

# Неделя 2. Functions. Modules.

## Цели лекции

- научиться писать функции, освоить базовый инструментарий функционального программирования в Python;
- научиться использовать генераторы списков, анонимные функции;
- научиться создавать и структурировать модули;

## План лекции

<b>План лекции</b> .....	1
<b>Functions</b> .....	2
ОПРЕДЕЛЕНИЕ ФУНКЦИИ И НЕЙМИНГ .....	2
ФУНКЦИЯ ТОЖЕ ОБЪЕКТ; ДОКУМЕНТИРОВАНИЕ .....	2
ДОКУМЕНТИРОВАНИЕ ФУНКЦИИ .....	3
ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИЮ .....	3
РАСПАКОВКИ ПРИ ПЕРЕДАЧЕ ПАРАМЕТРОВ .....	5
ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ РЕЗУЛЬТАТ .....	6
АННОТАЦИЯ ФУНКЦИИ .....	7
ПАРАМЕТРЫ ФУНКЦИИ ИЗМЕНЯЕМОГО ТИПА .....	7
ПАРАМЕТРЫ ФУНКЦИИ KEYWORD-ONLY .....	8
ОБЛАСТИ ВИДИМОСТИ ПЕРЕМЕННЫХ (SCOPE) .....	8
АНОНИМНЫЕ ФУНКЦИИ: LAMBDA .....	9
<b>Modules</b> .....	11
КРАТКОЕ ОПИСАНИЕ .....	11
КАК ИЗБЕЖАТЬ РАСПРОСТРАНЕННУЮ ОШИБКУ? .....	11
РЕШЕНИЕ КВАДРАТНОГО УРАВНЕНИЯ .....	11

### ОПРЕДЕЛЕНИЕ ФУНКЦИИ И НЕЙМИНГ

До сих пор мы использовали встроенные функции Питона и функции из модулей, которые входят в комплект его поставки. Но мощь структурных языков программирования заключается в том, что мы можем создавать свои собственные функции, причем делается это довольно просто. В структурном программировании функция представляет собой именованную (либо неименованную — см. об анонимных функциях далее) последовательность выражений, выполняющих необходимую операцию. В Питоне существует специальный оператор определения функции, имеющий следующий синтаксис:

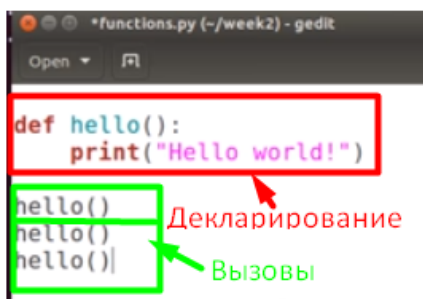
```
def имя_функции(параметры):  
    тело функции
```

Правила выбора имен функций:

- можно использовать любые латинские буквы, цифры и знак `_` (знак подчеркивания). Знак подчеркивания может использоваться для разделения слов составляющих имя переменной: например, `user_name` или `full_price`;
- не может начинаться с цифры;
- не должно совпадать с ключевыми словами (`and`, `class`, `print`, `return` ...)

Первая строка определения обычно называется заголовком функции, обозначенным ключевым словом `def` (от англ. «define» – «определить»). Заголовок функции в Питоне завершается двоеточием. После него может следовать любое количество выражений, но они должны быть записаны со смещением (четыре пробела) относительно начала строки.

Следует отличать объявление (определение, `declaring`) функции от ее **вызова** (`call`); описывая последовательность выражений, мы декларируем функцию; вызывая декларированную ранее функцию, интерпретатор python выполняет код функции, и только тогда мы получаем возвращаемый результат:



```
cadet@pb:~/week2$ python functions.py  
Hello world!  
Hello world!  
Hello world!
```

Результаты вызовов функции

### ФУНКЦИЯ ТОЖЕ ОБЪЕКТ; ДОКУМЕНТИРОВАНИЕ

В языке Python всё является объектом, и у любого объекта могут быть атрибуты и методы. Все функции имеют стандартный атрибут `__doc__`, содержащий строку документации, определённую в исходном коде функции. В Python функции — *объекты первого класса*. Функцию можно передать в качестве аргумента другой функции.

