**Функции 3**

**Функции, как объект**

До сих пор мы рассматривали функции как совершенно отдельный элемент языка со своим синтаксисом и механизмом работы. Но, оказывается, функции также что-то вроде особого типа объектов. Бывают числа, строки, списки, кортежи, словари, множества. А бывают — функции. У каждого из этих типов есть свои операции, свой синтаксис, но все они — объекты.

Любая функция в языке Python — объект типа function.

Приведенный ниже код:

**print**(**type**(**print**))

**print**(**type**(**sum**))

**print**(**type**(**abs**))

выводит:

<class 'builtin\_function\_or\_method'>

<class 'builtin\_function\_or\_method'>

<class 'builtin\_function\_or\_method'>

Обратите внимание: скобки не ставим при передаче аргумента в функцию type(), мы не вызываем функцию, а передаем ее название.

Мы также можем объявить свою функцию, которая тоже является объектом.

Поскольку функции тоже объекты, работать с ними можно и как с объектами: записывать их в переменные, передавать в качестве аргументов другим функциям, возвращать из функций и т.д.

Приведенный ниже код:

**def** hello():

 **print**('Hello from function')

func = hello # присваиваем переменной func функцию hello

func() # вызываем функцию

выводит:

Hello from function

Таким образом, теперь переменную func можно использовать как функцию hello().

Рассмотрим еще один пример. Если по какой-либо причине вам не нравится название встроенной функции print(), которая выводит указанный текст на экран, можно написать такой код:

writeln = **print** # как в языке Pascal 😀

writeln('Hello world!')

writeln('Python')

Такой код выводит:

Hello world!

Python

**Встроенные функции, принимающие функции в качестве аргументов**

Мы изучили встроенные функции:

* min(): поиск минимального элемента;
* max(): поиск максимального элемента;
* sorted(): сортировка данных.

Но что, если мы хотим написать код для поиска максимального по модулю элемента списка numbers? И вообще, сравнивать элементы не стандартным способом, а более специфическими?

На этот случай все выше перечисленные встроенные функции могут принимать необязательный аргумент **key** – функцию, определяющую условия сравнения элементов. Другими словами, значение key должно быть функцией, принимающей один аргумент и возвращающей на его основе ключ для сравнения.

numbers = [10, -7, 8, -100, -50, 32, 87, 117, -210]

**print**(max(numbers, **key**=abs))

Пусть в списке points хранятся в виде кортежей координаты точек плоскости в двумерной биполярной системе координат.

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

При использовании встроенной функции sorted() (или списочного метода sort()) сортировка пройдет по первым значениям пар кортежа, а в случае их совпадения, по вторым.

При этом можно отсортировать данный список по второму элементу кортежа или по сумме элементов кортежей.

**def** compare\_by\_second(point):

 **return** point[1]

**def** compare\_by\_sum(point):

 **return** point[0] + point[1]

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

**print**(sorted(points, **key**=compare\_by\_second))

**print**(sorted(points, **key**=compare\_by\_sum))

**Лямбда функции**

Помимо стандартного определения функции, состоящего из ее заголовка с ключевым словом def и блока кода – тела функции, в Python можно создавать короткие однострочные функции с использованием оператора **lambda.**

Общий формат определения анонимной функции:

**lambda** список\_параметров: выражение.

Тут список\_параметров – список параметров через запятую, выражение – значение, либо код, дающий значение.

Следующие два определения функций эквивалентны:

**def** standard\_function(x

 **return** x\*2

lambda\_function = **lambda** x: x\*2

Когда применение анонимных функций оправдано:

* однократное использование функции;
* передача функций в качестве аргументов другим функциям;
* возвращение функции в качестве результата другой функции.

Вернемся к примеру сортировки кортежей по второму элементу и сумме элементов. Очевидно, что такие функции как compare\_by\_second() и compare\_by\_sum() не особо нужны вне контекста сортировки, поэтому логично их заменить на анонимные функции:

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

**print**(sorted(points, **key**=**lambda** point: point[1]))

**print**(sorted(points, **key**=**lambda** point: point[0] + point[1]))

**Функции высшего порядка**

Как уже сказано, функции, которые принимают или/и возвращают другие функции, называются **функциями высшего порядка**.

Часто функции высшего порядка используются для обработки наборов данных. Рассмотрим две важные функции высшего порядка:

* map();
* filter();

**Функция map()**

При работе со списками часто требуется применить одно и то же преобразование к каждому элементу. Можно написать цикл, содержащий нужное преобразование.

Например, для преобразования списка чисел в список их квадратов, код может выглядеть так:

**def** f(x):

 **return** x \*\* 2

old\_list = [1, 2, 4, 9, 10, 25]

new\_list = []

**for** item **in** old\_list:

 new\_item = f(item)

 new\_list.append(new\_item)

**print**(old\_list)

**print**(new\_list)

Несложно понять, что цикл будет выглядеть одинаково практически во всех случаях. Меняться будет только преобразование, то есть применяемая функция f(). Так почему бы не обобщить код, чтобы функция была параметром?

**def** map(function, items):

 result = []

 **for** item **in** items:

 new\_item = function(item)

 result.append(new\_item)

 **return** result

Теперь мы можем совершать преобразования, используя функцию высшего порядка map().

Приведенный ниже код:

**def** square(x):

 **return** x\*\*2

**def** cube(x):

 **return** x\*\*3

numbers = [1, 2, -3, 4, -5, 6, -9, 0]

strings = map(str, numbers)

abs\_numbers = map(abs, numbers)

squares = map(square, numbers)

cubes = map(cube, numbers)

Нет необходимости делать преобразующую элементы функцию отдельно определенной именованной функцией.

squares = map(**lambda** x: x \*\* 2, numbers)

cubes = map(**lambda** x: x \*\* 3, numbers)

numbers = [1, 2, 3, 4, 5, 6]

# увеличиваем на 1

new\_numbers1 = list(map(lambda x: x+1, numbers))

# удваиваем

new\_numbers2 = list(map(lambda x: x\*2, numbers))

**Функция filter()**

Другая популярная задача при работе со списками: отобрать часть элементов списка по определенному критерию. Функция высшего порядка для решения такой задачи называется filter().

Реализация такой функции может выглядеть так:

**def** filter(function, items):

 result = []

 **for** item **in** items:

 **if** function(item):

 result.append(item)

 **return** result

Наша функция filter() применяет предикат function к каждому элементу и добавляет в итоговый список только те элементы, для которых предикат вернул True.

Например, чтобы из исходного списка чисел получить список с элементами, большими 10, можно написать такой код:

**def** is\_greater10(num):

 **return** num > 10

numbers = [12, 2, -30, 48, 51, -60, 19, 10, 13]

large\_numbers = filter(is\_greater10, numbers)

**print**(large\_numbers)

Здесь тоже возможно использование лямбда функции

large\_numbers = filter(**lambda** x: x > 10, numbers)

Приведем еще примеры

numbers = [-1, 2, -3, 4, 0, -20, 10, 30, -40, 50, 100, 90]

# положительные числа

positive\_numbers = list(filter(lambda x: x > 0, numbers))

# числа, большие 50

large\_numbers = list(filter(lambda x: x > 50, numbers))

# четные числа

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

words = ['python', 'stepik', 'beegeek', 'iq-option']

# слова длиною больше 6 символов

new\_words1 = list(filter(lambda w: len(w) > 6, words))

# слова содержащие букву e

new\_words2 = list(filter(lambda w: 'e' in w, words))