Функциональное программирование

на Python

Мы рассмотрим:

1) lambda-выражения;
2) замыкания;
3) функторы;
4) Функции Первого Класса;
5) модули functools и operator.

О функциональном программировании

Функциональное программирование – одна из парадигм программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних.

lambda-выражения

Lambda-выражение создаёт функцию (возможно, безымянную). В ней можно вызывать функции, использовать операторы, но нельзя использовать выражения типа присвоения, if, for и while. Она возвращает значение последнего вычисленного выражения.

```
>>> L = lambda x: x*2
>>> L(5)
10
```

lambda-выражения

```
>>> def double(x):
...     return 2*x
>>> L = lambda s: double(s) + v
>>> v = 10
>>> L(4)
18

>>> (lambda s: s*2)(5)
10
```

lambda-выражения - пример

```
>>> f = lambda sec: [sec/3600,
                     (sec/60)%60,
                     sec%60]

>>> f(12710)
[3, 31, 50]
```

lambda-выражения

```
>>> def f(s):
...     if s == 0:
...         return "zero"
...     elif s == 1:
...         return "one"
...     else:
...         return "other"
>>> f(0)
'zero'
```

Вместо if:
and и or

```
>>> L = lambda s: (s == 0 and "zero") or
(s == 1 and "one") or ("other")
>>> L(0)
'zero'
```

Справа от and стоит ненулевое значение, иначе даже в случае true «левой половины» будут выполняться следующие скобки!

lambda-выражения

```
>>> def f(k):
...     while k > 0:
...         print k
...         k -= 1
>>> f(2)
2
1
```

Вместо while:

рекурсия

```
>>> def my_print(x):
...     print x
...     return x
>>> L = lambda k: k and my_print(k) and L(k-1)
>>> nul = L(2)
2
1
>>> nul
0
```

Замыкания

Замыкание — функция, сохраняющая необходимые данные.

Что делать, если функция зависит от большого числа параметров?

• Передавать их в функцию при каждом вызове — неудобно;

• Создать глобальные переменные для части параметров, использовать их в функции - «засорение» глобального пространства имен, может привести к ошибкам.

• ...Хотелось бы, чтобы экземпляр функции «запоминал» свои параметры.

Замыкания

Вариант №1: параметр по умолчанию.

```
>>> N = 10
>>> def mulN(i, n=N):
...     return i * n

>>> mulN(5)
50
>>> N = 55
>>> mulN(5)
50

>>> L = lambda i, n=N: i * n
>>> L(5)
275
```

Замыкания — пример №1

Вариант №2: фабрика функций.

```
>>> def mulN(n):
...     return lambda i: i*n

>>> mul2 = mulN(2)
>>> mul2(7)
14
>>> mulN(3)(8)
24
```

Замыкания — пример №2

```
def logger_factory(filename):
    f = open(filename, "a")
    def logger(message):
        f.write(message + "\n")
        return f
    return logger

>>> logger=logger_factory("log.txt")
>>> f = logger("MESSAGE")
>>> f.close()
```

Замыкания — пример №3

```
def f(a, b):
    print "first: %s, second: %s" % (a, b)

def currying(func, f_param):
    def function(s_param):
        return func(f_param, s_param)
    return function

>>> func = currying(f, 5)
>>> func(7)
first: 5, second: 7
```

Функторы

- это классы с определённым оператором(), в Python — с реализованным методом __call__.

```
class Functor:
    def __call__(self, value):
        return value*2

>>> f = Functor()
>>> f(5)
10
```

Функторы — пример №1

```
class mulN:
    def __init__(self, n):
        self.n = n
    def __call__(self, value):
        return self.n * value

>>> f = mulN(7)
>>> f(3)
21
```

Функторы — пример №2

```
class logger:
  def __init__(self, filename):
    self.file = open(filename, "a")
  def __call__(self, message):
    self.file.write(message + "\n")
  def close(self):
    self.file.close()

>>> l = logger("log.txt")
>>> l("Message")
>>> l.close()
```

Функторы — пример №3

```
def f(a, b):
    print "first: %s, second: %s" % (a, b)

class currying:
    def __init__(self, func, param):
        self.func = func
        self.param = param
    def __call__(self, param):
        return self.func(self.param, param)

>>> c = currying(f, 5)
>>> c(7)
first: 5, second: 7
```

Функции Первого Класса

- это функции, принимающие в качестве параметра другие функции.