Вы можете присвоить имя функции другой переменной и вызвать функцию уже по новому имени. Переменная как функция; **удобно для передачи функции в качестве аргумента**:

```
>>> hello()
Hello world!
>>> hello
<function hello at 0x7fbb89d7bf28>
>>> type(hello)
<class 'function'>
>>> a = hello
>>> a
<function hello at 0x7fbb89d7bf28>
>>> a()
Hello world!
```

Справка о пользовательской функции:

```
>>> dir(a)
['_annotations_', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattr__', '__ibute__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

## ДОКУМЕНТИРОВАНИЕ ФУНКЦИИ

Документирование (введение блочной строки в теле функции):

```
def hello():
    """ This function prints "Hello world!" """
    print ("Hello world!")
```

Просмотр документации пользовательской функции **хелпом**:

```
>>> help(hello)

Help on function hello in module __main__:

hello()
    This function prints "Hello world!"
(END)
```

Просмотр документации **доком** (здесь функция ведет себя как объект: имеет свойство `__doc__`):

```
>>>hello.__doc__
'This function prints "Hello world!'
```


## ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИЮ

### ОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

Если при определении функции был определен параметр, то при вызове функции, обязательно нужно передать параметр в функцию. Можно передавать как константное значение так и переменную. Внутри функции

переменная попадает именно с тем именем, что было в определении.

```
def hello(name):  
    #name = "Peter"  
    print("Hello %s!" % name)  
  
hello("Peter")  
hello("John")  
hello("world")
```



```
ubuntu@ubuntu-VD:~/week2$ python3 functions.py  
Hello Peter!  
Hello John!  
Hello world!
```

Если мы не передаем обязательный параметр, т.е. вызываем функции `hello()`, то возникает соответствующая ошибка.

## ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ

**Значения по умолчанию** позволяют сделать отдельные аргументы функции **необязательными** - если значение не передать при вызове, аргумент получит значение по умолчанию.

Ниже приводится функция, в которой один аргумент является обязательным, а другой имеет значение по умолчанию:

```
>>> def hello(name, title=''):  
    print("Hello %s %s!" % (title, name))  
>>> n = «Peter»  
>>> hello(n, «Sir»)  
Hello Sir Peter!  
>>> hello("John")  
Hello John!  
>>> hello("world")  
Hello world!
```

При вызове такой функции мы обязаны передать значение для аргумента `name`, а значения для аргументов `title` можно опустить. Если функции передать только одно значение, аргумент `title` примет значение по умолчанию, а если два – значение по умолчанию не будет использовано.

**Порядок перечисления позиционных параметров** при вызове функции – нельзя менять!

## КОМБИНИРОВАНИЕ ИМЕНОВАННЫХ АРГУМЕНТОВ И ЗНАЧЕНИЙ ПО УМОЛЧАНИЮ

Мы имеем возможность указывать параметры в явном виде. Тогда порядок, в котором указываются аргументы, может быть изменен. Например:

```
>>> def hello(name, title=''):  
    print("Hello %s %s!" % (title, name))  
>>> hello(name = "John", title = "Mr.")  
Hello Mr. John!  
>>> hello("John", title = "Mr.")  
Hello Mr. John!  
>>> hello(title = "Mr.", name = "John")  
Hello Mr. John!
```

Когда в вызовах функции используются именованные аргументы, порядок их следования не имеет значения, потому что сопоставление выполняется по имени, а не по позициям.

**Важно:** сначала указываем неименованные, а потом именованные, иначе – ошибка. *Сначала позиционные параметры, а потом поименованные.*

**ВАЖНО:** дефолтные значения при объявлении функции должны **быть ее последними аргументами** (иначе – исключительная ситуация, см. листинг ниже); может быть несколько дефолтных значений:

```
>>> def showFullName(n, p = None, s):
...     return 'Fullname: {} {} {}'.format(n, s, p)
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>> def showFullName(n, p = None, s = None):
...     return 'Fullname: {} {} {}'.format(n, s, p)
...

```

## РАСПАКОВКИ ПРИ ПЕРЕДАЧЕ ПАРАМЕТРОВ

### РАСПАКОВКА ПАРАМЕТРОВ ФУНКЦИИ

В Python мы иногда не можем заранее знать, сколько аргументов будем передавать функции. В таких случаях применяется специальный синтаксис `*args` и `**kwargs`. Одна звездочка (`*args`) означает переменное количество позиционных аргументов, а две звездочки (`**kwargs`) – передают переменное количество именованных аргументов. Такое применение возможно в двух случаях: при определении функции и при ее вызове.

```
def hello(name, surname='', title='Mr.'):
    if surname:
        surname = ' ' + surname.strip()
    print "Hello %s %s%s" % (title, name, surname)
a = ["John", "Galt"]
kwa = {'surname': 'Galt', 'title': 'Sir'}
hello(*a) # распаковка списка
hello("John", **kwa) # распаковка словаря

```

- `*args` - распаковка списка в позиционные переменные
- `**kwargs` - распаковка словаря в именованные переменные
- можно по-всякому комбинировать

**Напоминаем:** в случае комбинированного использования позиционных и именованных аргументов: **сначала указываем позиционный параметр, потом – именованный.**

Пример использования набора значений:

```
>>> b = [(1,2,3),(4,5,6),(7,8,9)]
>>> for i in b:
...     print(i)
...
(1, 2, 3)
(4, 5, 6)
(7, 8, 9)
>>> for i in b:
...     f(*i)
...
1
2
3
4
5
6
7
8
9

```

```
>>> def f(a,b,c):
...     print(a)
...     print(b)
...     print(c)
```

Словарь распаковываем при помощи \*\* (т.е. как поименованные параметры).

Пример использования словаря при распаковке:

```
>>> d = {'a': 11, 'b': 22, 'c': 33}
>>> f(*a)
3
2
1
>>> f(**d)
11
22
33
>>> f(a=1, b=2, c=3)
1
2
3
```

Смешанная распаковка параметров:

```
def hello(name, surname='', *, title='Mr.'):
    full_name = name
    if surname:
        full_name = full_name + ' ' + surname
    if title:
        full_name = title + ' ' + full_name
    print("Hello %s!" % full_name)

l = ["Peter", 'Smith']
d = {'title': 'Sir'}
#hello("Peter", 'Smith', title="Sir")
hello(*l, **d) → Hello Sir Peter Smith!
```

## ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ РЕЗУЛЬТАТ

Функция **всегда** по дефолту возвращает **None**, из чего вытекает следующая особенность: в отличие от языков Java, C#, ..., при задании логики функции с оператором **if**, в явном виде не нужно указывать конструкцию **return None** для случая, если проверяемая переменная не соответствует ни одному условию, потому что при отсутствии иного возвращаемого значения будет по умолчанию возвращен **None**:

С помощью **return** можно явно вернуть значение

```
def some_function():
    pass

r = some_function() # r = None

def some_function():
    return 1

r = some_function() # r = 1
```

- выходов из функции может быть несколько, после выполнения инструкции **return**, функция сразу вернет значение и далее выполняться не будет

Само собой разумеется, переменная может хранить результат выполнения функции.

```
def max_value(a, b):
    if a > b:
        return a
    elif a < b:
        return b
    return None

r = max_value(1, 2) # r = 2
r = max_value(4, 3) # r = 4
```

```
r = max_value(5, 5) # r = None
```

## АННОТАЦИЯ ФУНКЦИИ

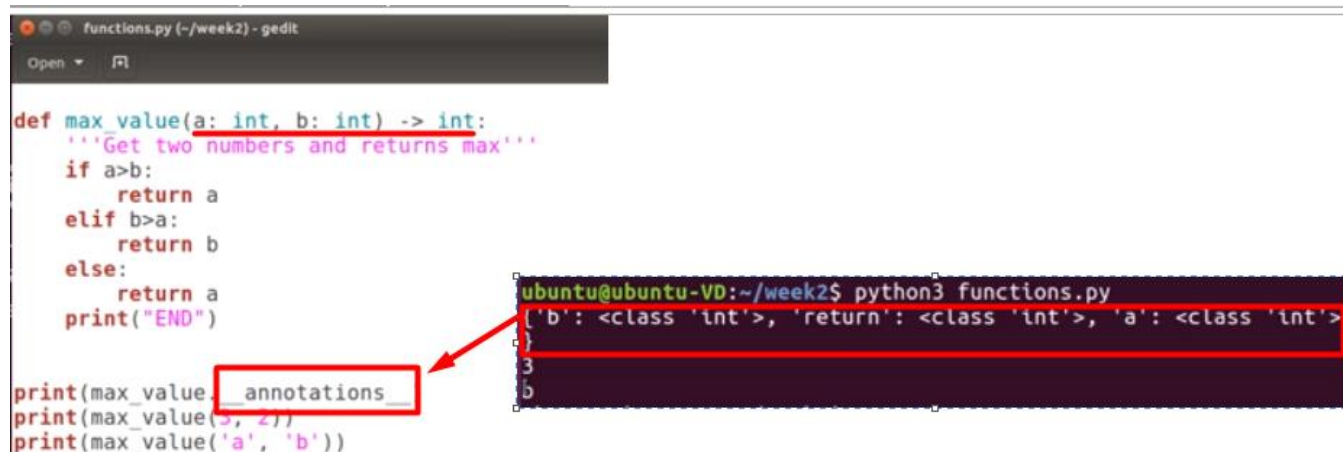
Язык Python поддерживает строгую динамическую типизацию. Результатом этого, является то, что мы можем использовать параметры любых типов типов, что может приводить к неожиданным результатам.

В Python 3 добавилась явная аннотация параметров и возвращаемого значения, которая делает код более читабельным и выразительным, но пока ни на что не влияет:

