Функции

1. Основные сведения и примеры

Функция — это обособленный участок кода, который можно вызывать, обратившись к нему по имени, которым он был назван. При вызове происходит выполнение команд *тела* функции.

Функции можно сравнить с небольшими программками, которые сами по себе, т. е. автономно, не исполняются, а встраиваются в обычную программу. Нередко их так и называют – *подпрограммы*.

С точки зрения Python функция — это объект, принимающий аргументы и возвращающий значение. В отличие от основной программы функции обычно получают данные не с устройства ввода (клавиатуры, файла и др.), а из вызывающей программы. Сюда же они возвращают результат своей работы.

Вы уже знакомы с *встроенными* функциями, например, `input` , `abs` , `min` , `print` и т.д. Сейчас речь пойдет о тех функциях, которые может создавать программист.

In [1]:
```
# синтаксис описания функции
def add2(x, y, z):
    result = x**2 + y**2 + z**2
    return result
```

In [2]:
```
print(add2(1,2,5))
```

```
30
```

В предыдущем примере `add2` — имя функции, `x, y, z` — аргументы функции. В данном случае функция вычисляет (или, как говорят, ***возвращает***) сумму квадратов своих аргументов.

Используем функцию `add2` для решения задачи:

Найти значение выражения

$$y = -7\frac{\sqrt[3]{a^2 + b^2 + c^2}}{a^2 + (b-a)^2 + \frac{c^2}{4}}$$

при следующих значениях переменных:

$$a = 3, b = 4, c = 10.$$

In [3]:
```
a, b, c = 3, 4, 10
y = -7*add2(a, b, c)**(1/3) / add2(a, b-a, c/2)
print(y)
```

```
-1.0
```

```
In [4]: y = -7*(a**2 + b**2 + c**2)**(1/3) / (a**2 + (b-a)**2 + (c/2)**2)
        print(y)
```

```
-1.0
```

Аргументы и параметры, или Немножко филологии

Математика:
$y = f(x)$
x – аргумент, а y – значение функции f.

Программирование:

```
def f(t):
    return t**5
y = f(x)
```

`t` – параметр (формальный параметр) функции `f`
`x` – аргумент (фактический параметр) функции `f`

Функция может быть любой сложности и возвращать любые объекты (числа, логические значения, строки, списки, кортежи, и даже функции). Функция может и не заканчиваться инструкцией `return`, при этом функция вернет значение `None`.

Пример с функцией `add2` выше не очень показателен — использование функции не привело к заметному *упрощению* или *сокращению* кода.
Рассмотрим другой пример.

Задача. Даны длины четырех отрезков: a, b, c, d. Сколько есть возможностей выбрать три из них так, чтобы можно было сложить треугольник?

Начинаем перебирать варианты:
a, b, c
a, b, d
a, c, d
b, c, d

Смотрим на код:

```
In [5]: a, b, c, d  = 1, 2, 3, 4
        k = 0
        if max(a, b, c) < a+b+c - max(a, b, c):
            k += 1
        if max(a, b, d) < a+b+d - max(a, b, d):
            k += 1
        if max(a, c, d) < a+c+d - max(a, c, d):
            k += 1
        if max(b, c, d) < b+c+d - max(a, b, c):
            k += 1
        print(k)
```

```
1
```

Читать такой код сложно. Явно видны повторы. Можно легко запутаться при разработке и при исправлении.

Проще один раз написать правильную функцию и пользоваться ей:

In [6]:
```
def is_3(a, b, c):
    '''
    функция возвращает True,
    если из отрезков a, b, c
    можно сложить треугольник
    '''
    return  max(a, b, c) < a+b+c - max(a, b, c)

k = is_3(a, b, c) + is_3(a, b, d) + is_3(a, c, d) + is_3(b, c, d)
print(k)
```

```
1
```

Польза функций не только в возможности многократного вызова одного и того же кода из разных мест программы. Не менее важно, что благодаря им программа приобретает **структуру**. Функции как бы разделяют ее на обособленные части, каждая из которых выполняет свою конкретную задачу. В результате код становится более *прозрачным*, его гораздо проще понимать, корректировать, находить и исправлять ошибки. Каждую функцию можно отлаживать отдельно, а потом из готовых работающих кирпичиков собирать программу.

Пусть надо написать программу, вычисляющую площади разных фигур. Пользователь указывает, площадь какой фигуры он хочет вычислить. После этого вводит исходные данные. Например, длину и ширину в случае прямоугольника. Поток выполнения разделяется на несколько ветвей с использованием оператора `if-elif-else` :

In [ ]:
```
figure = input('1-прямоуг., 2-треугольник, 3-круг: ')
if figure == '1':
    a = float(input('Ширина: '))
    b = float(input('Высота: '))
    print(f'Площадь: {a*b}')
elif figure == '2':
    a = float(input('Основание: '))
    h = float(input('Высота: '))
    print(f'Площадь: {0.5 * a * h}')
elif figure == '3':
    r = float(input('Радиус: '))
    print(f'Площадь: {3.14 * r**2}')
else:
    print('Неопознанная фигура')
```

В принципе, и без функций все хорошо. Но что, если нам еще где-то в программе потребуется вычислять площадь круга? Или мы решим вычислять площадь треугольника не по высоте и основанию, а по трем сторонам (по формуле Герона, например)? Лучше написать функции!

