Python:

Списки и итераторы

с разных ракурсов

Тимофеев Никита
АУ РАН

Мы рассмотрим:

1) Итераторы;
2) Списочные встраивания;
3) Генераторные выражения;
4) Ленивые вычисления;
5) Генераторы;
6) Модуль Itertools.

Итераторы

Итератор — сущность, указывающая на единицу данных в потоке данных. Эта сущность обладает единственным методом next(), который возвращает следующий элемент в потоке.

В качестве потока данных могут выступать итерируемые структуры данных (списки, кортежи, словари, строки), файлы и пр.

Все функции, принимающие структуры данных, неявно получают итератор для них. В них всегда можно передать итератор.

Итераторы

```
>>> L = [1, 2]
>>> it = iter(L)
>>> print it
<listiterator object at 0x8e828ac>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
Traceback (most recent call last):
File "<pyshell#7>", line 1, in <module>
it.next()
StopIteration
```

Итераторы

```
>>> for i in L:
...   print i

>>> for i in iter(L):
...   print i

-----------------------------
>>> iterator = iter(L)
>>> t = tuple(iterator) #list
>>> t
(1, 2)
```

Итераторы

```
>>> iterator = iter(L)
>>> a, b = iterator
>>> a, b
(1, 2)

------------------------------------

>>> L = [('a', 1), (5, True), (7, 'abc')]
>>> d = dict(iter(L))
>>> d
{'a': 1, 5: True, 7: 'abc'}
```

Итераторы

```
>>> d = {'Mon': 1, 'Tue': 2, ... , 'Sun': 7}
>>> for key in d:
...     print key, d[key]

Wed 3
Sun 7
Fri 5
Tue 2
Sat 6
Mon 1
Thu 4
```

У словарей есть методы iterkeys, itervalues, iteritems, возвращающие итератор на ключи, значения, кортежи (ключ, значение). Например, for key, value in d.iteritems(): будет последовательно записывать в key ключи, а в value — соответствующие им значения (распаковка кортежа).

Итераторы

```
f = file("text.txt", "r")
for line in f:
    print line
```

В результате получим:

```
string #1

string #2
```

Итераторы

Важно то, что итератор «запоминает» именно позицию единицы данных в изменяющемся списке (в случае со словарём будет сгенерировано исключение):

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it.next()
1
>>> L.insert(0, 0)
>>> L
[0, 1, 2, 3]
>>> it.next()
1 #а не 2, как можно было бы предположить
```

Пользовательские итераторы

Итераторы могут быть бесконечными. Понятно, что такие итераторы не указывают на структуру данных или файл, результат их работы генерируется «на ходу». Для этого обычно используются функции-генераторы (см. далее). При работе с бесконечными итераторами некоторые встроенные функции зацикливаются. Например, max(), min(), list(), tuple(), операторы in и not in, а также цикл for.

Итераторы — пример №1

```
class infinite_iterator:
    def __init__(self):
        self.counter = -1

    def __iter__(self):
        return self

    def next(self):
        self.counter += 1
        return self.counter
```

```
>>> it = infinite_iterator()
>>> it.next()
0 # и так далее до бесконечности
```

Итераторы — пример №2

```
class counter:
    def __init__(self, maximum):
        self.counter = -1
        self.maximum = maximum

    def __iter__(self):
        return self

    def next(self):
        self.counter += 1
        if self.counter < self.maximum:
            return self.counter
        raise StopIteration()
```

Итераторы — пример №2

```
>>> it = counter(4)
>>> for x in it:
...     print x,

0 1 2 3
>>> it.next()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    it.next()
  File "/home/nikita/example.py", line 9, in next
    raise StopIteration()
StopIteration
```

Итераторы — пример №2

```
    def send(self, value):
        self.counter = value - 1
        return self.next()
```

```
>>> c = counter(20)
>>> c.next()
0
>>> c.send(18)
18
>>> c.next()
19
>>> c.next()
StopIteration
```

Итераторы — пример №3

```
class fibnum:
     def __init__(self):
          self.fn1 = 1
          self.fn2 = 1
     def __iter__(self):
          return self
     def next(self):
          oldfn2 = self.fn2
          self.fn2 = self.fn1
          self.fn1 += oldfn2
          return oldfn2
-----------------------------------------
for i in fibnum():
     print i
     if i > 50:
          break
1 1 2 3 5 8 13 21 34 55
```

Итераторы — пример №4