```
def sum (a:int, b:int) -> int:
    """ Get two numbers and return sum """
    return a + b
```

Информация будет храниться в скрытом атрибуте `__annotations__`

Пример:



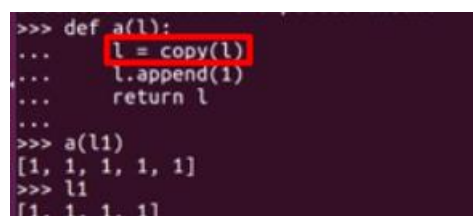
The screenshot shows a code editor with a file named `functions.py`. The code defines a function `max_value(a: int, b: int) -> int:` with a docstring and logic to return the maximum of two integers. Below the function, there are three print statements: `print(max_value.__annotations__)`, `print(max_value(3, 2))`, and `print(max_value('a', 'b'))`. A red box highlights `__annotations__` in the first print statement. To the right, a terminal window shows the output of `python3 functions.py`, which is `{'b': <class 'int'>, 'return': <class 'int'>, 'a': <class 'int'>}`. A red arrow points from the `__annotations__` attribute in the code to the terminal output.

## ПАРАМЕТРЫ ФУНКЦИИ ИЗМЕНЯЕМОГО ТИПА

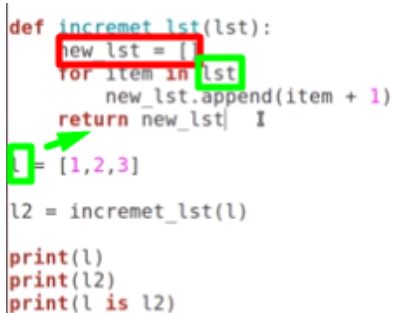
Обсудим как правильно передавать изменяемые объекты в функцию, чтобы они не изменяли оригинальное значение. Проблема возникает, поскольку параметр функции и оригинальное значение это один и тот же объект.

Два пути решения проблемы:

- создаем копию с помощью вызова функции `copy` (`from copy import copy`):
- не менять значение внутри функции.



The screenshot shows a terminal session. A function `a(l):` is defined, which takes a list `l` and creates a copy `l = copy(l)` (highlighted with a red box). It then appends 1 to the copy and returns it. The function is called with `a(l1)`, and the output shows `[1, 1, 1, 1, 1]` for `l1` and `[1, 1, 1, 1]` for `l`.



The screenshot shows a code editor with a function `incremet_lst(lst):`. Inside the function, a new list `new_lst = []` is created (highlighted with a red box). A loop iterates over `lst` (highlighted with a green box), and each item is appended to `new_lst` with an increment of 1. The function returns `new_lst`. Below the function, a list `l1 = [1, 2, 3]` is defined (highlighted with a green box). A new list `l2 = incremet_lst(l1)` is created. Finally, `print(l)`, `print(l2)`, and `print(l is l2)` are executed. The output shows that `l` and `l2` are different objects.

## ПАРАМЕТРЫ ФУНКЦИИ KEYWORD-ONLY

В Python 3 появилась возможность указывать, что некоторые параметры функции должны быть вызваны обязательно **по имени**.

Синтаксически мы отделяем \* те параметры, которые мы будем указывать только по имени.

\* означает окончание блока позиционных параметров.

Пример:

```
def hello(name, surname='', *, title, some=''):
    #name = "Peter"
    print("Hello %s %s %s!" % (title, name, surname))

n = "Peter"
hello(n, 'Smith', title="Sir")

hello("John", title='Sir', some='!|')

hello("world")
```

## ОБЛАСТИ ВИДИМОСТИ ПЕРЕМЕННЫХ (SCOPE)

Мы можем не только передавать параметры внутрь функции, но и создавать собственные переменные внутри нее. Функция создает свое пространство имен.

Три основные области видимости:

- Если присваивание переменной выполняется внутри инструкции def, переменная является **локальной** для этой функции.
- Если присваивание производится в пределах объемлющей инструкции def, переменная является **нелокальной** для этой функции.
- Если присваивание производится за пределами всех инструкций def, она является глобальной для всего файла.

```
>>> x = 99
>>> def func():
    x = 88

>>> print(x)
99
>>> func()
>>> print(x)
99
```

Инструкция присваивания x = 99 создает глобальную переменную с именем 'x' (она видима из любого места в файле), а инструкция x = 88 создает локальную переменную 'x' (она видима только внутри инструкции def).



Предусмотрена инструкция **global**, которая позволяет обратиться к глобальной переменной внутри функции.

Однако Python ZEN не советует “смешивать” области видимости: *Namespaces are one honking great idea--let's do more of those! To есть области видимости должны быть изолированными.*

**Совет:** передавайте глобальные переменные - явно, как параметр функции.

```
>>> a = 1
>>> def outerFunction(a):
...     a += 1
...     def innerFunction(b):
...         return a + b
...     return innerFunction(1)
...
>>> outerFunction(a)
3
```

Функцию можно объявить в теле другой функции. Соответственно у каждой функции создается свое пространство имен (по принципу матрешки).

## АНОНИМНЫЕ ФУНКЦИИ: LAMBDA

Альтернативный способ определения функции.

Используя зарезервированное слово **lambda** вы можете создать небольшую **анонимную(безымянную)** функцию. Здесь представлена функция, которая возвращает сумму двух её аргументов:

```
lambda a, b: a+b
```

Формы **lambda** могут быть использованы в любом месте, где требуется объект функции (в частности, для сохранения результата выполнения функции в переменную либо передачи функции в качестве аргумента другой функции). При этом они **синтаксически ограничены на одно выражение**. Семантически, они лишь **синтаксический сахар** для обычного определения функции, альтернативная форма объявления функции.



```
*lam.py (~/.week2) - gedit
def square(n):
    return n**2

square2 = square
r = square(4)
print(r)

r2 = square2(5)
print(r2)

square3 = lambda x: x**2
r3 = square3(6)
print(r3)
```

ubuntu@ubuntu-VD:~/week2\$ python3 lam.py

16  
25  
36

Примеры применения функций лямба – функциональное программирование. Пример использования функции **map()**:

```

SyntaxError: invalid syntax
>>> a = [i**2 for i in range(10)]
>>> a
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> help(map)