In [ ]:
```
def rectangle():
    a = float(input('Ширина: '))
    b = float(input('Высота: '))
    return a*b

def triangle():
    a = float(input('Основание: '))
    h = float(input('Высота: '))
    return 0.5 * a * h

def circle():
    r = float(input('Радиус: '))
    return 3.14 * r**2

figure = input('1-прямоуг., 2-треугольник, 3-круг: ')
if figure == '1':
    area = rectangle()
elif figure == '2':
    area = triangle()
elif figure == '3':
    area = circle()
print(f'Площадь: {area}')
```

Что может быть параметром функции?

Всё!!!

Параметром может быть любой объект, в том числе - функция.

Еще один пример с вычислением математического выражения.

Найти значение выражения

$$\frac{\ln\alpha + \ln\gamma + \ln\gamma^2}{\cos\beta + \cos(\beta + \pi) + \cos(\beta - \pi)}$$

при следующих значениях переменных:

$$\alpha = 1, \beta = \pi, \gamma = e.$$

In [7]:
```
from math import cos, log, pi, e
alpha = 1
beta = pi
gamma = e

# аргумент функции - некоторая другая функция
def add_f(f, a, b, c):
    return f(a) + f(b) +f(c)

z1 = add_f(log, alpha, gamma, gamma*gamma)
z2 = add_f(cos, beta, beta+pi, beta-pi)
print(z1/z2)
```

```
3.0
```

2. Решение задач

Func1.

Описать функцию `sign(x)` целого типа,
возвращающую для вещественного числа `x` следующие значения:

$$\begin{cases} -1, & \text{если } x < 0; \\ 0, & \text{если } x = 0; \\ 1, & \text{если } x > 0 \end{cases}$$

С помощью этой функции найти значение выражения `sign(a) + sign(b)` для данных вещественных чисел `a` и `b`.

In [9]:
```
# напишите текст функции ниже
def sign(x):
    if x>0:
        return 1
    elif x<0:
        return -1
    return 0
```

In [10]:
```
# блок тестирования функции
# тест 1
a = 1
b = -2
print(f'sign({a})+sign({b}) = {sign(a)+sign(b)}')

# тест 2
a = 1
b = 25
print(f'sign({a})+sign({b}) = {sign(a)+sign(b)}')

# тест 3
a = -25
b = -25
print(f'sign({a})+sign({b}) = {sign(a)+sign(b)}')
```

```
sign(1)+sign(-2) = 0
sign(1)+sign(25) = 2
sign(-25)+sign(-25) = -2
```

In [13]:
```
# блок тестирования функции - 2
assert sign(1)==1,   'Test 1'
assert sign(10)==1,  'Test 2'
assert sign(0)==0,   'Test 3'
assert sign(-2)==-1, 'Test 4'
```

Func9.

Описать функцию `even(k)` логического типа, возвращающую `True`, если целый параметр `k` является четным, и `False` в противном случае. С ее помощью найти количество четных чисел в наборе из 10 целых чисел.

In [14]:
```
# напишите текст функции ниже
def even(k):
    return k%2 == 0
```

In [15]:
```
# блок тестирования функции
integers10 = 1, 121, 14, 24, 23, 565, 1, 11, 0, -5
k = 0
for n in integers10:
    if even(n):
        k += 1
print(k)
```

```
3
```

Func15.

Описать функцию `digit_n(k, n)` целого типа, возвращающую `n`-ю цифру целого положительного числа `k` (цифры в числе нумеруются справа налево, правая цифра имеет номер 1). Если количество цифр в числе `k` меньше `n`, то функция возвращает −1. Для каждого из пяти данных целых положительных чисел `k1`, `k2`, …, `k5` вызвать функцию `digit_n` с параметром `n`, изменяющимся от 1 до 5.

In [17]:
```
# напишите текст функции ниже
def digit_n(k, n):
    k //= 10**(n-1)
    if k==0:
        return -1
    return k%10
```

In [18]:
```
# блок тестирования функции
k1 = 12345
k2 = 54321
k3 = 123
k4 = 23405600
k5 = 314159265
for k in k1,k2,k3,k4,k5:
    print(f'{k=}:')
    for n in range(1,6):
        print(digit_n(k,n), end=' ')
    print()
```

```
k=12345:
5 4 3 2 1
k=54321:
1 2 3 4 5
k=123:
3 2 1 -1 -1
k=23405600:
0 0 6 5 0
k=314159265:
5 6 2 9 5
```

Func19.

Описать функцию `fact(n)` целого типа, вычисляющую значение факториала

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

(`n` > 0 — параметр целого типа). С помощью этой функции найти факториалы пяти данных целых чисел.

In [19]:
```
# напишите текст функции ниже
def fact(n):
    p = 1
    for i in range(1, n+1):
        p *= i
    return p
```

In [20]:
```
# блок тестирования функции
n1 = 0
n2 = 5
n3 = 50
n4 = 1
n5 = 8
for n in n1, n2, n3, n4, n5:
    print(f'{n}! = {fact(n)}')
```

```
0! = 1
5! = 120
50! = 30414093201713378043612608166064768844377641568960512000000000000
1! = 1
8! = 40320
```

Func24.

Описать функцию `mean(x, y)`, вычисляющую среднее арифметическое $\frac{x+y}{2}$ и среднее геометрическое $\sqrt{x \cdot y}$ двух положительных чисел x и y и возвращающую результат в виде двух вещественных чисел (`x` и `y` — вещественные параметры). С помощью этой функции найти среднее арифметическое и среднее геометрическое для пар `(a, b)`, `(a, c)`, `(a, d)`, если даны `a`, `b`, `c`, `d`.

