# Лекция 8
# Элементы функционального программирования

7 апреля 2016 г.

# Итераторы, введение

# Зачем нужны итераторы?

```
for i in range(10 ** 6):
    ...
```

# Зачем нужны итераторы?

```
for i in range(10 ** 6):
    ...
```

- В памяти хранится миллион чисел.
- Можно обойтись двумя.

# Зачем нужны итераторы?

```
for i in range(10 ** 6):
    ...
```

- В памяти хранится миллион чисел.
- Можно обойтись двумя.

Неудобный способ:

```
end = 10 ** 6
i = 0
while i < end:
    ...
    i += 1
```

# Зачем нужны итераторы?

Правильный способ:

```
for i in xrange(10 ** 6):
    ...
```

# Зачем нужны итераторы?

Правильный способ:

```
for i in xrange(10 ** 6):
    ...
```

## xrange в Python 3

| Python 2 | Python 3 |
|---|---|
| range | - |
| xrange | range |

# Зачем нужны итераторы?

Правильный способ:

```
for i in xrange(10 ** 6):
    ...
```

## xrange в Python 3

| Python 2 | Python 3 |
|---|---|
| range | - |
| xrange | range |

- Почему этот код работает?
- Где можно использовать итераторы?
- Как создавать итераторы?

# Генераторы

# Простой генератор

```
def g():
    yield 1
    yield 'abc'
    yield [1, 2, 3]

for x in g():
    print x
```

```
1
abc
[1, 2, 3]
```

# Генератор range

```
def myrange(begin, end, step):
    current = begin
    while current < end:
        yield current
        current += step
```

# Генератор range

```
def myrange(begin, end, step):
    current = begin
    while current < end:
        yield current
        current += step


myrange(0, 5, 2)
```

```
<generator object myrange at ...>
```

```
list(myrange(0, 5, 2))
```

```
[0, 2, 4]
```

# Схема работы генераторов

Вызов функции-генератора:

- Позиция = начало функции

Необходимо отдать очередной элемент:

- Начать исполнение функции с текущей позиции
- Если встретился оператор `yield`:
  - Отдать аргумент оператора `yield`
  - Остановить исполнение
  - Запомнить текущую позицию
- Иначе сообщить, что больше элементов нет.

# Генератор случайной длины

```
def random_length_repeat(obj, prob):
    while True:
        if random.random() < prob:
            break
        yield obj
```

# Генератор случайной длины

```
def random_length_repeat(obj, prob):
    while True:
        if random.random() < prob:
            break
        yield obj


list(random_length_repeat('a', 0.2))
list(random_length_repeat('a', 0.2))
list(random_length_repeat('a', 0.2))
```

```
['a', 'a', 'a']
[]
['a', 'a', 'a', 'a']
```

# Генераторные выражения

Списковое выражение

```
[2*x + 1 for x in range(5)]
```

```
[1, 3, 5, 7, 9]
```

# Генераторные выражения

Списковое выражение

```
[2*x + 1 for x in range(5)]
```

```
[1, 3, 5, 7, 9]
```

Генераторное выражение

```
(2*x + 1 for x in range(5))
```

```
<generator object <genexpr> at 0x7f847f233d70>
```

```
list((2*x + 1 for x in range(5)))
```

```
[1, 3, 5, 7, 9]
```

# Генераторные выражения

Списковое выражение

```
sum([len(w) for w in words])
```

Генераторное выражение

```
sum((len(w) for w in words))
```

Упрощенное генераторное выражение

```
sum(len(w) for w in words)
```

# Итераторы, теория

# Последовательность

*Последовательность (sequence)* — упорядоченный набор объектов, к элементам которого можно обращаться по индексу.

# Последовательность

*Последовательность (sequence)* — упорядоченный набор объектов, к элементам которого можно обращаться по индексу.

Должны быть определены операции:

- `__len__`
- `__getitem__` (для индексов)

# Последовательность

*Последовательность (sequence)* — упорядоченный набор объектов, к элементам которого можно обращаться по индексу.

Должны быть определены операции:

- `__len__`
- `__getitem__` (для индексов)

Примеры:

- `list`
- `tuple`
- `str`

# `for` для последовательностей

## Что имитируем

```
for x in sequence:
    action(x)
```

# for для последовательностей

## Что имитируем

```
for x in sequence:
    action(x)
```

```
def for_sequence(sequence, action):
    length = len(sequence)
    i = 0
    while i < length:
        x = sequence[i]
        action(x)
        i += 1
```

# Итерируемые и итераторы

*Итерируемое (iterable)* — упорядоченный набор объектов, элементы которого можно получать по одному.

