

Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 4. Функция. Область видимости переменной

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

март, 2025

4.1 Определение функции, вызов функции, объект функции

Функция является способом группирования операторов и выражений *для многократного выполнения сгруппированного кода* за счет вызовов функции в разных местах программы.

Для функции в коде возможны два действия: *определение функции* и *вызов функции*. Вызвать функцию можно только после ее определения.

Определение функции, операторы `def` и `return`

Определение функции осуществляется с помощью оператора `def` или с помощью выражения, содержащего зарезервированное имя `lambda`.

Оператор `def` является составным. Общий формат определения функции с помощью оператора `def`:

```
def <имя функции>(<имя_аргумента1>, <имя_аргумента2>, ..., <имя_аргументаn>):  
    <тело функции из операторов и выражений>  
def <имя функции>(<имя_аргумента1>, <имя_аргумента2>, ..., <имя_аргументаn>): будем называть заголовком функции.
```

Важно отметить, что при определении функции код тела функции не выполняется, а выполняется только при вызове функции. При каждом вызове функции тело функции выполняется заново. Определение функции осуществляется один раз, вызов функции -- много раз.

Результатом вызова функции является `<объект-значение>`, которое посылается посредством оператора `return` вызывающему коду в виде `return <объект-`

значение> , или результатом вызова функции является объект `None` , если оператор `return` отсутствует в теле функции

```
In [1]: def f(x):  
        return x  
  
        def g(x):  
            x = 10  
  
        f(1), g(1)
```

Out[1]: (1, None)

При необходимости возвращения из функции нескольких объектов-значений их можно возвращать как элементы кортежа в виде `return obj1, obj2, obj3 ...`

```
In [2]: def f(x):  
        return "Return value:", x  
  
        f(1)
```

Out[2]: ('Return value:', 1)

Выход из тела функции осуществляется сразу после выполнения оператора `return` или при достижении конца тела функции, если оператор `return` отсутствует в теле функции

```
In [3]: def f(x):  
        return x  
        print(f'{x = }')  
  
        def g(x, y):  
            print(f'{x = }, {y = }')  
  
        f(x=1), g(y=2, x=1)
```

x = 1, y = 2

Out[3]: (1, None)

Вызов функции

Вызов функции осуществляется с помощью выражения вида `<имя функции> (аргумент1, аргумент2, ..., аргументn)` , где `аргумент1, аргумент2, ..., аргументn` являются передаваемыми в функцию объектами-значениями, которые используются для выполнения кода, записанного в теле функции.

При каждом вызове функции выполняется тело функции.

Результатом вызова функции является `<объект-значение>` .

При вызове функции все аргументы можно разделить на два типа: **позиционные аргументы** и **ключевые аргументы** (именованные, named, keyword).

Если при вызове функции значение аргумента задается только объектом, то он называется **позиционный аргумент**. Позиционные аргументы должны передаваться в том же порядке, в котором указаны при определении функции.

Ключевые аргументы при вызове указываются в виде пары `<ключ=объект>`, где *ключом является имя аргумента, указанное при определении функции*. Это позволяет при вызове функции указывать ключевые аргументы в любом порядке.

При вызове функции аргументы должны указываться в следующем порядке: сначала позиционные аргументы, затем ключевые аргументы:

```
In [4]: print(1, 2, 3, sep=' between ')
```

```
1 between 2 between 3
```

Примеры различных вызовов функции:

```
In [5]: def f(arg1, arg2, arg3):
        print(arg1, arg2, arg3)

f(1, 2, 3) # все аргументы позиционные
f(arg1=1, arg2=2, arg3=3) # все аргументы ключевые
f(arg3=3, arg2=2, arg1=1) # порядок ключевых аргументов не важен
f(1, 2, arg3=3) # сначала позиционные аргументы, затем ключевой
```

```
1 2 3
1 2 3
1 2 3
1 2 3
```

Вызов функции с ключевыми аргументами является более информативным, чем вызов с позиционными аргументами.