>>> def sq(i):
...     return i**2
...
>>> map(sq, range(10))
<map object at 0x7efe336c5e48>
>>> list(map(sq, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> list(map(lambda x: x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

```

«Продвинутые» темы: set comprehension, reduce

Больше информации об инструментах функционального программирования:

<https://docs.python.org/3/tutorial/datastructures.html#functional-programming-tools>

Книга о функциональном программировании: <http://www.oreilly.com/programming/free/functional-programming-python.csp>

## Modules

### КРАТКОЕ ОПИСАНИЕ

Часто по мере роста программы возникает желание разбить ее на несколько файлов, чтобы упростить ее разработку и дальнейшее сопровождение. Язык Python позволяет помещать определения в файлы и использовать их как модули, которые могут импортироваться другими программами и сценариями.

Чтобы определить модуль, достаточно воспользоваться текстовым редактором, с его помощью ввести некоторый программный код и сохранить его с расширением ".py" - любой такой файл автоматически будет считаться модулем.

Использовать модуль в других программах можно с помощью инструкций (на примере модуля datetime):

- **from datetime import timedelta** : используется для импорта конкретных функций модуля; в дальнейшем обращение к функциям упрощается – дот-нотация модуль.функция не используется (только функция);
- **from datetime:** используется для импорта всех функций модуля; обращение к функциям осуществляется через дот-нотацию модуль.функция (например, `datetime.timedelta(days=3)`)

Когда импортируется модуль, например `spam`, интерпретатор ищет файл с именем "spam.py" в **текущем** каталоге, затем в каталогах, указанных в переменной окружения **PYTHONPATH**, затем в **зависящих от платформы путях** по умолчанию. Каталоги, в которых осуществляется поиск, хранятся в переменной **sys.path**

Импортировать из модуля можно не только переменные, но и функции.

Модуль определяется наличием в директории файла `__init__.py`

### КАК ИЗБЕЖАТЬ РАСПРОСТРАНЕННУЮ ОШИБКУ?

Случается, что при выполнении практических заданий студенты, запуская скрипт из файла, называют файл именем модуля. К примеру, называют файл `datetime.py` и в скрипте, помещенном в данный файл, импортируют модуль `datetime`. При таком подходе ни одна из функций `datetime` не будет доступна, будет валиться `AttributeError`! И такое поведение вполне логично: Питон **сначала ищет модули в текущей директории**.

**Вывод:** не нужно называть исполняемые файлы именем модуля, который Вы импортируете! Если Вы все же наткнулись на эту ошибку, исправить ее можно следующим образом:

- переименовать файл из, к примеру, `datetime.py` в `datetimeScript.py`
- удалить скрытый файл `datetime.pyc`: в окне директории Убунту их можно вывести нажатием комбинации клавиш `Ctrl + H` (либо в терминале - команда `rm`).

Все вышеописанные действия следует выполнить в текущей директории.

### РЕШЕНИЕ КВАДРАТНОГО УРАВНЕНИЯ

Видео "Решение квадратного уравнения", демонстрирующее работу с модулями, поясняет проверку на тип запускаемого файла, которая позволяет избежать дублирования вызова функции, импортированной из главного файла. Проверка имеет вид:

**В чем предназначение блока проверки `if __name__ == 'main'?`**

Подобная проверка имеет в общем случае выглядит так:

```
if __name__ == 'main':  
    main()
```

где `__name__` - имя текущего (вызываемого, запускаемого) файла. Проверка: в случае, если файл является "главным" (содержащим логику реализации импортированного модуля), вызываем импортированную из него функцию `main()`; предусмотренные в скрипте субмодуля вызовы импортированной функции не будут осуществлены.

Проверка `if __name__ == 'main'` позволяет избежать дублирования вызова импортированной функции, когда и в главном модуле, и в субмодуле предусмотрен вызов "главной" функции (рассматриваемой; здесь - функции `main`). Приведем упрощенный пример:

#### **Содержимое файла `flat_script.py`:**

```
def main():
    print 'Wanna call main() ONCE!'

main()
```

#### **Содержимое файла `other_script.py`:**

```
from flat_script import main

main()
```

#### **Исполняя скрипт файла `other_script.py`, получим:**

```
$ python other_script.py
Wanna call main() ONCE!
Wanna call main() ONCE!
```

#### **Добавим в скрипт файла `other_script.py` проверку - скрипт имеет вид:**

```
from flat_script import main

# main()

if __name__ == 'main':
    main()
```

#### **РЕЗУЛЬТАТ исполнения:**

```
$ python other_script.py

Wanna call main() ONCE!
```

Итак, проверка `if __name__ == 'main':`

- предназначена для избежания дублирования вызова функции `main()`, импортированной субмодулем `other_script`, запуская импортируемую функцию `main()` только в случае успешного прохождения проверки
- позволяет определить имя текущего, запускаемого модуля, определенного в `__name__`
- позволяет определить, является ли текущий модуль "главным", основным, "main"
- позволяет определить, запускается ли скрипт непосредственно либо при импортировании основного модуля

**Подробнее об этом:** <http://stackoverflow.com/questions/419163/what-does-if-name-main-do>  
[https://docs.python.org/3/library/\\_main\\_.html](https://docs.python.org/3/library/_main_.html)

а также в книге Mark Lutz "Learning Python".