

Практикум по программированию на языке Python

Занятие 3: Пользовательские и встроенные функции, итераторы и генераторы

Мурат Апишев (mel-lain@yandex.ru)

Москва, 2019

Функции range и enumerate

```
In [8]: r = range(2, 10, 3)
print(type(r))

for e in r:
    print(e, end=' ')
```

```
<class 'range'>
2 5 8
```

```
In [13]: for index, element in enumerate(list('abcdef')):
          print(index, element, end=' ')
```

```
0 a  1 b  2 c  3 d  4 e  5 f
```

Функция zip

```
In [20]: z = zip([1, 2, 3], 'abc')
print(type(z))

for a, b in z:
    print(a, b, end='  ')
```

```
<class 'zip'>
1 a 2 b 3 c
```

```
In [21]: for e in zip('abcdef', 'abc'):
print(e)
```

```
('a', 'a')
('b', 'b')
('c', 'c')
```

```
In [24]: for a, b, c, d in zip('abc', [1,2,3], [True, False, None], 'xyz'):
print(a, b, c, d)
```

```
a 1 True x
b 2 False y
c 3 None z
```

Определение собственных функций

```
In [26]: def function(arg_1, arg_2=None):  
        print(arg_1, arg_2)  
  
        function(10)  
        function(10, 20)
```

```
10 None  
10 20
```

Функция - это тоже объект, её имя - просто символическая ссылка:

```
In [28]: f = function  
        f(10)  
  
        print(function is f)
```

```
10 None  
True
```

Определение собственных функций

```
In [30]: retval = f(10)  
print(retval)
```

10 None
None

```
In [32]: def factorial(n):  
         return n * factorial(n - 1) if n > 1 else 1 # recursion  
  
print(factorial(1))  
print(factorial(2))  
print(factorial(4))
```

1
2
24

Передача аргументов в функцию

Параметры в Python всегда передаются по ссылке

```
In [33]: def function(scalar, lst):  
         scalar += 10  
         print(f'Scalar in function: {scalar}')  
  
         lst.append(None)  
         print(f'Scalar in function: {lst}')
```

```
In [34]: s, l = 5, []  
         function(s, l)  
  
         print(s, l)
```

```
Scalar in function: 15  
Scalar in function: [None]  
5 [None]
```

Передача аргументов в функцию

```
In [225]: def f(a, *args):  
           print(type(args))  
           print([v for v in [a] + list(args)])  
  
f(10, 2, 6, 8)  
  
<class 'tuple'>  
[10, 2, 6, 8]
```

```
In [226]: def f(*args, a):  
           print([v for v in [a] + list(args)])  
           print()  
  
f(2, 6, 8, a=10)  
  
[10, 2, 6, 8]
```

```
In [227]: def f(a, *args, **kw):  
           print(type(kw))  
           print([v for v in [a] + list(args) + [(k, v) for k, v in kw.items()]])  
  
f(2, *(6, 8), **{'arg1': 1, 'arg2': 2})  
  
<class 'dict'>  
[2, 6, 8, ('arg1', 1), ('arg2', 2)]
```

Области видимости переменных

В Python есть 4 основных уровня видимости:

- Встроенная (builtins) - на этом уровне находятся все встроенные объекты (функции, классы исключений и т.п.)
- Глобальная в рамках модуля (global) - всё, что определяется в коде модуля на верхнем уровне
- Объемлющей функции (enclosed) - всё, что определено в функции верхнего уровня
- Локальной функции (local) - всё, что определено в функции нижнего уровня

Есть ещё области видимости переменных циклов, списковых включений и т.п.

Правило разрешения области видимости LEGB при чтении

```
In [38]: def outer_func(x):  
         def inner_func(x):  
             return len(x)  
         return inner_func(x)
```

```
In [39]: print(outer_func([1, 2]))
```

2

Кто определил имя `len` ?