In [21]:
```
# напишите текст функции ниже
def mean(x, y):
    a = (x+y)/2
    g = (x*y)**0.5
    return a, g
```

In [22]:
```
# блок тестирования функции
a = 10.0
b = 1000.0
c = 40.0
d = 360.0
print(mean(a,b))
mean_a, mean_g = mean(a,c)
print(mean_a, mean_g)
```

```
(505.0, 100.0)
25.0 20.0
```

3. Модули

Написанные и проверенные функции хороши тем, что их можно использовать не только в одной программе. Но если у программиста 100 полезных функций, копировать их текст из программы в программу не очень удобно. Для решения этой проблемы служат *модули*.

Строго говоря, модуль в Python – это любой файл с программой. Однако в данном контексте представляет интерес файл, содержащий написанные (и проверенные!) функции. Подключение его к программе осуществляется командой `import` .

В качестве примера рассмотрим задачу создания модуля `geometry` , содержащего набор функций для вычисления площадей геометрических фигур.

In [ ]:
```
# Файл geometry.py
# Файл необходимо поместить в ту же папку,
#      что и файл-notebook
# Еще раз обратите внимание на оформление комментариев!
def rectangle(a, b):
    '''
    функция вычисляет площадь прямоугольника
    ширины a и высоты b
    '''
    return a*b

def circle(r):
    '''
    функция вычисляет площадь круга
    радиуса r
    '''
    return 3.1415926535*r*r

def triangle(a, b, c):
    '''
    функция вычисляет площадь треугольника
    со сторонами a, b и c
    '''
    p = (a+b+c)/2
    return (p*(p-a)*(p-b)*(p-c))**0.5
```

```
In [23]: # для экспериментов
         import geometry

         print(geometry.circle(10))
```

```
314.15926535
```

Как узнать, какие функции есть в модуле?

```
In [24]: dir(geometry)
```

```
Out[24]: ['__builtins__',
          '__cached__',
          '__doc__',
          '__file__',
          '__loader__',
          '__name__',
          '__package__',
          '__spec__',
          'circle',
          'rectangle',
          'triangle']
```

```
In [25]: help(geometry.triangle)
```

```
Help on function triangle in module geometry:

triangle(a, b, c)
    функция вычисляет площадь треугольника
    со сторонами a, b и c
```

Как назвать модуль?

Поскольку модуль будет импортироваться и использоваться в качестве переменной, то к его названию предъявляются общие требования к именам. Имя модуля не может начинаться с цифры и не может совпадать с одним из *ключевых слов* языка Python, список которых можно посмотреть, выполнив следующий код

```
In [27]: import keyword
         print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'brea
k', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finall
y', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonloc
al', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yiel
d']
```

Не рекомендуется называть модуль так же, как какую-либо из встроенных функций.

Куда поместить модуль?

Самый простой вариант – в одну папку с программой, которая его вызывает. Если говорить более точно, то модуль должен находиться в той папке, в которой интерпретатор Python сможет его найти. Путь поиска модулей указан в переменной

`sys.path` стандартной библиотеки `sys`. В него включены текущая директория (та самая папка с основной программой), а также директории, в которых установлен Python. Кроме того, переменную `sys.path` можно изменять вручную, что позволяет программисту разместить все свои модули в удобном для себя месте.

In [28]:
```
import sys
sys.path
```

Out[28]:
```
['C:\\Users\\MK\\Desktop\\7 functions',
 'c:\\python38\\python38.zip',
 'c:\\python38\\DLLs',
 'c:\\python38\\lib',
 'c:\\python38',
 '',
 'c:\\python38\\lib\\site-packages',
 'c:\\python38\\lib\\site-packages\\win32',
 'c:\\python38\\lib\\site-packages\\win32\\lib',
 'c:\\python38\\lib\\site-packages\\Pythonwin',
 'c:\\python38\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\MK\\.ipython']
```

Можно ли использовать модуль как самостоятельную программу?

Можно. Обычно так часто делают, когда хотят проверить корректность работы всех функций модуля. Однако надо помнить, что при импортировании модуля его код выполняется полностью, то есть, если в модуле кроме описания функций есть и текст программы, их вызывающей и что-то при этом выводящей на экран, то в процессе импорта эта программа полностью выполнится, что-то напечатает и т.п.

Чтобы этого избежать, нужно проверить, запущен ли скрипт как программа, или импортирован. Это можно сделать с помощью переменной `__name__`, которая определена в любой программе и равна `'__main__'`, если скрипт запущен в качестве главной программы, и имени модуля, если он импортирован. Например, geometry.py может выглядеть вот так:

In [ ]:
```
# Файл geometry.py
# Файл необходимо поместить в ту же папку, что и файл-notebook
def rectangle(a, b):
    '''
    функция вычисляет площадь прямоугольника
    ширины a и высоты b
    '''
    return a*b
def circle(r):
    '''
    функция вычисляет площадь круга
    радиуса r

    '''
    return 3.1415926535*r*r
def triangle(a, b, c):
    '''
    функция вычисляет площадь треугольника
    со сторонами a, b и c
    '''
    p = (a+b+c)/2
    return (p*(p-a)*(p-b)*(p-c))**0.5

if __name__ == "__main__":
    # тестируем функции
    print(rectangle(2,3))
    print(circle(10))
    print(triangle(30,40,50))
```

In [32]:
```
import geometry_1
```

```
работает геометрия!
```

4. Тонкие вопросы

4.1 Ключевые и позиционные аргументы.