Например:

- filter();
- map();
- reduce();
- apply()...

filter(function, sequence)

Возвращает последовательность (по возможности того же типа, что и sequence), для которых предикат function вернул true.

```
>>> L = [1, 4, 6, 7, 8, 9]
>>> filter(lambda x: x % 2 == 0, L)
[4, 6, 8]

>>> T = (1, 4, 6, 7, 8, 9)
>>> filter(None, T)
(1, 4, 6, 7, 8, 9)
```

map(function, sequence [. . .])

Возвращает список значений, полученных применением функции function к элементам одной или нескольких последовательностей.
Аналогична transform() из библиотеки STL.

```
>>> def cube(x):
...   return x*x*x
>>> map(cube, range(5))
[0, 1, 8, 27, 64]

>>> map(lambda x: x*x*x, range(5))
[0, 1, 8, 27, 64]
```

map(function, sequence [. . .])

```
>>> x = [5, 2, 7]
>>> y = (1, 10,3)

>>> map(lambda x,y: x*y, x, y)
[5, 20, 21]

>>> y = (1, 10, 3, 1)
>>> map(max, x, y)
[5, 10, 7, 1]
```

map(function, sequence [. . .])

В случае, если function равно None, получаем список кортежей

```
>>> seq = xrange(4)
>>> square = lambda x: x*x
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9)]

>>> seq1 = range(1, 5)
>>> seq2 = ["one", "two", "three"]
>>> map(None, seq1, seq2)
[(1, 'one'), (2, 'two'),
 (3, 'three'), (4, None)]
```

zip(sequence [. . .])

Его действие аналогично map с None вместо функции — формирование кортежей из параметров. Количество кортежей будет равно длине кратчайшей последовательности.

```
>>> a = range(1,10)
>>> b = ("one", "two", "three")
>>> c = {"!" : 1, "@" : 2, "#" : 3}
>>> x = zip(a, b, c)
>>> x
[(1, 'one', '!'),
 (2, 'two', '@'),
 (3, 'three', '#')]
```

zip(sequence [. . .])

Если последовательности имеют одинаковую длину, то эта процедура обратима (будет список кортежей)

```
>>> x = zip(a, b, c)
>>> x
[(1, 'one', '!'),
 (2, 'two', '@'),
 (3, 'three', '#')]

>>> zip(x)
[((1, 'one', '!'),),
 ((2, 'two', '@'),),
 ((3, 'three', '#'),)]
```

Не получилось!!!

zip(sequence [. . .])

* - специальный символ, он используется только при передаче последовательности в функцию, чтобы передавать не один аргумент, а множество.

```
>>> x = zip(a, b, c)
>>> x
[(1, 'one', '!'),
 (2, 'two', '@'),
 (3, 'three', '#')]

>>> zip(*x)
[(1, 2, 3),
 ('one', 'two', 'three'),
 ('!', '@', '#')]
```

Получилось!!!

* и **

```
>>> def f(a,b,c,d,e):
        return a+b+c+d+e

>>> L = [1, 2, 3, 4, 5]
>>> f(L)
TypeError: f() takes exactly 5 arguments (1 given)
>>> f(*L)
15
>>> D =  {"a": 1, "b": 2, "c": 3,
         "d": 4, "e": 5}
>>> f(D)
TypeError: f() takes exactly 5 arguments (1 given)
>>> f(*D)
'acbed'
>>> f(**D)
15
```

reduce(function, sequence [, initial])

Возвращает значение, полученное последовательным применением бинарной функции function к элементам последовательности и результату предыдущего вычисления. Аналогична accumulate() из библиотеки STL.

```
>>> reduce(lambda x,y: x * y,
           range(1, 6),
           1)
120
# ((((1*1)*2)*3)*4)*5
```

reduce(function, sequence [, initial])

```
>>> def add(x,y):
...     return x+y
>>> reduce(add, xrange(1, 6))
15 # ((((1+2)+3)+4)+5)

>>> reduce (lambda x,y: x+y,
            xrange(1, 6))
15

>>> sum(xrange(1, 6))
15
# reduce(lambda x,y: x+y, seq, 0)
```

apply(function, parameters)

Возвращает значение, полученное выполнением функции function с параметрами parameters. Позиционные параметры задаются списком, именованные - словарем.

```
>>> def f(a, b, c, d = None, e = None):
...     print a, b, c, d, e
>>> apply  (f, [1, 2, 3],
            {"d" : "DDD", "e" : "EEE"})
1 2 3 DDD EEE

>>> f( *[1, 2, 3],
       **{"d" : "DDD", "e" : "EEE"})
1 2 3 DDD EEE
```