MIPT
MOSCOW INSTITUTE
OF PHYSICS AND TECHNOLOGY

АТП кафедра Алгоритмов
и Технологий
Программирования

Итераторы и генераторы.

Емельянов А. А.
login-const@mail.ru

2017

https://github.com/king-menin/python-course

Как работает цикл for

```
for index in range(10):
    pass

for k, v in {1: 1, 2: 2}.items():
    pass

try:
    for line in open("test.py"):
        pass
except Exception:
    pass

for sym in "some letters":
    pass
```

Итераторы

- **Итераторы** — это специальные объекты, предоставляющие последовательный доступ к данным из контейнера. Контейнер - любой объект, содержащий внутри себя другие объекты - например, список, кортеж, словарь и т.п.

- **Протоколы**
  - *Iterable*: должен быть определён метод __iter__, возвращающий итератор, экземпляр класса у которого определён метод __next__ (например self), этот метод будет вызываться пока он не вернёт исключение StopIteration.
  - *Sequence*: должен быть определен метод __getitem__ возвращающий элемент по индексу, или поднимающий исключение IndexError.

Функции iter и next

- В обоих случаях (протоколах итерирования) от объекта можно будет позвать функцию iter(object) возвращает итератор по object.

- От объекта, возвращаемого iter можно вызвать функцию next(it) возвращает следующее значение.

```
els = list(range(5))
it_els = iter(els)
# it_els = els.__iter__()
```

```
type(els), type(it_els), type(iter(it_els))
```

```
(list, list_iterator, list_iterator)
```

```
next(it_els), it_els.__next__(), next(it_els), next(it_els)
```

```
(0, 1, 2, 3)
```

```
print(*iter(it_els))
```

```
4
```

```
next(it_els)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-39-f4c8507427d2> in <module>()
----> 1 next(it_els)

StopIteration:
```

Собственные итерируемые объекты

- Метод __iter__ должен возвращать итератор. Может возвращать self, то есть итерируемое может быть иетратором у самого себя.

- Итератор должен иметь метод __next__ – возвращающий следующее значение в последовательности, или выкидывающий StopIteration если последовательность кончилась.

- Итератор должен возвращать self из метода __iter__, то есть в Python все итераторы являются итерируемыми.

- Примеры: список, file, dict

Собственные итерируемые объекты

In [23]:
```
class RangeIterator(object):
    def __init__(self, i, j):
        self.i = i #первое число
        self.j = j #последнее число

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.j:
            ret_val = self.i
            self.i += 1
            return ret_val
        else:
            raise StopIteration("No more elements")

    pass

it = RangeIterator(10, 20)
print([x for x in it])
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Последовательности

- Метод __getitem__ должен уметь возвращать элемент по индексу или поднимать исключение IndexError

- Пример: str

In [29]:
```
class Indexer(object):
    def __init__(self, max_index):
        self.max_index = max_index


    def __getitem__(self, idx):
        if idx > self.max_index:
            raise IndexError(idx)
        return idx
    pass
```

In [30]:
```
it = Indexer(5)
print([x for x in it])
```

```
[0, 1, 2, 3, 4, 5]
```

Преимущества итераторов в python

- Любой объект может быть итерируемым

- Память фактически не тратится, так как промежуточные данные выдаются по мере необходимости при запросе, поэтому фактически в памяти останутся только исходные данные и конечный результат, да и их можно читать и записывать, используя файл на диске.

Пере-использование итераторов (исчерпаемость)

Итератор самого себя

```
class RangeIterator1(object):
    def __init__(self, i, j):
        self.i = i #первое число
        self.j = j #последнее число
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.j:
            ret_val = self.i
            self.i += 1
            return ret_val
        else:
            raise StopIteration("No more elements")
    pass
it = RangeIterator(10, 20)
print(list(it), list(it))
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19] []
```

Итератор по новому объекту

```
class RangeIteratorHelper(object):
    def __init__(self, i, j):
        self.i = i #первое число
        self.j = j #последнее число
    def __next__(self):
        if self.i < self.j:
            ret_val = self.i
            self.i += 1
            return ret_val
        else:
            raise StopIteration("No more elements")
    pass

class RangeIterator2(object):
    def __init__(self, i, j):
        self.i = i #первое число
        self.j = j #последнее число
    def __iter__(self):
        return RangeIteratorHelper(self.i, self.j)
    pass
it = RangeIterator2(10, 20)
print(list(it), list(it), sep="\n")
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Устройство цикла for

- Возможная реализация цикла for

```
els = list(range(5))
it_els = iter(els)
```

```
for value in it_els:
    some_action(value)
```

```
0
1
2
3
4
```

```
els = list(range(5))
it_els = iter(els)
```

```
def some_action(value):
    print(value)

while True:
    try:
        value = next(it_els)
    except StopIteration:
        break
    some_action(value)
```

```
0
1
2
3
4
```

x in Iterable

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

```
d = Dict({1:1, 2:3})
1 in d
```

```
True
```

Генераторы

- Словом «генератор» обычно обозначается функция-генератор (или метод-генератор), возвращающая итератор генератора. Однако иногда слово может быть использовано и для обозначения самого итератора. В случаях, когда контекст непонятен лучше использовать полные термины: функция-генератор и итератор генератора.

