Лекция 4 "Функции"

Финансовый университет при Правительстве РФ, лектор С.В. Макрушин

v 0.8 11.08.2021

Разделы:

-

- к оглавлению

In [2]:

```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v1.css")
HTML(html.read().decode('utf-8'))
```

Out[2]:

Создание функции и ее вызов

- к оглавлению

Функция описывается с помощью ключевого слова `def` по следующей схеме:

```
def <Имя функции> ([<Параметры>]):
    ["""  Строка документирования  """]
    <Тело функции>
    [return <Значение>]
```

- **Имя функции** должно быть корректным уникальным идентификатором
  - состоящим из латинских букв, цифр и знаков подчеркивания (причем имя не может начинаться с цифры).

    - в качестве имени нельзя использовать ключевые слова, кроме того, следует избегать совпадений с названиями встроенных идентификаторов.
    - регистр символов в названии функции имеет значение.
- После имени функции в круглых скобках можно указать один или несколько **параметров** через запятую.
    - Если функция не принимает параметры, то просто указываются круглые скобки.
    - После круглых скобок ставится двоеточие.
- После двоеточия может следовать необязательная **строка документирования** фукнции (распространено использование многострочной строки, заключенной в тройные кавычки)
- **Тело функции** является составной конструкцией. Как и в любой составной конструкции, инструкции внутри функции выделяются одинаковым количеством пробелов слева.

In [1]:

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

In [2]:

```
func()
```

Необязательная инструкция `return` позволяет вернуть значение из функции. После исполнения этой инструкции выполнение функции будет завершено.

In [3]:

```
def  func():
    print ("Текст до инструкции return")
    return  "Возвращаемое  значение"
    print ("Эта инструкция никогда не будет выполнена")
```

In [6]:

```
r = func()
print(f'Результат: {r}')
```

```
Текст до инструкции return
Результат: Возвращаемое  значение
```

In [4]:

```
print(func())  # вызываем функцию
```

```
Текст до инструкции return
Возвращаемое  значение
```

In [8]:

```
def print_ok():
    """  Пример  функции  без
            параметров  """
    print("Сообщение при удачно выполненной операции")

def echo(m):
    """  Пример функции с параметром """
    print(m)

def summa(x, y):
    """  Пример функции с параметрами,
    возвращающей сумму двух переменных"""
    return x + y

def pow2(x, y):
    """  Пример функции с параметрами,
    возвращающей значение x в степени y"""
    return x ** y
```

In [9]:

```
print_ok()
```

```
Сообщение при удачно выполненной операции
```

In [10]:

```
echo("Сообщение")
```

```
Сообщение
```

In [11]:

```
x = summa(5, 2)
x
```

Out[11]:

```
7
```

In [13]:

```
y, x = 10, 40
z = summa(y, x)
z
```

Out[13]:

```
50
```

Имя переменной в вызове функции может не совпадать с именем переменной в определении функции. Кроме того, глобальные переменные x и y не конфликтуют с одноименными переменными в определении функции, т.к. они расположены в разных областях видимости. Переменные, указанные в определении функции, являются **локальными** и доступны только внутри функции.

In [14]:

```
# в Python нет ограничений на тип значений передваемых в функцию:
summa('str', 'ing')
```

Out[14]:

```
'string'
```

In [15]:

```
summa([1, 3], [5, 7])
```

Out[15]:

```
[1, 3, 5, 7]
```

Все в языке Python является объектом, например, строки, списки и даже типы данных и функции.

Инструкция `def` создает объект, имеющий тип `function`, и сохраняет ссылку на него в идентификаторе, указанном после инструкции `def`. Таким образом, мы можем сохранить ссылку на функцию в другой переменной. Для этого название функции указывается без круглых скобок.

In [16]:

```
f = summa
v = f(10, 20)
v
```

Out[16]:

```
30
```

In [17]:

```
type(summa)
```

Out[17]:

```
function
```

In [18]:

```
type(f)
```

Out[18]:

```
function
```

In [19]:

```
# функции в Python можно передавать в качестве аргументов других функций:
def  func(fp,  a,  b):
    """ Через переменную fp будет доступна ссылка на
    функцию summa() """
    return fp(a, b)  #  Вызываем функцию summa()
```

In [20]:

```
func(summa, 10, 20)
```

Out[20]:

```
30
```

In [23]:

```
func(f, 10, 20)
```

Out[23]:

```
30
```

In [24]:

```
def razn(a, b):
    return a - b
```

In [25]:

```
func(razn, 10, 20)
```

Out[25]:

```
-10
```

Объекты функций поддерживают множество **атрибутов**. Обратиться к атрибутам функции можно, указав атрибут после названия функции через точку. Например, через атрибут `__name__` можно получить название функции в виде строки, через атрибут `__doc__` - строку документирования и т.д.

