Лекция 8: Итераторы, генераторы и модуль `itertools`

Сергей Лебедев

sergei.a.lebedev@gmail.com

26 октября 2015 г.

Итераторы

Цикл for и модуль dis

- Напоминание: оператор **for** в Python работает с любой последовательностью.

```
>>> import dis
>>> dis.dis("for x in xs: do_something(name)")
      0 SETUP_LOOP              24 (to 27)
      3 LOAD_NAME                0 (xs)
      6 GET_ITER
>>    7 FOR_ITER                16 (to 26)
     10 STORE_NAME              1 (x)
     [...]
>>   26 POP_BLOCK
>>   27 LOAD_CONST              0 (None)
     30 RETURN_VALUE
```

- Инструкция `GET_ITER` вызывает у аргумента оператора **for** метод `__iter__`, который возвращает *итератор*.
- Инструкция `FOR_ITER` вызывает метод `__next__` у итератора до тех пор, пока не будет поднято исключение **StopIteration**.

- Протокол итераторов состоит из двух методов:
  - Метод `__iter__` возвращает экземпляр класса, реализующего протокол итераторов, например, `self`.
  - Метод `__next__` возвращает следующий по порядку элемент итератора. Если такого элемента нет, то метод должен поднять исключение **`StopIteration`**.
- Важный инвариант метода `__next__`: если метод поднял исключение **`StopIteration`**, то все последующие вызовы метода `__next__` тоже должны поднимать исключение.
- В отличие от, например, Java в Python *iterator* также является *iterable*.

- Для коллекций обычно нет смысла реализовывать протокол итераторов целиком, достаточно реализовать только метод `__iter__`.
- Иногда элементы коллекции можно перечислить более чем одним способом. В этом случае удобно реализовывать дополнительные методы, возвращающие итераторы:

```
class BinaryTree:
    def __iter__(self):
        return self.inorder_iter()

    def preorder_iter(self):
        # ...

    def inorder_iter(self):
        return InOrderIterator(self)

    def postorder_iter(self):
        # ...
```

- У функции `iter` две формы вызова:
  - принимает итератор и вызывает у него метод `__iter__`,
  - принимает функцию и терминальное значение и вызывает функцию до тех пор, пока она не вернёт нужное значение[1]:

```
from functools import partial
with open(path, "rb") as handle:
    read_block = partial(handle.read, 64)
    for block in iter(read_block, ""):
        do_something(block)
```

- Функция `next` принимает итератор и вызывает у него метод `__next__`. Можно также указать значение, которое нужно вернуть в случае возникновения исключения **`StopIteration`**:

```
>>> next(iter([1, 2, 3]))
1
>>> next(iter([]), 42)
42
```

[1] http://bit.ly/beautiful-python

- Напоминание:

```
for x in xs:
    do_something(x)
```

- Процесс исполнения оператора **for** можно концептуально записать так:

```
it = iter(xs)
while True:
    try:
        x = next(it)
    except StopIteration:
        break
    do_something(x)
```

Протокол итераторов и операторы in и not in

- Операторы **in** и **not in** используют “магический” метод `__contains__`, который возвращает **True**, если переданный элемент содержится в экземпляре класса.
- По умолчанию метод `__contains__` реализован через протокол итераторов:

```
class object:
    # ...

    def __contains__(self, target):
        for item in self:
            if item == target:
                return True
        return False
```

- Пример:

```
>>> id = Identity()
>>> 5 in id        # ≡ id.__contains__(5)
True
>>> 42 not in id   # ≡ not id.__contains__(42)
True
```

Протокол итераторов и реализация “по умолчанию”

- В Python предусмотрен упрощённый вариант реализации протокола итераторов с использованием метода `__getitem__`.
- Метод `__getitem__` принимает один аргумент — индекс элемента в последовательности и
  - либо возвращает элемент, соответствующий индексу,
  - либо поднимает **`IndexError`**, если элемента с таким индексом нет.
- Пример:

```
>>> class Identity:
...     def __getitem__(self, idx):
...         if idx > 5:
...             raise IndexError(idx)
...         return idx
...
>>> list(Identity())
[0, 1, 2, 3, 4, 5]
```