```
class simple_ints:
   def __init__(self):
        self.num = 1
   def __iter__(self):
        return self
   def next(self):
        while True:
           self.num += 1
           for x in xrange(2,self.num/2+1):
               if self.num % x == 0:
                   break
               else:
                   return self.num
```

Итераторы — пример №4

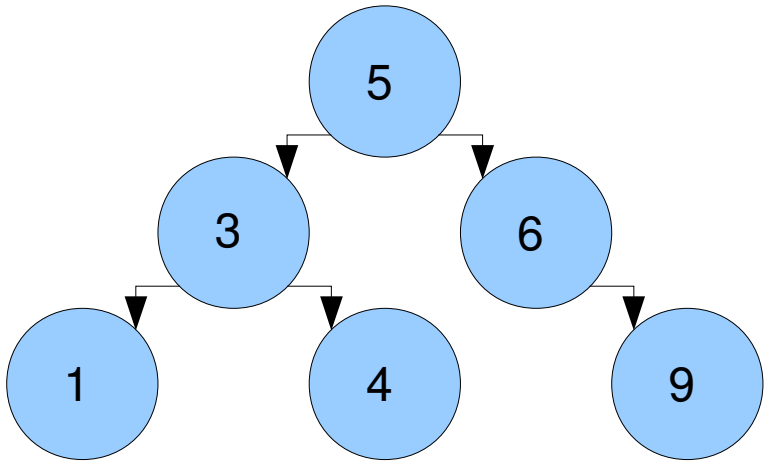
```
>>> s = simple_ints()
>>> for num in s:
...     print num,
...     if num > 100:
...         break

2 3 5 7 11 13 17 19 23 29 31 37
41 43 47 53 59 61 67 71 73 79 83
89 97 101
```

Итераторы — пример №5

```
class node:
    def __init__(self, value=0,
                 left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
    def __iter__(self):
        return inorder(self)

n1 = node(1)
n4 = node(4)
n3 = node(3, n1, n4)
n9 = node(9)
n6 = node(6, right=n9)
root = node(5, n3, n6)
```



Итераторы — пример №5

```
>>> list(root)
[1, 3, 4, 5, 6, 9]
-----------------------------------
class inorder:
    def __init__(self, t):
        self.value = t.value
        self.dir = "left"
        self.left = inorder(t.left)
                        if t.left != None else None
        self.right = inorder(t.right)
                        if t.right != None else None
    def __iter__(self):
         return self
```

Итераторы — пример №5

```
class inorder:
    ****** продолжение ******
    def next(self):
        if self.dir == "left":
            try:
                if self.left is None:
                    raise StopIteration()
                return self.left.next()
            except StopIteration:
                self.dir = "right"
                return self.value
        else:
            if self.right is None:
                raise StopIteration()
            return self.right.next()
```

ВОПРОС !!!

```
class cycle_iter:
    def __init__(self, L):
        self.L = L
        self.index = -1
    def __iter__(self):
        return self
    def next(self):
        if len(self.L) == 0:
            raise StopIteration()
        self.index += 1
        if self.index >= len(self.L):
            self.index = 0
        return self.L[self.index]
```

ВОПРОС !!!

```
>>> L = ["A", "B", "C"]
>>> it = cycle_iter(L)
>>> it.next()
'A'
>>> L.insert(0, "Z")
>>> L
['Z', 'A', 'B', 'C']
>>> it.next()
# ???
...
>>> it.next()
'Z'
>>> del L
>>> it.next()
# ???
```

ОТВЕТ !!!

```
>>> L = ["A", "B", "C"]
>>> it = cycle_iter(L)
>>> it.next()
'A'
>>> L.insert(0, "Z")
>>> L
['Z', 'A', 'B', 'C']
>>> it.next()
>>> 'A' #итератор помнит позицию в списке
...
>>> it.next()
'Z'
>>> del L
>>> it.next()
>>> 'A' #пока хоть одна ссылка жива...
```

Списочные встраивания
List comprehensions

```
>>> L = []
>>> for x in xrange(10):
...     L.append(x)

>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

-------------------------------

>>> [x for x in xrange(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Списочные встраивания

```
>>> L = []
>>> for x in xrange(10):
...     if x % 2 == 0:
...         L.append(x)

>>> L
[0, 2, 4, 6, 8]

----------------------------

>>> [x for x in xrange(10)
          if x % 2 == 0]
[0, 2, 4, 6, 8]
```

