14. Функции

- Определение функции
- Пустая функция
- Встроенные функции
- Аргументы (параметры) функции
- Оператор return
- Позиционные аргументы
- Именованные (ключевые) аргументы
- Переменное количество параметров в функции *args, **kwargs
- Строки документации
- Анонимные функции
- Точка входа
- Область видимости

Для того чтобы многократно использовать одни и те же фрагменты кода в своих программах, используются функции.

Определение функции

Функция создается путем объявления ключевого слова **def** (*definition - определение*), а затем **имени функции с круглыми скобками**, за которым следует символ **двоеточия**. Как и в случае с циклами, тело функции должно содержать **отступ в 4 пробела**.

```
def my_first_foo():
    print('This is my first function')
```

Рисунок.1 – Определение функции

Таким образом, мы создали простую функцию с именем ***my_first_foo***, при вызове которой на экране будет отображаться текст *"This is my first function "*

Важный момент: Код функции должен быть написан до ее вызова, иначе компилятор выдаст ошибку.

Чтобы вызвать эту функцию в нужном месте программы (а без вызова она не будет работать), достаточно **написать ее имя со скобками** (круглыми) *Рис.1.2*

```
def my_first_foo():
    print('This is my first function')


my_first_foo()  # вызов функции
```

Рисунок.1.2 – Вызов функции

Затем будет выполнено тело функции, и в этом случае будет отображен текст *"This is my first function".*

```
This is my first function

Process finished with exit code 0
```

Рисунок.1.3 – Результат Рис.1.2

Код функции может содержать такие конструкции, как *циклы*, *списки*, *условные операторы* и *любые другие конструкции*, используемые вне функций.

Пустая функция

Когда вы создаете первоначальный скелет программы, вы часто будете использовать *пустые функции*. Пустая функция ничего не содержит в себе и содержит заглушку (ключевое слово) **pass**, которая ничего не делает.

```
def my_second_foo():
    pass
```

Рисунок.2 – Создание пустой функции

Если вызвать такую функцию, ничего не произойдет. Пустые функции нужны, чтобы увидеть полную картину будущей программы. Одну за одной, в процессе создания программы, вы будете заполнять нужным функционалом.

```
def my_second_foo():
    pass


my_second_foo()  # вызов пустой функции, при вызове ничего не произойдет
```

Рисунок.2.1 – Вызов функции

Встроенные функции

Стоит сказать, что мы начали использовать встроенные в Python функции с самого начала изучения. Например, функция **print()**. Передавая ей в качестве параметра свой текст, мы выводим его на экран.

Точно так же мы можем передавать разный текст при каждом вызове нашей функции. Для этого в скобках нам нужно обозначить **имя параметра**, к которому мы будем далее ссылаться.

При написании кода вы часто будете использовать встроенные функции, такие как **len()**, **print()**, **sum()**, **max()**, **min()**, **sorted()**, **next()**, **open()**, **map()**, **zip()**, **any()**, **all()**, **isintance()**, **str()** и другие

Список всех встроенных функций **Python 3** можно найти здесь

Аргументы (параметры) функции

Параметр – это переменная в определении функции, указывающая аргумент, который функция должна принять.

```
def my_foo(some_text):
    print(some_text)
```

Рисунок.3 – Создание функции с параметром (аргументом)

В коде выше мы добавили функции **my_foo** один параметр, под названием ***some_text*** *Рис.3.* Теперь попробуем вызвать её с разными аргументами

```
def my_foo(some_text):
    print(some_text)


my_foo('Явное лучше неявного.')
my_foo('Сложное лучше запутанного.')
my_foo('Особые случаи не настолько особые, чтобы нарушать правила.')
```

Рисунок.3.1 – Вызов функции с разными аргументами

По сути, мы создали аналог встроенной функции **print()**, которая выводит переданный ей текст.

Мы можем передавать **в качестве параметра любые значения и переменные** и использовать их уже в коде функции *Рис.3.2.*

```
def my_sum_foo(a, b):
    print(a + b)


x, y = 10, 8
my_sum_foo(x, y)
```

Рисунок.3.2 – Функция, которая суммирует два переданных ей значение

Как видно из приведенного выше примера, функция получает два параметра в качестве входных данных, суммируя, результат выводит на экран. *Рис.3.3*

```
18

Process finished with exit code 0
```

Рисунок.3.3 – Результат Рис.3.2

Имена переменных могут быть любыми и не обязательно совпадать с именами параметров в функции. Мы можем получить тот же результат, не используя имена переменных, **сразу передавая необходимые значения в вызов функции** *Рис.3.4.*

```
def my_sum_foo(a, b):
    print(a + b)


my_sum_foo(10, 8)
```

Рисунок.3.4 – Вызов функции путем передачи необходимых значений в вызов

Оператор return

Мы так же можем **передать результат работы функции** в какую-то переменную. Для этого нам необходимо "*объяснить*" компилятору, что делать по завершению работы этой функции ключевым словом **return**

```
def my_multi_foo(a, b):
    res = a * b
    return res  # возвращает результат умножения


# вызов функции, возвращаемое значение присваивается в переменную result
result = my_multi_foo(3, 12)
print(result)
```

*Рисунок.4 – Оператор **return***

В приведенном выше коде результат функции был присвоен переменной result, которую мы распечатали *Рис.4.*

Функции **всегда возвращают значение**. И если мы явно не указали возврат значения с помощью **return**, то возвращается значение **None** (*нет, ничего, ничто*).

Мы можем распечатать и увидеть, что для **None** присвоен специальный тип данных **NoneType**.

```
print(None)
print(type(None))
```

*Рисунок.4.1 – Тип **NoneType***

```
None
<class 'NoneType'>

Process finished with exit code 0
```

Рисунок.4.2 – Результат Рис.4.1

Позиционные аргументы

Функции, которые вы видели выше, имели **позиционные аргументы**. **При вызове** функций значения в такие аргументы **подставляются согласно позиции имён аргументов** в определении функции.

То есть параметры, передаваемые функции, считываются последовательно, и **их количество должно совпадать** с тем, **которое указано в описании функции**.

```
def my_power_foo(a, b):
    result = a ** b
    print(result)


my_power_foo(5, 2)
```

*Рисунок.5 – Функция, возводящая число **a** в степень **b***

```
25

Process finished with exit code 0
```

Рисунок.5.1 – Результат Рис.5

Если изменить порядок параметров, то получим совершенно другой результат

```
def my_power_foo(a, b):
    result = a ** b
    print(result)


my_power_foo(2, 5)
```

*Рисунок.5.2 – Функция, возводящая число **a** в степень **b***

```
32

Process finished with exit code 0
```

Рисунок.5.3 – Результат Рис.5.2

Именованные (ключевые) аргументы

Часто бывает так, что вам нужно указать порядок параметров произвольно. В таких случаях **параметры передаются по имени** (*ключу*).

```
def my_multi_foo(a, b):
    res = a ** b
    print(res)


my_multi_foo(b=2, a=5)   # аргументы передаются по имени параметра
```

*Рисунок.6 – Функция, возводящая число **a** в степень **b***

```
25

Process finished with exit code 0
```

Рисунок.6.1 – Результат Рис.6

Иногда необходимо указать **значение по умолчанию** для параметра. В этом случае мы можем передать в параметр нужные нам данные, но если их нет, то будет подставлен тот, который был указан изначально

```
def my_multi_foo(a, b=10):
    res = a * b
    return res


# передается один аргумент, вместо b подставляется значение по умолчанию
result_1 = my_multi_foo(5)

# передаются оба аргумента, значение b по умолчанию игнорируется
result_2 = my_multi_foo(5, 20)

print(result_1)
print(result_2)
```

Рисунок.6.2 – Значение параметра по умолчанию

```
50
100

Process finished with exit code 0
```

Рисунок.6.3 – Результат Рис.6.2

Переменное количество параметров в функции, *args, **kwargs

В Python можно передать переменное (неизвестное) количество параметров двумя способами:

1. ***args** для **неименованных** (позиционных) аргументов
2. ****kwargs** для **именованных** (ключевых) аргументов

Аргументы функции в Python, обозначаемые символами * и **, на самом деле **являются коллекциями** – списками, кортежами или словарями. Первые два передают позиционные аргументы, а словари передают именованные аргументы.

Чтобы передать несколько аргументов функции в виде коллекции, коллекцию **необходимо распаковать**. Делается это с помощью **одной** или **двух звездочек**.

Имена не обязательно должны быть ***args** и ****kwargs**, они могут быть любыми, **главное здесь оператор звездочка**(ки).

```python
def check_parameters(*params):
    if len(params) == 0:
        return 'Параметры не найдены...'

    total = 0
    for i in params:
        total += i

    return total


# вызов функции без параметров
res_1 = check_parameters()  # Параметры не найдены...

# вызов функции с тремя параметрами
res_2 = check_parameters(5, 2, 3)  # сумма аргументов

print(res_1)
print(res_2)
```

Рисунок.7 – Неизвестное количество позиционных аргументов

При вызове функции все позиционные параметры заносятся в кортеж ***params***, работа с которым проходит по стандартным правилам.

```
Параметры не найдены...
10

Process finished with exit code 0
```

Рисунок.7.1 – Результат Рис.7

Вы также можете сделать распаковку прямо в вызове функции.

```
def login(*args):
    print(args)


admins = ['RedBanana', 'Yolo54', 'Cat101']
login('owner', *admins)
```

Рисунок.7.2 – Распаковка списка при вызове функции

```
('owner', 'RedBanana', 'Yolo54', 'Cat101')

Process finished with exit code 0
```

Рисунок.7.3 – Результат Рис.7.2

****kwargs** принимает **именованные аргументы** и отображает словарь.

```
def login_and_password(**kwargs):
    return kwargs


print('Логин\tПароль')
print(login_and_password(admin='myStrongPassword', lisa54='qwerty123',
guest='lolBadPassword'))
```

Рисунок.7.4 – Неизвестное количество ключевых аргументов

```
Логин   Пароль
{'admin': 'myStrongPassword', 'lisa54': 'qwerty123', 'guest': 'lolBadPassword'}

Process finished with exit code 0
```

Рисунок.7.5 – Результат Рис.7.4

Также можно **комбинировать позиционные** и **именованные** (*ключевые*) аргументы. Если вам нужно передать в функцию параметры того и другого типа, то вам достаточно указать их через запятую. Но при этом **позиционные аргументы передаются до ключевых**.

```
def pos_and_key_arguments(*args, **kwargs):
    # позиционные аргументы, выводятся в консоль в строковом виде
    print('\t'.join(args))

    return kwargs  # возвращается ключевые аргументы


print(pos_and_key_arguments('Логин', 'Пароль', admin='myStrongPassword',
lisa54='qwerty123', guest='lolBadPassword'))
```

Рисунок.7.6 – Комбинация позиционных и ключевых аргументов

```
Логин   Пароль
{'admin': 'myStrongPassword', 'lisa54': 'qwerty123', 'guest': 'lolBadPassword'}

Process finished with exit code 0
```

Рисунок.7.7 – Результат Рис.7.6

Строки документации

Python имеет одну особенность, называемую **строками документации**, обычно обозначаемую **сокращённо docstrings**. Этот инструмент помогает лучше документировать программу и облегчает ее понимание. Строку документации, в отличии от комментариев, можно получить из функции, даже во время выполнения программы.

Доступ к строке документации функции можно получить с помощью атрибута этой функции **__doc__**, который нужно записать через точку, и вывести с помощью встроенной функции **print()**.

```python
def number_max(x, y):
    """
    Выводит максимальное из двух чисел.
    Оба значения должны быть целыми числами.

    :param x: целое число
    :param y: целое число
    :return: None
    """

    x = int(x)  # конвертируем в целые, если это необходимо
    y = int(y)

    if x > y:
        print(x, 'наибольшее')

    elif x < y:
        print(y, 'наибольшее')

    else:
        print('Числа равны')


number_max(28, 72)
print(number_max.__doc__)  # вывод строки документации
```

Рисунок.8 – Строка документации

```
72 наибольшее

    Выводит максимальное из двух чисел.
    Оба значения должны быть целыми числами.

    :param x: целое число
    :param y: целое число
    :return: None


Process finished with exit code 0
```

Рисунок.8.1 – Результат Рис.8

Почти такой же результат можно получить, используя встроенную функцию **help()**

```
number_max(43, 10)
print(help(number_max))   # вывод строки документации
```

*Рисунок.8.2 – Встроенная функция **help()***

```
43 наибольшее
Help on function number_max in module __main__:

number_max(x, y)
    Выводит максимальное из двух чисел.
    Оба значения должны быть целыми числами.

    :param x: целое число
    :param y: целое число
    :return: None

None

Process finished with exit code 0
```

Рисунок.8.3 – Результат Рис.8.2

Анонимные функции

Анонимные функции создаются с помощью инструкции **lambda** **Они не поддерживают строку документации**. Кроме того, вам **не нужно делать отступы**, как мы это делали с инструкцией **def foo()**

Лямбда-функции **могут содержать только одну строку кода**. Также, **lambda** функции, в отличие от обычной, **не требует оператора return**.

```
my_sum_foo = lambda a, b: a + b  # лямбда функция, которая суммирует два
значения

print(my_sum_foo(5, 35))
```

Рисунок.9 – Лямбда функция

```
40

Process finished with exit code 0
```

Рисунок.9.1 – Результат Рис.9

Лямбда-функции **могут передаваться без необходимости предварительного присвоения переменной**. Её можно использовать внутри стандартной функции.

```
def power(n):
    return lambda x: x ** n


exp = power(6)  # в exp присваивается значение(выражение), которое
возвращается функцией power()

print(exp(3))
```

Рисунок.9.2 – Лямбда функция

```
729

Process finished with exit code 0
```

Рисунок.9.3 – Результат Рис.9.2

В лямбда выражениях **можно использовать условные операторы (тернарные операторы).** Синтаксис отличается от обычного оператора **if**

```
check_odd_even = lambda x: 'Четное' if x % 2 == 0 else 'Нечетное'

print(check_odd_even(17))
print(check_odd_even(48))
```

Рисунок.9.4 – Лямбда функция

```
Нечетное
Четное

Process finished with exit code 0
```

Рисунок.9.5 – Результат Рис.9.4

Лямбда-функции **отлично подходят для сортировки**, где они используются довольно часто.

Допустим, нам нужно отсортировать список по длине. Это легко сделать с помощью стандартной функции **len()**. А если мы возьмем более сложный вариант – *сортировку по последней букве значений* в списке?

Вот тут-то и пригодится лямбда.

```
fruits = ['Яблоко', 'Персик', 'Кокос', 'Айва', 'Ананас', 'Банан']

fruits.sort(key=lambda x: x[-1])  # сортировка по последней букве значений

print(fruits)
```

Рисунок.9.6 – Лямбда функция

```
['Айва', 'Персик', 'Банан', 'Яблоко', 'Кокос', 'Ананас']

Process finished with exit code 0
```

Рисунок.9.7 – Результат Рис.9.6

Точка входа

Многие языки программирования имеют функцию **main()**, которая запускает выполнение программы. В Python ее как таковой нет.

В программах, написанных на Python, часто можно увидеть следующую конструкцию:

```
if __name__ == '__main__':
    # Код основной программы
```

Рисунок.10 – Точка входа

Её часто называют **"точкой входа в программу"**. Эта конструкция работает как разделитель, поэтому файл можно использовать как:

- Основную программу (*и запустить код после конструкции if*)
- Модуль (*и не выполнять код после if*)

Если интерпретатор запускает исходный файл как основную программу, он присваивает для специальной переменной **__name__** значение **"__main__"**. Если этот файл импортирован из другого модуля, в переменной **__name__** будет присвоено имя этого модуля.

```
def foo():
    print('Hello World!')


if __name__ == '__main__':
    '''Выполнится только когда запускается, а не импортируется.'''
    foo()

print('Python is fun!')
```

Рисунок.10.1 – Точка входа

```
Hello World!
Python is fun!

Process finished with exit code 0
```

Рисунок.10.2 – Результат Рис.10.1

Тему модулей мы рассмотрим на следующем уроке, но вкратце, **любую программу можно импортировать как модуль**.

Например, если мы напишем **import foo_example** в другом файле (где **foo_example** – название программы (файла) с экрана выше) и запустим его, то получится только *'Python is fun!'* Так как условие **if не будет выполняться при импорте**.

Конструкция **if __name__ == '__main__':** позволяет логически организовать код и сделать его читабельным. Схематически это можно представить следующим образом:

```python
def my_foo():
    print('Hello Alice!')


def my_foo_2():
    print('Hello Siri!')


def my_foo_3():
    print('Hello Alexa!')


if __name__ == '__main__':
    my_foo()
    my_foo_2()
    my_foo_3()
```

Рисунок.10.3 – Точка входа

Область видимости

В Python **переменная доступна** только **в той области, в которой она создана**. Эта область называется **областью видимости**. Такая особенность позволяет **ограничить доступ** к определенным значениям, **чтобы избежать конфликтов** между **одинаковыми идентификаторами**.

Переменные бывают двух типов, **локальные** и **глобальные**.

Локальные переменные

Чтобы **создать переменные с локальной областью видимости**, просто нужно **поместить их в отдельный блок кода**, изолированный от остальной части программы.

```
def my_foo():
    a = 3 ** 5
    print(a)


my_foo()  # 243
print(a)  # NameError: name 'a' is not defined
```

Рисунок.11 – Локальные переменные

```
    print(a)  # NameError: name 'a' is not defined
NameError: name 'a' is not defined
243

Process finished with exit code 1
```

Рисунок.11.1 – Результат Рис.11

При попытке напечатать результат выполнения в переменной **a** мы **получили ошибку**, так как **переменная находится внутри функции**, и **является локальной**. В то же время **print()** внутри функции отработал успешно.

Глобальные переменные

Чтобы иметь возможность **использовать переменные в любой части программы**, вы **должны объявить глобальную переменную**. Для этого нужно **создать переменную отдельно от области кода**, ограниченной отдельным блоком кода, например, функцией.

```
a = 2 ** 5


def foo():
    print(a)


foo()  # 32
print(a)  # 32
```

Рисунок.12 – Глобальные переменные

```
32
32

Process finished with exit code 0
```

Рисунок.12.1 – Результат Рис.12

```python
a = 8 ** 3  # глобальная переменная


def foo():
    # локальная переменная a = 7 + 17
    a = 7 + 17
    print(a)


foo()  # 24
print(a)  # 512
```

Рисунок.12.2 – Глобальные и локальные переменные

Обратите внимание на этот пример – переменные, **а никак** между собой **не пересеклись** и имеют **разные значения**, так как **находятся в разных областях видимости**.

```
24
512

Process finished with exit code 0
```

Рисунок.12.3 – Результат Рис.12.2

Конечно, вы **не должны использовать одни и те же имена** переменных в реальной программе.

И если локальная переменная в функции недоступна извне, то она будет доступна любой функции, находящейся внутри этой функции.

```python
def some_outer_func():
    a = 'Переменная'  # локальная переменная

    def some_inner_func():
        print(a)  # вложенная функция

    some_inner_func()


some_outer_func()
```

Рисунок.12.4 – Глобальные и локальные переменные

```
Переменная

Process finished with exit code 0
```

Рисунок.12.5 – Результат Рис.12.4

Ключевое слово global

Использование ключевого слова **global** **необходимо тогда**, когда есть потребность **сделать локальную переменную глобальной**.

```python
def my_foo():
    global a
    a = 2 ** 5 + 8


my_foo()
print(a)  # 40
```

*Рисунок.13 – Ключевое слово **global***

```
40

Process finished with exit code 0
```

Рисунок.13.1 – Результат Рис.13

Вы также можете **использовать** ключевое слово **global**, чтобы **изменить значение глобальной переменной из локальной области видимости.**

```python
b = 16  # глобальная переменная


def my_foo():
    global b  # в локальной области видимости, но глобальная переменная
    global a  # глобальная переменная

    b = 50  # глобальной переменной b присваивается новое значение
    a = 2 ** 5 + 8
    a += b


my_foo()
print(a)  # 90
print(b)  # 50
```

*Рисунок.13.2 – Ключевое слово **global***

```
90
50

Process finished with exit code 0
```

Рисунок.13.3 – Результат Рис.13.2

Ключевое слово nonlocal

Нелокальные области видимости **возникают**, когда вы **определяете функции внутри функций**. Когда переменная объявлена через **nonlocal**, она **будет ссылаться на одноименную переменную в ближайшем замыкании**, исключая глобальные переменные.

```python
x = 0  # глобальная переменная


def some_outer():
    x = 1  # локальная переменная

    def some_inner():
        nonlocal x  # не локальная переменная
        x = 20  # новое значение присваивается для x в some_outer()
        print("Вложенный x:", x)

    some_inner()
    print("Внешний x:", x)


some_outer()
print("Глобальный x", x)
```

*Рисунок.14 – Ключевое слово **nonlocal***

```
Вложенный x: 20
Внешний x: 20
Глобальный x 0

Process finished with exit code 0
```

Рисунок.14.1 – Результат Рис.14

Если мы уберём **nonlocal** из примера *Рис.14*, то получим другой вывод, так как область видимости **изменится на локальную**.

```python
x = 0  # глобальная переменная


def some_outer():
    x = 1  # локальная переменная для some_outer()

    def some_inner():
        x = 20  # локальная переменная для some_inner()
        print("Вложенный x:", x)

    some_inner()
    print("Внешний x:", x)


some_outer()
print("Глобальный x", x)
```

*Рисунок.14.2 – Ключевое слово **nonlocal***

```
Вложенный x: 20
Внешний x: 1
Глобальный x 0

Process finished with exit code 0
```

Рисунок.14.3 – Результат Рис.14.2