**Lambda – для создания небольших маленьких функций**

**Map – применяет функцию к списку или словарю, возвращает измененный объект**

**Filter - применяет логическую функцию к списку или словарю, возвращает элементы, которые удовлетворяют условию**

**Zip – создает кортежи из элементов словарей или списков**

List comprehension**- удобная генерация списков**

Оглавление

[List comprehension 1](#_Toc114511198)

[lambda 2](#_Toc114511199)

[map 3](#_Toc114511200)

[filter 4](#_Toc114511201)

[Функция zip 5](#_Toc114511202)

[List comprehension 9](#_Toc114511203)

# lambda

лямбда-оператор или лямбда-функция используются для создания небольших, одноразовых и анонимных объектов функции в Python.

Базовый синтаксис:

lambda arguments : expression

lambda-оператор может иметь любое количество аргументов, но может иметь только одно выражение. Он не может содержать никаких операторов и возвращает объект функции, который может быть назначен любой переменной.

Пример:

def add(x, y):

return x + y

# Вызываем функцию

add(2, 3) # Результат: 5

Выше мы создали функцию с именем **add**, которая принимает два аргумента **x** и **y** и возвращает их сумму.

Давайте посмотрим как эта простейшая функция будет выглядить с применением lambda:

add = lambda x, y : x + y

print add(2, 3) # Результат: 5

В **lambda x, y: x + y;**xиyявляются аргументами функции, а x + y - это выражение, которое выполняется, и его значения возвращаются в качестве вывода.

lambda x, y: x + y возвращает объект функции, который может быть назначен любой переменной, в этом случае функциональный объект присваивается переменной **add**.

type (add) # Возвращет: function

Если мы проверим тип add, это function.

# map

Базовый синтаксис:

map(function\_object, iterable1, iterable2,...)

Функция map первым аргументом ожидает объект функции и любое количество объектов для перебора, таких как dict или list. Он выполняет **function\_object** для каждого элемента в последовательности и возвращает список элементов, измененных объектом функции.

def multiply2(x):

return x \* 2

map(multiply2, [1, 2, 3, 4]) # Вернет [2, 4, 6, 8]

В приведенном выше примере **map** выполняет функцию multiply2 для каждого элемента в списке [1, 2, 3, 4] и возвращает [2, 4, 6, 8]

Давайте посмотрим, как мы можем написать приведенный выше код с помощью map и lambda.

map(lambda x : x\*2, [1, 2, 3, 4]) #Вернет [2, 4, 6, 8]

Всего одна строка кода и все.

dict\_a = [{'name': 'python', 'points': 10}, {'name': 'java', 'points': 8}]

map(lambda x : x['name'], dict\_a) # Вернет: ['python', 'java']

map(lambda x : x['points']\*10, dict\_a) # Вернет: [100, 80]

map(lambda x : x['name'] == "python", dict\_a) # Вернет: [True, False]

В приведенном выше примере каждый объект dict\_a будет передан как параметр функции лямбда

# filter

Базовый синтаксис:

filter(function\_object, iterable)

Функция **filter** ожидает два аргумента, function\_object и iterable. function\_object должна возвращать логическое значение. function\_object вызывается для каждого элемента итерации, и фильтр возвращает только те элементы, для которых function\_object возвращает true.

Подобно функции map, функция filter также возвращает список элементов. В отличие от map функция filter может принять только один итерируемый объект.

Пример:

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

filter(lambda x : x % 2 == 0, a) # Вернет: [2, 4, 6]

Обработка объектов:

dict\_a = [{'name': 'python', 'points': 10}, {'name': 'java', 'points': 8}]

filter(lambda x : x['name'] == 'python', dict\_a) # Вернет: [{'name': 'python', 'points': 10}]

# Функция zip

В Pyhon функция zip позволяет пройтись одновременно по нескольким итерируемым объектам (спискам и др.):

>>> a = [10, 20, 30, 40]

>>> b = ['a', 'b', 'c', 'd', 'e']

>>> for i, j in zip(a, b):

... print(i, j)

...