In [26]:

```
summa.__name__
```

Out[26]:

```
'summa'
```

In [27]:

```
f.__name__
```

Out[27]:

```
'summa'
```

Выведем названия всех атрибутов функции с помощью встроенной функции dir():

In [13]:

```
dir(summa)
```

Out[13]:

```
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__get__',
 '__getattribute__',
 '__globals__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__kwdefaults__',
 '__le__',
 '__lt__',
 '__module__',
 '__name__',
 '__ne__',
 '__new__',
 '__qualname__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

>

Расположение определений функций

- к оглавлению

Все инструкции в программе на Python выполняются последовательно сверху вниз. Это означает, что прежде чем использовать идентификатор в программе, его необходимо предварительно определить, присвоив ему значение. Поэтому, **определение функции** должно быть выполнено **перед вызовом**

функции (обычно это означает, что определение находится раньше вызова).

In [28]:

```
# неверно!
f2()

def f2():
    print('Функция f2!')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-08d6de4e1d83> in <module>
      1 # неверно!
----> 2 f2()
      3
      4 def f2():
      5     print('Функция f2!')

NameError: name 'f2' is not defined
```

In [30]:

```
for i in range(5):
    if i > 2:
        f2()
    def f2():
        print('Функция f2!')
# работает т.к. определение функции было выполнено ранее, чем вызов
```

```
Функция f2!
Функция f2!
```

In [31]:

```
# Определение функции в зависимости от условия
n = input("Введите 1 для вызова первой функции:")
n = n.rstrip("\r")
if n == "1":
    def echo():
        print("Вызов первой функции")
else:
    def echo():
        print ("Апьтернативная функция")
echo() # Вызываем функцию
```

```
Введите 1 для вызова первой функции:1
Вызов первой функции
```

Инструкция `def` всего лишь присваивает ссылку на объект функции идентификатору, расположенному после ключевого слова def. Если определение одной функции встречается в программе несколько раз, то будет использоваться функция, которая расположена последней.

>

Анонимные функции

- к оглавлению

Помимо обычных функций язык Python позволяет использовать анонимные функции, которые называются **лямбда-функциями**. Анонимная функция описывается с помощью ключевого слова `lambda` по следующей схеме:

```
lamda [<Параметр1>[, ... , <ПараметрN>]]: <Возвращаемое значение>
```

После ключевого слова `lambda` можно указать передаваемые параметры. В качестве параметра `<Возвращаемое значение>` указывается **выражение** (не составное!), результат выполнения которого будет возвращен функцией.

Как видно из схемы, у лямбда-функций нет имени. По этой причине их и называют анонимными функциями.

В качестве значения **лямбда-функция возвращает ссылку на объект-функцию**, которую можно сохранить в переменной или передать в качестве параметра в другую функцию. Вызвать лямбда-функцию можно, как и обычную, с помощью круглых скобок, внутри которых расположены передаваемые параметры.

In [32]:

```
f1 = lambda: 10 + 20  # функция без параметров
f2 = lambda x, y:  x + y  # функция с двумя параметрами
f3 = lambda x, y, z: x + y + z  # функция с тремя параметрами
print(f1())
print(f2(5, 10))
print(f3(5, 10, 30))
```

```
30
15
45
```

In [34]:

```
arr  =  ["единица1",  "Единый",  "Единица2"]
arr.sort()
print(arr)
# распространенный пример использования лямбда-функций:
arr.sort(key=lambda s: s.lower())
print(arr)
```

```
['Единица2', 'Единый', 'единица1']
['единица1', 'Единица2', 'Единый']
```

In [35]:

```
ls = ['дыня', 'вишня', 'арбуз', 'яблоко', 'абрикос']
```

In [36]:

```
sorted(ls)
```

Out[36]:

```
['абрикос', 'арбуз', 'вишня', 'дыня', 'яблоко']
```

In [15]:

```
sorted(ls, key=lambda s:len(s))
```

Out[15]:

```
['дыня', 'вишня', 'арбуз', 'яблоко', 'абрикос']
```

In [37]:

```
sorted(ls, key=len)
```

Out[37]:

```
['дыня', 'вишня', 'арбуз', 'яблоко', 'абрикос']
```

>

Необязательные параметры функций и сопоставление по ключам

- к оглавлению

Чтобы сделать некоторые параметры необязательными, следует в определении функции присвоить этому параметру начальное значение (значение по умолчанию).

In [38]:

```
def summa(x, y=2):
    return x + y
```

In [39]:

```
summa(5)
```

Out[39]:

```
7
```

In [40]:

```
summa(5, 4)
```

Out[40]:

```
9
```

Синтаксис определения параметров функции не позволяет указывать параметры, не имеющие значений по умолчанию, после параметров со значениями по умолчанию.

In [41]:

```
# ошибка:
def bad(a, b=1, c):
    return a + b + c
```

```
  File "<ipython-input-41-b8660933d718>", line 2
    def bad(a, b=1, c):
            ^
SyntaxError: non-default argument follows default argument
```

Язык Python позволяет также передать значения в функцию, используя **сопоставление по ключам**. Для этого при вызове функции параметрам присваиваются значения. Последовательность указания параметров может быть произвольной.

```
def summa(x, y):
    """  Пример функции с параметрами,
    возвращающей  сумму двух  переменных"""
    return x + y
```

In [42]:

```
summa(y=10, x=20)
```

Out[42]:

```
30
```

Сопоставление по ключам очень удобно использовать, если функция имеет несколько необязательных параметров. В этом случае не нужно перечислять все значения, а достаточно присвоить значение нужному параметру.

In [43]:

```
def multi_summa(x=1, y=1, z=1):
    return x + 10 * y + 100 * z
```

In [44]:

```
multi_summa(1, 2, 3)
```

Out[44]:

```
321
```

In [45]:

```
multi_summa(1, 2)
```

Out[45]:

```
121
```

In [46]:

```
multi_summa(5)
```

Out[46]:

```
115
```

In [47]:

```
multi_summa()
```

Out[47]:

```
111
```

In [48]:

```
multi_summa(z=3)
```

Out[48]:

```
311
```

Значения по умолчанию создаются на этапе выполнения инструкции `def` (то есть в момент создания функции), а не в момент ее вызова. Для неизменяемых аргументов, таких как строки или числа, это не имеет никакого значения, но, при использовании изменяемых аргументов в качестве значения по умолчанию, может появиться труднообнаружимая проблема.

In [49]:

```
def append_if_even(x, lst=[]): # ОШИБКА!
    if x % 2 == 0:
        lst.append(x)
    return lst
```

In [50]:

```
append_if_even(2)
```

Out[50]:

```
[2]
```

In [51]:

```
append_if_even(4)
```

Out[51]:

```
[2, 4]
```

In [52]:

```
# пример решения задачи без ошибки:
def append_if_even(x, lst=None):
    if lst is None:
        lst = []
    if x % 2 == 0:
        lst.append(x)
    return lst
```

In [53]:

```
append_if_even(2)
```

Out[53]:

```
[2]
```

In [54]:

```
append_if_even(4)
```

Out[54]:

```
[4]
```

>

Возвращение нескольких значений из функции

- к оглавлению

In [78]:

```
def min_med_max(l):
    "Возвращает минимальное, медианное и максимальное значение итерируемого объекта (списка
    ls = sorted(l)
    l_min = ls[0]
    l_max = ls[-1]
    l_med = ls[len(ls)//2]
    return l_min, l_med, l_max # удобный синтаксис для запаковки результатов в кортеж
```

In [83]:

```
min_med_max([3, 1, 100, 11, 7])
```

Out[83]:

```
(1, 7, 100)
```

In [80]:

```
# распаковка кортежа, возвращаемого функцией:
min1, med1, max1 = min_med_max([3, 1, 100, 11, 7])
```

In [81]:

```
print(f'min1: {min1}, med1: {med1}, max1: {max1}')
```

```
min1: 1, med1: 7, max1: 100
```

In [82]:

```
all_res = min_med_max([3, 1, 100, 11, 7])
all_res
```

Out[82]:

```
(1, 7, 100)
```

Распаковка и запаковка параметров функции

- к оглавлению

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ * .

In [85]:

```
t1 = (5, 10 ,15)
t2 = [25, 35, 45]
t3 = (5, 10 ,15, 20)
```

In [86]:

