KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 3 урок > Основы программирования Python. Пользовательские функции

> Оглавление 3 урока

Скачать Ноутбук

Скачать в PDF

Скачать файл, использованный в лекции

> Работа с файлами

В Python существует встроенный интерфейс для работы с файлами. С помощью него можно читать и редактировать существующие файлы, а также

создавать новые.

Пример

Пусть наш файл называется `responses.txt` — простой текстовый файл.

```
# cat — консольная команда чтения файлов
!cat responses.txt
```

Output:

```
Хорошо бы снизить цены
Нужно снизить цены
Меня все устраивает
У конкурентов цены лучше
```

Note: В JupyterNotebook можно выполнять команды терминала, с которыми вы познакомитесь позднее. Выше приведена команда `cat` (это не Python).

Теперь сделаем так, чтобы Python смог обращаться к текстовым данным. Рекомендуемый вариант чтения файлов — с помощью контекстного менеджера `with open()`.

```
# Передаем путь
with open('responses.txt') as f:
    text = f.read()
```

где:

`with` — ключевое слово;

`open('path/to/file')` — функция, которая открывает файл, где `path/to/file` — путь до файла или название файла;

`as f` — переменная `f`, через которую идет обращение к файлу в дальнейшем;

`text = f.read()` — метод файла `read()` запишет переменную `f` в другую переменную `text`. При этом тип данных станет `str` , а строки текста будут

разделены пробельным символом `\n`.

`'Хорошо бы снизить цены\nНужно снизить цены\nМеня все устраивает\nУ конкурентов цены лучше'`

Поэтому по таким строкам легко итерироваться:

```
with open('responses.txt') as f:
    for line in f:
        print(line)
```

Output:

```
Хорошо бы снизить цены

Нужно снизить цены

Меня все устраивает

У конкурентов цены лучше
```

А теперь давайте посчитаем число вхождений слов в ответы пользователей в файле `response.txt`.

```
# Словарь, где будут храниться кол-ва элементов
word_counter = {}

# Открываем файл
with open('responses.txt') as f:

    # идем итеративно по строкам
        for line in f:

        # удалим пробельные символы с концов
        strip_line = line.strip()

        # разделим по пробелам
                words = strip_line.split()
```

```
        # для каждого слова предложения
                for word in words:

            # если оно уже есть в словаре
                        if word in word_counter:
                word_counter[word] += 1

            # иначе
                        else:
                word_counter[word] = 1

word_counter
```

Output:

```
{'Хорошо': 1,
 'бы': 1,
 'снизить': 2,
 'цены': 3,
 'Нужно': 1,
 'Меня': 1,
 'все': 1,
 'устраивает': 1,
 'У': 1,
 'конкурентов': 1,
 'лучше': 1}
```

Но как теперь записать получившийся словарь в новый файл?

Модификаторы доступа

На вход функция `open()` может принимать два аргумента. Как говорилось ранее, первый — это путь до файла, а второй — определенный режим или модификатор доступа. Он говорит, что конкретно можно делать с файлами. Ниже представлены доступные режимы.

Режим	Обоснование
`'r'`	открытие на чтение (является значением по умолчанию)
`'w'`	открытие на запись, содержимое файла удаляется; если файла не существует, создается новый
`'x'`	открытие на запись, если файла не существует, иначе исключение
`'a'`	открытие на дозапись, информация добавляется в конец файла
`'b'`	открытие в двоичном режиме
`'t'`	открытие в текстовом режиме (является значением по умолчанию)
`'+'`	открытие на чтение и запись

Мы хотим записать новый файл в файл с названием `result.txt`. Для этого используем модификатор доступа `'w'`, что означает `write`.

```
# Обязательно строковый тип
with open('result.txt', 'w') as f:
    f.write(str(word_counter))

# Чтение файлов через команду cat
!cat result.txt
```

Output:

```
{'Хорошо': 1, 'бы': 1, 'снизить': 2, 'цены': 3, 'Нужно': 1,
'Меня': 1, 'все': 1, 'устраивает': 1, 'У': 1, 'конкурентов':
1, 'лучше': 1}
```

Кодировка

В Python кодировка представляет собой способ, с помощью которого символы текста преобразуются в байты для хранения или передачи данных. Python поддерживает различные кодировки, такие как UTF-8, ASCII, ISO-8859-1 и другие, которые определяют, как символы Unicode будут представлены в байтах.