In [33]:
```
def velocity(distance, time):
    return distance/time

print(velocity(50, 2))
print(velocity(2, 50))
```

```
25.0
0.04
```

Проблема: глядя на код

```
print(velocity(2, 50))
```

трудно вспомнить/понять, где тут время, а где - расстояние.
Решение:

In [34]:
```
# поэкспериментируем
print(velocity(distance=2, time=50))
print(velocity(time=2, distance=50))
print(velocity(50, time=5))
```

```
0.04
25.0
10.0
```

In [35]:
```
# А так?
v = velocity(20, distance=5)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-35-498ec00b003e> in <module>
      1 # А так?
----> 2 v = velocity(20, distance=5)

TypeError: velocity() got multiple values for argument 'distance'
```

In [36]:
```
# А так?
v = velocity(time=20, 5)
```

```
  File "<ipython-input-36-67d05f104547>", line 2
    v = velocity(time=20, 5)
                         ^
SyntaxError: positional argument follows keyword argument
```

In [37]:
```
# или так
print(end='***', 'hello')
```

```
  File "<ipython-input-37-b7ef6e62ddfc>", line 2
    print(end='***', 'hello')
                    ^
SyntaxError: positional argument follows keyword argument
```

4.2 Значение по умолчанию

Предположим, мы хотим написать функцию вычисления веса объекта, имеющего массу `m`. Из школьной физики известна формула веса $P = mg$, где m – масса объекта, а g – ускорение свободного падения.

In [38]:
```
def p(m, g):
    return m*g
```

Приведенная функция годится для любой планеты, главное – правильно задать величину ускорения свободного падения. Но если большинство программистов будут использовать эту функцию для вычисления веса объектов, находящихся на Земле, то хорошо бы освободить их от необходимости постоянно писать 9.81 в качестве значения параметра g. Делается это так:

In [ ]:
```
def p(m, g=9.81):
    return m*g

mass = 10000 # 10 тонн
p_earth = p(mass) # вес одной тонны на Земле; использовано значение g=9.81
p_mars = p(mass, 3.86) # вес одной тонны на Марсе
print(p_earth, p_mars)
```

Число 3.86 – ускорение свободного падения на Марсе – найдено в табличке на [Википедии (https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D0%BA%D0%BE%D1%80%D0%B5%D0%E](https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D0%BA%D0%BE%D1%80%D0%B5%D0%E)

4.3 Функции с произвольным количеством аргументов

Мы уже знакомы со стандартными функциями, количество аргументов у которых может быть любым:

In [2]:
```
a = min(1,2,3,-1,-2,-10,20,30) # 8 аргументов
b = max(a*a, a+20, 25)          # 3 аргумента
print(a, b, a+b, a-b)           # 4 аргумента
```

```
-10 100 90 -110
```

In [3]:
```
sum(1,2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-a91f35d5101e> in <module>
----> 1 sum(1,2)

TypeError: 'int' object is not iterable
```

In [ ]:
```

```

Напишем функцию `summ` , которая вычисляет сумму всех своих аргументов, сколько бы их не было

In [4]:
```
def summ(*args):
    s = 0
    for x in args:
        s += x
    return s

print(summ(1))
print(summ(1,2))
print(summ(1,3,10))
print(summ())
```

```
1
3
14
0
```

А как написать ***хорошую*** функцию, вычисляющую среднее арифметическое своих аргументов?

In [5]:
```
def mean(a, *args):
    s = a
    for x in args:
        s += x
    return s/(len(args)+1)
```

In [6]:
```
mean(1,2,3)
```

Out[6]:
```
2.0
```

In [7]:
```
# а так?
mean()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-ae6b33bfef6c> in <module>
      1 # а так?
----> 2 mean()

TypeError: mean() missing 1 required positional argument: 'a'
```

Количество ключевых аргументов тоже может быть произвольным. Но чтобы работать с ними, нужно сначала познакомиться с такой структурой данных как словарь. Займемся этим позже...

4.4 Области видимости переменных

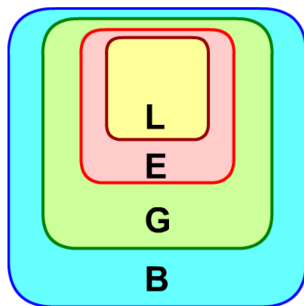
Областью видимости имени переменной называется та часть программного кода, в которой к переменной с этим именем гарантируется однозначный доступ. Под переменной в этом контексте понимаются все *именованные* объекты языка –

собственно переменные, функции, объекты и т.д. Переменную можно использовать только в пределах ее области видимости.

Переменная *видна* в том блоке, где она первый раз появилась или определена. В таблице ниже – *причины* появления переменной:

Действие	**Оператор**
Присваивание	`x = value`
Подключение модулей	`import module` `from module import name`
Определение функции	`def my_func():` `    ...`
Задание параметров функции	`def my_func(arg1, arg2,... argN):` `    ...`
Описание класса	`class MyClass:` `    ...`

Области видимости в Python



Local – локальная,

Enclosing – объемлющая (другое название - nonlocal),

Global – глобальная,

Built-in – встроенная

- Локальная область видимости – это блок кода (или тело) любой функции. В ней находятся имена, которые вы определяете внутри функции. Эти имена будут видны только в пределах тела функции. Локальная область видимости создается при вызове функции, а не при ее определении, поэтому у вас будет столько же различных локальных областей, сколько различных вызовов функций произошло в программе. Каждый вызов приводит к созданию новой локальной области.
- Объемлющая (или охватывающая, или нелокальная) область видимости – это локальная область видимости функции *с точки зрения* вложенной (описанной внутри нее) функции. В примере ниже переменная `a` будет создана в объемлющей области с точки зрения функции `g` и в локальной области с точки зрения функции `f` .

```
def f(x):
    a = 1
    def g(z):   # пример вложенной функции
        return z+2
    return g(a+x)
```

- Глобальная область видимости – это самая верхняя область в программе или модуле. Имена в этой области видны (т. е. доступны) всюду в программном коде.