“Семантика” упрощённого протокола итераторов: `seq_iter`

```
class seq_iter:
    def __init__(self, instance):
        self.instance = instance
        self.idx = 0

    def __iter__(self):
        return self

    def __next__(self):
        try:
            res = self.instance[self.idx]
        except IndexError:
            raise StopIteration

        self.idx += 1
        return res
```

“Семантика” упрощённого протокола итераторов: `object`

```
class object:
    # ...

    def __iter__(self):
        if not hasattr(self, "__getitem__"):
            cls = self.__class__
            msg = "{} object is not iterable"
            raise TypeError(msg.format(cls.__name__))
        return seq_iter(self)
```

- В Python *iterator* также является *iterable*.
- Итератор — это экземпляр класса, который реализует два метода `__init__` и `__next__`.
- Альтернативно можно воспользоваться реализацией этих методов по умолчанию и определить метод `__getitem__`.
- Протокол итераторов используется:
  - оператором **`for`**,
  - операторами **`in`** и **`not in`**.
- Протокол итераторов реализуется всеми встроенными коллекциями, а также, например, файлами и объектами типа `map`, `filter` и `zip`.

Генераторы

- Генератор — это функция, которая использует не только оператор **return**, но и оператор **yield**.
- В результате выполнения оператора **yield** работа функции **приостанавливается**, а не прерывается, как при использовании оператора **return**.
- Пример:

```
>>> def g():
...     print("Started")
...     x = 42
...     yield x
...     x += 1
...     yield x
...     print("Done")
```

```
>>> type(g)
<class 'function'>
>>> gen = g()
>>> type(gen)
<class 'generator'>
>>> next(gen)
Starting ...
42
>>> next(gen)
43
```

[2] http://python.org/dev/peps/pep-0255

Примеры генераторов: unique

```
>>> def unique(iterable, seen=None):
...     seen = set(seen or [])
...     for item in iterable:
...         if item not in seen:
...             seen.add(item)
...             yield item
...
>>> xs = [1, 1, 2, 3]
>>> unique(xs)
<generator object unique at 0x1027c5798>
>>> list(unique(xs))
[1, 2, 3]
>>> 1 in unique(xs)
True
```

```
>>> def map(func, iterable, *rest):
...     for args in zip(iterable, *rest):
...         yield func(*args)
...
>>> xs = range(5)
>>> map(lambda x: x * x, xs)
<generator object map at 0x103122510>
>>> list(map(lambda x: x * x, xs))
[0, 1, 4, 9, 16]
>>> 9 in map(lambda x: x * x, xs)
True
```

Примеры генераторов: chain

```
>>> def chain(*iterables):
...     for iterable in iterables:
...         for item in iterable:
...             yield item
...
>>> xs = range(3)
>>> ys = [42]
>>> chain(xs, ys)
<generator object chain at 0x10311d708>
>>> list(chain(xs, ys))
[0, 1, 2, 3, 42]
>>> 42 in chain(xs, ys)
True
```

Примеры генераторов: count и enumerate

```
>>> def count(start=0):
...     while True:
...         yield start
...         start += 1
...
>>> next(count())
0
>>> counter = count()
>>> next(counter)
0
>>> next(counter)
1
>>> def enumerate(iterable, start=0):
...     pass  # как?
...
>>> next(enumerate(count(42)))
(0, 42)
```

- Основное правило переиспользования генераторов: **не делайте этого**.

```
>>> def g():
...     yield 42
...
>>> gen = g()
>>> list(gen)
[42]
>>> list(gen)  # не тут-то было!
[]
```

- Если вы хотите переиспользовать генератор, подумайте ещё раз.
- Если вы уверены, что без переиспользования не обойтись, воспользуйтесь функцией `tee` из модуля `itertools`.

- Генераторы позволяют компактно реализовывать метод `__iter__` у коллекций.
- Рассмотрим уже знакомый нам класс бинарного дерева:

```
class BinaryTree:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left, self.right = left, right

    def __iter__(self):  # inorder
        for node in self.left:
            yield node.value
        yield self.value
        for node in self.right:
            yield node.value
```

- Плюс генераторов в том, что они позволяют обойтись без лишних классов, например, `InOrderIterator`.

