# Python

## Лекция 7. Генераторы

Сайфулин Дмитрий, Слободкин Евгений
ИТМО, 7 декабря 2023

MASTER OF
SOFTWARE
ENGINEERING

# PEP 255 – Simple Generators

https://peps.python.org/pep-0255/

# Мотивация

```
# Функция, которая разбирает «число + число»:

def tokenize_expr(s):
    if not first_num_found:
        return find_first_num(s)

    if not op_found:
        return find_op(s)

    if not second_num_found:
        return find_second_num(s)
```

# Мотивация

```
# Функция, которая разбирает «число + число»:

def tokenize_expr(s):
    if not first_num_found:   #  Откуда брать и где хранить эти переменные?
        return find_first_num(s)

    if not op_found:
        return find_op(s)

    if not second_num_found:
        return find_second_num(s)
```

# Мотивация

```
# Функция, которая разбирает «число + число»:

def tokenize_expr(s):
    if not first_num_found:   #  Откуда брать и где хранить эти переменные?
        return find_first_num(s)

    if not op_found:    #  По факту это некоторое текущее «состояние» разбора.
        return find_op(s)

    if not second_num_found:
        return find_second_num(s)
```

# Мотивация

```
# Функция, которая разбирает «число + число»:

def tokenize_expr(s):
    if not first_num_found:   #  Откуда брать и где хранить эти переменные?
        return find_first_num(s)

    if not op_found:    #  По факту это некоторое текущее «состояние»
разбора.
        return find_op(s)

    if not second_num_found:   #  Часто состояние делают аргументом функции,
                               # и вызывающая сторона его передает.
        return find_second_num(s)
```

# Iteration Abstraction in Sather (1996)

```
# Core разработчики Python вдохновлялись генераторами в языке Sather, где они
# в основном использовались для упрощения написания итераторов:

upto!(once limit:INT):INT is
     r:INT:=self;
     loop while!(r<=limit);
          yield r;  # состояние — текущий индекс.
          r:=r+1
     end
end
```

# Примеры генераторов

# Простейший генератор

```
def simple_gen():
    yield


>>> simple_gen
<function simple_gen at 0x1023c4720>

>>> simple_gen()
<generator object simple_gen at 0x102300ca0>
```

# Простейший генератор

```
def simple_gen():
    yield


>>> simple_gen
<function simple_gen at 0x1023c4720>

>>> simple_gen()
<generator object simple_gen at 0x102300ca0>

>>> g = simple_gen()

>>> next(g)   # Генератор удовлетворяет протоколу итератора.
None          # Не в этот ли момент мы начали исполнение генератора?

>>> next(g)
StopIteration
```

# Более сложный генератор

```
def one_two_three():
    yield 1
    yield 2
    yield 3
```

# Более сложный генератор

```
def one_two_three():
    yield 1
    yield 2
    yield 3

>>> g = one_two_three()
>>> next(g)
1

>>> next(g)
2

>>> next(g)
3

>>> next(g)
StopIteration
```

# return и yield

```
# Согласно спецификации функция называется генератором, если содержит хотя бы
# один yield
def gen(x):
    if x % 2 == 0:
        return x + 1
    else:
        yield 2 * x
```

# return и yield

```
# Согласно спецификации функция называется генератором, если содержит хотя
бы
# один yield
def gen(x):
    if x % 2 == 0:
        return x + 1
    else:
        yield 2 * x
```

```
>>> gen(2)
<generator object gen at 0x104981780>

>>> next(gen(2))
StopIteration: 3

>>> next(gen(1))
2
```

# Бесконечный генератор

```
def fib():
    prev, cur = 1, 1

    while True:
        yield prev
        prev, cur = cur, prev + cur
```

# Бесконечный генератор

```
def fib():
    prev, cur = 1, 1

    while True:
        yield prev
        prev, cur      = cur, prev + cur
```

```
>>> fg = fib()
<generator object fib at 0x1027ff510>

>>> type(fg)
<class 'generator'>
```

# Бесконечный генератор

```
def fib():
    prev, cur = 1, 1

    while True:
        yield prev
        prev, cur      = cur, prev + cur

>>> fg = fib()
<generator object fib at 0x1027ff510>

>>> type(fg)
<class 'generator'>

>>> [next(fg) for _ in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

# Бесконечный генератор

```
def fib():
    prev, cur = 1, 1

    while True:
        yield prev
        prev, cur      = cur, prev + cur

>>> fg = fib()
<generator object fib at 0x1027ff510>

>>> type(fg)
<class 'generator'>

>>> [next(fg) for _ in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

>>> list(itertools.islice(fib(), 10))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

# Генераторы и исключения

# Распространение исключений

```
def f():
    yield 1
    raise Exception
    yield 2


>>> g = f()
>>> next(g)
1

>>> next(g)
Exception

>>> next(g)
StopIteration
```

# Распространение исключений

```
>>> def f():
...     try:
...         yield 1
...         try:
...             yield 2
...             1/0
...             yield 3  # never get here
...         except ZeroDivisionError:
...             yield 4
...             yield 5
...             raise
...         except:
...             yield 6
...         yield 7     # the "raise" above stops this
...     except:
...         yield 8
...     yield 9
...     try:
...         x = 12
...     finally:
...        yield 10
...     yield 11
```

```
>>> print list(f())
[1, 2, 4, 5, 8, 9, 10, 11]
```

Пример взят из оригинального PEP:
https://peps.python.org/pep-0255/#specification-generators-and-exception-propagation

# Генераторные выражения

https://peps.python.org/pep-0289

# Пример

```
>>> (x + 1 for x in range(10))
<generator object <genexpr> at 0x1029db510>
```

# Пример

```
>>> (x + 1 for x in range(10))
<generator object <genexpr> at 0x1029db510>


# Такую конструкцию можно использовать для создания коллекций:
>>> {c: idx for idx, c in enumerate('bcadfe')}
{'b': 0, 'c': 1, 'a': 2, 'd': 3, 'f': 4, 'e': 5}

>>> set(ord(c) for c in 'abcdefedcba')
{97, 98, 99, 100, 101, 102}
```

# Аккуратно! (1)

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

# Аккуратно! (1)

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

>>> wtf = (x for x in even if x not in even)

>>> list(wtf)
#  ???
```

# Аккуратно! (1)

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

>>> wtf = (x for x in even if x not in even)

>>> list(wtf)
[0]

>>> list(wtf)
#  ???
```

# Аккуратно! (1)

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

>>> wtf = (x for x in even if x not in even)

>>> list(wtf)
[0]

>>> list(wtf)
[]
```

# Аккуратно! (1)

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

>>> wtf = (x for x in even if x not in even)

>>> list(wtf)
[0]

>>> list(wtf)
[]
```

```
def even():
    for x in range(10):
        yield x

def wtf():
    e = even()
    for x in e:
        if x not in e:
            yield x
```

# Аккуратно! (2)

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

>>> wtf = (x for x in even if x not in even)
>>> even = [2, 4, 42]
>>> list(wtf)
#  ???
```

# Аккуратно! (2)

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

>>> wtf = (x for x in even if x not in even)
>>> even = [2, 4, 42]
>>> list(wtf)
[0, 6, 8]  # Почему?!
```

# Аккуратно! (2)

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

>>> wtf = (x for x in even if x not in even)
>>> even = [2, 4, 42]
>>> list(wtf)
[0, 6, 8]   # Почему?!
```

```
# Позднее связывание мы уже видели в лямбдах:
# fns = [lambda: print(i) for i in range(10)]

# Почитать почему был выбран именно такой подход можно в PEP 289:
```

https://peps.python.org/pep-0289/#early-binding-versus-late-binding

# List Comprehension

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

>>> not_wtf = [x for x in even if x not in even]
>>> even = [2, 4]
>>> not_wtf
[]
```

# map/filter/reduce/…

# map

```
>>> map(lambda x: x**2, [1, 2, 3])
<map object at 0x102d0f1f0>

>>> list(map(lambda x: x**2, [1, 2, 3]))
[1, 4, 9]
```

# filter

```
>>> filter(lambda x: x != 1, [1, 2, 3])
<filter object at 0x102d0f1f0>

>>> list(filter(lambda x: x != 1, [1, 2, 3]))
[2, 3]
```

# zip

```
>>> zip([1, 2, 3], ('a', 'b', 'c'))
<zip object at 0x102d0f1f0>

>>> [f'{n} -> {c}' for n, c in zip([1, 2, 3], ('a', 'b', 'c'))]
['1 -> a', '2 -> b', '3 -> c']
```

# functools.reduce

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
```

# itertools.chain.from_iterable

```
>>> from itertools import chain
>>> chained = chain.from_iterable([[1, 2], (3, 4), '56'])

>>> next(chained)
1
>>> next(chained)
2
>>> next(chained)
3
>>> next(chained)
4
>>> next(chained)
'5'
>>> next(chained)
'6'
```

# itertools.chain.from_iterable

```
# itertools.chain.from_iterable по смыслу эквивалентна:

def from_iterable(iterables):
    for it in iterables:
        for element in it:
            yield element
```

# Ещё itertools

