Лекция №6

Специальные конструкции Python

- Итераторы
- Генераторы
- Менеджер контекста
- Практика

Обход последовательностей

Для обхода последовательностей используется оператор цикла **for**

```
filenames = ["mytext.txt", "cv.doc", "myphoto.jpg"]
for fname in filenames:
    print(fname)
```

Для обхода последовательностей в стиле C/C++ (с итерируемой переменной) используется встроенная функция enumerate:

```
filenames = ["mytext.txt", "cv.doc", "myphoto.jpg"]

# start - optional parameter (by default enumeration starts with 0)
for i, fname in enumerate(filenames, start=1):
    print("reading file {} - {}".format(i, fname))
```

Итераторы

Итераторы — это специальные объекты, предоставляющие последовательный доступ к данным из контейнера. Любой объект, поддерживающий интерфейс итератора, имеет метод **__next__()**, позволяющий переходить на следующую ступень вычисления. Чтобы получить итератор по объекту (например, по списку), к нему нужно применить функцию **iter()**. Цикл **for** тоже задействует итератор, только автоматически.

Итераторы

```
>>> l = [1, 2, 3, 4]
>>> l
[1, 2, 3, 4]
>>> li = iter(l)
>>> li
<list_iterator object at 0x7f45cefa5f98>
>>> next(li)
1
>>> next(li)
2
>>> next(li)
3
>>>
next(li)
4
>>> next(li)
Traceback (most recent call last):
    File "<input>", line 1, in <module>
StopIteration
```

Итераторы

```
# Можно создать свой итератор
# для этого нужно лишь обеспечить интерфейс итератора:
# то есть определить методы __iter__ и __next__
>>> class Fibonacci:
...     def __init__(self, N):
...         self.n, self.a, self.b, self.max = 0, 0, 1, N
...     def __iter__(self):
...         # сами себе итератор: в классе есть метод next()
...         return self
...     def __next__(self):
...         if self.n < self.max:
...             a,self.n,self.a,self.b = self.a, self.n+1, self.b, self.a+self.b
...             return a
...         else:
...             raise StopIteration
...
...
for i in Fibonacci(10):
... print(i, end=' ')
...
0 1 1 2 3 5 8 13 21 34
```

Генераторы

Генераторы — это итерируемые функциональные объекты. Фактически это функции, которые сохраняют контекст вычислений между вызовами, возвращая при каждом вызове результат выполнения очередной итерации. Простейший генератор можно представить как цикл внутри функции, и при каждом вызове этой функции выполняется только одна очередная итерация этого цикла.

```
>>> def gen(n):
        i = 0
        while i < n:
            yield i
            i += 1
>>> g = gen(3)
>>> g
<generator object gen at 0x02A9DEE0>
>>> next(g)
0
>>> next(g)
1
```

Генераторы

Yield - это ключевое слово, которое используется примерно как **return** в обычных функциях. Отличие состоит в том, что генератор сохраняет свое состояние и при следующем вызове выполнение генератора продолжится с кода, находящегося непосредственно после **yield**.

При обращении к генераторной функции возвращается генераторный объект, выполнение функции при этом не начинается. При первом вызове функции **next** (в Python3 **next** – встроенная функция, которая вызывает метод **__next__** генераторного объекта, в Python2 **next** – метод генераторного объекта), функция начинает выполнение, пока не достигнет **yield**, и возвращает результат выражения **yield**.

Все это позволяет организовывать т.н. "ленивые" вычисления. То есть когда код в функции выполняется не сразу, а по требованию.

Генераторы

Данный пример поясняет порядок итерирования генераторных объектов:

```
>>> def gen_fun():
...     print("begin")
        for i in [1, 2]:
            print("before yield", i)
            yield i
            print("after yield", i)
        print("end")
>>> f = gen_fun()
>>> f
<generator object gen_fun at 0x00000221C73F5660>
>>> next(f)
begin
before yield 1
1
>>> next(f)
after yield 1
before yield 2
2
>>> next(f)
after yield 2
end
Traceback (most recent call last):
    File "<input>", line 1, in <module>
StopIteration
```