```
# неудобный способ передачи параметров:
multi_summa(t1[0], t1[1], t1[2])
```

Out[86]:

```
1605
```

In [87]:

```
# распаковка (картеж распаковывается в позиционные аргументы):
multi_summa(*t1)
```

Out[87]:

```
1605
```

In [88]:

```
multi_summa(*t2)
```

Out[88]:

```
4875
```

In [89]:

```
# Ошибка! количество передаваемых параметров
# должно равняться количеству объявленных в функции параметров:
multi_summa(*t3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-89-22606a357499> in <module>
      1 # Ошибка! количество передаваемых параметров
      2 # должно равняться количеству объявленных в функции параметров:
----> 3 multi_summa(*t3)

TypeError: multi_summa() takes from 0 to 3 positional arguments but 4 were gi
ven
```

In [90]:

```
# решение проблемы:
multi_summa(*t3[:3])
```

Out[90]:

```
1605
```

Если значения параметров содержатся в словаре, то распаковать словарь можно, указав перед ним две звездочки ** :

In [91]:

```
d1 = {'x': 11, 'y': 12, 'z': 13}
```

In [92]:

```
multi_summa(x=d1['x'], y=d1['y'], z=d1['z'])
```

Out[92]:

```
1431
```

In [93]:

```
multi_summa(**d1)
```

Out[93]:

```
1431
```

In [74]:

```
t3 = [0, -1]
d2 = {'z': -2}
# сначала позиционные параметры, потом пары имя - значение:
multi_summa(*t3, **d2)
```

Out[74]:

```
-210
```

Переменное число параметров в функции

In [94]:

```
print(1, 3, 5, 'val')
```

```
1 3 5 val
```

In [95]:

```
# упаковка последовательности параметров в параметр-кортеж:
def all_summa(*t):
    """Функция принимает произвольное количество параметров"""
#     print(type(t)) # проверка типа параметра
    res = 0
    for i in t:
        res += i
    return  res
```

In [96]:

```
all_summa(10, 20, 30, 40, 50)
```

Out[96]:

```
150
```

In [97]:

```
t4 = [10, 20, 30, 40, 50, 60, 70]
# одновременная распаковка (t4) и упаковка (в параметр функции t):
all_summa(*t4)
```

Out[97]:

```
280
```

In [98]:

```
all_summa(t4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-98-7394efbb6a08> in <module>
----> 1 all_summa(t4)

<ipython-input-95-7166c714d171> in all_summa(*t)
      5     res = 0
      6     for i in t:
----> 7         res += i
      8     return  res

TypeError: unsupported operand type(s) for +=: 'int' and 'list'
```

Если перед параметром в определении функции указать две звездочки ** , то все именованные параметры будут сохранены в словаре.

In [99]:

```
def d_summa(**d):
    for k, v in d.items():  #  Перебираем словарь с переданными параметрами
        print(f"{k} => {v}", end="; ")
```

In [100]:

```
d_summa(a=1, x=10, z=-2)
```

```
a => 1; x => 10; z => -2;
```

In [59]:

```
d3 = {'f': 3, 'g': 4, 'e': 5}
d_summa(**d3)
```

```
f => 3; g => 4; e => 5;
```

При комбинировании параметров параметр с двумя звездочками указывается самым последним.

In [101]:

```
def c_summa(*t, **d):
    for v in t:
        print(v, end="; ")
    for k, v in d.items():
        print(f"{k} => {v}", end="; ")
```

In [102]:

```
c_summa(10, 20, 30, 42, a=1, b=2)
```

```
10; 20; 30; 42; a => 1; b => 2;
```

In [62]:

```
c_summa(*t4, **d3)
```

```
10; 20; 30; 40; 50; 60; 70; f => 3; g => 4; e => 5;
```

>

Аннотации и документирование функций

- к оглавлению

In [103]:

```
def echo(m):
    """  Пример функции с параметром """
    print(m)
```

In [105]:

```
?echo
```

In [104]:

```
help(echo)
```

```
Help on function echo in module __main__:

echo(m)
    Пример функции с параметром
```

In [107]:

```
echo.__doc__
```

Out[107]:

```
'  Пример функции с параметром '
```

In [108]:

```
summa.__doc__
```

Среди разработчиков на Python популярно несколько форматов документирования функции в docstring:

reST

Возможно наиболее распространенный стиль. Формат используется инструментом **Sphinx** (https://www.sphinx-doc.org/en/master/ (https://www.sphinx-doc.org/en/master/)) для генерации документации. Этот формат по умолчанию используется IDE PyCharm.

Пример:

```
"""
This is a reST style.