10 a

20 b

30 c

40 d

Здесь выражение zip(a, b) создает объект-итератор, из которого при каждом обороте цикла извлекается кортеж, состоящий из двух элементов. Первый берется из списка a, второй - из b.

>>> for i in zip(a, b):

... print(i, type(i))

...

(10, 'a') <class 'tuple'>

(20, 'b') <class 'tuple'>

(30, 'c') <class 'tuple'>

(40, 'd') <class 'tuple'>

В списке b на один элемент больше. Функция zip возвращает итератор, который останавливается, когда исчерпывается самая короткая последовательность. Если требуется учесть все значения из самой длинной, то следует использовать функцию zip\_longest из модуля itertools:

>>> a = [10, 20, 30, 40]

>>> b = ['a', 'b', 'c', 'd', 'e']

>>> c = [1.1, 1.2]

>>> for i in itertools.zip\_longest(a,b,c):

... print(i)

...

(10, 'a', 1.1)

(20, 'b', 1.2)

(30, 'c', None)

(40, 'd', None)

(None, 'e', None)

Как мы видим, если элемента не хватает, то по-умолчанию подставляется объект None. Можно указать свой вариант заполнения:

>>> for i in itertools.zip\_longest(a,b,c, fillvalue=0):

... print(i)

...

(10, 'a', 1.1)

(20, 'b', 1.2)

(30, 'c', 0)

(40, 'd', 0)

(0, 'e', 0)

Если требуется получить не итератор, возвращаемый zip(), а список из элементов, то к объекту zip применима функция list, которая преобразует итератор в список:

>>> a = [10, 20, 30, 40]

>>> c = [1.1, 1.2, 1.3, 1.4]

>>> ac = zip(a, c)

>>> type(ac)

<class 'zip'>

>>> ac = list(ac)

>>> type(ac)

<class 'list'>

>>> ac

[(10, 1.1), (20, 1.2), (30, 1.3), (40, 1.4)]

При написании программ на Python функцию zip нередко применяют для выполнения совместных или одновременных действий над элементами разных списков.

>>> values = [1.34, 3.25, 2.99]

>>> coefficient = [3, 2, 2]

>>> for i, j in zip(values, coefficient):

... print(i\*j)

...

4.0200000000000005

6.5

5.98

>>> a = []

>>> b = []

>>> for i, j in zip(range(10,20), range(1,10)):

... a.append(i)

... b.append(j)

...

>>> a

[10, 11, 12, 13, 14, 15, 16, 17, 18]

>>> b

[1, 2, 3, 4, 5, 6, 7, 8, 9]

В Python есть еще одна встроенная функция, которая часто используется в заголовке **for**. Это функция enumerate(). Если range() позволяет получить только индексы элементов списка, то enumerate() – сразу индекс элемента и его значение.

Функция enumerate() применяется для так называемых итерируемых объектов (список относится к таковым) и создает объект-генератор, который генерирует кортежи, состоящие из двух элементов – индекса элемента и самого элемента.

>>> spisok = [16, 46, 26, 36]

>>> for i in enumerate(spisok):

... print(i)

...

(0, 16)

(1, 46)

(2, 26)

(3, 36)

>>> b = "hello"

>>> for i in enumerate(b):

... print(i)

...

(0, 'h')

(1, 'e')

(2, 'l')

(3, 'l')

(4, 'o')

Эти кортежи можно распаковывать, то есть извлекать индекс и значение, в теле цикла:

>>> for item in enumerate(spisok):

... print(item[0], item[1])

...

0 16

1 46

2 26

3 36

Однако чаще это делают еще в заголовке **for**, используя две переменные перед **in**:

>>> for id, val in enumerate(spisok):

... print(id, val)

...

0 16

1 46

2 26

3 36

**Функция enumerate() используется для упрощения прохода по коллекциям в цикле, когда кроме самих элементов требуется их индекс:**

>>> a = [10, 20, 30, 40]

>>> for id, item in enumerate(a):

... a[id] = item + 5

...