Генераторы

Еще один пример организации "ленивых" вычислений:

```
>>> def lazy_range():
...     for i in [1, 2, 3, 4]:
...         print('now i process {}'.format(i))
...         yield i
...
>>> l = lazy_range()
>>> l
<generator object lazy_range at 0x7f45cef68a40>
>>> next(l)
now i process 1
1
>>> next(l)
now i process 2
2
>>> next(l)
now i process 3
3
>>> next(l)
now i process 4
4
>>> next(l)
Traceback (most recent call last):
    File "<input>", line 1, in <module>
StopIteration # как видно print выполняется не сразу, а по частям
```

Генераторы

Также генераторный объект можно обойти как последовательность с помощью цикла **for**.

```
>>> l = lazy_range()
>>> l
<generator object lazy_range at 0x7f45cef68a40>
>>> for i in l:
...     print(i)
...
now i process 1
1
now i process 2
2
now i process 3
3
now i process 4
4
```

Генераторы

Для задания генераторного объекта можно использовать круглые скобки. При указании же квадратных скобок создается итераторный объект.

```
>>> mygen = (x*x for x in range(3))
>>> mygen
<generator object <genexpr> at 0x7f45cefa3048>
>>> for i in mygen:
...     print(i)
...
0
1
4
>>> mygen
<generator object <genexpr> at 0x7f45cefa3048>
>>> for i in mygen:
...     print(i)
...
>>> mygen2 = (x*x for x in range(3))
>>> list(mygen2)
[0, 1, 4]
>>> list(mygen2)
[]
```

Менеджер контекста (context manager)

Контекстные менеджеры это специальные конструкции, которые представляют из себя блоки кода, заключенные в инструкцию with.

```
# Простой пример - работа с файлом:
# после работы с файлом нужно позаботиться о его закрытии

>>> fp = open("./file.txt", "w")
>>> fp.write("Hello, World")
>>> fp.close() # без with файл нужно закрывать самому


# с with файл закроется сам, как только уйдет из его блока
>>> with open("./file.txt", "w") as fp:
...     fp.write("Hello, World")
```

Менеджер контекста (context manager)

Создать свой менеджер контекста достаточно просто. Надо всего лишь определить методы **__enter__** и **__exit__** соответствующего класса.

```
>>> class TestManager:
...     def __enter__(self):
...         print('Starting code inside manager')
...     def __exit__(self, exc_type, exc_val, exc_tb):
...         print('Code completed. Error info: type - {}'.format(exc_type))
>>>
>>> with TestManager():
...     1 + 1
...
Starting code inside manager
2
Code completed. Error info: type - None
```

Практика

1. В чем ошибка в данной программе и как ее можно исправить?

```
def chargen():
    while True:
        for c in '0123456789':
            yield c
words = [c+c for c in chargen()][:10]
```

2. Функции на вход подаётся последовательность чисел source и множитель m. На выходе функции ожидается новая последовательность на основе source, где каждый член был умножен на m. Если source не был указан, то берётся последовательность [1,2,3]. Укажите ошибки, допущенные в данной функции, и предложите свою реализацию.

```
def multiplier(m=1, source=[1,2,3]):
    result = source
    for i, x in enumerate(source):
        result[i] *= m
    return result
>>> multiplier(5)
[5, 10, 15]
>>> multiplier(12, [1,2])
[12, 24]
```

3. Напишите свой менеджер контекста, замеряющий и показывающий время исполнения кода внутри него.

Практика*

4. Часто задача программиста заключается в том, чтоб найти в документации готовую функцию, которая реализует необходимое решение. Данное задании предполагает самостоятельное изучение документации к библиотеке itertools (это набор готовых итераторов), чтобы подобрать те функции, которые дадут правильные ответы на следующие вопросы (иногда надо будет добавить свои аргументы при вызове функций помимо тех, что указаны в задании):
   - Функция должна принимать три массива ([1, 2, 3], [4, 5], [6, 7]), а вернуть лишь один массив ([1, 2, 3, 4, 5, 6, 7])
   - Функция принимает массив (['hello', 'i', 'write', 'cool', 'code']) и возвращает массив из элементов, у которых длина не меньше пяти (['hello', 'write'])
   - Функция выдает на строку 'password' все возможные комбинации вида ([('p', 'a', 's', 's'), ('p', 'a', 's', 'w'), ('p', 'a', 's', 'o'), ...)

   Требуется написать код, который использует указанные входные данные и выводит на экран возвращаемое значение.

   Помните, что функции могут возвращать генератор, который нужно "развернуть" для вывода на экран.