

# Функции в Python

# Функции в Python

Функция - это блок кода, выполняющий определенные действия:

- у функции есть имя, с помощью которого можно запускать этот блок кода сколько угодно раз
  - запуск кода функции, называется вызовом функции
- функциям можно передавать аргументы
  - соответственно, код функции будет выполняться с учетом указанных аргументов

# Создание функции

# Создание функции

- функции создаются с помощью зарезервированного слова `def`
- за `def` следуют имя функции и круглые скобки
- внутри скобок могут указываться параметры, которые затем передаются функции
- после круглых скобок идет двоеточие и с новой строки, с отступом, идет блок кода, который выполняет функция
- первой строкой, опционально, может быть комментарий, так называемая `docstring`
- в функциях может использоваться оператор `return`
  - он используется для прекращения работы функции и выхода из нее
  - Как правило, оператор `return` возвращает какое-то значение

# Пример функции

Пример создания функции:

```
In [1]: def printstr( s ):
...:     """Documentation string"""
...:     print s
...:
```

Эта функция ожидает строку в качестве аргумента, и выводит ее:

```
In [2]: printstr('Test string')
Test string
```

Но так как функция просто выводит строку на стандартный поток вывода, а не возвращает ее:

```
In [3]: s = printstr('Test string')
Test string
```

Переменная s содержит значение None

```
In [4]: print s
None
```

Первая строка в определении функции - это docstring, строка документации. Это комментарий, который используется как описание функции. Его можно отобразить так:

```
In [5]: printstr.__doc__
Out[5]: 'Documentation string'
```

# Оператор return

Оператор return используется для прекращения работы функции, выхода из нее, и, как правило, возврата какого-то значения.

Выражения, которые идут после return, не выполняются:

```
In [5]: def printstr( s ):
...:     """Documentation string"""
...:     print "String: ", s
...:     return s
...:     print "After return"
...:
```

```
In [6]: printstr('Test string')
String:  Test string
Out[6]: 'Test string'
```

```
In [7]: s = printstr('Test string')
String:  Test string
```

```
In [8]: print s
Test string
```

Обратите внимание, что строка 'After return' не выводится.

# Оператор return

Оператор return может использоваться несколько раз:

```
In [52]: def compare(a,b):  
        .....:     if a > b:  
        .....:         return a  
        .....:     elif b > a:  
        .....:         return b  
        .....:     else:  
        .....:         return "Equal"  
        .....:
```

```
In [53]: compare(10,2)  
Out[53]: 10
```

```
In [54]: compare(2,10)  
Out[54]: 10
```

```
In [55]: compare(2,2)  
Out[55]: 'Equal'
```

# **Пространства имен. Области видимости**



# LEGB

У переменных в Python есть область видимости. В зависимости от места в коде, где переменная была определена, определяется и область видимости, то есть, где переменная будет доступна.

Когда мы использовали имена переменных в программе, Python каждый раз, искал, создавал или изменял эти имена в соответствующем пространстве имен. Пространство имен, которое доступно нам в каждый момент, зависит от области в которой мы находимся.

У Python есть правило, которым он пользуется при поиске переменных. Оно называется LEGB.

Например, если внутри функции, мы обращаемся к имени переменной, Python ищет переменную в таком порядке по областям видимости (до первого совпадения):

- L (local) - в локальной (внутри функции)
- E (enclosing) - в локальной области объемлющих функций (это те функции, внутри которых находится наша функция)
- G (global) - в глобальной (в скрипте)
- B (built-in) - в встроенной (зарезервированные значения Python)

# Локальные и глобальные переменные

- локальные переменные:
  - переменные, которые определены внутри функции
  - эти переменные становятся недоступными после выхода из функции
- глобальные переменные
  - переменные, которые определены вне функции
  - эти переменные 'глобальны' только в пределах модуля
    - например, чтобы они были доступны в другом модуле, их надо импортировать

```
In [7]: s = 'string'
```

```
In [8]: def printstr( s ):
...:     s = s + ' test'
...:     return s
...:
```

```
In [9]: printstr('function')
Out[9]: 'function test'
```

```
In [10]: s
Out[10]: 'string'
```

# Аргументы функций

# Аргументы функций

Цель создания функции, как правило, заключается в том, чтобы вынести кусок кода, который выполняет определенную задачу, в отдельный объект. Это позволяет нам использовать этот кусок кода многократно, не создавая его заново в программе.

Как правило, функция должна выполнять какие-то действия с входящими значениями и на выходе выдавать результат.

Для того чтобы функция могла принимать входящие значения, мы должны ее создать с переменными:

```
In [11]: def sum_ab(a,b):  
...:     return a + b  
...:
```

Тогда мы сможем при использовании функции, передавать ей аргументы:

```
In [12]: sum_ab(10,20)  
Out[12]: 30
```

```
In [13]: sum_ab('test','string')  
Out[13]: 'teststring'
```

# Аргументы функций

В данном случае мы создали функцию `sum_ab`, которая ожидает на вход два аргумента `a` и `b`. И на выходе, возвращает сумму этих аргументов.

```
In [11]: def sum_ab(a,b):  
...:     return a + b  
...:
```

При таком определении функции, мы обязаны передать оба аргумента. Если мы передадим только один аргумент, возникнет ошибка (как и в случае, если мы передадим 3 и больше аргументов):

```
In [14]: sum_ab(50)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-8-cc30973e46d7> in <module>()  
----> 1 sum_ab(50)
```

```
TypeError: sum_ab() takes exactly 2 arguments (1 given)
```

# Типы аргументов

# Типы аргументов

Функции можно **вызывать** используя такие типы аргументов:

- обязательные аргументы (позиционные)
- ключевые аргументы
- аргументы по умолчанию
- аргументы переменной длины

# Обязательные аргументы

Обязательные аргументы:

- надо передать ровно столько, сколько указано параметров функции (нельзя пропустить а или b, или указать больше)
- надо передать в правильном порядке

Функция с обязательными аргументами:

```
In [15]: def divide(a,b):  
...:     return a/b  
...:
```

```
In [16]: divide(10,2)  
Out[16]: 5
```



# Ключевые аргументы

Ключевые аргументы:

- передаются с указанием именем аргумента
- засчет этого, они могут передаваться в любом порядке

Передачи функции аргументов, как ключевых:

```
In [17]: def divide(a,b):  
        .....:     return a/b  
        .....:
```

```
In [18]: divide(b=10, a=100)  
Out[18]: 10
```

# Аргументы со значением по умолчанию

При создании функции, можно указывать значение по умолчанию для аргумента:

```
In [19]: def divide(a,b=2):  
        ....:     return a / b  
        ....:
```

В данном случае, у аргумента b есть значение по умолчанию. И, если вызывать функцию, не указывая значение для b, будет использоваться значение 2:

```
In [20]: divide(a=100,b=5)  
Out[20]: 20
```

```
In [21]: divide(a=100)  
Out[21]: 50
```

# Аргументы переменной длины

Иногда заведомо неизвестно сколько аргументов надо передавать функции. Для таких ситуаций мы можем создавать функцию с параметрами переменной длины.

Такой параметр может быть, как ключевым, так и позиционным.

# Позиционные аргументы переменной длины

```
In [22]: def sum_arg(a,*arg):  
        .....:     print a, arg  
        .....:     return a + sum(arg)  
        .....:
```

Функция `sum_arg` создана с двумя параметрами:

- параметр `a`
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен
- параметр `*arg` - ожидает аргументы переменной длины
  - сюда попадут все остальные аргументы в виде кортежа
  - эти аргументы могут отсутствовать
  - имя `arg` не имеет никакого специального значения. Это может быть и любое другое имя. Главное в обозначении, это звездочка (\*)

# Позиционные аргументы переменной длины

```
In [22]: def sum_arg(a,*arg):  
        .....:     print a, arg  
        .....:     return a + sum(arg)  
        .....:
```

Попробуем вызвать функцию:

```
In [23]: sum_arg(1,10,20,30)  
1 (10, 20, 30)  
Out[23]: 61
```

```
In [24]: sum_arg(1,10)  
1 (10, )  
Out[24]: 11
```

```
In [25]: sum_arg(1)  
1 ()  
Out[25]: 1
```

# Позиционные аргументы переменной длины

```
In [26]: def sum_arg(*arg):  
        .....:     print arg  
        .....:     return sum(arg)  
        .....:
```

```
In [27]: sum_arg(1,10,20,30)  
(1, 10, 20, 30)  
Out[27]: 61
```

```
In [28]: sum_arg()  
( )  
Out[28]: 0
```

# Ключевые аргументы переменной длины

Аналогичным образом можно создать параметр переменной длины для ключевых аргументов:

```
In [29]: def sum_arg(a, **karg):  
.....:     print a, karg  
.....:     return a + sum(karg.values())  
.....:
```

Функция `sum_arg` создана с двумя параметрами:

- параметр `a`
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен
- параметр `**karg` - ожидает ключевые аргументы переменной длины
  - сюда попадут все остальные ключевые аргументы в виде словаря
  - эти аргументы могут отсутствовать
  - имя `karg` не имеет никакого специального значения. Это может быть и любое другое имя. Главное в обозначении, это две звездочки (`**`)
  - Хотя, как правило именно имена `*arg` и `**karg` и используются

# Ключевые аргументы переменной длины

```
In [29]: def sum_arg(a,**karg):  
.....:     print a, karg  
.....:     return a + sum(karg.values())  
.....:
```

Попробуем вызвать функцию:

```
In [30]: sum_arg(a=10,b=10,c=20,d=30)  
10 {'c': 20, 'b': 10, 'd': 30}  
Out[30]: 70
```

```
In [31]: sum_arg(b=10,c=20,d=30,a=10)  
10 {'c': 20, 'b': 10, 'd': 30}  
Out[31]: 70
```



# Анонимная функция lambda

# Анонимная функция lambda

Конструкция lambda называется анонимной функцией, из-за того, что она создается без определения функции стандартным образом через def.

В конструкции lambda:

- может содержаться только одно выражение
- аргументов может передаваться сколько угодно

Пример анонимной функции:

```
In [32]: sum_arg = lambda a, b: a + b
```

```
In [33]: sum_arg(1, 2)  
Out[33]: 3
```

```
In [34]: sum_arg(10, 22)  
Out[34]: 32
```

# **Полезные встроенные функции**

**Функция zip()**

# Функция zip

Функция zip():

- на вход функции передаются последовательности
- zip() возвращает список кортежей, каждый из которых состоит из элементов
  - например, десятый кортеж будет содержать десятый элемент каждой из переданных последовательностей
- если на вход были переданы последовательности разной длины, то все они будут отрезаны по самой короткой последовательности
- последовательность элементов соблюдается

# Функция zip

Пример использования zip():

```
In [35]: a = [1,2,3,4,5]
```

```
In [36]: b = [10,20,30,40,50]
```

```
In [37]: zip(a,b)
```

```
Out[37]: [(1, 10), (2, 20), (3, 30), (4, 40), (5, 50)]
```

```
In [38]: zip(b,a)
```

```
Out[38]: [(10, 1), (20, 2), (30, 3), (40, 4), (50, 5)]
```

Использование zip() со списками разной длины:

```
In [39]: c = [100,200,300]
```

```
In [40]: zip(a,b,c)
```

```
Out[40]: [(1, 10, 100), (2, 20, 200), (3, 30, 300)]
```

# Пример использования функции zip

```
In [10]: d_keys = ['hostname', 'vendor', 'model', 'IOS', 'IP']
```

```
In [11]: data = {  
.....: 'r1': ['london_r1', 'Cisco', '4451', '15.4', '10.255.0.1'],  
.....: 'r2': ['london_r2', 'Cisco', '4451', '15.4', '10.255.0.2'],  
.....: 'sw1': ['london_sw1', 'Cisco', '3850', '3.6.XE', '10.255.0.101']  
.....: }
```

```
In [12]: london_co = {}
```

```
In [13]: for k in data.keys():  
.....:     london_co[k] = dict(zip(d_keys, data[k]))  
.....:
```

```
In [14]: london_co
```

```
Out[14]:  
{  
  'r1': {'IOS': '15.4',  
         'IP': '10.255.0.1',  
         'hostname': 'london_r1',  
         'model': '4451',  
         'vendor': 'Cisco'},  
  'r2': {'IOS': '15.4',  
         'IP': '10.255.0.2',  
         'hostname': 'london_r2',  
         'model': '4451',  
         'vendor': 'Cisco'},  
  ...  
}}
```

**Функция map()**



# Функция map

Функция map():

- Функция map() применяет указанную функцию к каждому элементу последовательности и возвращает список результатов.

Пример использования функции map():

```
In [41]: a = ['aaa', 'bbb', 'ccc']
```

```
In [42]: def to_upper(s):  
.....:     return s.upper()  
.....:
```

```
In [43]: map(to_upper, a)  
Out[43]: ['AAA', 'BBB', 'CCC']
```

# Функция map и lambda

Вместе с map() очень удобно использовать lambda:

```
In [41]: a = ['aaa', 'bbb', 'ccc']
```

```
In [44]: map(lambda s: s.upper(), a)
```

```
Out[44]: ['AAA', 'BBB', 'CCC']
```

Если функция, которую использует map(), ожидает два аргумента, то передаются два списка:

```
In [45]: a = ['a', 'b', 'c', 'd']
```

```
In [46]: b = [1, 2, 3, 4]
```

```
In [47]: map(lambda x, y: x*y, a, b)
```

```
Out[47]: ['a', 'bb', 'ccc', 'dddd']
```

# Функция map

Вместе с map() можно использовать стандартные функции

```
In [48]: l = ['1', '2', '3', '4', '5']
```

```
In [49]: map(int, l)  
Out[49]: [1, 2, 3, 4, 5]
```

```
In [50]: s = ['aaa', 'bbbbbb', 'ccccccccc', 'ddd']
```

```
In [51]: map(len, s)  
Out[51]: [3, 5, 9, 3]
```

**Функция filter()**

# Функция filter

Функция filter():

- Функция filter() применяет функцию ко всем объектам списка, и возвращает те объекты, для которых функция вернула True.

Например, из списка чисел оставляем только нечетные:

```
In [48]: filter(lambda x: x%2,[10,111,102,213,314,515])
```

```
Out[48]: [111, 213, 515]
```

А теперь только четные:

```
In [49]: filter(lambda x: not x%2,[10,111,102,213,314,515])
```

```
Out[49]: [10, 102, 314]
```

# **Python для сетевых инженеров**

**Автор курса: Наташа Самойленко**  
**[nataliya.samoylenko@gmail.com](mailto:nataliya.samoylenko@gmail.com)**