- Итератор генератора — это объект, порождаемый функцией-генератором.

- Генераторы являются простым средством для создания итераторов. Всё, что можно сделать при помощи генераторов можно также сделать при помощи итераторов, построенных на классах. Но в случае генераторов методы __iter__() и __next__() создаются автоматически, также автоматически возбуждается StopIteration, да и поддерживать генераторы проще и удобнее, чем реализовывать то же с использованием классов.

- Обычно генератор – это функция, имеющая внутри слово yield.

Пример генератора

```
def some_gen(x):
    yield x
    x += 10
    yield x
    print("End of generator")

gen = some_gen(10)
print(type(gen))
print(next(gen)), print(list(gen)), print(next(gen)), print(list(gen))
```

```
<class 'generator'>
10
End of generator
[20]

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-11-1674f2c6b768> in <module>()
      7 gen = some_gen(10)
      8 print(type(gen))
----> 9 print(next(gen)), print(list(gen)), print(next(gen)), print(list(gen))

StopIteration:
```

Пример генератора

```
def fibonacci(max_num=10):
    a, b = 0, 1
    while a < max_num:
        # return a, + запоминает место рестарта для следующего вызова
        yield a
        a, b = b, a + b
    pass

# Используем генератор как итератор
[number for number in fibonacci()]
```

```
[0, 1, 1, 2, 3, 5, 8]
```

Пример генератора

```
def even(iterable):
    for idx in iterable:
        if not idx % 2:
            yield idx

for idx in even(range(10)):
    print(idx)
```

```
0
2
4
6
8
```

Функция map

```
def _map(function, iterable):
    for item in iterable:
        yield function(item)
    pass
```

```
_ = list(_map(print, [1, 2, 3]))
```

```
1
2
3
```

```
_ = list(map(print, [1, 2, 3]))
```

```
1
2
3
```

Функция filter

```
def _filter(function, iterable):
    for item in iterable:
        if function(item):
            yield item
    pass
```

```
_ = list(_map(print, _filter(lambda x: x % 2, range(10))))
```

```
1
3
5
7
9
```

```
_ = list(_map(print, filter(lambda x: x % 2, range(10))))
```

```
1
3
5
7
9
```

Функция zip

```
def _zip(*iterables):
    def zipped(iterables):
        while True:
            res = []
            end = False
            for iterable in iterables:
                try:
                    res.append(next(iterable))
                except StopIteration:
                    end = True
                    break
            if end:
                break
            else:
                yield tuple(res)
    return zipped(list(_map(iter, iterables)))
```

```
_ = list(map(print, _zip([1, 2, 3], range(4))))
```

```
(1, 0)
(2, 1)
(3, 2)
```

```
_ = list(map(print, zip([1, 2, 3], range(4))))
```

```
(1, 0)
(2, 1)
(3, 2)
```

Задача

- Дана последовательность N чисел. Необходимо посчитать суммы всех идущих друг за другом под-последовательностей длины M. Т. е. если на вход подается range(9), то ответом будет [3, 12, 21].

Задача

- Дана последовательность N чисел. Необходимо посчитать суммы всех идущих друг за другом под-последовательностей длины M. Т. е. если на вход подается range(9), то ответом будет [3, 12, 21].

- Решение:

```
N = 25
M = 5
it = iter(range(N))
list(map(sum, zip(*([it]*M))))
```

```
[10, 35, 60, 85, 110]
```

Цепочки

```
def chain(*iterables):
    for iterable in iterables:
        for it in iterable:
            yield it
list(chain(range(5), [10, 20], "test"))
```

```
[0, 1, 2, 3, 4, 10, 20, 't', 'e', 's', 't']
```

Функция enumerate

```
def _enumerate(iterable):
    i = 0
    for it in iterable:
        yield i, it
        i += 1

list(_enumerate("test"))
```

```
[(0, 't'), (1, 'e'), (2, 's'), (3, 't')]
```