:param param1: this is a first param
:param param2: this is a second param
:returns: this is a description of what is returned
:raises keyError: raises an exception
"""
```

Google

Компания Google поддерживает собственный формат комментариев функций. Он тоже может использоваться инструментом Sphinx (необходимо использование плагина Napoleon).

Пример:

```
"""
This is an example of Google style.

Args:
    param1: This is the first param.
    param2: This is a second param.

Returns:
    This is a description of what is returned.

Raises:
    KeyError: Raises an exception.
"""
```

Numpydoc

Библиотека NumPy рекомендует использовать собственный стиль описания функций (базируется на стиле Google и может использоватьс в Sphinx).

Пример:

```
"""
My numpydoc description of a kind
of very exhautive numpydoc format docstring.

Parameters
----------
first : array_like
    the 1st param name `first`
second :
    the 2nd param
third : {'value', 'other'}, optional
    the 3rd param, by default 'value'

Returns
-------
string
    a value in a string

Raises
------
KeyError
    when a key error
OtherError
    when an other error
"""
```

Epytext

Стиль поддерживаемый инструментом для генерации документации Epydoc (http://epydoc.sourceforge.net (http://epydoc.sourceforge.net)) называется форматом Epytext format.

Пример:

```
"""
This is a javadoc style.

@param param1: this is a first param
@param param2: this is a second param
@return: this is a description of what is returned
@raise keyError: raises an exception
"""
```

Аннотации типов (type hints) просто считываются интерпретатором Python и никак более не обрабатываются, но доступны для использования из стороннего кода и упрощают работу:

- статических анализаторов;
- интегрированных сред разработки (IDE);
- по документированию кода.

Применение аннотаций позволяет:

- быстрее обнаруживать ошибки
- повышает эффективность использования IDE

- упрощает разработку в больших проектах

Подробнее см.:

- https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html (https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html)
- https://docs.python.org/3/library/typing.html (https://docs.python.org/3/library/typing.html)

In [109]:

```
from typing import Callable, Iterator, Iterable, Mapping, Union, Optional, Tuple, List, Dic
from typing import get_type_hints
import __main__
```

Простые примеры аннотации функций:

In [110]:

```
# аннотация для аргумента и возвращаемого значения:
def stringify(num: int) -> str:
    return str(num)

# аннотация для функции многих аргументов:
def plus(num1: int, num2: int) -> int:
    return num1 + num2

# аннотация для функций с несколькими параметрами:
def f(num1: int, my_float: float = 3.5) -> float:
    return num1 + my_float
```

In [111]:

```
# получение аннотации функции:
stringify.__annotations__
```

Out[111]:

```
{'num': int, 'return': str}
```

Аннотации можно использовать не только для параметров функций, но и для переменных:

In [113]:

```
li1: List[int] = [1, 2]
```

In [114]:

```
di1: Dict[str, float] = {'length': 10.0, 'width': 100.0}
```

In [115]:

```
tu1: Tuple[str, float, float] = ("rect", 10.0, 100.0)
```

In [116]:

```
# аннтация переменной, которая может указывать на кортежа разного размера
tu1: Tuple[int, ...] = (1, 2, 3)
```

In [117]:

```
# можно даже проаннотировать не инициализированную переменную:
wo_init: str
```

In [118]:

```
# получение аннотаций переменных текущего модуля:
get_type_hints(__main__)
```

Out[118]:

```
{'li1': typing.List[int],
 'di1': typing.Dict[str, float],
 'tu1': typing.Tuple[int, ...],
 'wo_init': str}
```

Более сложные варианты:

In [144]:

```
# Union используется если что-то может относиться к нескольким типам:
x: List[Union[int, str]] = [3, 5, "test", "fun"]
```

In [119]:

```
# Optional[X] эквивалентно: Union[X, None]
st1: Optional[str] = None
```

In [120]:

```
st1 = 'My string'
```

In [121]:

```
def mystery_function():
    return None
```

In [122]:

```
# Аннотация Any для случая, когда тип неизвестен или слишком "динамичен":
x: Any = mystery_function()
```

Аннотация функций работающих с контейнерами:

In [123]:

```
def scale(scalar: float, vector: List[float]) -> List[float]:
    return [scalar * num for num in vector]

new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

In [124]:

```
scale.__annotations__
```

Out[124]:

```
{'scalar': float, 'vector': typing.List[float], 'return': typing.List[float]}
```

Определение собственных типов для аннотации:

In [125]:

```
Vector = List[float]

def scale2(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

new_vector = scale2(2.0, [1.0, -4.2, 5.4])
```

In [128]:

```
scale2.__annotations__
```

Out[128]:

```
{'scalar': float, 'vector': typing.List[float], 'return': typing.List[float]}
```

In [129]:

```
Более сложные типы аннотаций:
```

```
  File "<ipython-input-129-519fdc4bb7cd>", line 1
    Более сложные типы аннотаций:
                 ^
SyntaxError: invalid syntax
```

In [130]:

```
# Анотация Iterable используется для любых представителей
# iterables (что-то что можно обойти с помощью "for"),
# и Sequence (поддерживают "len" и "__getitem__")
def f(ints: Iterable[int]) -> List[str]:
    return [str(x) for x in ints]

f(range(1, 3))
```

Out[130]:

```
['1', '2']
```

In [131]:

```
# Mapping описывает объекты с интерфейсом словаря (имеющие "__getitem__") (для не поддержив
# и для поддерживающих изменения: MutableMapping (имеющие и "__getitem__" и "__setitem__")
def f(my_mapping: Mapping[int, str]) -> List[int]:
#     my_mapping[5] = 'maybe'  # if we try this, mypy will throw an error...
    return list(my_mapping.keys())
```

>

Глобальные и локальные переменные

- к оглавлению

Передача параметров функций

Объекты в функцию передаются по ссылке. Если объект относится к неизменяемому типу, то изменение значения внутри функции не затронет значение переменной вне функции.

In [132]:

```
def change(a, b):
    a = 20
    b = 'str'
```

In [133]:

```
v1 = 30
v2 = 'val'
change(v1, v2)
print(f'v1: {v1}')
print(f'v2: {v2}')
```

```
v1: 30
v2: val
```

In [134]:

```
def change2(a):
    a.append(10)
```

In [135]:

```
s1 = [1]
# изменяемый объект может быть неявно изменен внутри функции:
change2(s1)
s1
```

Out[135]:

```
[1, 10]
```

Глобальные и локальные переменные

Глобальные переменные - это переменные, объявленные в программе вне функции. В Python глобальные персменные видны в любой части модуля, включая функции.

In [136]:

```
def func_glob(glob2):
    print("----Вход в функцию func_glob")
    print("    Значение  глобальной переменной glob = ",  glob)
    print("    Значение  ЛОКАЛЬНОЙ переменной glob2 = ",  glob2)
    glob2 += 10
    print("    Значение  ЛОКАЛЬНОЙ переменной glob2 = ", glob2)
    print("----Выход из функции func_glob")

glob, glob2 = 10, 5
print("Значение глобальной переменной glob = ",  glob)
print("Значение глобальной переменной glob2 = ",  glob2)
func_glob(77)  #  Вызываем функцию
print("Значение глобальной переменной glob = ",  glob)
print("Значение глобальной переменной glob2 = ",  glob2)
```

```
Значение глобальной переменной glob =  10
Значение глобальной переменной glob2 =  5
----Вход в функцию func_glob
    Значение  глобальной переменной glob =  10
    Значение  ЛОКАЛЬНОЙ переменной glob2 =  77
    Значение  ЛОКАЛЬНОЙ переменной glob2 =  87
----Выход из функции func_glob
Значение глобальной переменной glob =  10
Значение глобальной переменной glob2 =  5
```

Переменной `glob2` внутри функции присваивается значение, переданное при вызове функции. По этой причине создается новое имя `glob2`, которое является **локальным**. Все изменения этой переменной внутри функции не затронут значение одноименной глобальной переменной.

Локальные переменные - это переменные, которым внутри функции присваивается значение.

- Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной.
- Значение глобальной переменной "затененной" локальной переменной не изменяется.
- Локальные персменные видны только внутри тела функции.