Списочные встраивания

```
>>> L = []
>>> for i in xrange(3):
...     for j in xrange(3):
...         L.append(10*i+j)

>>> L
[0, 1, 2, 10, 11, 12, 20, 21, 22]

-----------------------------------

>>> [10*i+j for i in xrange(3)
            for j in xrange(3)]
[0, 1, 2, 10, 11, 12, 20, 21, 22]
```

Списочные встраивания

```
>>> L = []
>>> for i in xrange(4):
...     for j in xrange(4):
...         if i < j:
...             L.append((i, j))

>>> L
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]

----------------------------------------

>>> [(i, j) for i in xrange(4)
            for j in xrange(4) if i < j]
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

Списочные встраивания

[expression for item1 in sequence1 if condition1

for item2 in sequence2 if condition2 …]

Такое выражение действует как вложенные циклы (с необязательными условиями), к результатам которого применяется выражение expression. При этом весь список сохраняется в памяти. Например,

```
>>> t = 100
>>> [x**2 + t for x in range(10)
                  if x % 2 == 0]
[100, 104, 116, 136, 164]
```

Списочные встраивания

Извлечём первые элементы из списков...

```
>>> list_of_lists = [[1,2,3],
                     [4,5,6],
                     [7,8,9]]

>>> for list in list_of_lists:
...     list[0] #кладем его в список

>>> [list[0] for list in list_of_lists]
[1, 4, 7]
```

Списочные встраивания

Извлечём элементы номер 0 и 2 из списков...

```
>>> list_of_lists = [[1,2,3],
                     [4,5,6],
                     [7,8,9]]

>>> for list in list_of_lists:
...     list[0] #кладем его в список
...     list[2] #кладем его в список

#то же самое:
>>> [x for list in list_of_lists
       for x in (list[0], list[2])]
[1, 3, 4, 6, 7, 9]
```

ВОПРОС !!!

Что будет выведено на консоль?

```
>>> list_of_lists = [[1,2,3],
                     [4,5,6],
                     [7,8,9]]

>>> [func  for list in list_of_lists
           for first_x in (list[0],)
           for func in (first_x ** 2,)][1]
```

ОТВЕТ !!!

16

```
>>> list_of_lists = [[1,2,3],
                     [4,5,6],
                     [7,8,9]]

>>> [func  for list in list_of_lists
           for first_x in (list[0],)
           for func in (first_x ** 2,)]
[1, 16, 49]

>>> [list[0] ** 2
     for list in list_of_lists]
[1, 16, 49]
```

Списочные встраивания

Недостатки списочных встраиваний:

- Они расходуют ресурсы;
- Списки всегда конечны.

Хочется итерироваться по списку, который может быть бесконечным, не расходуя ресурсы. Т.е. хотелось бы иметь возможность получить итератор вместо списка. Аналогично тому, как вместо функции range() можно использовать функцию xrange().

Генераторные выражения

Действует аналогично генератору списков, но генерируется не весь список, а итератор на него. Значения будут вычисляться по мере вызова у итератора метода next(). Т. о. генераторные выражения вобрали в себя всё лучшее от итераторов (экономия ресурсов, возможность работы с бесконечными последовательностями) и списочных встраиваний (простой и короткий синтаксис).

Выражение при этом пишется в круглых скобках.

Генераторные выражения

```
>>> line_list = ["   abc ", "ab", "a b "]

>>> [line.strip() for line in line_list]
['abc', 'ab', 'a b']

>>> stripped_iter =
    (line.strip() for line in line_list)
>>> stripped_iter.next()
'abc'
>>> stripped_iter.next()
'ab'
...
```

Списочные встраивания vs Генераторные выражения

- Списочные встраивания нельзя использовать для генерации бесконечных списков, а генераторные выражения — можно.
- Генераторные выражения всегда пишутся в (), но если мы передаем их в функцию, скобки от функции считаются (вторые не ставим):

```
>>> sum(len(line)
        for line in line_list)
13
```

Списочные встраивания vs Генераторные выражения

- При работе с генераторными выражениями надо быть осторожным с переопределением глобальных переменных и функций:

```
>>> def f(x):
...     return x**2
>>> it = (f(x) for x in xrange(2, 10))
>>> it.next()
4
>>> def f(x):
...   return x**3
>>> it.next()
27
```

Ленивые вычисления (ЛВ)

Ленивые вычисления (lazy evaluation) — концепция, согласно которой вычисления следует откладывать до тех пор, пока не понадобится их результат.

Уже изученные примеры:

- xrange();
- итераторы;
- генераторные выражения.

Рассмотрим «ленивость» операторов and и or — если результат логического выражения определяется первым операндом, то второй операнд игнорируется

«Ленивость» and и or

```
>>> def a():
...     print "a"
...     return True
```

```
>>> def b():
...     print "b"
...     return False
```

```
>>> if a() or b():
...     print "or"
a
or

>>> if b() and a():
...     print "and"
b
```

«Ленивость» краткой записи if

```
>>> def f():
...     print "f"
...     return "f"
```

```
>>> def g():
...     print "g"
...     return "g"
```

```
>>> f() if True else g()
f
'f'
>>> f() if False else g()
g
'g'
```

ВОПРОС !!!

```
def f(s):
    return (s == 0 and "A") or
           (s == 1 and "B") or
           ("C")


>>> f(0)
# ???
>>> f(1)
# ???
>>> f(2)
# ???
```

OTBET !!!

```
def f(s):
    return (s == 0 and "A") or
           (s == 1 and "B") or
           ("C")

>>> f(0)
'A'
>>> f(1)
'B'
>>> f(2)
'C'
```

Генераторы — пример №1

- это функции, возвращающие генераторные объекты, реализующие интерфейс итераторов (и не только).

```
>>> def infinite_generator():
...     i = 0
...     while (True):
...         yield i
...         i += 1

>>> it = infinite_generator();
>>> it.next()
0
>>> it.next()
1
>>> max(it) #выполнения этой строки вы
не дождетесь
```

Генераторы — пример №2

```
>>> def counter(maximum):
...     for i in xrange(maximum):
...         yield i

>>> gen_obj = counter(10)
>>> gen_obj.next()
0
...
>>> gen_obj.next()
9
>>> gen_obj.next()
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    gen_obj.next()
StopIteration
```

Генераторы

```
>>> for i in counter(10):
...     print i

>>> a, b, c = counter(3)
```

Генераторы выбрасывают исключение StopIteration и прекращают итерирование генераторного объекта, когда:

- выполняется инструкция return, в генераторах она не может содержать возвращаемое значение (например, return 5);
- выполнение фукнции оказывается «за последней строчкой» генератора, т.е. инструкции закончились;
- в генераторе сгенерировано исключение StopIteration.

Генераторы

Генераторный объект – это больше, чем итератор. Он содержит методы:

- `next()`;
- `send(value)` - задать значение генераторному объекту;
- `throw(ex_type)` - выбросить исключение в генераторе (на месте инструкции `yield`);
- `close()` - генерирует исключение `GeneratorExit` внутри генератора. Генератор может отреагировать на это, сгенерировав исключение `StopIteration` или `GeneratorExit`. Отлавливание исключения и генерация чего-либо другого приведет к `RuntimeError`! Этот метод вызывается сборщиком мусора перед уничтожением генераторного объекта.

Генераторы — пример №2

```
>>> def counter(maximum):
...     i = 0
...     while i < maximum:
...         try:
...             val = (yield i)
...             if val is not None:
...                 i = val
...             else:
...                 i += 1
...         except Exception:
...             i = -10
...
```

```
>>> c = counter(5)
>>> c.next()
0
>>> c.next()
1
>>> c.send(4)
4
```

```
>>> c.throw(Exception)
-10
>>> c.next()
-9
>>> c.close()
>>> c.next() #StopIteration
```

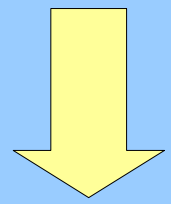
Генераторы — пример №2

Программа

```
>>> c = counter(5)
>>> c.next()
0
>>> c.next()
1
>>> c.send(4)
4
>>> c.throw(Exception)
-10
>>> c.next()
-9
>>> c.close()
>>> c.next()
~~~some info~~~
StopIteration
```

«Сопрограмма»-генератор

```
>>> def counter(maximum):
...     i = 0
...     while i < maximum:
...         try:
...             val = (yield i)
...             if val is not None:
...                 i = val
...             else:
...                 i += 1
...         except Exception:
...             i = -10
```