- Встроенная область видимости создается или загружается всякий раз, когда вы запускаете скрипт или открываете интерактивный сеанс. Эта область содержит

In [8]:
```
x1 = 'global'
def f():
    def g():
        print(x1)
    g()

f()
```

```
global
```

In [9]:
```
def f():
    x2 = 'enclosing'
    def g():
        print(x2)
    g()

f()
```

```
enclosing
```

In [10]:
```
x2
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-10-581de19c31fb> in <module>
----> 1 x2

NameError: name 'x2' is not defined
```

In [11]:
```
def f():
    def g():
        x3 = 'local'
        print(x3)
    g()

f()
```

```
local
```

In [12]:
```
def f():
    def g():
        print(min)
    g()

f()
```

```
<built-in function min>
```

Еще раз. Локальные переменные доступны только внутри тела функции. Попытка их использовать "снаружи" приводит к ошибке:

In [13]:
```
def add(a,b):
    local = a+b
    print(f'{a}+{b}={local}')

a = 5
b = 10
add(a,b)
print(local)
```

```
5+10=15

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-56de837c48d3> in <module>
      6 b = 10
      7 add(a,b)
----> 8 print(local)

NameError: name 'local' is not defined
```

Механизмы устранения "конфликтов" имен

Что выведет программа, 10 или 100?

In [14]:
```
z = 10
def z_print():
    z = 100
    print(f'::::::> {z}')

z_print()
print(z)
```

```
::::::> 100
10
```

В общем случае поиск переменной идет по схеме **L** → **E** → **G** → **B**

Но посмотрите на следующий пример. Почему при вызове второй функции возникает ошибка?

In [15]:
```
x = 100

def x_print1():
    print(f'x плюс 1 = {x+1}')

def x_print2():
    x = x+1
    print(f'x плюс 1 = {x}')

x_print1()
x_print2()
```

```
x плюс 1 = 101

---------------------------------------------------------------------------
-
UnboundLocalError                         Traceback (most recent call las
t)
<ipython-input-15-b86d5cc22326> in <module>
      9 
     10 x_print1()
---> 11 x_print2()

<ipython-input-15-b86d5cc22326> in x_print2()
      5 
      6 def x_print2():
----> 7     x = x+1
      8     print(f'x плюс 1 = {x}')
      9 

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

Кроме правила поиска имени **L** → **E** → **G** → **B** нужно помнить правила *появления* переменных (Таблица выше). Если мы что-то присваиваем переменной внутри тела функции, переменная создается в этот момент, а значит, является локальной. Локальная переменная `x` внутри функции `x_print2` еще не имеет значения, поэтому выражение `x + 1` не может быть вычислено.

А если очень-очень нужно...

Может ли функция `x_print2` все-таки изменить значение глобальной переменной `x` ?

Да. Для этого используется описатель `global` в теле функции. В Python версии 3 появился ещё и описатель `nonlocal` .

In [16]:
```
x = 100

def x_print2():
    global x
    x = x+1
    print(f'x плюс 1 = {x}')

x_print2()
print(x)
```

```
x плюс 1 = 101
101
```

Глобальные переменные и нелокальные переменные

In [17]:
```
# использование глобальной переменной
def enclosing_f():
    p = 20
    def local_f(x):
        global  p
        p += x
        return p
    print (f'Значение локальной функции: {local_f(1)}')

p = 10 # используется это значение!
enclosing_f()
```

```
Значение локальной функции: 11
```

In [18]:
```
# использование нелокальной переменной
def enclosing_f():
    p = 20 # используется это значение!
    def local_f(x):
        nonlocal p
        p += x
        return p
    print (f'Значение локальной функции: {local_f(1)}')

p = 10
enclosing_f()
```

```
Значение локальной функции: 21
```

In [19]:
```
# попытка объявления нелокальной переменной,
# отсутствующей в объемлющей функции,
# приводит к ошибке на этапе компиляции
def enclosing_function():
    q = 20
    def local_function(x):
        nonlocal p
        p += x
        return p
    print (local_function(10))

p = 10
enclosing_function()
```

```
  File "<ipython-input-19-b79f4f91b855>", line 7
    nonlocal p
    ^
SyntaxError: no binding for nonlocal 'p' found
```

Можно ли придумать осмысленный пример, когда функция **должна** менять значение глобальной переменной?

Да, один пример см. ниже. Таким же образом будет использована глобальная переменная при решении задачи **Recur4**. Однако существует общее мнение: по возможности лучше избегать использования глобальных переменных вообще, и их изменения внутри функции – в особенности.

In [20]:
```
# Пример с изменением глобальной переменной
def get_candy():
    global candy
    candy += 1
    print(f'Теперь у меня {candy} конфет.')

candy = 5
get_candy()
get_candy()
get_candy()
print(candy)
```

```
Теперь у меня 6 конфет.
Теперь у меня 7 конфет.
Теперь у меня 8 конфет.
8
```

Про области видимости, локальные и глобальные переменные можно ещё почитать, например, тут:
https://realpython.com/python-scope-legb-rule/ (https://realpython.com/python-scope-legb-rule/) (на английском)
https://habr.com/ru/company/otus/blog/487952/ (https://habr.com/ru/company/otus/blog/487952/) (на русском)

Для закрепления.
Сколько раз будет выведено `x=20` , а сколько `x=10` ?

In [21]:
```
def f(x):
    print(f'внутри f. 1: x={x}, id={id(x)}')
    x = 10
    print(f'внутри f. 2: x={x}, id={id(x)}')

x = 20
print(f'** вне f. 1: x={x}, id={id(x)}')
f(x)
print(f'** вне f. 2: x={x}, id={id(x)}')
```