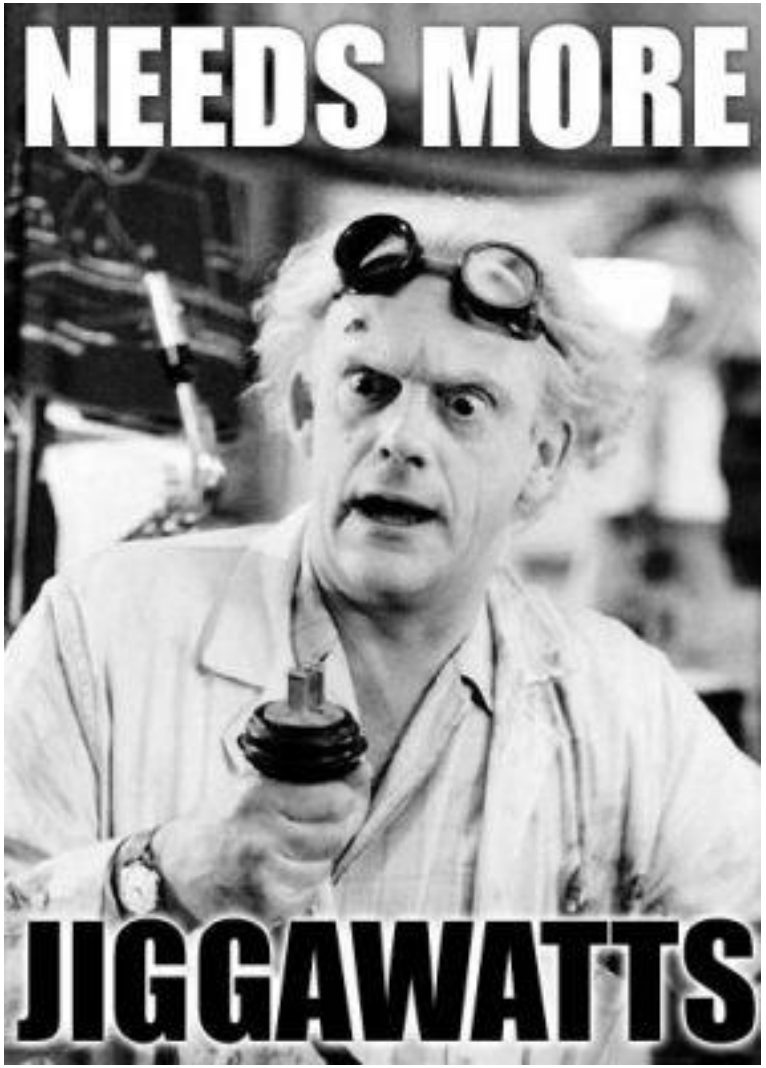
Применение генераторов

- С помощью генераторов можно просто реализовывать разные итераторы по коллекциям.

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

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

Need more power!



yield from

- yield from iterable по существу, является лишь сокращенной формой for item in iterable: yield item.

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

    def __iter__(self):
        if self.left: yield from self.left
        yield self.value
        if self.right: yield from self.right
```

generator.send

- generator.send(value) – Продолжает выполнение и «отправляет» значение в функцию генератора. Аргумент value становится результатом текущего выражения yield.

- Метод send () возвращает следующее значение, полученное генератором, или вызывает StopIteration, если генератор выходит, не давая другого значения.

- Когда send () вызывается для запуска генератора, он должен быть вызван с None как аргумент, потому что нет выражения yield, которое могло бы получить значение.

generator.throw

- Вызывает исключение типа type в точке, где генератор был приостановлен, и возвращает следующее значение, полученное функцией генератора.

- Если генератор выходит без получения другого значения, возникает исключение StopIteration.

- Если функция-генератор не улавливает исключение прошедшего или создает другое исключение, то это исключение распространяется на вызывающего.

generator.close

- Вызывает генератор GeneratorExit в точке, где функция генератора была приостановлена.
- Если функция генератора затем выходит правильно, уже закрыта или вызывает GeneratorExit (не вылавливая исключение), close возвращает его вызывающему.
- Если генератор возвращает значение (через yield), возникает RuntimeError. Если генератор вызывает какое-либо другое исключение, он распространяется на вызывающую.
- Функция close () ничего не делает, если генератор уже завершил работу из-за исключения или обычного выхода.

Пример генератора 1

```
def echo(value=None):
    print("Execution starts when 'next()' is called for the first time.")
    try:
        while True:
            try:
                value = (yield value)
            except Exception as e:
                value = e
    finally:
        print("Don't forget to clean up when 'close()' is called.")

generator = echo(1)
print(next(generator))

print(next(generator))

print(generator.send(2))

generator.throw(TypeError, "spam")

generator.close()
```

```
Execution starts when 'next()' is called for the first time.
1
None
2
Don't forget to clean up when 'close()' is called.
```

Пример генератора 2

```
def accumulate():
    tally = 0
    while 1:
        next = yield
        if next is None:
            return tally
        tally += next

def gather_tallies(tallies):
    while 1:
        tally = yield from accumulate()
        tallies.append(tally)

tallies = []
acc = gather_tallies(tallies)
# Ensure the accumulator is ready to accept values
next(acc)
for i in range(4):
    acc.send(i)

acc.send(None)  # Finish the first tally
for i in range(5):
    acc.send(i)

acc.send(None)  # Finish the second tally
acc.close()
tallies
```

```
[6, 10]
```

Выражения генераторы

- Помимо генераторов списков, словарей и множеств в python есть и просто выражения генераторы:

```
%%time
a = filter(lambda x : not (x % 33), (x * (x + 2) for x in range(10 ** 7)))
```

```
Wall time: 177 ms
```

```
%%time
a = filter(lambda x : not (x % 33), [x * (x + 2) for x in range(10 ** 7)])
```

```
Wall time: 1.84 s
```

itertools

- Стандартная библиотека генераторов:
  - islice, count, repeat, cycle
  - dropwhile, takewhile
  - product, permutations
  - chain, chain.from_iterable, tee