# Итерируемые и итераторы

*Итерируемое (iterable)* — упорядоченный набор объектов, элементы которого можно получать по одному.

Должна быть определена операция `__iter__` (возвращает итератор).

# Итерируемые и итераторы

*Итерируемое (iterable)* — упорядоченный набор объектов, элементы которого можно получать по одному.

Должна быть определена операция `__iter__` (возвращает итератор).

*Итератор (iterator)* — объект, представляющий поток данных.

# Итерируемые и итераторы

*Итерируемое (iterable)* — упорядоченный набор объектов, элементы которого можно получать по одному.

Должна быть определена операция `__iter__` (возвращает итератор).

*Итератор (iterator)* — объект, представляющий поток данных.

Должна быть определена операция `next` (возвращает очередной элемент).

# Итерируемые и итераторы

*Итерируемое (iterable)* — упорядоченный набор объектов, элементы которого можно получать по одному.

Должна быть определена операция `__iter__` (возвращает итератор).

*Итератор (iterator)* — объект, представляющий поток данных.

Должна быть определена операция `next` (возвращает очередной элемент).
Окончание потока: исключение `StopIteration`

# `for` для iterable

## Что имитируем

```
for x in iterable:
    action(x)
```

# for для iterable

## Что имитируем

```
for x in iterable:
    action(x)
```

```
def for_iterable(iterable, action):
    iterator = iter(iterable)
    try:
        while True:
            x = next(iterator)
            action(x)
    except StopIteration:
        pass
```

# Код для iterator

```
class MyRangeIterator(object):
    def __init__(self, end):
        self.end = end
        self.current = 0
    def next(self):
        if self.current == self.end:
            raise StopIteration()
        result = self.current
        self.current += 1
        return result
```

# Код для iterable

```
class MyRangeIterator(object):
    ...

class MyRange(object):
    def __init__(self, end):
        self.end = end
    def __iter__(self):
        return MyRangeIterator(self.end)
```

# Истощение генераторов

```
x = MyRange(5)
print list(x)
print list(x)
```

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

# Истощение генераторов

```
x = MyRange(5)
print list(x)
print list(x)
```

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

```
y = myrange(0, 5, 1)
print list(y)
print list(y)
```

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

# Итерируемые и истощение

Последовательность

- Итерируемое
- Не истощается

# Итерируемые и истощение

Последовательность

- Итерируемое
- Не истощается

Итерируемое, но не последовательность

- Может истощаться (генераторы)
- Может не истощаться (xrange)

# Итерируемые и истощение

Последовательность

- Итерируемое
- Не истощается

Итерируемое, но не последовательность

- Может истощаться (генераторы)
- Может не истощаться (xrange)

Итератор

- Итерируемое
- Истощается

# Итерируемость итераторов

`x` — итератор

- `iter(x)` возвращает `x`
- `x` истощается

# Итерируемость итераторов

`x` — итератор

- `iter(x)` возвращает `x`
- `x` истощается

Пример использования

```
def all_equal(iterable):
    iterator = iter(iterable)
    first_element = next(iterator)
    for element in iterator:
        if element != first_element:
            return False
    return True
```

# Итерируемость итераторов

`x` — итератор

- `iter(x)` возвращает `x`
- `x` истощается

Пример использования

```
def all_equal(iterable):
    iterator = iter(iterable)
    first_element = next(iterator)
    for element in iterator:
        if element != first_element:
            return False
    return True
```

(где ошибка?)

# Инструменты функционального программирования

# Функция map

`map(func, iterable)`

- Применяет `func` ко всем элементам `iterable`.
- Возвращает список результатов.

```
map(str, [1, 2, 3])
```

```
['1', '2', '3']
```

# Функция map

`map(func, iterable)`

- Применяет `func` ко всем элементам `iterable`.
- Возвращает список результатов.

```
map(str, [1, 2, 3])
```

```
['1', '2', '3']
```

`map` + `lambda` → списковые выражения

```
map(lambda x: x + 1, sequence)
```

```
[x + 1 for x in sequence]
```

# Еще примеры для map

Пользователь вводит числа:

```
1 2 3 4 5
```

Считывание списка чисел:

```
numbers = map(int, raw_input().split())
```

Подсчет длин слов в тексте

```
text = 'Dog barks cat meows'
map(len, text.split())
```

```
[3, 5, 3, 5]
```

# Функция enumerate

`enumerate(iterable)`

- Нумерует элементы в `iterable`
- Возвращает истощаемое итерируемое (вне зависимости от истощаемости `iterable`)

```
for index, el in enumerate(['a', 'b', 'c']):
    print index, el
```

```
0 a
1 b
2 c
```