- Напоминание: в Python есть генераторы списков, множеств и словарей.
- Выражения-генераторы работают аналогичным образом, но не порождают коллекцию в процессе работы:

```
>>> gen = (x ** 2 for x in range(10**42) if x % 2 == 1)
>>> gen
<generator object <genexpr> at 0x10311d708>
>>> next(gen)
1
>>> list(filter(lambda x: x % 2 == 1,
...                 (x ** 2 for x in range(10))))
[1, 9, 25, 49, 81]
```

- Если выражение-генератор — единственный аргумент функции, скобки можно опустить:

```
>>> sum(x ** 2 for x in range(10) if x % 2 == 1)
165
```

- Оператор **yield** можно использовать как выражение:

```
>>> def g():
...     res = yield     # точка входа 1
...     print("Got {!r}".format(res))
...     res = yield 42  # точка входа 2
...     print("Got {!r}".format(res))
...
>>> gen = g()
>>> next(gen)  # "промотаем" до первого yield
>>> next(gen)  # "промотаем" до второго yield
Got 'None'
42
>>> next(gen)  # выполним оставшуюся часть генератора
Got 'None'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- На первый взгляд выражение **yield** выглядит бесполезно, но первое впечатление обманчиво.

Интерфейс генераторов: send

- Метод `send` **возобновляет** выполнение генератора и “отправляет” свой аргумент в следующий **`yield`**:

```
>>> gen = g()
>>> gen.send("foobar")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send [...] to a just-started generator
```

- Чтобы инициализировать генератор нужно “отправить” ему **`None`**. Функция `next` делает ровно это:

```
>>> gen = g()
>>> next(gen)
```

- Результатом метода `send` является следующее значение генератора или исключение **`StopIteration`**, если такого исключения нет.

```
>>> gen = g()
>>> gen.send(None)  # ≡ next(gen)
>>> gen.send("foobar")
Got 'foobar'
42
```

Интерфейс генераторов: throw

- Метод `throw` поднимает переданное исключение в месте, где генератор приостановил исполнение и возвращает следующее значение генератора.

```
>>> def g():
...     try:
...         yield 42
...     except Exception as e:
...         yield e
...
>>> gen = g()
>>> next(gen)
42
>>> gen.throw(ValueError, "something is wrong")
ValueError('something is wrong',)
>>> gen.throw(RuntimeError, "another error")
???
```

- Если генератор не обработал брошенное в него исключение, то выполнение генератора прекращается и исключение передаётся наверх по стеку вызовов.

Интерфейс генераторов: `close`

- Метод `close` поднимает специальное исключение **`GeneratorExit`** в месте, где генератор приостановил исполнение:

```
>>> def g():
...     try:
...         yield 42
...     finally:
...         print("Done")
...
```

```
>>> gen = g()
>>> next(gen)
42
>>> gen.close()
Done
```

- Если всё хорошо, то метод `close` завершает работу генератора и ничего не возвращает.
- Что может пойти не так? Генератор может обработать исключение **`GeneratorExit`** и поднять другое исключение.

- Сопрограмма — это программа, которая может иметь больше одной точки входа, а также поддерживает остановку и продолжение с сохранением состояния.
- Звучит как определение генератора наоборот:

```
>>> def grep(pattern):
...     print("Looking for {!r}".format(pattern))
...     while True:
...         line = yield
...         if pattern in line:
...             print(line)
...
>>> gen = grep("Gotcha!")
>>> next(gen)
Looking for 'Gotcha!'
>>> gen.send("This line doesn't have \
...           what we're looking for")
>>> gen.send("This one does. Gotcha!")
This one does. Gotcha!
```

[3] http://dabeaz.com/coroutines

- Прежде, чем начать работать с сопрограммой, её нужно инициализировать с помощью вызова функции `next`.
- Объявим декоратор `coroutine`, который скроет эту деталь реализации:

```
>>> def coroutine(g):
...     @functools.wraps(g)
...     def inner(*args, **kwargs):
...         gen = g()
...         next(gen)
...         return gen
...     return inner
...
>>> grep = coroutine(grep)
>>> gen = grep("Gotcha!")
>>> gen.send("One more line for ya!")
```

- Зачем это всё нужно? Ответ в домашнем задании.

Не сейчас

[4] http://dabeaz.com/coroutines

- Оператор **`yield from`** позволяет делегировать выполнение другому генератору:

```
def chain(*iterables):
    for iterable in iterables:
        yield from iterable
```

- Любые вызовы методов `send` и `throw` у родительского генератора будут переданы вложенному генератору без изменений.

[5] http://python.org/dev/peps/pep-0380

Оператор return и исключение StopIteration

- Кроме оператора **yield** в теле генератора можно использовать оператор **return**.
- На человеческом языке использование **return** означает: «У меня больше нет элементов, извини, возьми лучше вот это.»

```
>>> def g():
...     yield 42
...     return []  # держи!
...
>>> gen = g()
>>> next(gen)
42
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: []  # вот и оно.
```

return ≠ raise StopIteration

- Несмотря на схожесть, использование оператора **return** в генераторе не эквивалентно поднятию исключения **StopIteration**.
- Контрпример:

```
def g():
    try:
        yield 42
        raise StopIteration([])  # ≠ return []
    except Exception as e:
        pass
```

- Оператор **`yield from`**, как и оператор **`yield`**, можно использовать в качестве выражения.
- При этом значением выражения **`yield from`** будет значение атрибута `value` у поднятого вложенным генератором исключения **`StopIteration`**:

```
>>> def f():
...     yield 42
...     return []
...
>>> def g():
...     res = yield from f()
...     print("Got {!r}".format(res))
...
>>> gen = g()
>>> next(gen)
42
>>> next(gen, None)
Got []
```

- Протокол менеджеров контекста требует реализации двух методов: `__enter__` и `__exit__`,
- Если мы хотим, чтобы у менеджера было какое-то состояние, то мы вынуждены также добавить метод `__init__`.
- В итоге получаем:

```
class cd:
    def __init__(self, path):
        self.path = path

    def __enter__(self):
        self.saved_cwd = os.getcwd()
        os.chdir(self.path)

    def __exit__(self, *exc_info):
        os.chdir(self.saved_cwd)
```

Менеджеры контекста и генераторы: @contextmanager

- Декоратор `contextmanager` из модуля `contextlib` принимает генератор специального вида и строит по нему менеджер контекста.

```
from contextlib import contextmanager

@contextmanager
def cd(path):                    # __init__
    old_path = os.getcwd()       # __enter__
    os.chdir(path)
    try:
        yield                    # ---------
    finally:
        os.chdir(old_path)       # __exit__
```

- Генераторы позволяют сократить количество синтаксического шума при реализации менеджеров контекста.

Ещё один пример использования @contextmanager

Метод `__enter__`, построенный декоратором `contextmanager`, возвращает аргумент оператора **`yield`**:

```
import tempfile
import shutil

@contextmanager
def tempdir():                          # __init__
    outdir = tempfile.mkdtemp()         # __enter__
    try:
        yield outdir                    # ---------
    finally:
        shutil.rmtree(outdir)           # __exit__

with tempdir() as path:
    print(path)                         # ==> /tmp/tmpvfzsmvsv
```

- Генератор в Python — это функция, которая использует операторы `yield` или `yield from`.
- В мире Python генераторы вездесущи не менее, чем любимые всеми декораторы.
- Мы поговорили о том, что генераторы можно использовать
  - как итераторы,
  - как сопрограммы,
  - как легкие потоки,
  - для компактной реализации менеджеров контекста.

Модуль itertools

- Функция `islice` обобщает понятие слайса на произвольный итератор:

```
>>> from itertools import islice
>>> xs = range(10)
>>> list(islice(xs, 3))          # ≡ xs[:3]
[0, 1, 2]
>>> list(islice(xs, 3, None))    # ≡ xs[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> list(islice(xs, 3, 8, 2))    # ≡ xs[3:8:2]
[3, 5, 7]
```

- Прелесть функций из модуля `itertools` в том, что с помощью них легко выражаются самые разнообразные операции над последовательностями.

Вопрос

Как будет выглядеть функция `drop`, “выкидывающая” префикс длины `n` из переданного ей итератора?

- Для удобства реализуем родственника функции drop: функцию take, которая строит список из более, чем n первых элементов переданного ей итератора.