```
** вне f. 1: x=20, id=8791470770432
внутри f. 1: x=20, id=8791470770432
внутри f. 2: x=10, id=8791470770112
** вне f. 2: x=20, id=8791470770432
```

In [22]:
```
# А если так?

def f():
    print(f'внутри f. 1: x={x}, id={id(x)}')
    x = 10
    print(f'внутри f. 2: x={x}, id={id(x)}')

x = 20
print(f'** вне f. 1: x={x}, id={id(x)}')
f()
print(f'** вне f. 2: x={x}, id={id(x)}')
```

```
** вне f. 1: x=20, id=8791470770432

---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call las
t)
<ipython-input-22-85a49beaeaeb> in <module>
      8 x = 20
      9 print(f'** вне f. 1: x={x}, id={id(x)}')
---> 10 f()
     11 print(f'** вне f. 2: x={x}, id={id(x)}')

<ipython-input-22-85a49beaeaeb> in f()
      2 
      3 def f():
----> 4     print(f'внутри f. 1: x={x}, id={id(x)}')
      5     x = 10
      6     print(f'внутри f. 2: x={x}, id={id(x)}')

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

4.5 Оператор-заглушка. Ключевое слово `pass`

При разработке программ сложной структуры часто сначала составляется список функций, а потом начинается пошаговое написание их кода и тестирование. Однако код, приведенный ниже, не позволит запустить тестирование из-за синтаксических ошибок.

In [23]:
```python
def rectangle(a, b):
    return a*b

def circle(r):
    # напишем потом

if __name__ == "__main__":
    # тестируем функции
    print(rectangle(2,3))
    # потестируем потом
    # print(circle(10))
```

```
  File "<ipython-input-23-7b14c7f68a03>", line 7
    if __name__ == "__main__":
    ^
IndentationError: expected an indented block
```

Возможный выход – использование ключевого слова `pass`, которое означает в данном контексте "не делаем ничего, переходим к следующему участку кода". Данное ключевое слово часто используется в процессе разработки и отладки собственных классов (тема "ООП на Питон" будет позже).

In [25]:
```python
def rectangle(a, b):
    return a*b

def circle(r):
    pass

if __name__ == "__main__":
    # тестируем функции
    print(rectangle(2,3))
    print(circle(10))
```

```
6
None
```

5. Рекурсия

Чтобы понять рекурсию,
нужно сначала понять рекурсию

Мы видели, функция может вызывать *другую* функцию. Но функция также может вызывать и саму себя! Такой вызов называется **рекурсией**, а сама функция называется рекурсивной.

Рассмотрим это на примере функции вычисления факториала.

Recur1

Описать **рекурсивную** функцию `fact_r(n)` целого типа, вычисляющую значение факториала

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

(n > 0 — параметр целого типа). С помощью этой функции найти факториалы пяти данных целых чисел.

Рекурсивное определение факториала:

$$n! = (n-1)! \cdot n$$
$$0! \equiv 1$$

In [26]:
```
# напишите текст функции ниже
def fact_r(n):
    if n==0:
        return 1
    else:
        return fact_r(n-1)*n
```

In [27]:
```
# блок тестирования функции
nn = 10, 5, 50, 1, 8
for n in nn:
    print(f'{n}! = {fact_r(n)}')
```

```
10! = 3628800
5! = 120
50! = 30414093201713378043612608166064768844377641568960512000000000000
1! = 1
8! = 40320
```

Некоторые выводы

- В описании рекурсивной функции должна присутствовать проверка условия, определяющего, по какой ветке пойдет вычисление: по рекурсивной — в этой ветке произойдёт вызов функцией себя самой, или **терминальной**, которая закончит вычисление и вернёт результат.
- Отсутствие терминальной ветки или ошибки в реализации условия перехода на эту ветку приведут к зацикливанию программы.
- К тестированию рекурсивных функций нужно относиться особенно ответственно. Особенно важно проверять срабатывание условия завершения рекурсии.

Знаменитая рекурсия – но программировать так не надо :)

In [ ]:
```
def short_story():
    print("У попа была собака, он её любил.")
    print("Она съела кусок мяса, он её убил,")
    print("В землю закопал,")
    print("Надпись написал, что")
    short_story()
```

In [ ]:
```
# Запускать на свой страх и риск!
# В 99% случаев вызывает крах jupyter-сервера
short_story()

#Можно запустиь в IDLE - там не страшно, см. ниже про RecursionError
```

В литературе рекурсия встречается довольно часто. Вот, например, цитата из Станислава Лема ("Звездные дневники Ийона Тихого. Путешествие четырнадцатое"), где главный герой ищет в энциклопедии, что такое ***сепульки***:

```
     Нашел следующие краткие сведения:
     "СЕПУЛЬКИ  -  важный  элемент цивилизации ардритов  (см.)  с  планеты
Энтеропия (см.). См. СЕПУЛЬКАРИИ".
     Я последовал этому совету и прочел:
     "СЕПУЛЬКАРИИ - устройства для сепуления (см.)".
     Я поискал "Сепуление"; там значилось:
     "СЕПУЛЕНИЕ  - занятие ардритов (см.) с планеты Энтеропия  (см.).  См.
СЕПУЛЬКИ".
     Круг  замкнулся, больше искать было негде.
```

Рекурсия без терминальной ветви имеет еще одно название – **порочный круг.**

Глубина рекурсии. Стек.