Указание правильной кодировки важно для корректного отображения и обработки текстовых данных.

В функции `open()` в Python используется параметр `encoding` для указания кодировки, которая будет использоваться при чтении или записи файла.

В большинстве своем, если вы работаете с файлом, содержащим текст на русском языке, вы можете указать кодировку UTF-8, которая широко используется для работы с различными языками.

Пример:

```
with open('file.txt', 'r', encoding='utf-8') as f:
    print(f.read())
```

Этот код открывает файл для чтения с использованием кодировки UTF-8. Это гарантирует, что текст из файла будет правильно интерпретирован и прочитан как строки Unicode.

Если мы предположим, что файл `file.txt` сохранен в кодировке `cp1251` (Windows-1251), то при попытке прочитать такой файл с использованием неправильной кодировки, например, `utf-8`, может возникнуть ошибка `UnicodeDecodeError`.

Таким образом, кодировка в Python играет важную роль при работе с текстовыми данными, а правильное указание кодировки помогает избежать проблем с интерпретацией символов и обработкой текста.

Подробнее про кодировку тут.

> Пользовательские функции

Допустим, вы написали код, который решает бóльшую часть вашей работы, или несколько участков кода, которые дублируются и выполняют одинаковые логические операции. Такой код можно преобразовать в **пользовательские функции**.

Зачем нужны пользовательские (или именные) функции:

- Переиспользование кода

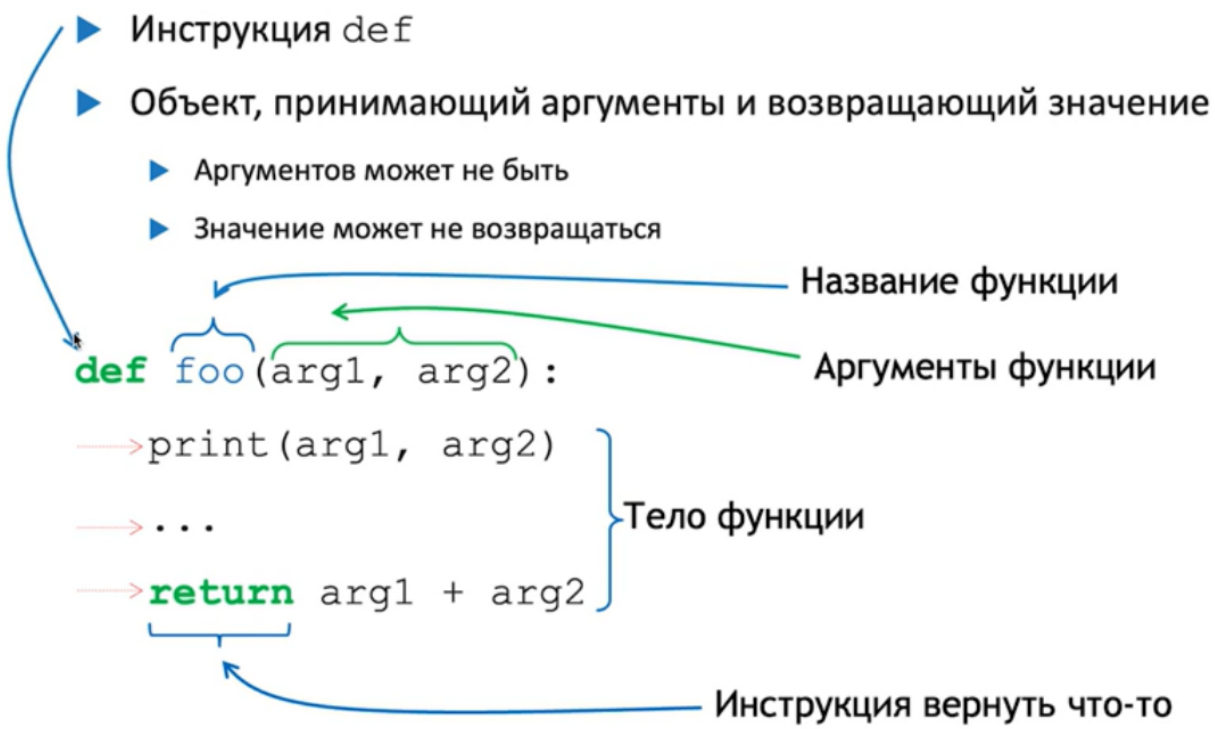
- Уменьшение вероятности ошибки
- Структурирование кода