Следующие вызовы функции будут некорректными:

```
def f(arg1, arg2):
    print(arg1, arg2)

f()
f(arg1=2, {1})
f('second', arg1='first')
f(3, arg3=4)

def f(arg1, arg2):
    print(arg1, arg2)

f() # должно быть два аргумента
f(arg1=2, {1}) # позиционный аргумент должен указываться до
               # ключевого арг.
f('second', arg1='first') # одному позиционному аргументу соответствуют
                           # два значения
f(3, arg3=4) # неизвестный ключевой аргумент
```

Передача значений аргументов при вызове функции

При вызове функции значения аргументов передаются путем *автоматического неявного присваивания* аргументам-переменным, указанным в заголовке функции при ее определении, соответствующих объектов-значений.

```
In [6]: def f(arg1, arg2):  
        ...  
  
        f(1, 2) # неявные присваивания arg1 = 1 и arg2 = 2
```

При этом в период выполнения функции аргументы-переменные будут ассоциироваться с пространством имен функции и являться **локальными переменными**. По завершению выполнения функции аргументы-переменные удаляются из памяти, а объекты-значения, на которые они ссылались, могут быть подвергнуты автоматической сборке мусора, если на них нету других ссылок.

Важно : При вызове функции копирования объектов, указанных в качестве значений аргументов, не происходит. Другими словами, значения аргументов функций в Python *передаются по ссылке, а не путем создания новых объектов*.

Это приводит к тому, что *изменяемые объекты* (списки, словари, множества), указанные в качестве значений аргументов при вызове функции, могут быть изменены внутри функции и, как следствие, оказать влияние на вызывающий код

```
In [7]: def f(arg1):  
        arg1.pop()  
  
        list1 = [1,2,3]; set1 = set(list1)  
        f(list1); f(set1)  
        print(list1, set1)
```

```
[1, 2] {2, 3}
```

Для того, чтобы не оказывать влияния на изменяемые объекты, передаваемые при вызове функции, необходимо *внутри функции создавать копию передаваемого объекта*:

```
In [8]: def f(arg1):  
        arg1 = arg1[:] # создание копии списка  
        arg1.pop()  
  
        list1 = [1,2,3]  
        f(list1)      # вызов функции со списком в качестве аргумента  
        print(list1)
```

```
[1, 2, 3]
```

Режимы сопоставления объектов-значений и аргументов-переменных при вызове функции

При вызове функции возникает **задача сопоставления** передаваемых объектов-значений и аргументов функции, указанных в заголовке оператора `def`.

Позиционный режим сопоставления при *вызове функции* только с позиционными аргументами -- *автоматическое неявное присваивание* аргументам-переменным

объектов-значений по позиции слева направо

```
In [9]: def f(arg1, arg2, arg3, arg4, arg5):  
        ...  
  
        f(1, 2, 3, 4, 5) # arg1 = 1, arg2 = 2, arg3 = 3, arg4 = 4, arg5 = 5
```

Позиционный режим сопоставления **с распаковкой последовательностей**: когда при *вызове функции* только с позиционными аргументами указывается символ ***** перед последовательностью с объектами-значениями, то происходит распаковка последовательности на элементы и *автоматическое неявное присваивание* объектов-значений из последовательности аргументам-переменным по позиции слева направо

```
In [10]: def f(arg1, arg2, arg3, arg4, arg5):  
          print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')  
  
          f(*range(1,6)) # arg1 = 1, arg2 = 2, arg3 = 3, arg4 = 4, arg5 = 5  
          # совмещение позиционных режимов с распаковкой и без распаковки  
          f(2, *"st", 5, 6)
```

```
arg1=1, arg2=2, arg3=3, arg4=4, arg5=5  
arg1=2, arg2='s', arg3='t', arg4=5, arg5=6
```

Ключевой режим сопоставления, когда при *вызове функции* явно указывается связь между аргументом и передаваемым значением **аргумент = значение**

```
In [11]: def f(arg1, arg2, arg3, arg4, arg5):  
          print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')  
  
          f(arg5=1, arg3=2, arg1=3, arg2=4, arg4=5)
```

```
arg1=3, arg2=4, arg3=2, arg4=5, arg5=1
```