Существует предел глубины возможной рекурсии, который зависит от реализации Python. Когда предел достигнут, возникает исключение `RecursionError` .

Сколько раз сработала функция в предыдущем примере?
Чтобы узнать это, немножко откорректируем текст, убрав вывод на печать - из-за большого объема этого вывода и может произойти крах jupiter.

In [2]:
```
story_counter = 0
def short_story():
    global story_counter
#     print("У попа была собака, он её любил.")
#     print("Она съела кусок мяса, он её убил,")
#     print("В землю закопал,")
#     print("Надпись написал, что")
    story_counter += 1
    short_story()

short_story()
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-2-6f15d91f70be> in <module>
      9     short_story()
     10 
---> 11 short_story()

<ipython-input-2-6f15d91f70be> in short_story()
      7 #     print("Надпись написал, что")
      8     story_counter += 1
----> 9     short_story()
     10 
     11 short_story()

<ipython-input-2-6f15d91f70be> in short_story()
      7 #     print("Надпись написал, что")
      8     story_counter += 1
```

In [3]:
```
story_counter
```

Out[3]: 63

Узнать и/или изменить предел глубины рекурсии можно с помощью модуля `sys`:

```
import sys
sys.setrecursionlimit(limit)
sys.getrecursionlimit()
```

In [1]:
```
# покспериментируем здесь и потом снова вызовем функцию short_story:
import sys
sys.setrecursionlimit(100)
```

Recur3

Описать рекурсивную функцию `power_n(x, n)` вещественного типа, находящую значение `n`-й степени числа `x` по формулам:

$$x^n = \begin{cases} 1, & \text{если } n = 0, \\ \left(x^{\frac{n}{2}}\right)^2, & \text{при четных } n > 0, \\ x \cdot x^{n-1}, & \text{при нечетных } n > 0, \\ 1/x^{-n}, & \text{при } n < 0 \end{cases}$$

($x \neq 0$ — вещественное число, n — целое; в формуле для четных n должна

In [4]:
```
# напишите текст функции ниже
def power_n(x, n):
    if n==0:
        return 1
    elif n<0:
        return 1/power_n(x, -n)
    elif n%2==0:
        y = power_n(x, n//2)
        return y*y
    else:
        return x*power_n(x, n-1)
```

In [5]:
```
# блок тестирования функции
x = 2.0
nn = -2, 0, 10, 5, 1000
for n in nn:
    print(power_n(x, n))
```

```
0.25
1
1024.0
32.0
1.0715086071862673e+301
```

In [6]:
```
2**1000.0
```

Out[6]:
```
1.0715086071862673e+301
```

Recur 4.

Описать рекурсивную функцию `fib1(n)` целого типа, вычисляющую `n`-й элемент последовательности *чисел Фибоначчи* (`n` — целое число):

$$f_1 = f_2 = 1,\ f_k = f_{k-2} + f_{k-1},\ k = 3, 4, \dots .$$

С помощью этой функции найти пять чисел Фибоначчи с данными номерами, и вывести эти числа вместе с количеством рекурсивных вызовов функции `fib1`, потребовавшихся для их нахождения.

In [8]:
```
counter = 0
# напишите текст функции ниже
def fib1(n):
    global counter
    counter += 1
    if n==1 or n==2:
        return 1
    else:
        return fib1(n-2) + fib1(n-1)
```

In [11]:
```
# блок тестирования функции
nn =  10, 15, 20, 25, 30
for n in nn:
    counter = 0
    print(fib1(n), counter)
```

```
55 109
610 1219
6765 13529
75025 150049
832040 1664079
```

Попробуем сделать вывод...

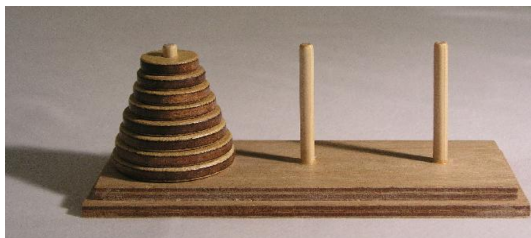
Связь рекурсии и стека: задача «Разворот последовательности» с сайта http://pythontutor.ru (http://pythontutor.ru)

Дана последовательность целых чисел, заканчивающаяся числом 0. Выведите эту последовательность в обратном порядке. При решении этой задачи ***нельзя*** пользоваться массивами и прочими динамическими структурами данных.

In [ ]:
```
def reverse():
    x = int(input("x = (0 для завершения)"))
    if x!=0:
        reverse()
    print(x)

reverse()
```

Ханойские башни



Оригинальная головоломка. Даны три стержня, на один из которых нанизаны восемь колец, причём кольца отличаются размером и лежат меньшее на большем. Задача состоит в том, чтобы перенести пирамиду из восьми колец за наименьшее число ходов на другой стержень. За один раз разрешается переносить только одно кольцо, причём нельзя класть большее кольцо на меньшее.

Легенда. В Великом храме города Бенарес, под собором, отмечающим середину мира, находится бронзовый диск, на котором укреплены 3 алмазных стержня, высотой в один локоть и толщиной с пчелу. Давным-давно, в самом начале времён, монахи этого монастыря провинились перед богом Брахмой. Разгневанный Брахма воздвиг три

высоких стержня и на один из них возложил 64 диска, сделанных из чистого золота. Причём так, что каждый меньший диск лежит на большем. Как только все 64 диска будут переложены со стержня, на который Брахма сложил их при создании мира, на другой стержень, башня вместе с храмом обратятся в пыль и под громовые раскаты погибнет мир.

