel и «привяжем» его к главному окну. Привязка означает, что при закрытии главного окна автоматически закроется и дополнительное окно. Но при закрытии дополнительного окна главное окно останется открытым.

Дополнительное окно имеет те же параметры, что и главное окно и может содержать любые элементы управления. Нам понадобятся:

- элемент управления Text, который позволяет выводить в окно текст с различным стилевом оформлением, а также содержать элементы управления и изображения;

- полоса прокрутка Scrollbar, так как в статичное окно может не поместиться вся информация.

Основные возможности элемента управления Text.

1) индексация знакомест: текстовый литерал вида 'строка.позиция'. Нумерация строк начинается с единицы, а нумерация позиций — с нуля. Для номера позиции есть специальное значение end, которое равно значению индекса последнего символа строки;

2) вставка объекта в заданную позицию: insert('строка.позиция', объект);

3) вставка объекта после самого последнего символа: insert(END, объект);

4) задание стиля оформления: tag_config('имя',параметры), где 'имя' есть имя стиля, а именованные параметры задают набор характеристик стиля. Перечислим основные параметры оформления текста:

font — гарнитура, размер и начертание символов:

justify — выравнивание текста;

foreground — цвет символов;

background — цвет фона;

underline — подчеркнутый текст (если underline=True)

5) стилевое оформление диапазона позиций: tag_add('имя','начало','конец'), где 'имя' есть имя стиля, а параметры 'начало' и 'конец' определяют начало и конец диапазона.

Элемент управления Scrollbar позволяет прокручивать содержимое элементов Text, Canvas, Entry или Listbox. Взаимодействие этих элементов достигается следующим образом. При создании полосы прокрутки параметр command ссылается на метод xview элемента Text при горизонтальной прокрутке, или на метод yview элемента Text при вертикальной прокрутке. Одновременно, в элементе Text параметр xscrollcommand при горизонтальной прокрутке или параметр yscrollcommand при вертикальной прокрутке ссылается на метод set полосы прокрутки.

```
def helpgame():
   winhelp=Toplevel(root)
   winhelp.title('Описание игры')
   help=Text(winhelp,width=53,height=21,
             font=('Courier New',16))
   help.pack(side='left')
   scroll=Scrollbar(winhelp,command=help.yview)
   help.config(yscrollcommand=scroll.set)
   scroll.pack(side='right',fill='y')
```

Создадим стиль оформления подзаголовков;

```
   help.tag_config('style_c',justify='center',
                   font=('Arial',14,'bold'))
```

Выводим текст и изображение блоков

```
   help.insert(END,'\n')
   help.insert(END,
'''  "Сокобан" в переводе с японского - это кладовщик.
 Сокобан должен передвинуть несколько ящиков, чтобы
 поставить их на места, отмеченные крестиками.\n\n''')
   help.insert(END,'Условные обозначения:\n\n    ')
   help.tag_add('style_c','6.0','6.end')
   help.image_create(END,image=wall)
   help.insert(END,
```

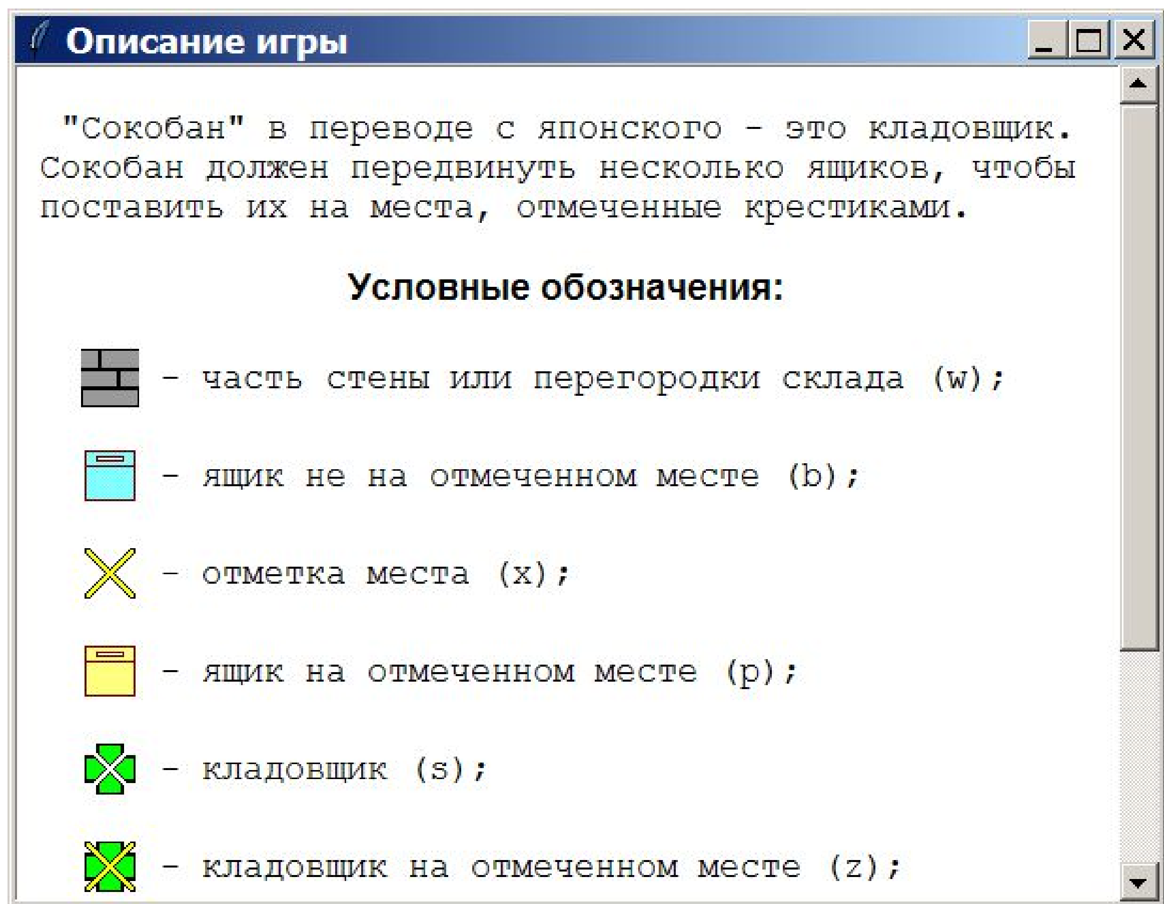
```
' - часть стены или перегородки склада (w);\n\n   ')
   help.image_create(END,image=box)
   help.insert(END,
' - ящик не на отмеченном месте (b);\n\n   ')
   help.image_create(END,image=xplace)
   help.insert(END,' - отметка места (x);\n\n   ')
   help.image_create(END,image=boxoplace)
   help.insert(END,
' - ящик на отмеченном месте (p);\n\n   ')
   help.image_create(END,image=sokoban)
   help.insert(END,' - кладовщик (s);\n\n   ')
   help.image_create(END,image=sokoplace)
   help.insert(END,
' - кладовщик на отмеченном месте (z);\n\n')
   help.insert(END,'Управление:\n\n')
   help.tag_add('style_c','20.0','20.end')
   help.insert(END,
'''  При нажатии стрелки на клавиатуре кладовщик делает
 шаг в этом направлении, если соседнее поле свободно.
 Если перед кладовщиком стоит ящик, за которым есть
 место, то кладовщик передвигает ящик на это место.\n''')
```

Внешний вид окна описания игры приведен на следующей странице.

Приведём в заключение несколько задач различной степени сложности.

task_01.sbn	**task_02.sbn**	**task_03.sbn**	**task_04.sbn**
01 wwwwwwww w..xxxxw w.b....w wwwswwww w......w w.b.bb.w w......w wwwwwwww #	02 ..wwwww. www...w. w.b.w.ww w.w..x.w w....w.w wwbwx..w .ws..www .wwwww.. #	03 wwwwwwww w...xxxw w.bbb.sw ww...www .ww..w.. ..wwww.. #	04 .wwww... .w..w... ww..ww.. w.xbxwww w.b.b..w wwzbx..w .ww..www ..wwww.. #

Задание для самостоятельной работы. Решите предложенные задачи.



42. Исполнитель «Муравей»

Последний проект — исполнитель «Муравей». Этот известный исполнитель придуман А.Г.Звенигородским и получил дальнейшее развитие в книге «Муравьиные сказки» авторов Гутмана Г.Н. и Карпиловой О.М.

Это достаточно большой проект, который объединяет рассмотренные ранее приёмы программирования и иллюстрирует новые возможности языка программирования Python.

Исполнитель Муравей живёт на доске 8х8 клеток. Размер доски определяется практическими соображениями, так как при отсутствии компьютера исполнитель мог «работать» на шахматной доске. Кроме исполнителя на доске могут находиться кубики с символами, а в «расширенной» версии и цветные кубики. Каждый объект занимает одну клетку. Исполнитель может перемещаться на другое поле в той же горизонтали или вертикали, при этом он может толкать кубики,

встречающиеся на его пути. Ни выйти за пределы доски, ни столкнуть с неё кубики Муравей на может.

Система команд исполнителя состоит из команд left(k), right(k), up(k), down(k), где k — длина хода в клетках доски. Команды исполнителю отдаёт некоторое устройство, управляющее исполнителем. В нашем проекте это человек, который управляет исполнителем с помощью стрелок клавиатуры, или программа, которая «руководит» исполнителем.

На рисунке ниже показан внешний вид окончательного варианта разрабатываемого приложения. В левой части размещена игровая доска с расставленными на ней кубиками и самим исполнителем. Справа — окно редактора с набранной программой. Внизу расположены две кнопки. Кнопка «Управление исполнителем» включает режим ручного управления. Кнопка «Выполнение программы» запускает написанную программу на выполнение.

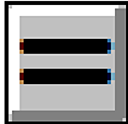
Разработаем классы Cell – поле с кубиком и Ant – поле с исполнителем. Для удобства изменения внешнего вида зададим глобальные параметры:

n – размер доски в клетках;
size – размер поля доски в пикселях;
p – величина отступа игровой доски от границы;
foncolor – цвет свободных полей доски;
progcolor – цвет фона окна редактора.

```
#--- параметры ---
n=8; size=32; p=10
foncolor='#ddeeff'
```

```
progcolor='#ddffee'
#-----------------
```

Класс Cell содержит единственный метод — конструктор __init__(). В нём мы пропишем все графические объекты, из которых состоит кубик с символом. Это серый квадрат, символ и линии белого и темно-серого цвета, которые создают впечатление объёма. На рисунке в увеличенном виде представлен один из кубиков предыдущего примера.



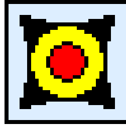
Для перемещения изображения кубика на игровой доске методом move, зададим для всех составляющих кубик графических объектов один и тот же тег. Но он должен быть разным для всех экземпляров класса (иначе все кубики будут перемещаться одновременно!). Поэтому добавим в тег координаты кубика:

```
self.tag='cell'+str(i)+str(j)
```

Ещё один нюанс связан с необходимостью восстанавливать изображение начального положения кубиков на игровой доске для новой попытки решить задачу. Поэтому, кроме текущих координат self.i, self.j кубика, введём переменные self.old_i, self.old_j, в которых запомним начальное положение кубика на игровой доске.

```
class Cell:
   def __init__(self,i,j,ch):
      self.i=self.old_i=i
      self.j=self.old_j=j
      self.tag='cell'+str(i)+str(j)
      x=p+j*size
      y=p+i*size
      canvas.create_rectangle(x,y,x+size,y+size,
             fill='silver',tag=self.tag)
      canvas.create_text(x+size/2,y+size/2,
             text=ch,font=('Courier New',size,'bold'),
             tag=self.tag)
      canvas.create_line(x+2,y+size-2,x+2,y+2,
             x+size-2,y+2,width=3,fill='white',
             tag=self.tag)
      canvas.create_line(x+2,y+size-2,
             x+size-2,y+size-2,x+size-2,y+2,width=3,
             fill='gray',tag=self.tag)
```

Класс Ant также содержит только конструктор, который формирует внешний вид исполнителя:



```
class Ant():
   def __init__(self,i,j):
      self.i=self.old_i=i
      self.j=self.old_j=j
      x=p+j*size
      y=p+i*size
      self.tag='ant'
      canvas.create_line(x+3,y+3,x+size-3,y+size-3,
             width=size/5,arrow='both',tag=self.tag)
      canvas.create_line(x+size-3,y+3,x+3,y+size-3,
             width=size/5,arrow='both',tag=self.tag)
      canvas.create_oval(x+size/5,y+size/5,x+4*size/5,
             y+4*size/5,fill='yellow',tag=self.tag)
      canvas.create_oval(x+size/3,y+size/3,x+2*size/3,
             y+2*size/3,fill='red',tag=self.tag)
```

Начальное расположение кубиков и исполнителя будем задавать в текстовых файлах с использованием следующей кодировки:

символ `'.'` (точка) обозначает пустое поле;

символ `'@'` обозначает исполнителя.

Все остальные символы обозначают кубик с этим символом.

В частности, для демонстрации работы программы создадим файл task-road.txt:

```
........
........
........
==....==
........
..====..
.@......
........
```

Загрузим данные из файла в переменную board и одновременно в двумерном списке cells создадим всех «действующих лиц» задачи. Заметим, что список cells содержит значения разных типов: для пустого поля — значение None, для поля с кубиком — экземпляр класса Cell, для исполнителя — экземпляр класса Ant.

```
f=open('task-road.txt','r')
board=[]; cells=[]
for i in range(n):
   line=f.readline().rstrip()
   b=[]; w=[]; j=0
   for ch in line:
      b.append(ch)
      if ch=='@':
         ant_i=old_i=i
         ant_j=old_j=j
         w.append(Ant(i,j))
      elif ch=='.': w.append(None)
      else: w.append(Cell(i,j,ch))
      j+=1
   board.append(b)
   cells.append(w)
f.close
```

Текущее положение исполнителя будем хранить в глобальных переменных ant_i, ant_j, которые будут изменяться при его перемещениях. Начальное положение исполнителя продублируем в переменных old_i, old_j, нужных для отображения начального состояния игровой доски.

Для того, чтобы начать сначала мало восстановить внешний вид задачи, нужно восстановить и списки board и cells, которые будут изменяться в ходе решения задачи.

Скопируем оба списка в переменные brd и cls:

```
brd=deepcopy(board)
cls=deepcopy(cells)
```

Заметим, что простого присваивания brd=board недостаточно для копирования списка, так как в этом случае и board и brd будут связаны с одним и тем же местом в памяти, выделенном для хранения значений списка. Только метод deepcope («глубокое» копирование) из модуля copy позволяет получить новый, независимый от исходного, список.

Покажем, как использовать имеющиеся у нас данные, чтобы восстановить как внешний вид, так и внутреннее состояние задачи. Напишем функцию restore:

```
def restore():
   global board,cells,ant_i,ant_j
   # восстановление внешнего вида задачи
   ant_i=old_i; ant_j=old_j
   for i in range(n):
      for j in range(n):
         w=cells[i][j]
```

```
            if w!=None:
               canvas.move(w.tag, (w.old_j-w.j)*size,
                                  (w.old_i-w.i)*size)
      # восстановление внутреннего состояния заачи
      board=deepcopy(brd)
      cells=deepcopy(cls)
      # обновление приложения
      root.update()
```

Перемещение исполнителя. Для каждого из четырёх событий (нажата стрелка влево, стрелка вправо, стрелка вверх и стрелка вниз) напишем свой обработчик. Для примера приведём обработчик события «нажата стрелка влево»:

```
   def moveleft(event):
      global board,cells,ant_i,ant_j
      i=ant_i; j=ant_j
      # находим крайний кубик
      while j>0 and board[i][j-1]!='.': j-=1
      if j==0:
         showinfo('Ошибка','Не могу выполнить команду
   ВЛЕВО')
         return
      else:
         for k in range(j,ant_j+1):
            canvas.move(cells[i][k].tag,-size,0)
            cells[i][k].j-=1
            cells[i][k-1]=cells[i][k]
            board[i][k],board[i][k-1] =\
                        board[i][k-1],board[i][k]
         cells[ant_i][ant_j]=None
         ant_j-=1
```

Разберёмся в алгоритме работы функции. Сначала координаты исполнителя переписываем в переменные i, j. Затем мы идём влево по i-ой строке, пока не обнаружим пустое поле.

```
   while j>0 and board[i][j-1]!='.': j-=1
```

Если дойдя до края доски мы не обнаруживаем пустого поля, то генерируем сообщение об ошибке:

```
      if j==0:
         showinfo('Ошибка',
                 'Не могу выполнить команду ВЛЕВО')
         return
```

В противном случае перемещаем Муравья и стоящие перед ним кубики на одну клетку влево: сначала самый левый кубик, потом его соседа справа и т.д. В последнюю очередь перемещаем Муравья.

Аналогично пишутся обработчики остальных событий. Принцип перемещения кубиков во всех случаях один и тот же — по убыванию расстояния от кубика до исполнителя, т.е. начинаем перемещение с самого дальнего кубика.

Выполнение команд исполнителя сводится к генерированию и обработке нескольких одинаковых событий. Так, функция left(k) генерирует в цикле k событий '<Left>':

```
def left(k):
   for i in range(k):
      root.event_generate('<Left>')
      root.update()
      root.after(500)
```

Аналогично реализуются остальные команды.

Напишем основную программу

```
from copy import *
from tkinter import *
from tkinter.messagebox import *
root=Tk()
root.title('Исполнитель "Муравей"')
root['bg']='white'

# игровая доска
canvas=Canvas(root,width=2*p+n*size,
                  height=2*p+n*size,bg=foncolor)
for i in range(n+1):
   canvas.create_line(p, p+i*size, p+n*size, p+i*size)
for j in range(n+1):
   canvas.create_line(p+j*size, p, p+j*size, p+n*size)
# редактор программы
program=Text(root, font=('Courier New',int(0.42*size)),
               width=25,height=13,padx=p,pady=p,
               bg=progcolor)
# кнопки выбора режима
button1=Button(root,width=30,
               text='Управление исполнителем',
               command=manage_ant)
button2=Button(root,width=30,
               text='Выполнение программы',
               command=exec_prog)
# размещение элементов управления
```

```
canvas.grid(row=0,column=0)
program.grid(row=0,column=1)
button1.grid(row=1,column=0)
button2.grid(row=1,column=1)
```

При нажатии на кнопку «Управление исполнителем» фокус ввода передаётся канве и восстанавливается начальная позиция задачи:

```
def manage_ant():
   canvas.focus_set()
   restore()
```

При нажатии на кнопку «Выполнение программы» фокус ввода передаётся редактору, восстанавливается начальная позиция задачи и вызывается метод exec. Метод получает из редактора весь набранный текст и выполняет его как обычную Python-программу:

```
def exec_prog():
   canvas.focus_set()
   restore()
   root.after(500)
   exec(program.get('1.0','end'))
```

Чтобы приложение реагировало на действия пользователя добавим в главную программу вызовы метода bind, который связывает канву, события нажатия на клавиши-стрелки и обработчики этих событий.

```
canvas.bind('<Left>',moveleft)
canvas.bind('<Right>',moveright)
canvas.bind('<Up>',moveup)
canvas.bind('<Down>',movedown)
```

Для перехода в режим набора программы достаточно щёлкнуть левой кнопкой мыши в области редактора.

Задания для самостоятельного выполнения.

1) Соберите все фрагменты в единую программу

2) Добавьте вывод сообщения в случае ошибки в программе

3) Добавьте в программу меню с пунктом «Выбор задачи», при выборе которого вызывается стандартный диалог выбора файла и несколько файлов с задачами.

4) Реализуйте возможность определения исполнителем цвета соседнего кубика — функцию GetColor(сторона), которая возвращает цвет кубика, стоящего с заданной стороны от исполнителя. Составьте несколько задач с цветными кубиками.

43. Приложения

Приложение 1. Код модуля geom_sys.py

```
"""
   модуль geom_sys содержит функции для работы
   в полярной системе координат и функции
   преобразования списка точек (векторов)
"""

from math import *
def create_circle(obj, S, R,**params):
   """
   S - центр круга
   R - радиус круга
   params - набор именованных параметров
   """
   (xs,ys) = S
   return obj.create_oval(xs-R,ys-R,xs+R,ys+R,params)

# функция перехода к декартовым координатам
def ps2ds(S,L):
   """
   S - центр системы координат
   L - точка (кортеж) или список точек (кортежей)
       в полярной системе координат с центром в точке S
   результат - точка (кортеж) или список точек (кортежей)
       в экранной системе координат
   """
   (xs,ys)=S
   if type(L) == tuple:
      (R,fi)=L
      return (xs + R*cos(pi*fi/180),
              ys - R*sin(pi*fi/180))
   if type(L) == list:
      return [(xs + R*cos(pi*fi/180),
               ys - R*sin(pi*fi/180)) for (R,fi) in L]

# функция перехода к полярным координатам
def ds2ps(S,L):
   """
   S - центр системы координат
   L - точка (кортеж)или список точек (кортежей)
       в экранной системе координат
```

```
    результат - точка (кортеж) или список точек (кортежей)
        в полярной системе координат с центром в точке S
    """
    (xs,ys)=S
    grd =180/pi
    if type(L) == tuple:
       (x,y)=L
       return (sqrt((x-xs)**2+(y-ys)**2),
               -atan2(y-ys,x-xs)*grd)
    if type(L) == list:
       return [(sqrt((x-xs)**2+(y-ys)**2),
                -atan2(y-ys,x-xs)*grd) for (x,y) in L]

# сложение векторов в экранной системе координат
def add(S,*L):
    """
    S - начальная точка
    L - список векторов
    xs, ys - координаты конечной точки
    """
    (xs, ys)=S
    for (x,y) in L:
       xs=xs+x
       ys=ys+y
    return (xs, ys)

# умножение векторов на число в экранной системе
координат
def mlt(L,k):
    """
    L - список векторов
    k - целое или вещественное число
    результат - вектор V*k
    """
    if type(L)==list:
       return [mlt(P,k) for P in L]
    else:
       (x,y)=L
       return (x*k, y*k)

# поворот списка точек на заданный угол
def turn(S, L, df):
    """
    S - центр поворота
    L - точка (кортеж) или список точек (кортежей)
```

```
    df - угол поворота. При df>0 поворот против часовой
        стрелки, при df<0 - по часовой стрелке
    результат - точка (кортеж) или список точек (кортежей)
        после поворота
    """
    P = ds2ps(S,L)
    if type(P)==tuple:
       (R,fi)=P
       return ps2ds(S,(R,fi+df))
    if type(P)==list:
       W=[(R,fi+df) for R,fi in P]
       return ps2ds(S,W)

# сдвиг списка точек на заданный вектор
def move(V, L):
    """
    V - вектор слвига
    L - точка или список точек
    результат - точка или список точек после сдвига
    """
    if type(L)==list:
       return [add(V,P) for P in L]
    else:
       return add(V,L)

# зеркальное отображение списка точек
def symmetry(P,Q,L):
    """
    P,Q - эти точки задают положение "зеркала"
    L - точка (кортеж) или список точек (кортежей)
    результат - точка (кортеж) или список точек (кортежей)
        после отображения
    """
    (rq,fq)=ds2ps(P,Q)
    if type(L)==tuple:
       (rw,fw)=ds2ps(P,L)
       return (ps2ds(P,(rw,2*fq-fw)))
    if type(L)==list:
       res=[]
       for W in L:
          (rw,fw)=ds2ps(P,W)
          res.append(ps2ds(P,(rw,2*fq-fw)))
       return res
```

Приложение 2. Модуль turtles

```
# модуль turtles, версия 2.0
# импорт математической библиотеки Питона
from math import *
# импорт графической библиотеки Питона
from tkinter import *

def create_win(w_title, c_width=400, c_height=300,
color='white'):
   global root, canvas, xs, ys
   xs=c_width/2; ys=c_height/2
# создание окна приложения с "холстом" для рисования
   root=Tk()
   root.title(w_title)
   canvas=Canvas(root, width=c_width, height=c_height,
                 bg=color)
   canvas.pack()
   return root
d=5
class Turtle:
   def __init__(self,
               angle=0,color='black',width=1,stype=1):
      self.xp=xs
      self.yp=ys
      self.fi=angle*pi/180
      self.cv=color
      self.wl=width
      if stype==1: self.st=1
      else : self.st=-1
      x=self.xp + 2*d*cos(self.fi)
      y=self.yp - 2*d*sin(self.fi);
      self.oval=canvas.create_oval(
                self.xp-d,self.yp-d,self.xp+d,self.yp+d,
                width=1,outline=self.cv)
      self.line=canvas.create_line(
                self.xp,self.yp,x,y,fill=self.cv)
   def move(self,len):
      x=self.xp + len*cos(self.fi)
      y=self.yp - len*sin(self.fi);
      canvas.move(self.oval,x-self.xp,y-self.yp)
      canvas.move(self.line,x-self.xp,y-self.yp)
      self.xp=x; self.yp=y
   def forw(self, len):                 # вперёд
      x=self.xp + len*cos(self.fi)
      y=self.yp - len*sin(self.fi);
```

```
        canvas.create_line(self.xp,self.yp,x,y,
                           fill=self.cv,width=self.wl)
        canvas.move(self.oval,x-self.xp,y-self.yp)
        canvas.move(self.line,x-self.xp,y-self.yp)
        self.xp=x; self.yp=y
    def back(self, len):                # назад
        x=self.xp-len*cos(self.fi)
        y=self.yp+len*sin(self.fi);
        canvas.create_line(self.xp,self.yp,x,y,
                           fill=self.cv,width=self.wl)
        canvas.move(self.oval,x-self.xp,y-self.yp)
        canvas.move(self.line,x-self.xp,y-self.yp)
        self.xp=x; self.yp=y
    def left(self, ug):
        self.fi+=self.st*ug*pi/180
        x=self.xp+2*d*cos(self.fi)
        y=self.yp-2*d*sin(self.fi);
        canvas.coords(self.line,self.xp,self.yp,x,y)
    def right(self, ug):
        self.fi-=self.st*ug*pi/180
        x=self.xp+2*d*cos(self.fi)
        y=self.yp-2*d*sin(self.fi);
        canvas.coords(self.line,self.xp,self.yp,x,y)

class Turtles:
    def __init__(self):
        self.Lst=[]
        self.state='draw'
    def addTurtle(self,obj):
        self.Lst.append(obj)
    def move(self,len):
        for w in self.Lst: w.move(len)
    def forw(self,len):
        if self.state=='draw':
            for w in self.Lst: w.forw(len)
        else:
            for w in self.Lst: w.move(len)
    def back(self,len):
        if self.state=='draw':
             for w in self.Lst: w.back(len)
        else:
            for w in self.Lst: w.move(-len)
    def left(self, ug):
        for w in self.Lst: w.left(ug)
    def right(self, ug):
```

```
        for w in self.Lst: w.right(ug)
    def setColor(self,color):
        for w in self.Lst: w.cv=color
    def reaction(self,event):
        key = event.keysym
        if key == 'Up'  : self.forw(10)
        if key == 'Down': self.back(10)
        if key == 'Left': self.left(15)
        if key == 'Right': self.right(15)
        if key == 'Next': self.state='draw'
        if key == 'Prior': self.state='undraw'
        if key == '1' : self.setColor('red')
        if key == '2' : self.setColor('green')
        if key == '3' : self.setColor('blue')
        if key == '4' : self.setColor('cyan')
        if key == '5' : self.setColor('magenta')
        if key == '6' : self.setColor('yellow')
        if key == '7' : self.setColor('black')
        for w in self.Lst:
           canvas.itemconfig(w.oval,outline=w.cv)
           canvas.itemconfig(w.line,fill=w.cv)
        if key == 'space':
           for w in self.Lst:
              canvas.itemconfig(w.oval,outline='')
              canvas.itemconfig(w.line,fill='')
    def execEvent(self, cmd):
        e=Event()
        e.keysym=cmd
        self.reaction(e)
```

Приложение 3. Модуль drawers

```
# модуль drawers, версия 1.0
# импорт математической библиотеки Питона
#from math import *
# импорт графической библиотеки Питона
from tkinter import *

def create_win(w_title, c_width, c_height,
color='white'):
   global root, canvas, xs, ys
   xs=c_width/2; ys=c_height/2
# создание окна приложения с "холстом" для рисования
   root=Tk()
```

```
    root.title(w_title)
    canvas=Canvas(root, width=c_width, height=c_height,
                  bg=color)
    canvas.pack()
    return root

class Drawer:
    def __init__(self,xp=0,yp=0,color='black',width=1):
        self.xp=xs+xp
        self.yp=ys-yp
        self.cv=color
        self.wl=width
    def moveto(self,x,y):
        self.xp=xs+x
        self.yp=ys-y
    def lineto(self,x,y):
        canvas.create_line(self.xp,self.yp,xs+x,ys-y,
                           fill=self.cv,width=self.wl)
        self.xp=xs+x
        self.yp=ys-y
    def moveon(self,dx,dy):
        self.xp+=dx
        self.yp-=dy
    def lineon(self,dx,dy):
        xp=self.xp+dx
        yp=self.yp-dy
        canvas.create_line(self.xp,self.yp,xp,yp,
                           fill=self.cv,width=self.wl)
        self.xp+=dx
        self.yp-=dy

class Drawers:
    def __init__(self):
        self.t=0
        self.Lst=[]
    def addDrawer(self,obj):
        self.Lst.append(obj)
    def setColor(self,color):
        for w in self.Lst: w.cv=color
    def startPos(self,pos,col,r=3):
       self.moveto(pos)
       k=len(col)
       for i,(px,py) in enumerate(pos):
          canvas.create_oval(xs+px-r,ys-py-r,\
                xs+px+r,ys-py+r,fill=col[i%k])
```

```
   def nextPos(self,pos,func):
      dt=0.5
      while True:
         V=[(dt*func(x,y)[0], dt*func(x,y)[1])
            for x,y in pos]
         if max(dx**2+dy**2 for dx, dy in V)<2: break
         else: dt=dt*0.9
      R=[(px+Vx, py+Vy)
         for (px,py), (Vx,Vy) in zip(pos,V)]
      W=[(dt*func(x,y)[0], dt*func(x,y)[1]) for x,y in R]
      self.t+=dt
      return [(px+(Vx+Wx)/2, py+(Vy+Wy)/2)
              for (px,py),(Vx,Vy),(Wx,Wy) in
zip(pos,V,W)]
   def moveto(self, D):
      for (w,d) in zip(self.Lst,D): w.moveto(*d)
   def lineto(self, D):
      for (w,d) in zip(self.Lst,D): w.lineto(*d)
   def moveon(self, D):
      for w in self.Lst:
         for d in D: w.moveon(*d)
   def lineon(self, D):
      for w in self.Lst:
         for d in D: w.lineon(*d)
   def axes(self,x,y):
      global xs,ys
      xs=x; ys=y; m=5
      w=int(canvas['width'])
      h=int(canvas['height'])
      canvas.create_line(m,ys,w-
m,ys,width=1,fill='black')
      canvas.create_text(w-2*m,ys+2*m,text='X')
      canvas.create_line(xs,m,xs,h-
m,width=1,fill='black')
      canvas.create_text(xs-2*m,2*m,text='Y')
```

Приложение 4. Программа «Собери картинку»

```
from tkinter import *
from tkinter.messagebox import showinfo
from random import *
'''
Глобальные параметры:
    n – количество фрагментов по горизонтали;
```

```
    m – количество фрагментов по вертикали;
    xsize, ysize – ширина и высота места для размещения
фрагмента. Для визуального разделения фрагментов эти
параметры на один пиксель больше соответствующих размеров
самого фрагмента;
    xp – отступ от границ рабочей области по горизонтали;
    yp – отступ от границ рабочей области по вертикали.
'''
n=4; m=3; xsize=100; ysize=111; xp=2; yp=2
# именованные константы
NUM=0; PART=1; IMG=2; RECT=3
# окно приложения
root=Tk()
root.title('Бабочка')
canvas=Canvas(root,width=2*xp+n*xsize,
              height=2*xp+m*ysize, bg='white')
canvas.pack()
# функции выделения фрагментов
def select(A):
   canvas.itemconfig(A[RECT],width=5)
def unselect(A):
   canvas.itemconfig(A[RECT],width=0)
def change(A,B):
# обмен параметров фрагментов
   A[NUM], B[NUM]= B[NUM], A[NUM]
   A[PART],B[PART]=B[PART],A[PART]
# отображение изменений на экране
   canvas.itemconfig(A[IMG],image=A[PART])
   canvas.itemconfig(B[IMG],image=B[PART])
def check():
   for i in range(m):
      for j in range(n):
         if cells[i][j][NUM]!=n*i+j+1: return False
   return True
# обработчик событий мыши
def reaction(event):
   global state,ns,np
   x=event.x
   y=event.y
   if state==0:
   # выделяем фрагмент картинки
      np=(x-xp)//xsize
      ns=(y-yp)//ysize
      if 0<=np<n and 0<=ns<m:
         select(cells[ns][np])
```

```
         state=-1
   else:
   # меняем местами два фрагмента
      j=(x-xp)//xsize
      i=(y-yp)//ysize
      if 0<=j<n and 0<=i<m:
         unselect(cells[ns][np])
         change(cells[ns][np],cells[i][j])
   # и проверяем окончание игры
         if check():
            showinfo('Игра окончена',
                     'Задание выполнено!')
         state=0
# -------------- главная программа ---------------
# получаем случайную последовательность фрагментов
kmap=list(range(1,n*m+1))
shuffle(kmap)
# создаём структуру данных cells
cells=[]; k=0
for i in range(m):
   w=[]
   y=yp+i*ysize
   for j in range(n):
      num=kmap[k]
      x=xp+j*xsize
      name = 'D:/BF/bf_{:02}'.format(num)+'.gif'
      part = PhotoImage(file=name)
      img=canvas.create_image(x+1,y+1,
                        anchor=NW, image=part)
      rect=canvas.create_rectangle(x+3,y+3,
                         x+xsize-3,y+ysize-3,fill='',
                         outline='red',width=0)
      w.append([num,part,img,rect])
      k+=1
   cells.append(w)
# обнуляем состояние игры
state=0
# связываем событие мыши с обработчиком
canvas.bind('<Button-1>',reaction)
```

Список литературы

1. Лутц М. Программирование на Python, том I, 4-е издание.— Пер. с англ.— Спб.: Символ-Плюс, 2011.— 992 с.

2. Python 3. Самое необходимое/ Н.А.Прохоренок, В.А.Дронов.— 2-е изд.— Спб.: БХВ-Петербург, 2018.— 608с.

3. Эл Свейгарт.— Учим Python, делая крутые игры.— Пер. с англ.- М.: Эксмо, 2018.— 416 с.

4. Гутман Г.Н., Карпилова О.М. Муравьиные сказки.— М.: Просвещение, 1993.— 95 с.

5. Мочалов Л.П. Головоломки — М.: Просвещение, 1996.— 190 с.

6. Журавлев А.П. Языковые игры на компьютере.— М.: Просвещение, 1988.— 144 с.

7. Шень А. Игры и стратегии с точки зрения математики.— 5-е изд.— М.: МЦНМО, 2018.— 56 с.

Источники в интернете

7. Tkinter. Программирование GUI на Python.— сайт https://younglinux.info/tkinter.php (Лаборатория линуксоида)

Оглавление

1. Графика в Tkinter 2
2. Пропорции и прогрессии 14
3. Полярная система координат........ 25
4. Узоры из окружностей и дуг 33
5. Зеркальная симметрия 41
6. Паркеты и мозаики........ 48
7. Анимация........ 59
8. Анимация (продолжение) 63
9. Моделирование механизмов........ 70
10. Управление окружностью........ 74
11. Игра «Сквош»........ 80
12. Исполнитель «Черепашка» 85
13. Исполнитель «Чертёжник» 94
14. Рекурсия в графике 109
15. Рисование мышкой........ 119
16. Учебный проект «Буква А». 124
17. Меню. Элементы управления 127
18. Игра «Таблица чисел»........ 132
19. Проект «Тест по математике»........ 140
20. Игра «ним» 145
21. Проект «Спирограф»........ 154
22. Проект «Подбор цвета» 158
23. Игра «Полоски»........ 161
24. Проект «Ход конём» 165
25. Игра «Обмен коней» 171
26. Головоломка «Флип-флоп» 172
27. Головоломка «Открой холодильник»........ 177
28. Проект «Тест памяти»........ 178
29. Игра «Найди пару»........ 178
30. Игра «15»........ 183
31. Игра «Минигольф» 186
32. Проект "Шарики в лабиринте" 191
33. Игра «Столбики»........ 192
34. Игра «Обменяй лягушек». 199
35. Игра «Собери картинку»........ 204
36. Игра «Сложи узор» 209
37. Проект «График функции» 214
38. Проект «Построение поверхности»........ 220
39. Проект «Красота математических функций»........ 225
40. Игра «Сокобан»........ 230
41. Игра «Сокобан» (продолжение) 237
42. Исполнитель «Муравей»........ 241
43. Приложения........ 249

Распространение материалов книги без ссылки на источник запрещается!

Замечания и пожелания можно отправить на электронную почту
gutgut@mail.ru (Гутман Геннадий Натанович)