- на уровне вложенной функции такого имени нет, смотрим выше
- на уровне объемлющей функции такого имени нет, смотрим выше
- на уровне модуля такого имени нет, смотрим выше
- на уровне `builtins` такое имя есть, используем его

На `builtins` можно посмотреть

```
In [83]: import builtins

counter = 0
lst = []
for name in dir(builtins):
    if name[0].islower():
        lst.append(name)
        counter += 1

    if counter == 5:
        break

lst
```

```
Out[83]: ['abs', 'all', 'any', 'ascii', 'bin']
```

Кстати, то же самое можно сделать более pythonic кодом:

```
In [81]: list(filter(lambda x: x[0].islower(), dir(builtins)))[: 5]
```

```
Out[81]: ['abs', 'all', 'any', 'ascii', 'bin']
```

Локальные и глобальные переменные

In [47]:

```
x = 2
def func():
    print('Inside: ', x) # read

func()
print('Outside: ', x)
```

Inside: 2
Outside: 2

In []:

```
x = 2
def func():
    x += 1 # write
    print('Inside: ', x)

func() # UnboundLocalError: local variable 'x' referenced before assignment
print('Outside: ', x)
```

In [50]:

```
x = 2
def func():
    x = 3
    x += 1
    print('Inside: ', x)

func()
print('Outside: ', x)
```

Inside: 4
Outside: 2

Ключевое слово global

In [52]:

```
x = 2
def func():
    global x
    x += 1 # write
    print('Inside: ', x)

func()
print('Outside: ', x)
```

Inside: 3
Outside: 3

In [55]:

```
x = 2
def func(x):
    x += 1
    print('Inside: ', x)
    return x

x = func(x)
print('Outside: ', x)
```

Inside: 3
Outside: 3

Ключевое слово nonlocal

```
In [73]: a = 0
def out_func():
    b = 10
    def mid_func():
        c = 20
        def in_func():
            global a
            a += 100

            nonlocal c
            c += 100

            nonlocal b
            b += 100

            print(a, b, c)

        in_func()
    mid_func()

out_func()
```

100 110 120

Главный вывод: не надо злоупотреблять побочными эффектами при работе с переменными верхних уровней

Пример вложенных функций: замыкания

- В большинстве случаев вложенные функции не нужны, плоская иерархия будет и проще, и понятнее
- Одно из исключений - фабричные функции (замыкания)

```
In [75]: def function_creator(n):  
         def function(x):  
             return x ** n  
  
         return function  
  
f = function_creator(5)  
f(2)
```

```
Out[75]: 32
```

Объект-функция, на который ссылается `f`, хранит в себе значение `n`

Анонимные функции

- `def` - не единственный способ объявления функции
- `lambda` создаёт анонимную (`lambda`) функцию

Такие функции часто используются там, где синтаксически нельзя записать определение через `def`

```
In [88]: def func(x): return x ** 2  
func(6)
```

Out[88]: 36

```
In [94]: lambda_func = lambda x: x ** 2 # should be an expression  
lambda_func(6)
```

Out[94]: 36

```
In [91]: def func(x): print(x)  
func(6)
```

6

```
In [93]: lambda_func = lambda x: print(x ** 2) # as print is function in Python 3.*  
lambda_func(6)
```

36

Встроенная функция sorted

```
In [102]: lst = [5, 2, 7, -9, -1]
```

```
In [109]: def abs_comparator(x):  
            return abs(x)  
  
            print(sorted(lst, key=abs_comparator))  
  
            [-1, 2, 5, 7, -9]
```

```
In [111]: sorted(lst, key=lambda x: abs(x))
```

```
Out[111]: [-1, 2, 5, 7, -9]
```

```
In [112]: sorted(lst, key=lambda x: abs(x), reverse=True)
```

```
Out[112]: [-9, 7, 5, 2, -1]
```


Встроенная функция `filter`

```
In [220]: lst = [5, 2, 7, -9, -1]
```

```
In [221]: f = filter(lambda x: x < 0, lst)  # True condition  
          type(f)  # iterator
```

```
Out[221]: filter
```

```
In [222]: list(f)
```

```
Out[222]: [-9, -1]
```

Встроенная функция map

```
In [123]: lst = [5, 2, 7, -9, -1]
```

```
In [124]: m = map(lambda x: abs(x), lst)
           type(m)  # iterator
```

```
Out[124]: map
```

```
In [125]: list(m)
```

```
Out[125]: [5, 2, 7, 9, 1]
```

Ещё раз сравним два подхода

Напишем функцию скалярного произведения в императивном и функциональном стилях:

```
In [151]: def dot_product_imp(v, w):  
           result = 0  
           for i in range(len(v)):  
               result += v[i] * w[i]  
           return result
```

```
In [148]: dot_product_func = lambda v, w: sum(map(lambda x: x[0] * x[1], zip(v, w)))
```

```
In [152]: print(dot_product_imp([1, 2, 3], [4, 5, 6]))  
           print(dot_product_func([1, 2, 3], [4, 5, 6]))
```

32
32

Функция `reduce`

`functools` - стандартный модуль с другими функциями высшего порядка.