Ключевой режим сопоставления **с распаковкой словарей**: когда при *вызове функции* указывается ****** перед словарем с элементами **аргумент: значение**, то происходит *автоматическое неявное присваивание* аргументам, указанным ключами словаря, соответствующих объектов-значений из словаря

```
In [12]: def f(arg1, arg2, arg3, arg4, arg5):  
          print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')  
  
          myDict = {'arg2':2, 'arg3':3, 'arg5':5, 'arg4':4}  
  
          f(**myDict, arg1=100)
```

```
arg1=100, arg2=2, arg3=3, arg4=4, arg5=5
```

Возможно совмещение ключевого и позиционного режимов сопоставления объектов-значений и аргументов-переменных при вызове функции.

```
In [13]: def f(arg1, arg2, arg3, arg4, arg5):  
          print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')  
  
          f(1, 2, 3, arg5=5, arg4=4)
```

arg1=1, arg2=2, arg3=3, arg4=4, arg5=5

Определение функции со значениями по умолчанию для аргументов

При определении функции в заголовке оператора **def**, помимо имени аргумента, можно указать его значение, например, `def <имя функции>(<имя аргумента>=<значение>):` .

Значение аргумента функции, заданное при ее определении, называют *значением по умолчанию*, или **стандартным значением** аргумента. Аргумент, имеющий значение по умолчанию, является *необязательным* при вызове функции: если он отсутствует, функция будет выполнена со стандартным значением аргумента.

```
In [14]: def f(arg1, arg2=2, arg3=3):  
         return arg1, arg2, arg3  
  
f(100), f(100, 0)
```

```
Out[14]: ((100, 2, 3), (100, 0, 3))
```

Использование символа равенства `=` при задании стандартного значения во время определения функции и при задании ключевого аргумента во время вызова функции НЕ является вызовом оператора присваивания.

Значения по умолчанию создаются только один раз при ОПРЕДЕЛЕНИИ функции. Использование изменяемых объектов в качестве значений по умолчанию могут вызвать проблемы при повторных вызовах одной и той же функции. Рекомендация: не используйте изменяемые объекты (списки, словари, множества) в качестве значений по умолчанию

```
In [15]: def f(arg=[]):  
         arg.append(1)  
         return arg  
print(f())  
print(f())
```

```
[1]  
[1, 1]
```

Все аргументы в заголовке функции, находящиеся после первого аргумента со стандартным значением, также должны иметь стандартные значения.

```
In [16]: def f(arg1, arg2=2, arg3=3):  
         return arg1, arg2, arg3  
  
f(100, arg3=300), f(arg1=100)
```

```
Out[16]: ((100, 2, 300), (100, 2, 3))
```

Определение функции с произвольным количеством аргументов

Если при определении функции в заголовке оператора `def` указать символ `*` перед именем аргумента, например, `def <имя функции>(*<имя_аргумента>):`, то *при вызове* функции на месте этого аргумента может быть указано **произвольное количество позиционных аргументов**, которые *при выполнении* функции будут доступны как элементы *кортежа* `<имя_аргумента>`:

```
In [17]: def f(*pargs):
          print(pargs)

f() # аргументы со * в определении являются необязательными при вызове!
f(1,2,3)
f(1,2,3,4,5,6)

()
```

Пример функции с произвольным количеством позиционных аргументов

```
In [18]: def adder(*pargs):
          """ применяет операцию сложения ко всем аргументам и возвращает результат оп
          s = pargs[0]
          for i in pargs[1:]:
              s += i
          return s
```

```
In [19]: adder(1,2,3,4,5,6), adder('word1', 'word2', 'word3'), adder([1],[2],[3],[4])
```

```
Out[19]: (21, 'word1word2word3', [1, 2, 3, 4])
```

Если при определении функции в заголовке оператора `def` указать `**` перед именем аргумента, например, `def <имя функции>(**<имя_аргумента>):`, то *при вызове* функции на месте этого аргумента может быть указано **произвольное количество ключевых аргументов**, которые *при выполнении* функции будут доступны как элементы *словаря* `<имя_аргумента>`:

```
In [20]: def f(x,**kargs):
          print(kargs)

f(10) ## аргументы с ** в определении являются необязательными при вызове!
f(10, arg1=1, arg2=2, arg3=3)
f(10, x1=2, x2=3, x3=4, x5=5, x6=6)

{}
{'arg1': 1, 'arg2': 2, 'arg3': 3}
{'x1': 2, 'x2': 3, 'x3': 4, 'x5': 5, 'x6': 6}
```