In [137]:

```
def func():
    loc = 77 # Локальная  переменная
    glob = 25 # Локальная  переменная
    print('Значение glob внутри функции: ', glob)

glob = 10 #  Глобальная  переменная
func() # Вызываем функцию

print("Значение glob вне  функции: ", glob)
try:
    print(loc)  #  Вызовет исключение NameError
except NameError:  #  Обрабатываем исключение
    print("Переменная loc не видна вне функции")
```

```
Значение glob внутри функции:  25
Значение glob вне  функции:  10
Переменная loc не видна вне функции
```

Как видно из примера, переменная `loc` , объявленная внутри функции `func()` , недоступна вне функции. Объявление внутри функции локальной переменной `glob` не изменило значения одноименной глобальной переменной. Если обращение к переменной производится до присваивания значения (даже если существует одноименная глобальная переменная), то будет возбуждено исключение `UnboundLocalError` .

Для того **чтобы значение глобальной переменной можно было изменить внутри функции**, необходимо объявить переменную глобальной с помощью ключевого слова `global` .

In [138]:

```
def func():
    global glob
    loc = 77 #  Локальная  переменная
    glob = 25 #  Локальная  переменная
    print('Значение glob внутри функции: ', glob)

glob = 10 #  Глобальная  переменная
func() # Вызываем функцию
print("Значение glob вне  функции: ", glob)
```

```
Значение glob внутри функции:  25
Значение glob вне  функции:  25
```

Поиск идентификатора, используемого внутри функции, производится в следующем порядке:

1. Поиск объявления идентификатора внутри функции (в локальной области видимости).
2. Поиск объявления идентификатора в глобальной области.
3. Поиск во встроенной области видимости (встроенне функции, классы и т. д.).

Получить все идентификаторы и их значения позволяют следующие функции:

- `globals()` - возвращает словарь с глобальными идентификаторами;
- `locals()` - возвращает словарь с локальными идентификаторами;

- `vars( [<Объект>] )` - если вызывается без параметра внутри функции, то возвращает словарь с локальными идентификаторами. Если вызывается без параметра вне функции, то возвращает словарь с глобальными идентификаторами. При указании объекта в качестве параметра возвращает идентификаторы этого объекта (эквивалентно вызову `<Объект>.__dict__`)

In [139]:

```
globals()
```

Out[139]:

```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environm
ent',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['',
  'def func(): \n    pass ',
  'func()',
  'def  func(): \n    print ("Текст до инструкции return") \n    return
"Возвращаемое  значение" \n    print ("Эта инструкция никогда не будет вып
олнена") ',
  'func()',
  "r = func()\nprint(f'Результат {r}')",
  "r = func()\nprint(f'Результат: {r}')",
  'def print_ok(): \n    """  Пример  функции  без  \n            параметр
```

In [140]:

```
locals()
```

Out[140]:

```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environm
ent',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['',
  'def func(): \n    pass ',
  'func()',
  'def  func(): \n    print ("Текст до инструкции return") \n    return
"Возвращаемое  значение" \n    print ("Эта инструкция никогда не будет вып
олнена") ',
  'func()',
  "r = func()\nprint(f'Результат {r}')",
  "r = func()\nprint(f'Результат: {r}')",
  'def print_ok(): \n    """  Пример  функции  без  \n            параметр
```

In [141]:

```
def func():
    loc = 54
    glob2 = 25
    print("Локальные идентификаторы внутри  функции", sorted(vars().keys()))
```

In [142]:

```
func()
```

```
Локальные идентификаторы внутри  функции ['glob2', 'loc']
```

Вложенные функции

Одну функцию можно вложить в другую функцию, причем уровень вложенности неограничен. В этом случае вложенная функция получает свою собственную локальную область видимости и имеет доступ к идентификаторам внутри функции-родителя.

In [247]:

```
def f1(x):
    def f2():
        print(x)
    return f2
```

In [256]:

```
fa = f1(12)
fa()
```

```
12
```

In [257]:

```
fb = f1(22)
fb()
```

```
22
```

>

Задание к следующей лекции

По книге Н. Прохоренок:

повторить Глава 11 Пользовательские функции Глава 14 Обработка исключений

По книге М. Саммерфильд: повторить Глава 4 Управляющие структуры и функции (разедел "Собственные функции") Глава 4 Управляющие структуры и функции (разедел "Обработка исключений")

Задание к следующей лекции

По книге Н. Прохоренок:

повторить Глава 11 Пользовательские функции Глава 14 Обработка исключений

По книге М. Саммерфильд: повторить Глава 4 Управляющие структуры и функции (разедел "Собственные функции") Глава 4 Управляющие структуры и функции (разедел "Обработка исключений")