Пользовательская (или именная) функция имеет следующий вид:

```
def function_name(arg1=default_value1, arg2=default_value2,...)
    Код
    return результат
```

где

- `def` — ключевое слово, которое указывает на то, что дальше начинается функция;
- `function_name` — название функции;
- `args1`, `args2`... `argsn` — названия аргументов, которые будут использоваться внутри функции.



Важно: перед кодом должен быть отступ в 4 пробела или tab, иначе функция не заработает.

Пример:

Напишем код, который будет считывать файл `responses.txt` , считать число вхождений слов в ответы пользователей в нем и записывать результат в файл `result.txt`.

```
def file_processing():
    # Словарь, где будут храниться кол-ва элементов
    word_counter = {}

    # Открываем файл
        with open('responses.txt') as f:

        # идем итеративно по строкам
                for line in f:

            # удалим пробельные символы с концов
            strip_line = line.strip()

            # разделим по пробелам
            words = strip_line.split()

            # для каждого слова предложения
                        for word in words:

                # если оно уже есть в словаре
                                if word in word_counter:
                    word_counter[word] += 1

                # иначе
                                else:
                    word_counter[word] = 1
```

```
    # Обязательно строковый тип
        with open('result.txt', 'w') as f:
        f.write(str(word_counter))
```

Аргументы функции

- Каждый аргумент должен быть инициализирован
- Никакой аргумент не может быть инициализирован дважды
- Именованные аргументы только после позиционных (неименованных)
- Можно создавать дефолтные значения

```
def function_name( [ позиционные_аргументы,
                   [ *дополнительные_поз_арг,
                   [ именованные_аргументы,
                   [ **доп_имен_арг]]]]):
```

Именованные и позиционные (неименованные) аргументы в функциях Python — это два способа передачи аргументов в функцию.

Позиционные аргументы — это аргументы, которые передаются в функцию в том порядке, в котором они указаны в определении функции. При вызове функции значения аргументов передаются по порядку, начиная с первого аргумента и заканчивая последним.

Пример:

```
def add_numbers(a, b):
    return a + b

# Вызов функции
add_numbers(3, 5)
```

В этом примере аргументы `3` и `5` передаются в функцию `add_numbers` по порядку, где первый аргумент `a` равен `3`, а второй аргумент `b` равен `5`.

Именованные аргументы — это аргументы, которые передаются в функцию с указанием имени аргумента. При вызове функции значения аргументов передаются с указанием имени аргумента, что позволяет передавать аргументы в любом порядке и явно указывать, какое значение относится к какому аргументу.

Пример:

```
def greet(name, greeting):
    print(f'{greeting}, {name}!')

# Вызов функции
greet(greeting='Hi', name='Alice')
```

В этом примере аргументы `name` и `greeting` передаются в функцию `greet` с указанием их имени, что позволяет передавать их в любом порядке.

Использование именованных и позиционных (неименованных) аргументов в Python обеспечивает гибкость при передаче аргументов в функцию и делает код более читаемым и понятным.