# Функция filter

`filter(func, iterable)`

- Вычисляет значение `func` для всех элементов `iterable`
- Возвращает список из элементов, для которых функция вернула `True`

```
filter(callable, [[], int, (), abs])
```

```
[<type 'int'>, <built-in function abs>]
```

# Функция filter

`filter(func, iterable)`

- Вычисляет значение `func` для всех элементов `iterable`
- Возвращает список из элементов, для которых функция вернула `True`

```
filter(callable, [[], int, (), abs])
```

```
[<type 'int'>, <built-in function abs>]
```

`filter` + `lambda` → списковые выражения

```
filter(lambda x: x % 2 == 0, sequence)
```

```
[x for x in sequence if x % 2 == 0]
```

# Функция reduce

`reduce(func, iterable)`

- Если `a, b, c, ...` — элементы последовательности, то функция вычисляет `func(func(func(a, b), c)...)`

```
l = [1, 2, 3, 4, 5]

reduce(lambda x, y: x + y, l)
```

```
15
```

```
reduce(lambda x, y: x * y, l)
```

```
120
```

# Библиотека itertools

# Итераторные аналоги

`imap(func, iterable)`

- Итераторный аналог `map`

`izip(p, q, ...)`

- Итераторный аналог `zip`

`ifilter(func, iterable)`

- Итераторный аналог `filter`

# Комбинаторные функции

`product(p, q, ...)`

- Декартово произведение.
- Эквивалентно многоуровневому `for`.

# Комбинаторные функции

`product(p, q, ...)`

- Декартово произведение.
- Эквивалентно многоуровневому `for`.

```
print list(product('AB', (1, 2, 3)))
```

```
[('A', 1), ('A', 2), ('A', 3),
 ('B', 1), ('B', 2), ('B', 3)]
```

# Комбинаторные функции

Пример использования `product`

```
table = [[1, 2], [3, 4]]
rows = xrange(len(table))
cols = xrange(len(table[0]))
for r, c in product(rows, cols):
    ...
```

# Комбинаторные функции

Пример использования `product`

```
table = [[1, 2], [3, 4]]
rows = xrange(len(table))
cols = xrange(len(table[0]))
for r, c in product(rows, cols):
    ...
```

Другие комбинаторные функции:

- `permutations()`
- `combinations()`
- `combinations_with_replacement()`

# Преобразования

`islice(iterable, begin, end, step)`

- Аналогично операции `[begin:end:step]`

```
print list(islice(xrange(10), 3, 7, 2))
```

```
[3, 5]
```

# Преобразования

`islice(iterable, begin, end, step)`

- Аналогично операции `[begin:end:step]`

```
print list(islice(xrange(10), 3, 7, 2))
```

```
[3, 5]
```

`chain(p, q, ...)`

- Последовательно итерируется по другим итераторам

```
print list(chain(xrange(3), 'abc',
                 xrange(5)))
```

```
[0, 1, 2, 'a', 'b', 'c', 0, 1, 2, 3, 4]
```

# Бесконечные итераторы

`count(n)`

- Арифметическая прогрессия, начиная с `n`

```
print list(islice(count(10), 5))
```

```
[10, 11, 12, 13, 14]
```

# Бесконечные итераторы

`count(n)`

- Арифметическая прогрессия, начиная с `n`

```
print list(islice(count(10), 5))
```

```
[10, 11, 12, 13, 14]
```

`cycle(iterable)`

- Циклически повторяет элементы `iterable`

```
print list(islice(cycle('abc'), 5))
```

```
['a', 'b', 'c', 'a', 'b']
```

# Разные примеры

Обычный код

```
def smth_loops(width, height, depth, ...):
    # do smth for all elements
    for i in xrange(width):
        for j in xrange(height):
            for k in xrange(depth):
                ...
```

Код в стиле функционального программирования

```
from itertools import product

def smth_loops(width, height, depth, ...):
    shape = [width, height, depth]
    for i, j, k in product(*map(xrange, shape)):
        ...
```

# Разные примеры

Группируем элементы на блоки фиксированной длины

```
from itertools import izip_longest

def grouper(iterable, n, fillvalue=None):
    args = [iter(iterable)] * n
    return izip_longest(*args,
                        fillvalue=fillvalue)

list(grouper('ABCDEFG', 3, 'x'))
```

```
[('A', 'B', 'C'), ('D', 'E', 'F'),
('G', 'x', 'x')]
```