Рассмотрим пока только функцию `reduce`:

```
In [155]: from functools import reduce  
  
lst = list(range(1, 10))  
  
reduce(lambda x, y: x * y, lst)
```

```
Out[155]: 362880
```

Итерирование, функции `iter` и `next`

In [176]:

```
r = range(3)

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

0
1
2

In [178]:

```
it = iter(r)  # r.__iter__() - gives us an iterator
```

```
print(next(it))
print(it.__next__())
print(next(it))
print(next(it))
```

0
1
2

```
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-178-08a2b9f30bb6> in <module>
      4 print(it.__next__())
      5 print(next(it))
----> 6 print(next(it))
```

StopIteration:

Функции-генераторы

- Генераторы - подмножество итераторов
- Итератор в общем случае не хранит никакого состояния, он просто выполняет функцию `__next__()`
- Генератор - это функция (или результат генераторного выражения), сохраняющий состояние
- Сразу: `range` возвращает не итератор!
- Примеры функций-генераторов:
 - `zip`
 - `enumerate`
 - `reversed`
 - `map`
 - `filter`

```
In [193]: print('__next__' in dir(zip([], [])))  
          print('__next__' in dir(range(3)))
```

```
True  
False
```

Ключевое слово `yield`

- `yield` - это слово, по смыслу похожее на `return`
- Но используется в функциях, возвращающих генераторы
- При вызове такой функции тело не выполняется, функция только возвращает генератор
- В первый запуск функция будет выполняться от начала и до `yield`
- После выхода состояние функции сохраняется
- На следующий вызов будет проводиться итерация цикла и возвращаться следующее значение
- И так далее, пока не кончится цикл каждого `yield` в теле функции
- После этого генератор станет пустым

Пример генератора

```
In [194]: def my_range(n):  
           yield 'You really want to run this generator?'  
  
           i = -1  
           while i < n:  
               i += 1  
               yield i
```

```
In [197]: gen = my_range(3)  
while True:  
    try:  
        print(next(gen), end='    ')  
    except StopIteration: # we want to catch this type of exceptions  
        break
```

You really want to run this generator? 0 1 2 3

```
In [199]: for e in my_range(3):  
           print(e, end='    ')
```

You really want to run this generator? 0 1 2 3

Модуль `itertools`

- Модуль представляет собой набор инструментов для работы с итераторами и последовательностями
- Содержит три основных типа итераторов:
 - бесконечные итераторы
 - конечные итераторы
 - комбинаторные итераторы
- Позволяет эффективно решать небольшие задачи вида:
 - итерирование по бесконечному потоку
 - слияние в один список вложенных списков
 - генерация комбинаторного перебора сочетаний элементов последовательности
 - аккумуляция и агрегация данных внутри последовательности

Модуль `itertools`: примеры

In [203]: `from itertools import count`

```
for i in count(start=0):  
    print(i, end=' ')  
    if i == 5:  
        break
```

0 1 2 3 4 5

In [205]: `from itertools import cycle`

```
count = 0  
for item in cycle('XYZ'):  
    if count > 4:  
        break  
    print(item, end=' ')  
    count += 1
```

X Y Z X Y

Модуль `itertools`: примеры

```
In [210]: from itertools import accumulate

for i in accumulate(range(1, 5), lambda x, y: x * y):
    print(i)
```

```
1
2
6
24
```

```
In [211]: from itertools import chain

for i in chain([1, 2], [3], [4]):
    print(i)
```

```
1
2
3
4
```

Модуль `itertools`: примеры

```
In [215]: from itertools import groupby

vehicles = [('Ford', 'Taurus'), ('Dodge', 'Durango'),
            ('Chevrolet', 'Cobalt'), ('Ford', 'F150'),
            ('Dodge', 'Charger'), ('Ford', 'GT')]

sorted_vehicles = sorted(vehicles)

for key, group in groupby(sorted_vehicles, lambda x: x[0]):
    for maker, model in group:
        print('{model} is made by {maker}'.format(model=model, maker=maker))

    print ("**** END OF THE GROUP ***\n")
```

```
Cobalt is made by Chevrolet
**** END OF THE GROUP ***
```

```
Charger is made by Dodge
Durango is made by Dodge
**** END OF THE GROUP ***
```

```
F150 is made by Ford
GT is made by Ford
Taurus is made by Ford
**** END OF THE GROUP ***
```

Спасибо за внимание!