Параметры по умолчанию

- В функциях можно передавать сколько угодно аргументов через запятую
- Функция может иметь параметры по умолчанию

Пример:

```
def simple_args(phrase='Hi there'):
    print(phrase)

simple_args()
```

Output:

```
Hi there
```

Еще пример, где считывается входящий файл, указанный в переменной `input_file`, происходит подсчет числа вхождений слов в ответы пользователей в этом файле, а результат выполнения записывается в файл с именем, указанным в переменной `output_file`.

```
def file_processing(input_file, output_file):
    # Словарь, где будут храниться кол-ва элементов
    word_counter = {}
    print(f'Работаю с файлом {input_file}')

    # Открываем файл
        with open(input_file) as f:

        # идем итеративно по строкам
                for line in f:

            # удалим пробельные символы с концов
            strip_line = line.strip()

            # разделим по пробелам
            words = strip_line.split()

            # для каждого слова предложения
                        for word in words:

                # если оно уже есть в словаре
                                if word in word_counter:
                    word_counter[word] += 1

                # иначе
                                else:
                    word_counter[word] = 1

    # Обязательно строковый тип
        with open(output_file, 'w') as f:
        f.write(str(word_counter))
```

```
    print(f'Данные сохранены в {output_file}')

file_processing('responses.txt', 'result.txt')
```

Output:

```
Работаю с файлом responses.txt
Данные сохранены в result.txt
```

Неизвестное количество аргументов

Когда число параметров неизвестно, то в качестве одного из аргументов добавляется параметр `*args` со звёздочкой. Тут важна именно звёздочка, а вместо `args` можно подставить любое другое имя.

Если мы передадим таким образом список, то Python «подставит» элементы списка в аргументы по их названию.

```
# Функция с *args
def sum_any(*args):
    return sum(args)

# Функция с именованными аргументами
def sum_3_numbers(a, b, c):
    return a + b + c

a = [1, 2, 3]
print(sum_3_numbers(*a) == sum_3_numbers(1, 2, 3))
```

Output:

```
True
```

```
sum_any(1, 3, 4, 10, 12)
```

Output:

```
30
```

```
a = {
    'b': 10,
    'a': 1,
    'c': 20
}
print(sum_3_numbers(**a))
```

Output:

```
31
```

Возвращаемое значение функции

Формально в языках программирования нельзя возвращать несколько значений одновременно. Однако в языке Python это в какой-то мере возможно. Если возвращается несколько значений, то они преобразовываются в кортеж (тапл).

Пример:

```
# Объявляем функцию check_and_divide с аргументами a и b
def check_and_divide(a, b):
    # в checked записываем True, если a кратно b, и False, если
    checked = True if a % b == 0 else False
    # в result записываем остаток от деления a на b
    result = a // b
    # возвращаем checked и result
    return checked, result

# печатаем результат функции
print(check_and_divide(4, 2))
```

Output:

```
(True, 2)
```

Проверяем, что действительно функция `check_and_divide(4, 2)` возвращает тапл.

```
result = check_and_divide(4, 2)
print(type(result))
```

Output:

```
<class 'tuple'>
```

Для получения нескольких значений, можно провести операцию unboxing.

Unboxing

Unboxing — распаковка кортежа (тапла). Если мы знаем, сколько значений будет содержать тапл, то мы можем через запятую создать новые названия переменных. После этого к этим переменным можно обращаться по отдельности, как к числу или строке и т.д.

Пример:

```
check_result, result = check_and_divide(a=4, b=2)
print('Результат проверки:', check_result)
print('Частное:', result)
```

Output:

```
Результат проверки: True
Частное: 2
```

Если какая-то переменная, получаемая функцией, нам не нужна, то мы можем ее «убрать». Для этого используется системная переменная «`_`»

(нижнее подчёркивание). Теперь в «`_`» будет лежать `False`. И далее в процессе разработки будет ясно, что данная переменная не будет использоваться.

```
_, result = check_and_divide(b=4, a=2)
print('Результат проверки:', _)
print('Частное:', result)
```

Output:

```
Результат проверки: False
Частное: 0
```

Ключевое слово return

- После return функция далее не выполняется
- Не обязательно возвращать что-то (эквивалентно «пустому» `return`). В таком случае возвращаемое значение — `None`

```
def simple_args(phrase):
    print(phrase)
    return 'Успех'

simple_args('Тест')
```

Output:

```
Тест
'Успех'
```

Встроенные функции

На предыдущих уроках мы уже сталкивались с функциями.

Наиболее популярной встроенной функцией является `print()`.

Пример:

```
print("Hello world")
```

Output:

```
Hello world
```

```
print(len("I am Data Analyst"))
```

Output:

```
17
```

Также есть такие популярные функции, как `len()`, `abs()`, `sum()`, `str()`, `int()` и др.

Больше встроенных функций можно найти тут.

Больше про функции тут.

> Пространство имен и область видимости

Создадим функцию `simple_access()`

```
def simple_access():
    access_test = 10
    print(access_test)
```

Output:

```
10
```

Если мы попробуем обратиться к переменной `access_test`, то получим ошибку, так как она не была возвращена внутри функции.

```
access_test
```

Output:

```
---------------------------------------------------------------
NameError                                 Traceback (most recent
<ipython-input-30-26541ee0b68c> in <module>
----> 1 access_test
```

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

А если мы вне функции создадим переменную, а потом ее напечатаем внутри функции, то это сработает.

```
username = 'Alex'
def second_access():
    print(username)
second_access()
```

Output:

```
Alex
```

Это происходит потому, что Python создает пространство имен, которые вложены друг в друга.

Фактически все, что программа Python создает или с чем работает, является объектом. Для кода, который вы написали, создается файл, в котором обозначены все объекты, с которыми работает данный код. Они включают в себя все переменные, функции, а также встроенные имена, к которым можно обращаться всегда, не создавая их заново. Пример встроенного имени — функция `print()`.

Подробнее об областях видимости тут.

> Генераторы

Генератор в Python (generator function) — это функция, внутри которой обычно присутствует какой-нибудь цикл по элементам (итератор). Возвращает объект типа generator.

То есть генератор создаётся по принципу обычной функции, но вместо `return` там находится инструкция `yield`.

`return` исполняется всегда последним в вызове функции, в то время как `yield` временно приостанавливает исполнение, сохраняет состояние и

затем может продолжить работу позже.

Генератор имеет вид:

```
def generator_name(arg1, arg2,...):
    некоторый цикл:
        yield result
```

где:

- `def` — ключевое слово, которое указывает на то, что дальше начинается функция
- `generator_name` — название генератора
- `args1`, `args2`... `argsn` — названия аргументов, которые будут использоваться внутри функции

После двоеточия следует некоторый цикл, который перебирает какие-нибудь параметры

- `yield` — инструкция внутри цикла, которая временно приостанавливает исполнение, сохраняет состояние и затем может продолжить работу цикла.

Важно: инструкция `yield` является частью цикла

Пример:

Генератор, который возвращает квадраты чисел от 0 до n:

```
def square_gen(n): # объявляем функцию square_gen и вводим аргум
    print('Start') # печатаем момент старта последующего цикла
        for e in range(n):
        yield e ** 2 # возвращаем квадрат числа e

gen = square_gen(10)
print(gen)
```

Если мы попытаемся вызвать функцию и напечатать результат, то получаем невыполненный объект-генератор.

Output:

```
<generator object square_gen at 0x7f5ed42fc190>
```

Важно: вызов генератора не выполняет его. Мы это знаем, так как строка `Start` не напечаталась. Вместо этого генератор возвращает объект-генератор, который используется для управления выполнением.

Вывод элементов генератора

`next()` — функция, которая говорит генератору сделать итерацию

```
print(next(gen))
```

Output:

```
Start
0
```

Вывелась строка `Start`, так как генератор начал работать и вывелся первый элемент. Если мы снова применим функцию `next()` к генератору `gen`, то выведется уже квадрат единицы (1), при следующем обращении к функции выведется квадрат двойки (4) и так далее, пока мы не дойдем до последнего элемента итератора.

Помимо функции `next()` элементы генератора можно вывести при помощи цикла, т.е. по генератору можно итерироваться:

```
for e in gen:
    print(e)
```

Output:

```
1
4
```

```
9
16
25
36
49
64
81
```

Важно: в данном случае выводятся квадраты чисел начиная с единицы, потому что нулевой элемент в самой первой итерации уже прошёл.

Что надо знать о генераторах

1. Генератор позволяет экономить память, так как в определённый момент времени он хранит только одно значение.
2. Генераторы одноразовые. Если мы проитерировались по генератору, и в коде программы понадобится ещё раз использовать такой генератор, его нужно будет сделать заново (`gen = square_gen(10)`).

Выражение-генератор

Генераторы можно задавать как в виде функций, так и в виде выражений. Визуально это похоже на comprehensions, только разница в том, что мы используем круглые скобки.

Выражение-генератор, который возвращает квадраты чисел от 0 до n, будет выглядеть следующим образом:

```
gen = (e**2 for e in range(10))

# печатаем генератор
print(gen)
```

Output:

```
<generator object <genexpr> at 0x7f77cec92c00>
```

```
print(next(gen))
```

Output:

```
0
```

Итерируемся по генератору:

```
for e in gen:
    print(e)
```

Output:

```
1
4
9
16
25
36
49
64
81
```

> lambda-функции

Бывают случаи, когда нам необходима функция в одну строчку. Например, функция делает какое-нибудь простое действие, и вместо того чтобы писать `def` и несколько строчек кода, можно воспользоваться lambda-функцией.

lambda-функция имеет вид:

```
variable = lambda arg1, arg2,... : код
```

где:

- `variable` — переменная, в которой будет храниться функция;
- `lambda` — ключевое слово, чтобы Python понял, что дальше будет описано действие функции;
- `arg1`, `arg2`, ... — аргументы функции;
- код — код функции, идёт после двоеточия.

Пример:

lambda-функция, которая говорит, чётное число или нет:

```
lam = lambda x: 'Чётное' if x % 2 == 0 else 'Нечётное'

print(type(lam))
```

Output:

```
<type 'function'>
```

Видим, что `lam` действительно является функцией.

```
print(lam(5))
```

Вызываем lambda-функцию `lam` от числа 5, функция возвращает ответ, что число нечётное.

Output:

```
Нечётное
```

> Функции map(), filter(), reduce(), sorted()

Существует парадигма разработки, которая называется функциональное программирование. Это значит, что передаваемые аргументы являются единственными факторами, которые могут определить результат. Такие функции могут принимать любую другую функцию в качестве аргумента и могут быть переданы другим функциям в качестве аргументов.

Функция map()

`map(func, iterable)` — принимает два аргумента:

- первый — функция (это может быть как lambda-функция, так и функция, заданная через `def`);
- второй — какая-нибудь структура, по которой можно итерироваться (список, сет, тапл, словарь или строка).

Под капотом эта функция запускает цикл `for` и для каждого элемента структуры применяет функцию, которую мы передали в качестве первого аргумента.

Важно: структур, по которым мы итерируемся, может быть несколько по аналогии с функцией `zip()`.

Документация map()

Документация zip()

Пример

Задача — создать список квадратов чисел.

При помощи цикла `for` код можно написать следующим образом:

```
squared_list = [] # создаем пустой список, куда будем складыват

for e in range(10): # цикл for от 0 до 10 , не включая 10
    squared_list.append(e ** 2) # добавляем в список квадрат чи

print(squared_list) # печатаем итоговый список
```

Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Для этой задачи воспользуемся функцией `map()`, где первым аргументом будет lambda-функция, которая возводит число в квадрат, а вторым аргументом — список чисел от 0 до 9:

```
squared_list = map(lambda e: e ** 2, range(10))

print(list(squared_list))
```

Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Функция filter()

`filter(func, iterable)` — используется для фильтрации элементов.

Функция, которая передается первым аргументом, возвращает логическое значение: либо `True`, либо `False`. Тем самым она вернет только те элементы, для которых получилось значение `True`.

Документация filter()

Пример:

Задача — сохранить в список все нечётные числа из данного списка.

При помощи цикла `for`:

```
odd_list = [] # создаем пустой список

for e in range(10): # цикл for от 0 до 10 , не включая 10
    if e % 2 == 1: # "если остаток деления числа на 2 равен одно
        odd_list.append(e) # то добавь в список число

print(odd_list) # печатаем итоговый список нечетных чисел
```

Output:

```
[1, 3, 5, 7, 9]
```

Теперь воспользуемся функцией `filter()`, где первым аргументом будет lambda-функция, которая проверяет число на нечётность, а вторым аргументом — список чисел от 0 до 9.

К каждому числу от 0 до 9 применяется lambda-функция, если число нечётное, то функция возвращает значение `True` и записывает число, для которого функция оказалась истиной.

```
odd_list = filter(lambda e: e % 2 == 1, range(10))

print(list(odd_list))
```

Output:

```
[1, 3, 5, 7, 9]
```

Функция reduce()

`reduce(func, iterable)` — сокращает структуру до одного агрегированного значения.

Важно: начиная с третьей версии Python эту функцию нужно импортировать дополнительно.

Функция, применяемая для агрегации структуры (первый аргумент в `reduce`), должна принимать 2 аргумента: первый — куда складываем аккумулируемый результат, второй — элемент структуры, по которой происходит итерация.

Документация reduce()

Пример:

```
from functools import reduce # импортируем функцию reduce
```

```
a = [1, 2, 3, 4, 5] # заранее создаем список
result = reduce(lambda acc, e: acc + e ** 2, a)

print(result)
```

Output:

```
55
```

lambda-функция принимает два аргумента: аргумент `acc`, куда складываются все значения, и `e` — это элемент списка `a`.

Заранее не нужно присваивать значение переменной `acc`, python это делает за нас «под капотом».

Чтобы стало понятнее, откуда взялось число 55, рассмотрим аналогию выражения выше в цикле `for`:

```
acc = 0 # переменная куда складываются все значения

for i in a: # для каждого числа в списке a
    acc += i**2 # посчитай квадрат числа, сложи с тем, что лежит

print(acc) # печатаем итоговое число
```

Output:

```
55
```

То есть сначала `acc` пустой, и туда добавляется квадрат единицы. Значит, теперь в `acc` единица. Затем итератор переходит к двойке из списка `a`, возводит в квадрат (получаем 4) и складывает с тем, что в переменной `acc` (1+4=5), затем число 5 сохраняется в переменную `acc`, итератор переходит к третьему элементу списка `a` — числу 3, и проделывает все те же операции. И так до конца списка.

Функция sorted()

`sorted(iterable, key=None, reverse = False)` — используется для сортировки элементов итерируемого объекта.

Функция `sorted()` может принимать три аргумента: первый (обязательный) `iterable`, то есть итерируемый объект, элементы которого нужно отсортировать; второй `key` (необязательный) — функция, которая принимает элемент и возвращает значение, по которому будет производиться сортировка (это может быть как lambda-функция, так и функция, заданная через `def`); третий `reverse` (необязательный) — указывает, нужно ли сортировать в обратном порядке (по убыванию).

При вызове функции `sorted()` элементы итерируемого объекта будут упорядочены в порядке возрастания по умолчанию (значение `reverse` по умолчанию равно `False`).

Документация sorted()

Пример:

```
# объявляем функцию sort_students_by_age
def sort_students_by_age(students):
    # указываем в качестве итерируемого объекта список students,
    # в key передаем lambda-функцию;
    # reverse по умолчанию равен False
    sorted_students = sorted(students, key=lambda x: x[1])
    # возвращаем отсортированный список
    return sorted_students
```

```
# создаем список students
students = [('Alice', 20, 3.5), ('Bob', 22, 4.0), ('Charlie', 19

# Применяем функцию sort_students_by_age к списку students
sorted_students = sort_students_by_age(students)
print(sorted_students)
```

Output:

```
[('Charlie', 19, 3.2), ('Alice', 20, 3.5), ('Bob', 22, 4.0)]
```

Таким образом, мы написали код, которые сортирует список со студентами по возрасту в порядке возрастания.

> Дополнительные материалы

1. Функции в Python
2. Аргументы функции
3. Больше о lambda-функциях тут и тут
4. Какие встроенные функции нужно знать
5. Топ 3 функций в Python
6. Как сделать функции еще лучше