```
>>> from itertools import *

>>> list(combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

# Ещё itertools

```
>>> from itertools import *

>>> list(combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]

>>> list(islice(cycle('ABCD')))
['A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B']
```

# Ещё itertools

```
>>> from itertools import *

>>> list(combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]

>>> list(islice(cycle('ABCD')))
['A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B']

>>> list(product('ABC', (1, 2)))
[('A', 1), ('A', 2), ('B', 1), ('B', 2), ('C', 1), ('C', 2)]
```

# Ещё itertools

```
>>> from itertools import *

>>> list(combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]

>>> list(islice(cycle('ABCD')))
['A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B']

>>> list(product('ABC', (1, 2)))
[('A', 1), ('A', 2), ('B', 1), ('B', 2), ('C', 1), ('C', 2)]

>>> product('ABC', (1, 2))
<itertools.product object at 0x104492740>
```

# yield as expression

https://peps.python.org/pep-0342/

# yield as statement

```
# До 2.5 yield был только statement-ом:
def counter(maximum):
    i = 0
    while i < maximum:
        yield i
        i += 1


# Из вызывающего кода вы не могли нормально передавать значения в генератор.
# Приходилось использовать разделяемое состояние...
```

# yield as expression

```
# Начиная с 2.5 yield может выступать в роли expression:
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:
            i = val
        else:
            i += 1
```

# yield as expression

```
# Начиная с 2.5 yield может выступать в роли expression:
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:
            i = val
        else:
            i += 1
```

```
# Произойдет yield i, как и в обычном генераторе.
# В val запишется то, что мы отправим в генератор из вызывающего кода.
# i не поменяется до момента явного присваивания!
```

# yield as expression

```
# Начиная с 2.5 yield может выступать в роли expression:
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:
            i = val
        else:
            i += 1
```

```
>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
```

# yield as expression

```
# Начиная с 2.5 yield может выступать в роли expression:
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:
            i = val
        else:
            i += 1

>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
```

# yield as expression

```
# Начиная с 2.5 yield может выступать в роли expression:
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:
            i = val
        else:
            i += 1

>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
```

# yield as expression

```
# Начиная с 2.5 yield может выступать в роли expression:
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:
            i = val
        else:
            i += 1

>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
>>> next(it)
StopIteration
```

# yield as expression

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:  # Q: Зачем нужна эта проверка?
            i = val
        else:
            i += 1
```

# yield as expression

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:  # Q: Зачем нужна эта проверка?
                             # A: Вызов next в точности равен gen.send(None).
                             # Мы не знаем как будет возобновляться наш
                             # генератор, поэтому подстрахуемся.
            i = val
        else:
            i += 1
```

# yield as expression (2)

# Приколы

```
>>> def f(): lambda: (yield)
>>> type(f())
# ???
```

# Приколы

```
>>> def f(): lambda: (yield)  # Не прикол же, всё понятно 🤔.
>>> type(f())
<class 'NoneType'>
```

# Приколы

```
# Интереснее:

>>> def f(): lambda x=(yield): 1
>>> type(f())
#  ???
```

# Приколы

```
# Интереснее:

>>> def f(): lambda x=(yield): 1
>>> type(f())
<class 'generator'>  # Тоже понятно 😴.
```

# Приколы

```
>>> def f(): return; yield 1
>>> type(f())
# ???
```

# Приколы

```
>>> def f(): return; yield 1
>>> type(f())
<class 'generator'>
```

# Приколы

```
def f():
    if (yield):
        v = (yield)
        yield 'v: ' + str(v)
```

# Приколы

```
def f():
    if (yield):
        v = (yield)
        yield 'v: ' + str(v)
```

```
>>> g = f()
>>> next(g)
>>> print('after first send', g.send(True))
after first send None

>>> print('after second send', g.send(42))
after second send v: 42
```

# yield from

# Как пробрасывать значение из генератора?

```
>>> def g1(): yield 1

>>> def g2(): yield g1()

>>> next(g2())
<generator object g1 at 0x104c10b40>  # Мы же просто хотели 1-ку 😭...
```

# Как пробрасывать значение из генератора?

```
>>> def g1(): yield 1

>>> def g2(): yield from g1()

>>> next(g2())
1                  #  То, что надо! 👍
```

# Как пробрасывать значение из генератора?

```
# Помогает сократить код при работе с итерируемыми объектами.

def from_iterables(iterables):
    for iterable in iterables:
        for it in iterable:
            yield it


# Можно переписать как:

def flatten(nested_list):
    for sublist in nested_list:
        yield from sublist
```