Источник информации: статья в Википедии (https://ru.wikipedia.org/wiki/%D0%A5%D0%B0%D0%BD%D0%BE%D0%B9%D1%81%D0%%I

Демонстрация (towers.swf)

Задача. Разработать программу, перемещающую диски с левого стержня на правый по указанному правилу.

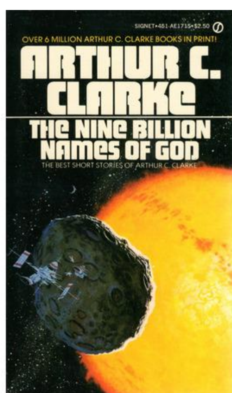
In [15]:
```
# текст программы
def move(n, start, end, empty):
    if n>0:
        move(n-1, start, empty, end)
        print(f'{start} ==> {end}')
        move(n-1, empty, end, start)

move(3, start=1, end=3, empty=2)
```

```
1 ==> 3
1 ==> 2
3 ==> 2
1 ==> 3
2 ==> 1
2 ==> 3
1 ==> 3
```

In [13]:
```
(2**64-1)//60//60//24//365.25
```

Out[13]:
```
584542046090.0
```



Идея использована в рассказе: Артур Кларк. Девять миллиардов имён Бога. (https://librebook.me/the_nine_billion_names_of_god/vol1/1)

Чтобы хорошо сдать экзамен, лучше бы прочитать!

Визуализация рекурсии (для самостоятельной работы)

Источник: http://aliev.me/runestone/Recursion/intro-VisualizingRecursion.html (http://aliev.me/runestone/Recursion/intro-VisualizingRecursion.html)

Черепашья графика.

http://acm.mipt.ru/twiki/bin/view/Cintro/PythonTurtleInfo (http://acm.mipt.ru/twiki/bin/view/Cintro/PythonTurtleInfo) (кратко, на русском)

https://docs.python.org/3/library/turtle.html (https://docs.python.org/3/library/turtle.html)

In [ ]:
```
# программа откроется в отдельном окне
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

Подключение (упрощенной) черепашьей графики для jupyter notebook

Выполнить в командной строке (Windows PowerShell)

```
pip install ipyturtle
jupyter nbextension enable --py --sys-prefix ipyturtle
```

Основные команды управления черепашкой:

```
back(length)
forward(length)
left(degree)
right(degree)
pendown()
penup()
pencolor(r, g, b)
showturtle()
hideturtle()
reset()
```

Инициализация окна рисования

In [ ]:
```
from ipyturtle import Turtle
my_t = Turtle(fixed=False)
my_t
```

In [ ]:
```
sys.setrecursionlimit(1000)
# рекурсивное рисование спирали
def draw_spiral(t, line_len):
        if line_len > 0:
            t.forward(line_len)
            t.right(90)
            draw_spiral(t,line_len-3)

# холст в исходном положении
my_t.reset()
# опускаем черепашку в левый нижний угол
my_t.penup()
my_t.back(150)
my_t.left(90)
my_t.forward(150)
my_t.right(90)
my_t.pendown()
# рисуем спираль
draw_spiral(my_t,300)
```

In [ ]:
```
# рекурсивное рисование дерева
def tree(branch_len,t):
    t.pencolor(r=4*branch_len, g=255-3*branch_len, b = 2*branch_len)
    t.pendown()
    if branch_len > 5:
        t.forward(branch_len)
        t.right(20)
        tree(branch_len-7,t)
        t.left(40)
        tree(branch_len-7,t)
        t.right(20)
        t.penup()
        t.back(branch_len)

# холст в исходном положении
my_t.reset()
# опускаем черепашку вниз
my_t.penup()
my_t.back(100)
my_t.pendown()
# рисуем дерево
tree(50,my_t)
my_t.hideturtle()
```

Вместо послесловия

> **12.1.1 Why Functions?**
>
> There is a long and disreputable tradition of writing very long functions – hundreds of lines long. I once encountered a single (handwritten) function with more than 32,768 lines of code. Writers of such functions seem to fail to appreciate one of the primary purposes of functions: to break up complicated computations into meaningful chunks and name them. We want our code to be comprehensible, because that is the first step on the way to maintainability. The first step to comprehensibility is to break computational tasks into comprehensible chunks (represented as functions and classes) and name those. Such functions then provide the basic vocabulary of computation, just as the types (built-in and user-defined) provide the basic vocabulary of data. ... Next, we can compose functions representing common or specialized tasks into larger computations.
>
> The number of errors in code correlates strongly with the amount of code and the complexity of the code. Both problems can be addressed by using more and shorter functions. Using a function to do a specific task often saves us from writing a specific piece of code in the middle of other code; making it a function forces us to name the activity and document its dependencies. Also, function call and return saves us from using error-prone control structures...
>
> The most basic advice is to keep a function of a size so that you can look at it in total on a screen. Bugs tend to creep in when we can view only part of an algorithm at a time. For many programmers that puts a limit of about 40 lines on a function. My ideal is a much smaller size still, maybe an average of 7 lines.
>
> In essentially all cases, the cost of a function call is not a significant factor ... Use functions as a structuring mechanism.

Bjarne Stroustrup. The C++ Programming Language. Fourth Edition. Addison-Wesley, 2013.

In [ ]: