

Функции. Работа с файлами

Цель занятия

После освоения темы вы:

- узнаете назначение функций в языках программирования;
- узнаете особенности работы с функциями в Python;
- сможете написать собственную функцию на языке Python;
- сможете читать и записывать данные из файла;
- сможете создать программу по описанию задачи на естественном языке с использованием функций языка Python.

План занятия

1. [Именные функции, инструкция `def`.](#)
2. [Возврат значений из функции.](#)
3. [Множественное присваивание, упаковка и распаковка значений.](#)
4. [Аргументы по умолчанию и именованные аргументы.](#)
5. [Инструкция `pass\(\)`. Согласованность аргументов.](#)
6. [Функция как объект. Функции высших порядков.](#)
7. [Лямбда-функция.](#)
8. [Принципы работы с файлами на Python.](#)
9. [Разбор задач на работу с файлами.](#)
10. [Правила записи кода PEP 8.](#)

Используемые термины

Функция в программировании — выделенный участок кода, к которому можно обратиться из другого места программы.

Конспект занятия

1. Именные функции, инструкция def

Рассмотрим пример программы, определяющей среднюю температуру за год.

```
t = [-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3]

s = 0

mm = 1000

mx = -1000

for e in t:

    s += e

    if e < mm:

        mm = e

    if e > mx:

        mx = e

print(s / len(t))

print(mm)

print(mx)
```

В первой строке определяется список `t = [-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3]`. Помимо среднего значения программа определяет минимальное (`mm`) и максимальное значения (`mx`) температур. Переменные `mm` и `mx` заведомо больше и заведомо ниже значений в исходном списке. Программа в цикле с помощью условного оператора `if` определяет значения минимума `mm` и максимума `mx`.

В рассмотренном примере поиск среднего значения, минимума и максимума выполняется вручную.

В самом языке Python уже есть встроенные функции нахождения минимального, максимального значения, а также суммы элементов списка. Поэтому код примера выше можно переписать:

```
temperatures = [-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3]

average_temperature = sum(temperatures) / len(temperatures)

print(average_temperature)

print(min(temperatures))

print(max(temperatures))
```

В разобранным примере используются функции:

- `min()` — определяет минимальное значение;
- `max()` — определяет максимальное значение;
- `sum()` — суммирует элементы.

Таким образом, с использованием функций код становится короче. Кроме того, функции повышают удобство использования программ.

Функция в программировании — выделенный участок кода, к которому можно обратиться из другого места программы. Функции выполняют некоторые «простые» действия.

Важно! Взаимодействие функции и остальной программы происходит только через глобальные переменные и изменяемые контейнеры — например, списки.

Функция в Python может возвращать результат работы. Можно считать, что функция — это преобразование аргументов функции в возвращаемое значение.

Синтаксис определения функции:

```
def <имя_функции> ([аргументы]) :

    <тело_функции>
```

Аргументы передаются в скобках после названия функции. Аргументов у функции может не быть. Тело функции записывается с отступом и может состоять из одной и более строк.

Синтаксис вызова функции:

```
<имя_функции> ([аргументы])
```

При вызове функции указываются конкретные значения аргументов, либо переменные, которые являются конечно-значимыми на данную строку выполнения программы.

Рассмотрим пример простейшей функции `simple_greetings()`, которая выводит приветствие «Привет, username!». Функция не содержит аргументов и конкретного возвращаемого значения:

```
def simple_greetings():  
    print('Привет, username!')
```

Тело функции состоит из одной строки — `print('Привет, username!')`.

После вызова функции `simple_greetings()` выполняется тело функции и осуществляется вывод строки, записанной в `print()` в консоль.

Важно! Порядок вызова имеет большое значение: функция должна быть определена до того, как ее вызывают. В противном случае будет выведено сообщение об ошибке.

Рассмотрим примеры, когда использование функции будет удобно и целесообразно. В этом коде выполняется повторение одних и тех же действий — ввод имени и печать приветствия:

```
print('Как тебя зовут?')  
  
name_1 = input()  
  
print('Привет', name_1)  
  
print('А тебя?')  
  
name_2 = input()  
  
print('Привет', name_2)  
  
print('А твоего пса?')  
  
name_3 = input()  
  
print('Привет', name_3)
```

Код можно упростить и структурировать, создав функцию `greet()` и вызывая ее в нужных местах программы. В рассмотренном примере функция `greet()`

вызывается три раза. Таким образом, применение функций уменьшает дублирование кода и делает его более понятным:

```
def greet():  
    name=input()  
    print('Привет, ', name)  
  
print('Как тебя зовут?')  
  
greet()  
  
print('А тебя?')  
  
greet()  
  
print('А твоего пса?')  
  
greet()
```

Определять типы аргументов при объявлении функции в Python не требуется. Аргументы функции могут иметь разный тип, главное, чтобы корректно выполнялись действия над описанными переменными в теле функции.

Рассмотрим пример функции вывода элементов списка через цикл `for`:

```
def print_array(array):  
    for element in array:  
        print(element)  
  
print_array(['Hello', 'world'])  
  
print_array([123, 456, 789])
```

Функция `print_array()` вызывается дважды. В первом случае в качестве аргументов функции используются строки `print_array(['Hello', 'world'])`, во втором — целые числа `print_array([123, 456, 789])`. В обоих случаях описанная функция `print_array()` работает корректно.

При вызове функции в качестве аргумента может быть передан результат некоторой операции, например:

```
print_array(['Hello'] + ['world'])
```

В этом случае нужно определить порядок действий:

- 1) Выполняется операция над аргументами, результат которой становится аргументом функции.
- 2) Выполняется вызов функции

```
print_array(['Hello', 'world']).
```

Рассмотрим пример трех функций:

```
def print_hello(arg_1, arg_2):  
    print('hello')  
  
def print_comrade():  
    print('comrade')  
  
def print_petrov():  
    print('Petrov')
```

Далее мы будем в качестве аргументов при вызове первой функции использовать вторую и третью функцию:

```
print_hello(print_comrade(), print_petrov())
```

Результатом будет следующий вывод в консоль:

```
comrade  
  
Petrov  
  
hello
```

То есть, сначала вызываются функции, указанные как аргументы `print_hello()`: `print_comrade()` и затем `print_petrov()`. Сама функция `print_hello()` выполняется в последнюю очередь.

2. Возврат значений из функции

Рассмотрим пример возврата значения из функции. Для этого используется команда `return`.

В примере определена функция `double_it()`, которая удваивает переданное ей значение:

```
def double_it(x):  
    return x*2  
  
radius = 3  
  
length = double_it(3.14) *radius
```

Как только выполнение доходит до инструкции `return`, выполнение функции завершается, и интерпретатор возвращается к месту, где функция была вызвана.

В примере описан вызов функции с аргументом 3,14. После вызова функции будет вычислено `length = 6.28*radius`. То есть значение, переданное `return`, является результатом вычисления функции. Таким образом, написанная программа позволяет вычислить примерное значение длины окружности.

Результат функции можно использовать в других вычислениях — например, присвоить другой переменной `double_pi`. Далее значение переменной `double_pi` выводится на экран и используется в выражении для определения длины окружности:

```
double_pi = double_it(3.14)  
  
print(double_pi)  
  
length = double_pi*radius
```

Удобная техника – использование локальной переменной `result`. Данный прием не является стандартом, но его часто можно встретить на практике.

Пример ниже рассчитывает сумму элементов списка. Результат функции сохраняется в переменную `result` и возвращается с помощью `return`:

```
def my_sum(arr):  
    result=0  
  
    for element in arr:  
        result += element  
  
    return result  
  
print(my_sum([1, 2, 3, 4]))
```

Важно! Переменные, определенные внутри функции, являются **локальными**. Эти переменные существуют и имеют определенное значение при вызове функции.

После выхода из функции `my_sum()` переменной `result` не существует. Но само значение результата никуда не исчезает и может получить новое имя снаружи:

```
s = my_sum([1, 2, 3, 89])          # => 95
```

В языке Python допускаются множественные точки возврата из функции. Рассмотрим функцию определения модуля числа:

```
def my_abs(x):  
    if x >= 0:  
        result = x  
    else:  
        result = -x  
    return result
```

Функцию `my_abs()` можно преобразовать, добавив оператор `return` в каждую ветвь условного оператора:

```
def my_abs(x):  
    if x >= 0:  
        result = x  
        return result  
    else:  
        result = -x  
        return result
```

Программу можно сделать еще короче:

```
def my_abs(x):  
    if x >= 0:  
        return x
```



```
else:  
    return -x
```

В функциях допускается две и более точек возврата. Но нужно помнить, что как только в ходе выполнения появляется `return`, функция завершает свою работу.

Команда `return` завершает выполнение функции из любого уровня вложенности. Пример показывает функцию для работы с матрицей. В зависимости от условий функция будет возвращать значение `True` или `False`:

```
def matrix_has_close_value(matrix, value, eps):  
    found = False  
    for row in matrix:  
        for cell in row:  
            if abs(cell-value) <= eps:  
                found = True  
                break  
        if found:  
            break  
    if found:  
        return True  
    else:  
        return False
```

Программу можно сократить. При этом оператор `return` сработает с любого уровня вложенности:

```
def matrix_has_close_value(matrix, value, eps):  
    for row in matrix:  
        for cell in row:  
            if abs(cell-value) <= eps:
```

```
        return True

    return False
```

Функции без команды `return` имеют определенное возвращаемое значение. Если работа функции завершается без выполнения `return` (закончились инструкции), то функция возвращает `None`. Команда `return` без аргумента также возвращает `None` и мгновенно завершает работу функции.

В Python просто реализуется возврат нескольких значений. Возвращаемые значения записываются через запятую после `return`:

```
def get_coordinates():

    return 1, 2

print(get_coordinates())    # => (1, 2)
```

На выходе функции, возвращающей несколько значений, получается кортеж, состоящий из этих значений. То есть команда `return 1, 2` практически идентична команде возврата кортежа с этими значениями `return (1, 2)`.

Полученный кортеж можно записать в одну переменную `result = get_coordinates()`, а можно воспользоваться множественным присваиванием и разделить:

```
x, y = get_coordinates()
```

Тогда в переменной `x` будет находиться значение 1, а в переменной `y` значение 2.

Важно! Всегда возвращайте однотипные данные в разных точках возврата. Иначе программы будут «хрупкими».

Рассмотрим пример функции:

```
def get_coordinates(index):

    if index % 2 == 0:

        return 1.5, 2.5

    else:

        return 1.5, 2.5, 0
```

При вызове функции в виде `x, y = get_coordinates()` программа «сломается» на нечетных индексах. При вызове `x, y, z = get_coordinates()` программа «сломается» на четных индексах. Чтобы подобных проблем не возникало, используйте одинаковое количество возвращаемых значений для любого сценария выполнения программы.

3. Множественное присваивание, упаковка и распаковка значений

Рассмотрим пример, когда функция возвращает два значения:

```
def get_coordinates():  
    return 1, 2  
  
x, y = get_coordinates()  
  
print(x)                # => 1  
print(y)                # => 2
```

Функция возвращает значения в виде кортежа. В примере кортеж будет содержать два значения (1, 2). Результат функции, возвращающей несколько значений, можно разложить по отдельным переменным:

```
x, y = get_coordinates()
```

На отдельные составляющие можно разложить не только кортеж, но и список:

```
x, y = [1.5, 2.5]  
  
print(x)                # => 1.5  
print(y)                # => 2.5
```

Значения отдельных составляющих будут иметь тот же тип, что и элементы исходного кортежа или списка.

Одной переменной можно одновременно присвоить два значения. В этом случае значения «по умолчанию» запаковывают в кортеж:

```
z = 1.5, 2.5                # z = (1.5, 2.5)
```

Общие правила:

- если справа от знака равенства больше одного значения, они запаковываются в кортеж;

- если слева от знака равенства больше одной переменной, то присваиваемое значение распаковывается по отдельным переменным;
- запаковывание может комбинироваться с распаковыванием.

Рассмотрим пример запаковывания. В некоторых случаях перед именем переменной может находиться символ звездочки «*». Если перед именем переменной стоит звездочка, то все «лишние» значения запаковываются в список и записываются в эту переменную:

```
x, y, *rest = 1, 2, 3, 4, 5, 6
print(x)                # => 1
print(y)                # => 2
print(rest)             # => [3, 4, 5, 6]
```

В примере первые два значения 1 и 2 помещаются в переменные `x` и `y` соответственно, остальные переменные запаковываются в список `rest`.

Переменная со звёздочкой всегда будет списком, даже если она будет содержать один или ни одного элемента.

В этом примере переменная `rest` содержит один элемент:

```
x, y, *rest = 1, 2, 3
print(rest)    # => [3]
```

Значений для переменной `rest` может «не хватить», в этом случае создается пустой список:

```
x, y, *rest = 1, 2
print(rest)    # => []
```

Если справа количество переменных без звездочки больше, чем количество значений слева, будет выведено сообщение об ошибке:

```
x, y, z, *rest = 1, 2          # Ошибка выполнения
```

Переменная со звездочкой может стоять на любом месте: в конце списка, в начале, в середине.

Если переменная со звездочкой находится в начале, то сначала определяются значения для всех остальных переменных. Значения распределяются с конца. Оставшиеся после присваивания обычным переменным значения передаются переменной со звездочкой.

Если переменная со звездочкой находится посередине, количество помещаемых в нее элементов определяется с учетом количества переменных без звездочки, которые находятся слева и справа.

Важно! Может быть не больше одной переменной со звездочкой.

В рассматриваемом примере переменная со звездочкой `*names` находится в начале. После следует переменная `surname`. Это значит, что все значения кроме последнего будут записаны в переменную со звездочкой `*names`:

```
*names, surname = 'Анна Мария Луиза Медичи'.split()
print(names)           # => ['Анна', 'Мария', 'Луиза']
print(surname)         # => Медичи
```

При помощи скобок можно распаковывать даже вложенные кортежи и списки:

```
a, (b, c), d = [1, [2, 3], 4]
```

Если вы хотите распаковать единственное значение в кортеже, то после имени переменной должна идти запятая:

```
a = (1,)
b, = (1,)
print(a)           # => (1,)
print(b)           # => 1
```

В рассматриваемом примере переменная `a` является кортежем, а переменная `b` будет содержать единственное значение из кортежа.

Переменные со звездочкой используются и при написании функции. Звёздочка в списке аргументов позволяет передавать в функцию произвольное число дополнительных аргументов:

```
def product(first, *rest):
```

```
result = first

for value in rest:
    result *= value

return result

product (2, 3, 5, 7)          # => 210
```

В рассматриваемом примере функция `product()` имеет два аргумента `first` и `*rest`. Второй аргумент дополнительно имеет знак звездочки. Таким образом, при вызове рассматриваемой функции, ей необходимо передать как минимум один аргумент. Все остальные значения попадают в `*rest`.

Рассматриваемая функция `product()` определяет произведения элементов списка. Количество элементов списка должно быть больше одного.

Звездочка в передаваемых аргументах позволяет распаковать список или кортеж. Ниже приведен пример списка, состоящего из строк. Команда `print(arr)` выводит список. При этом выводятся кавычки для выделения отдельных элементов списка и квадратные скобки, показывающие, что выводится именно список.

```
arr = ['cd', 'ef', 'gh']

# Здесь мы передаем просто список как один аргумент

print(arr) # => ['cd', 'ef', 'gh']
```

Если перед именем переменной `arr` добавить звездочку, то элементы списка будут выведены как три отдельных аргумента без дополнительных символов. Такой вывод аналогичен тому, как будто бы осуществлялся вывод трех отдельных строковых переменных.

```
# А здесь мы раскрыли список

# и функция print получила три отдельных аргумента

print(*arr) # => cd ef gh

# Это аналогично вызову

print('cd', 'ef', 'gh')          # => cdefgh
```

При выводе переменная со звездочкой может сочетаться с любыми другими значениями:

```
print('ab', *arr, 'yz')          # => ab cd ef gh yz
```

Также можно использовать при выводе несколько аргументов со звездочкой:

```
print(*arr, *arr)                # => cd ef gh cd ef gh
```

4. Аргументы по умолчанию и именованные аргументы

Рассмотрим функцию `int()`, которая возвращает целочисленное значение переданного аргумента. Функция `int()` имеет два аргумента, второй аргумент функции `int()` указывает основание системы счисления переданного аргумента. По умолчанию он равен 10, и его зачастую не указывают.

Поэтому две строки такого кода вернут одинаковый результат:

```
int('101')                      # => 101
```

```
int('101', 10)                  # => 101
```

В этом примере перевод осуществляется из двоичной системы счисления:

```
int('101', 2)                   # => 5
```

При написании функции также можно использовать аргументы по умолчанию:

```
def make_burger(typeOfMeat, withOnion = False, withTomato = True):  
    print('Булочка')  
    if withOnion:  
        print('Луковые колечки')  
    if withTomato:  
        print('Ломтик помидора')  
    print('Котлета из', typeOfMeat)  
    print('Булочка')
```

В рассматриваемой функции есть три аргумента, последние два уже определены в самом заголовке функции `withOnion = False, withTomato = True`.

Это означает, что при вызове функции обязательно необходимо указать только первый аргумент.

Примеры вызова функции:

```
make_burger('свинина') # бургер из свинины без лука, с помидорами
make_burger('свинина', True) # с луком и помидорами
make_burger('свинина', True, False) # с луком, без помидоров
```

В примерах выше функции с аргументами вызывались в том порядке, в котором они были определены при ее объявлении. Чтобы не приходилось запоминать неважные детали, при вызове функции аргументы можно передавать не по порядку, а по имени:

```
matrix_has_value(matrix=[[1, 2, 3], [4, 5, 6]], value=7)
```

или:

```
matrix_has_value(value=7, matrix=[[1, 2, 3], [4, 5, 6]])
```

С использованием именованных аргументов в определении функции ничего менять не требуется. А при вызове вы можете определять только те значения по умолчанию, которые требуется. Если вернуться к примеру функции `make_burger()`, рассмотренному выше, ее вызов может быть записан как

```
make_burger(typeOfMeat = 'говядина', withTomato = False)
```

5. Инструкция `pass()`. Согласованность аргументов

Функция `pass()` представляет собой заглушку на случай, если необходимо запустить ту или иную программу без определения функции. То есть `pass()` используется, когда вам нужно сказать «ничего не делать», а синтаксис требует наличия команды.

Например, такая конструкция ничего не возвращает, но и ничего не делает:

```
if game_over:
    pass                # ToDo: написать вывод итогового результата
```

Функция `pass()` позволяет создавать функцию «ничегонеделанья».

В примере рассматривается функция `nop()`:

```
def nop():
    pass

nop()
```


Такая функция работает. Но неудобно, что мы не можем передать в функцию аргументы. Функцию `nop()` можно переписать, используя аргумент со звездочкой:

```
def nop(*rest):  
    pass
```

Функция также не будет ничего возвращать, вне зависимости от того, что будет ей передано:

```
nop()  
  
nop("Любое", "сказанное", "вами слово", "будет проигнорировано")  
  
nop(100500, None, [1, 2, 3, 4, 5])
```

Теперь можно заменить любую функцию на `nop()`, не меняя аргументы.

«Сломать» функцию `nop()` можно, вызвав ее следующим образом:

```
print(<???)          # Работает  
  
nop(<???)            # Выдаёт ошибку
```

Функция `print()` не принимает именованные аргументы:

```
print(1, 2, 3, sep=', ')    # Работает  
  
nop(1, 2, 3, sep=', ')     # Выдаёт ошибку
```

Ошибку можно исправить, используя именованные аргументы и определив значение по умолчанию `None`:

```
def nop(*rest, sep=None, end=None):  
    pass
```

Рассмотрим использование аргумента с двумя звездочками. Аргумент с одной звёздочкой «захватывает» все позиционные параметры, с двумя звездочками — все именованные. Переменная со звездочкой записывает значения в список, переменная с двумя звездочками — в словарь:

```
def nop(*rest, **kwargs):  
    pass  
  
nop(1, [2, 3], debug=True, file="debug.log")
```

В переменную `*rest` будут записаны значения первых двух аргументов — получим список. В переменную `**kwargs` запишутся вторые два аргумента, но уже мы получим словарь. Ключами будут названия аргументов, а значения — соответствующие им значения. То есть будет создан словарь `{'debug': True, 'file': 'debug.log'}`.

В программе показан пример использования различных видов аргументов:

```
def profile(name, surname, city, *children, **additional_info):  
    print("Имя:", name)  
    print("Фамилия:", surname)  
    print("Город проживания:", city)  
    if len(children) > 0:  
        print("Дети:", ", ".join(children))  
    print(additional_info)  
  
profile("Сергей", "Михалков", "Москва", "Никита Михалков",  
"Андрей Кончаловский", occupation="writer", diedIn=2009)
```

После выполнения функции будет получен результат:

Имя: Сергей

Фамилия: Михалков

Город проживания: Москва

Дети: Никита Михалков, Андрей Кончаловский

`{'occupation': 'writer', 'diedIn': 2009}`

С помощью аргументов с одной и двумя звездочками можно передать любой список аргументов другой функции неизменным.

В рассматриваемом ниже примере функция `perforated_print()` выводит значение переданных ей аргументов и с помощью команды `print('- '*20)` подводит черту из двадцати знаков «-». Функцию можно вызывать с различным количеством аргументов, причем среди них могут встречаться именованные аргументы:

```
def perforated_print(*args, **kwargs):  
    print(*args, **kwargs)  
    print('-'*20)  
  
perforated_print('Теперь текст выводится с линией перфорации.')  
  
perforated_print('И', 'можно', 'использовать', 'любые', 'опции',  
end=':\n')  
  
perforated_print('end', 'sep', 'прочие', sep=', ', end='!\n')
```

6. Функция как объект. Функции высших порядков

В языке Python функция является таким же объектом, как число, строка или список. Единственное ее отличие — функцию необходимо вызвать, написав скобки. Чтобы получить функцию как объект, достаточно написать имя функции без скобок.

Рассмотрим пример, в котором получим объект функции ввода `input()`. Если использовать конструкцию `print(input)`, то получим информацию непосредственно о функции `input()` — `<built-in function input>`.

Если объект можно получить, его можно записать в переменную, а затем использовать:

```
vyvod = print  
vyvod('Privet mir!')
```

Рассмотрим две функции для печати списков. Первая функция `print_boxed()` выводит список в виде таблицы, вторая функция `print_simple()` выводит список элементов через запятую:

```
def print_boxed(arr):  
    arr_stringified = [str(element) for element in arr]  
    mid = ' | '.join(arr_stringified)  
    bar = '-' * (2 + len(mid))  
    print(' ' + bar + ' ')  
    print('| ' + mid + ' |')
```

```
print(' ' + bar + ' ')\n\ndef print_simple(arr):\n    arr_stringified = [str(element) for element in arr]\n    print(', '.join(arr_stringified))
```

Теперь мы можем выбирать стиль форматирования для всей программы:

```
formatting = 'boxed'\n\nif formatting == 'boxed':\n    print_formatted = print_boxed\nelse:\n    print_formatted = print_simple\n\n# Далее в программе можно использовать print_formatted повсюду\nprint_formatted([1, 1, 2, 3, 5, 8, 13, 21])\nprint_formatted([1, 2, 4, 8, 16, 32, 64, 128])\nprint_formatted(['abc', 'def', 'ghi'])
```

По умолчанию стиль форматирования выбран как `boxed`. Если в операторе `if` условие выполняется, то вызывается функция `print_boxed`, если нет — `print_simple`.

Существуют так называемые функции высшего порядка — это функции, которые могут принимать другие функции как аргумент или возвращать другие функции как результат вычислений.

Рассмотрим функцию `is_word_long()`, которой в качестве аргумента поступает строка. Она возвращает `True` или `False` в зависимости от того, превышает ли длина строки 6 символов. Примером функции высшего порядка выступает функция `filter()` — выполняет отбор элементов по критерию. В примере списком является `words`, в качестве аргумента выступает функция `is_word_long()`, которая определяет критерий отбора.

```
def is_word_long(word):  
    return len(word) > 6  
  
words = ['В', 'новом', 'списке', 'останутся', 'только', 'длинные',  
        'слова']  
  
# аргументы функции filter: критерий и список  
for word in filter(is_word_long, words):  
    print(word)
```

Результатом выполнения примера будет вывод в консоль:

```
останутся  
  
длинные
```

Функция `filter()` — пример функции высшего порядка. Функция `filter()` возвращает не список, а специальный итерируемый (перебираемый циклом `for`) объект.

Если необходимо перевести объект в список, используется функция `list()`:

```
long_words = list(filter(is_word_long, words))
```

7. Лямбда-функция

Часто при программировании требуются очень простые функции — например, критерий отбора. Они используются единожды, и им не нужно даже имя. Такие «безымянные» функции называются лямбда-функциями.

Синтаксис лямбда-функции:

```
lambda <аргументы>: <выражение>
```

Рассмотрим пример функции, возвращающей `True` или `False` в зависимости от того, длиннее слово 6 символов или нет:

```
lambda word: len(word) > 6
```

Аргументом лямбда-функции является `word`, выражением — `len(word) > 6`.

Рассмотренную лямбда-функцию можно использовать как аргумент функции `filter()`:

```
long_words = list(filter(lambda word: len(word) > 6, words))
```

Лямбда-функция — полноценная, хоть и безымянная функция. Ее можно записать в переменную и использовать.

В переменную `add` записана лямбда-функция, вычисляющая сумму двух аргументов:

```
add = lambda x, y: x + y
```

Далее `add` используется как объект. Аргументы записываются в скобках:

```
add(3, 5)                # => 8
```

Аргументом может служить и сама функция:

```
add(1, add(2, 3))        # => 6
```

Лямбда-функция может содержать только одно выражение, при этом ему необязательно зависеть от аргумента. Например, критерий, чтобы выбрать все элементы списка:

```
lambda x: True
```

Критерий, записанный с помощью лямбда-функции, теперь можно использовать в другой функции:

```
def print_some_primes(criterion):
    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
    for number in filter(criterion, primes):
        print(number)

print_some_primes(lambda x: True)

# => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Также, если необходимо, мы можем написать свою функцию `filter()`.

В примере показана функция `simple_filter()`, в качестве аргументов поступают функция `criterion()` и список `arr`. При этом критерий определяется через лямбда-функцию `lambda x: x % 12 == 7`, то есть все числа, которые при делении нацело на 12 дают остаток 7. Список `arr` определяется через функцию `range()`:

```
def simple_filter(criterion, arr):
```

```
result = []

for element in arr:

    if criterion(element):

        result.append(element)

    return result

simple_filter(lambda x: x % 12 == 7, range(1, 100))
```

После выполнения программы получаем результат:

```
[7, 19, 31, 43, 55, 67, 79, 91]
```

Еще одна функция высшего порядка — функция `map()`.

Функция `map()` берет функцию для преобразования одного элемента и список и выполняет преобразование всех элементов списка.

В примере записан итерируемый объект — список, задаваемый с помощью `range()`. С помощью лямбда-функции задается действие, которое необходимо сделать с каждым элементом — каждый элемент необходимо возвести в квадрат:

```
list(map(lambda x: x ** 2, range(1, 10)))
```

Таким образом, получаем следующий результат:

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

Заменой функции `map()` могут служить списочные выражения. Записанную ранее конструкцию можно переписать как

```
[x ** 2 for x in range(10)]
```

Замена функции `filter()` при помощи списочных выражений:

```
[x for x in range(10) if x % 2 == 0 and x % 3 != 0]
```

Рассмотрим пример комбинации функций `filter()` и `map()`. В результате останутся слова длиннее 6 символов, каждое слово будет переведено в верхний регистр:

```
words=['В', 'новом', 'списке', 'останутся', 'только', 'длинные',  
'слова']
```

```
long_words = list(map(lambda word: word.upper(), filter(lambda word: len(word) > 6, words)))
```

Код можно записать с использованием списочных выражений:

```
long_words = [word.upper() for word in words if len(word) > 6]
```

Результат обоих примеров будет одним и тем же: ['ОСТАНУТСЯ', 'ДЛИННЫЕ']

Иногда лямбда-функция не нужна, потому что уже есть существующая полноценная функция. Например, необходимо определить длину каждого слова в строке:

```
words = 'the quick brown fox jumps over the lazy dog'.split()
```

Пример реализации с помощью лямбда-функции:

```
list(map(lambda word: len(word), words))
```

Пример реализации с помощью функций `list()` и `map()`:

```
list(map(len, words))
```

Еще один пример, когда лямбда-функция не нужна:

```
numbers = list(map(float, input().split()))
```

```
#Можно вместо лямбды, вызывающей метод объекта, взять
```

```
# функцию метода: <тип>.<метод>
```

```
words = ['list', 'of', 'several', 'words']
```

```
list(map(lambda word: word.upper(), words))
```

```
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
```

```
list(map(str.upper, words))
```

```
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
```

8. Принципы работы с файлами на Python

Файлы можно условно разделить на текстовые и двоичные. При записи на определенный носитель информации все файлы являются двоичными.

Содержимое текстовых файлов — текст, разбитый на строки. Из специальных символов используются только символы перехода на новую строку.

Двоичные файлы используют любые символы.

Для открытия файла используется функция `open()`, которая возвращает файловый объект: `open(имя_файла, режим_доступа)`. В качестве имени файла может использоваться полный или относительный путь.

Режимы доступа:

- `'r'` — открыть файл для чтения;
- `'w'` — открыть файл для записи;
- `'x'` — открыть файл с целью создания, если файл существует, то вызов функции `open()` завершится с ошибкой;
- `'a'` — открыть файл для записи, при этом новые данные будут добавлены в конец файла, без удаления существующих;
- `'b'` — бинарный режим;
- `'t'` — текстовый режим;
- `'+'` — открывает файл для обновления.

Режимы чтения и записи — это режимы для работы с текстовыми данными.

По умолчанию в функции `open()` используется чтение в текстовом режиме.

У файлового объекта есть атрибуты:

- `file.closed` — возвращает `True` если файл закрыт и `False` в противном случае;
- `file.mode` — возвращает режим доступа к файлу, при этом файл должен быть открыт;
- `file.name` — имя файла.

Для закрытия файла используется метод `close()`.

Пример программы на использование простых команд работы с файлом:

```
f = open("test.txt", "w")           # создаёт test.txt
print("file.mode: " + f.mode)       #file.mode: w
```

```
f = open("test.txt", "r")

print("file.closed: " + str(f.closed))      #file.closed: False

print("file.mode: " + f.mode)               #file.mode: r

print("file.name: " + f.name)               #file.name: test.txt
```

Последовательность работы с файлом: открыть файл → работа с файлом (чтение или запись) → закрыть файл. Закрывать файл особенно важно при режиме записи, так как другие процессы в системе не смогут получить доступ к этому файлу.

Пример чтения и записи в файл: `Fin`, `Fout` — файловые переменные-указатели.

```
Fin = open ( "input.txt" )

Fout = open ( "output.txt", "w" )

# здесь работаем с файлами

Fin.close()

Fout.close()
```

Рассмотрим пример чтения данных из файла `input.txt`:

```
Fin = open("input.txt")

# Чтение строки

s = Fin.readline()                # "1 2"

# Чтение строки и разбивка по пробелам

s = Fin.readline().split()        # ["1","2"]

# Чтение целых чисел

s = Fin.readline().split()        # ["1","2"]

a, b = int(s[0]), int(s[1])

# или так

a, b = [int(x) for x in s]

# или так
```

```
a, b = map( int, s )
```

В читаемом файле содержится строка, содержащая «1 2». Чтение выполняется с помощью метода `readline()`, далее строка разделяется методом `split()`. Затем числа можно разделить с помощью множественного присваивания или с помощью списочных выражений.

Файл может быть использован для вывода определенных данных. В примере файл открывается на запись. Для записи в файл применяется метод `write()`:

```
a = 1
b = 2

Fout = open("output.txt", "w")

Fout.write (":d} + {:d} = {:d}\n".format(
    a, b, a+b) )

Fout.close()
```

В методе `write()` используется форматированный вывод. В файл записывается значение переменных и их сумма.

Важно! При записи в файл все данные преобразуются в строку.

Чтение данных из файла осуществляется с помощью методов `read(размер)` и `readline()`. Метод `read(размер)` считывает из файла определенное количество символов, переданное в качестве аргумента. Если использовать этот метод без аргументов, то будет считан весь файл.

Рассмотрим пример создания файла `test.txt`, в который будет записана последовательность чисел от 0 до 9. После чего содержимое файла читается в список с помощью `readlines()` и выводится в консоль. В данном примере список будет представлять один элемент из всей строки:

```
f = open("test.txt", "w")          # создаёт test.txt

for i in range(10):
    s = str(i)+" "
    f.write(s)
```

```
f = open("test.txt", "r")
text = f.readlines()
print(text)                                # ['0 1 2 3 4 5 6 7 8 9 ']
```

В качестве аргумента метода `read()` можно передать количество символов, которое нужно считать. Пример демонстрирует код программы, которая читает первые 10 символов, в которые входит и пробел:

```
f = open("test.txt", "r")
text = f.read(10)
print(text)                                #0 1 2 3 4
```

Метод `readline()` позволяет считать строку из открытого файла:

```
f = open("test.txt", "w")
text = f.readline()
# text = f.read() и text = f.readline()
# обе функции выполняют одно и то же действие
print(text)                                #0 1 2 3 4
```

Построчное считывание можно организовать с помощью оператора `for`:

```
f = open("test.txt", "r")
text = f.read().split()
for simvol in text:
    print(simvol)
```

После выполнения примера в столбик будут выведены значения из файла.

Для записи данных файл используется метод `write(строка)`. При успешной записи метод вернет количество записанных символов:

```
f = open("test.txt", "a")
f.write(" This text would be written in the test.txt")
f = open("test.txt", "r")
```

```
text=f.read()

print(text)

#0 1 2 3 4 5 6 7 8 9   This text would be written in the text.txt
```

Рассмотрим задачу: в файле записано в столбик неизвестное количество чисел, необходимо найти их сумму. Алгоритм решения задачи может быть следующим:

```
пока не конец файла

    прочитать число из файла

    добавить его к сумме
```

Код программы на Python:

```
Fin = open ( "input.txt" )

sum = 0

while True:

    s = Fin.readline() # если конец файла, вернет пустую строку

    if not s: break

    sum += int(s)

Fin.close()
```

Программу можно записать в более коротком виде. Например, с помощью [конструкции with ... as:](#)

```
sum = 0

with open ( "input.txt" ) as Fin:

    for s in Fin:

        sum += int(s)
```

Та же программа, но записанная с использованием цикла `for`:

```
sum = 0

for s in open ( "input.txt" ):
```

```
sum += int(s)
```

9. Разбор задач на работу с файлами

Рассмотрим примеры использования файлов для некоторых практических задач.

Пусть в файле записаны в столбик целые числа. Необходимо вывести в другой текстовый файл те же числа, отсортированные в порядке возрастания.

Прежде всего числа необходимо передать в список. В программе определяем список и цикл. Далее считываем каждую строку и добавляем в файл:

```
A = []  
  
while True:  
  
    s = Fin.readline()  
  
    if not s: break  
  
    A.append ( int(s) )
```

Код можно записать короче. Файл считывается, содержимое разделяется с помощью `split()` и затем преобразуется в список:

```
s = Fin.read().split()  
  
A = list(map(int, s))
```

Для сортировки используется метод `sort()`:

```
A.sort()
```

Есть несколько способов вывода результата (таблица 1).

Таблица 1. Способы вывода результата.

Код программы	Вывод результата
<pre>Fout = open ("output.txt", "w") Fout.write (str(A)) Fout.close()</pre>	<pre>[1, 2, 3]</pre>

<pre>for x in A: Fout.write (str(x)+"\n")</pre>	<pre>1 2 3</pre>
<pre>for x in A: Fout.write ("{:4d}".format(x))</pre>	<pre>1 2 3</pre>

Рассмотрим еще один пример. В файле записаны данные о собаках. В каждой строчке – кличка собаки, ее возраст и порода:

```
Мухтар 4 немецкая овчарка
```

Необходимо вывести в другой файл сведения о собаках, которым меньше 5 лет.

Алгоритм реализации программы может быть следующим:

```
пока не конец файла Fin
```

```
    прочитать строку из файла Fin
```

```
    разобрать строку – выделить возраст
```

```
    если возраст < 5 то
```

```
        записать строку в файл Fout
```

На языке Python программа может быть написана следующим образом:

```
s = Fin.readline()    # чтение одной строки  
data = s.split()      # разбивка по пробелам  
sAge = data[1]  
age = int(sAge)       # выделение возраста
```

Кратко рассмотренный выше код можно записать в две строки:

```
s = Fin.readline()  
age = int ( s.split()[1] )
```

Код задачи полностью:

```
Fin = open ( "input.txt" )
Fout = open ( "output.txt", "w" )
while True:
    s = Fin.readline()
    if not s: break
    age = int ( s.split()[1] )
    if age < 5:
        Fout.write ( s )
Fin.close()
Fout.close()
```

Код программы можно сократить. В примере выполняется построчное чтение. После определения возраста, если он удовлетворяет условию, возраст записывается в выходной файл:

```
lst = Fin.readlines()
for s in lst:
    age = int ( s.split()[1] )
    if age < 5:
        Fout.write ( s )
```

Другой способ записи кода:

```
for s in open ( "input.txt" ):
    age = int ( s.split()[1] )
    if age < 5:
        Fout.write ( s )
```


10. Правила записи кода PEP 8

Одна из целей создания языка Python — повышение читаемости кода. Написанный код должен быть понятен не только разработчику, но и его коллегам.

Помимо самих конструкций языка Python, были разработаны рекомендации по написанию кода на языке. С ними можно ознакомиться, зайдя на официальный сайт в раздел документации и перейдя по ссылке [PEP Index](#). Далее нужно выбрать индекс [8 Style Guide for Python Code](#).

PEP 8 содержит рекомендации по названию переменных, функций, других объектов. Документация объясняет, когда необходимо выполнять отступ.

Также несложно найти перевод данного документа на русский язык.

Дополнительные материалы для самостоятельного изучения

1. [More on Defining Functions](#)
2. [pass Statements](#)
3. [The return statement](#)
4. [Methods of File Objects](#)
5. [PEP 8 – Style Guide for Python Code](#)