```
>>> def take(n, iterable):
...     return list(islice(iterable, n))
...
>>> list(take(range(10), 3))
[0, 1 2]
```

- Названия бесконечных итераторов говорят сами за себя:

```
>>> from itertools import count, cycle, repeat
>>> take(3, count(0, 5))
[0, 5, 10]
>>> take(3, cycle([1, 2, 3]))
[1, 2, 3]
>>> take(3, repeat(42))
[42, 42, 42]
>>> take(3, repeat(42, 2))  # не совсем ∞
[42, 42]
```

Модуль itertools: dropwhile и takewhile

- Функции `dropwhile` и `takewhile` обобщают логику функций `drop` и `take` на произвольный предикат.
- Обратите внимание, что обе функции возвращают **итератор**, а не список, как реализованная нами функция `take`:

```
>>> from itertools import dropwhile, takewhile
>>> list(dropwhile(lambda x: x < 5, range(10)))
[5, 6, 7, 8, 9]
>>> it = takewhile(lambda x: x < 5, range(10))
>>> it
<itertools.takewhile object at 0x1022ca748>
>>> list(it)
[0, 1, 2, 3, 4]
```

- В модуле `itertools` реализован уже знакомый нам генератор `chain`, который конкатенирует произвольное число итераторов:

```
>>> from itertools import chain
>>> take(5, chain(range(2), range(5, 10)))
[0, 1, 5, 6, 7]
```

- Сконкатенировать итератор итераторов (!) можно с помощью метода `chain.from_iterable`:

```
>>> it = (range(x, x ** x) for x in range(2, 4))
>>> take(5, chain.from_iterable(it))
[2, 3, 3, 4, 5]
```

Вопрос

Чем `chain.from_iterable(it)` отличается от `chain(*it)`?

- Функция `tee` создаёт `n` независимых копий переданного ей итератора:

```
>>> from itertools import tee
>>> it = range(3)
>>> a, b, c = tee(it, 3)
>>> list(a), list(b), list(c)
([0, 1, 2], [0, 1, 2], [0, 1, 2])
```

- Использовать `it` после копирования не рекомендуется, потому что в этом случае скопированные итераторы `a, b, c` могут пропустить элемент:

```
>>> it = iter(range(3))
>>> a, b = tee(it, 2)
>>> used = list(it)
>>> list(a), list(b)
([], [])
```

Вопрос

Что изменится, если убрать вызов функции `iter` из второго примера?

Модуль itertools: комбинаторные итераторы

В модуле itertools в виде итераторов реализованы полезные комбинаторные операции, например:

- декартово произведение итераторов,

```
>>> list(itertools.product("AB", repeat=2))
[('A', 'A'), ('A', 'B'), ('B', 'A'), ('B', 'B')]
>>> list(itertools.product("AB", repeat=3))
[('A', 'A', 'A'), ('A', 'A', 'B'), ('A', 'B', 'A'), ...]
```

- перестановки элементов итератора,

```
>>> list(itertools.permutations("AB"))
[('A', 'B'), ('B', 'A')]
```

- сочетания (с повторениями и без) из элементов итератора.

```
>>> from itertools import combinations, \
...     combinations_with_replacement
>>> list(combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(combinations_with_replacement("ABC", 2))
[('A', 'A'), ('A', 'B'), ('A', 'C'),
 ('B', 'B'), ('B', 'C'), ('C', 'C')]
```

```
def build_graph(words, mismatch_percent):
    g = ...
    n_words = len(words)
    for u, v in itertools.combinations(range(n_words), 2):
        if len(words[u]) != len(words[v]):
            continue

        distance = hamming(words[u], words[v])
        # ...

    return g
```

Вопрос

Выглядит неплохо, но можно лучше. Как?

- Модуль `itertools` предоставляет обширный набор компонент для реализации операций над последовательностями.
- Мы обсудили:
  - `islice`,
  - бесконечные итераторы `count`, `cycle`, `repeat`,
  - `chain`,
  - `tee`,
  - комбинаторные итераторы `product`, `permutations`, `combinations` и `combinations_with_replacement`.