Генераторы — пример №3

```
def fibnum():
    fn1 = 1
    fn2 = 1
    while True:
        yield fn2
        (fn1, fn2) = (fn1 + fn2, fn1)

-----------------------------------

for x in fibnum():
   print x,
   if x > 50:
      break

1 1 2 3 5 8 13 21 34 55
```

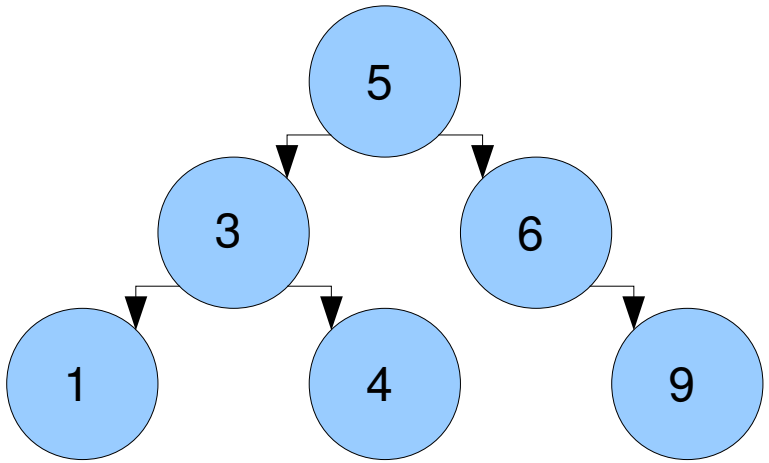
Генераторы — пример №4

```
def simple_ints():
    num = 1
    while True:
        num += 1
        for x in xrange(2, num/2+1):
            if num % x == 0:
                break
        else:
            yield num
-------------------------------------
for x in simple_ints():
   print x,
   if x > 100:
      break;
2 3 5 7 11 13 17 19 23 29 31 37 41 43
47 53 59 61 67 71 73 79 83 89 97 101
```

Итераторы — пример №5

```
class node:
    def __init__(self, value=0,
                 left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
    def __iter__(self):
        return inorder(self)

n1 = node(1)
n4 = node(4)
n3 = node(3, n1, n4)
n9 = node(9)
n6 = node(6, right=n9)
root = node(5, n3, n6)
```



Генераторы — пример №5

```
>>> list(root)
[1, 3, 4, 5, 6, 9]
-------------------------------

>>> def inorder(t):
...   if t:
...     for x in inorder(t.left):
...       yield x
...     yield t.value
...     for x in inorder(t.right):
...       yield x
```

ВОПРОС !!!

Найдите логическую ошибку.

```
import random
def limitation(limit=0, iters=10):
    i = 1
    for x in xrange(iters):
        j = random.random()
        if limit <= j < i:
            yield j
            i = j


>>> it = limitation(0.05, 5)
>>> list(it)
[0.8751810341356698 3,
 0.4438514385887991 6, …] # 5 приближений
```

ОТВЕТ !!!

```
import random
def limitation(limit=0, iters=10):
    i = 1
    while iters > 0:
        j = random.random()
        if j < i and j >= limit:
            yield j
            i = j
            Iters -= 1

#иначе число итераций может быть
#меньше пяти
```

Модуль itertools

Содержит функции для:

1) создания итераторов (в том числе - на основе имеющихся итераторов);
2) обработки итерируемых элементов;
3) фильтрования возвращаемых итератором значений;
4) группировки возвращаемых итератором значений.

Itertools

```
>>> import itertools
>>> it = itertools.count(10)
>>> it.next()
10
>>> it.next()
11

>>> it = itertools.repeat("abc") #n
>>> it.next()
'abc'
>>> it.next()
'abc'
```

Itertools

- cycle(list); //итерация по кругу
- chain(lists); //сложение списков
- izip(lists); //умножение списков
- islice(list, begin, end, step); //срез
- tee(list):

```
>>> it1, it2 = itertools.tee( xrange(10) )
>>> list(it1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(it2)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Itertools

```
>>> city_list = [("Moscow", "Russia"),
                 ("St. Petersburg", "Russia"),
                     ("New York", "USA"),
                     ("Paris", "France")]
>>> def get_country((city, state)):
...    return state
>>> it_country = itertools.groupby
                     (city_list, get_country)
>>> (country, it_city) = it_country.next()
>>> it_city.next()
('Moscow', 'Russia')
#при каждом вызове next у it_country будем
#получать новый it_city
```

ВОПРОСЫ ?

СПАСИБО
ЗА
ВНИМАНИЕ !!!