>>> a

[15, 25, 35, 45]

В данном случае на каждой итерации цикла из объекта, полученного от вызова функции **enumerate**, извлекается очередной кортеж. Этот кортеж состоит из индекса очередного элемента списка и значения этого элемента. Элементы кортежа связываются с идентификаторами id и item.

Без использования **enumerate** в цикл пришлось бы вводить переменную-счетчик:

>>> a = [10, 20, 30, 40]

>>> id = 0 # используется счетчик

>>> for num in a:

... a[id] = num + 5

... id += 1

...

>>> a

[15, 25, 35, 45]

Или извлекать элементы по индексу:

>>> a = [10, 20, 30, 40]

>>> for i in range(len(a)): # перебор по индексам

... a[i] += 5

...

>>> a

[15, 25, 35, 45]

Другими словами, без функции **enumerate** можно обойтись. Однако в ряде случаев она может быть удобней.

List comprehension трудно перевести правильно на русский, потому, раз он генерирует новый список, будем называть его просто генератором списков. Это одна из самых приятных вещей в python, научившись писать которую, будешь применять её везде. Как это выглядит? Пожалуй Вы точно видели записи такого вида:

squares = [x \*\* 2 **for** x **in** range(10)]

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

это и есть генератор списка, результатом которого будет список квадратов последовательности 0..9. Что здесь происходит? Это обычный цикл for, только записан в удобочитаемом виде, в развёрнутом виде это выглядело бы так:

squares = []

**for** x **in** range(10):

squares.append(x \*\* 2)

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

Вот тут и видна разница в этих записях — генератор списка условно можно назвать синтаксическим сахаром для цикла for, но у них разное время выполнения. Под капотом генератор списка также использует цикл for но выигрывает по скорости из-за того, то не вызывает метод append у списка (подробности [здесь](http://stackoverflow.com/a/22108640/3355831)).

Так же как и в обычных циклах в генераторах можно применять условия:

>>> odds = [x **for** x **in** range(10) **if** x % 2 != 0]

*# [1, 3, 5, 7, 9]*

если в условии нужен else, то всё условие пишется до for:

>>> [x \*\* 2 **if** x % 2 == 0 **else** x \*\* 3 **for** x **in** range(10)]

*# [0, 1, 4, 27, 16, 125, 36, 343, 64, 729]*

Гораздо удобнее итерироваться по двум спискам:

first = []

**for** x **in** range(1, 5):

**for** y **in** range(5, 1, -1):

**if** x != y:

first.append((x, y))

second = [(x, y) **for** x **in** range(1, 5) **for** y **in** range(5, 1, -1) **if** x != y]

>>> first == second

True

Отличный пример из документации (раскрытие списка списков), усложним его:

>>> vec = [[[1, 2, 3], [4, 5, 6], [7, 8, 9]], [[10, 11, 12], [13, 14, 15], [16, 17, 18]]]

>>> [digit **for** lst **in** vec **for** elem **in** lst **for** digit **in** elem]

*# [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]*

Подобным образом с помощью генераторов списков мы можем создать словарь:

>>> dict([(key, value) for (key, value) **in** zip([1, 2, 3], ['a', 'b', 'c'])])

*# {1: 'a', 2: 'b', 3: 'c'}*

И тут нас ожидает приятная новость — в python есть и генераторы словарей, записываются так же, как и генераторы списков, только в фигурных скобках { ... }:

>>> {key: value **for** key, value **in** zip([1, 2, 3], ['a', 'b', 'c'])}

*# {1: 'a', 2: 'b', 3: 'c'}*

К слову, словарь можно создать и без генератора — dict(zip(list1, list2)).

Если мы генератор списка запишем в круглых скобках, то получим обычный генератор:

>>> gen = (x **for** x **in** range(10))

>>> gen

<generator object <genexpr> at 0x7f635076ea98>

>>> gen.\_\_next\_\_()

0

>>> gen.\_\_next\_\_()

1

.......