Порядок следования аргументов при определении функции

При определении функции аргументы должны указываться в следующем порядке: позиционные аргументы, далее форма `*pargs`, затем аргументы со стандартными значениями, затем аргументы с передачей только по ключу и, наконец, форма `**kargs`:

```
In [21]: def f(a, *pargs, b=50, c, **kargs):
          print(f'{a=}, {pargs=}, {b=}, {c=}, {kargs=}')

          f(1, 2, 3, 4, 5, c=0, x=1, y=2)
```

```
a=1, pargs=(2, 3, 4, 5), b=50, c=0, kargs={'x': 1, 'y': 2}
print(*args, sep=' ', end='\n', file=None, flush=False)
```

В Python 3.X при определении функции аргументы можно указывать также в следующем порядке: позиционные аргументы, затем аргументы со стандартными значениями, далее форма `*`, затем аргументы с передачей только по ключу и, наконец, форма `**kargs` :

```
In [22]: def f(a, b=5, *, c, **kargs):
          print(f'{a=}, {b=}, {c=}, {kargs=}')

          f(1, c=10, x=1, y=2)
```

```
a=1, b=5, c=10, kargs={'x': 1, 'y': 2}
```

Такой синтаксис означает, что при вызове функции все аргументы, следующие за символом `*`, должны быть заданы ключевыми.

```
FuncAnimation(fig, func, frames=None, init_func=None, fargs=None,
save_count=None, *, cache_frame_data=True, **kwargs)
```

Объект функции

При выполнении оператора `def` происходит *автоматическое неявное присваивание* объекта функции имени функции. Имя функции становится ссылкой на объект функции.

```
In [23]: def f(x):
          ...
          return "function f is called"

          f
```

```
Out[23]: <function __main__.f(x)>
```

Объект функции имеет тип `function`

```
In [24]: f.__class__
```

```
Out[24]: function
```

С помощью атрибута `__code__` объекта функции можно получить, в частности, информацию об аргументах функции

```
In [25]: f.__code__, f.__code__.co_argcount, f.__code__.co_varnames
```

```
Out[25]: (<code object f at 0x000001C8C79B7910, file "C:\Users\Olga\AppData\Local\Temp\ipykernel_15144\3391401968.py", line 1>,
1,
('x',))
```

В дальнейшем в коде объекты функций можно присваивать другим переменным, передавать другим функциям в качестве аргумента, использовать в коллекциях и т.д. Обращаться с объектами функций можно так же, как и с объектами встроенных типов.

```
In [26]: g = f;  
         g(f), [1, f, "string"]
```

```
Out[26]: ('function f is called', [1, <function __main__.f(x)>, 'string'])
```

Аннотация функции

Аннотация типов -- это информация о типах данных для аргументов функции и для значений функции, которая указывается при определении функции в строке заголовка оператора `def`. Синтаксис определения функции с использованием аннотаций:

```
def <имя функции>(<имя_аргумента1> [: type1], <имя_аргумента2> [:  
type2], ..., <имя_аргументаN> [: typeN]) [-> typeN+1]:  
    <тело функции из операторов и выражений>
```

```
In [27]: def conc(s1: str, s2: str = 'second') -> str:  
         return ' and '.join([s1,s2])
```

```
In [28]: conc('first')
```

```
Out[28]: 'first and second'
```

Аннотации хранятся в атрибуте `__annotations__` объекта функции в виде словаря. Ключами словаря являются строковые объекты с именами атрибутов, а также строковый объект `'return'`. Значениями словаря являются имена типов, указанные в аннотациях.

```
In [29]: conc.__annotations__
```

```
Out[29]: {'s1': str, 's2': str, 'return': str}
```

Аннотации НЕ имеют никакого эффекта на выполнение функции и являются не обязательными при определении функции.

В качестве аннотаций может быть указано ЛЮБОЕ выражение, которое может быть в дальнейшем использовано

```
In [30]: def compile(source: "something compilable",  
                    filename: "where the compilable thing comes from",  
                    mode: "is this a single statement or a suite?"):  
        ...
```