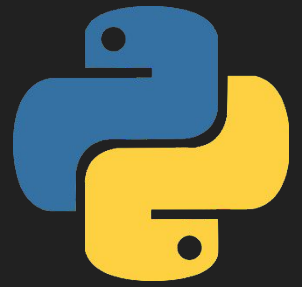


Section XII: Итераторы и генераторы

Приближаемся к Pythonic коду…

Section XII: Итераторы и генераторы

Итерабельные объекты (Iterable Objects)

```
>>> for i in [1, 2, 3, 4]:
...     print(i)
...
1
2
3
4
```

```
>>> for c in "python":
...     print(c)
...
p
y
t
h
o
n
```

Итерабельные объекты – «ага, вот эти ребята»

```
>>> for k in {"x": 1, "y": 2}:
...     print(k)
...
y
x
```

```
>>> for line in open("a.txt"):
...     print(line, end="")
...
first line
second line
```

```
>>> ",".join(["a", "b", "c"])
'a,b,c'
>>> ",".join({"x": 1, "y": 2})
'y,x'
```

```
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list({"x": 1, "y": 2})
['y', 'x']
```

Но это еще не итераторы!

Section XII: Итераторы и генераторы

Итерационный протокол (The Iteration Protocol). Ключевое слово `iter`

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> next(x)
1
>>> next(x)
2
>>> next(x)
3
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Ключевое слово `next` – следующий элемент

Что еще за `iter`? И откуда взялся `next`?

Section XII: Итераторы и генераторы

Итерационный протокол (The Iteration Protocol)

Волшебные методы `__iter__()` и `__next__()` образуют итерационный протокол.

И являются методами класса!

`__iter__()`: Выполняет инициализацию объекта-итератора и возвращает его (self). Такая инициализация используется, например, в цикле for.

`__next__()`: Возвращает следующее значение итератора. В случае, если объекты закончились, метод должен вызывать исключение `StopIteration`.

Хммм, `StopIteration`...

Так какого же класса?

Section XII: Итераторы и генераторы

Итераторы (Iterators)

Def. ***Итератор*** *- это объект, который позволяет программисту перемещаться по всем элементам коллекции, независимо от ее конкретной реализации.*

А как же итерабельные объекты?

```
>>> for i in [1, 2, 3, 4]:
...     print(i)
...
1
2
3
4
```

Не итератор

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
```

А вот теперь итератор

Функция `iter()` (которая вызывает `__iter__()`) возвращает итератор.

Def. Объект называется **итерабельным**, если мы можем получить из него итератор. Большинство встроенных контейнеров в Python, таких как: list, tuple, string и т.д., являются итерабельными.

Section XII: Итераторы и генераторы

Как на самом деле работает волшебный `for`?

```
for element in iterable:
    # do something with element
```

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

Без `while` никуда

Section XII: Итераторы и генераторы

Создание итератора. Пересобираем `range()`

`__iter__()`: Выполняет инициализацию объекта-итератора и возвращает его (self).

`__next__()`: Возвращает следующее значение итератора. В случае, если объекты закончились, метод должен вызывать исключение StopIteration.

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

```
>>> y = yrange(3)
>>> next(y)
0
>>> next(y)
1
>>> next(y)
2
>>> next(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in __next__
StopIteration
```

```
>>> list(yrange(5))
[0, 1, 2, 3, 4]
>>> sum(yrange(5))
10
```

Section XII: Итераторы и генераторы

Создание итератора

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

```
>>> a = PowTwo(4)
>>> i = iter(a)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
4
>>> next(i)
8
>>> next(i)
16
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

Кстати, зачем это все?

- Чище
- Компактнее
- Оптимальнее…

Но об этом чуть позже.

```
>>> for i in PowTwo(5):
...     print(i)
...
1
2
4
8
16
32
```

Section XII: Итераторы и генераторы

Немного деталей… Iterator vs. Iterable

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

```
class zrange_iter:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        # Iterators are iterables too.
        # Adding this functions to make them so.
        return self

    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

```
class zrange:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return zrange_iter(self.n)
```

```
>>> y = yrange(5)
>>> list(y)
[0, 1, 2, 3, 4]
>>> list(y)
[]
>>> z = zrange(5)
>>> list(z)
[0, 1, 2, 3, 4]
>>> list(z)
[0, 1, 2, 3, 4]
```

Section XII: Итераторы и генераторы

Немного деталей… Бесконечные итераторы

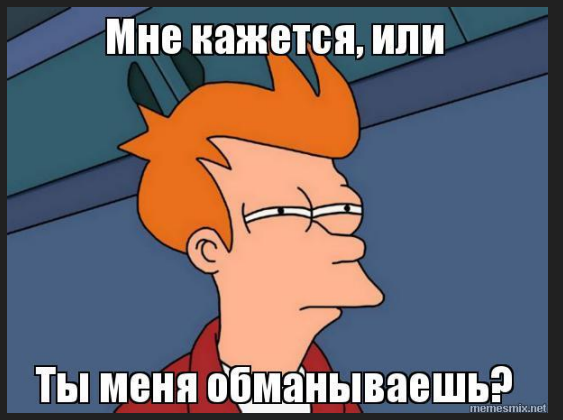
Callable object (function)

Sentinel ~~(часовой?)~~(метка)

```
>>> int()
0

>>> inf = iter(int,1)
>>> next(inf)
0
>>> next(inf)
0
```

```
class InfIter:
    """Infinite iterator to return all
        odd numbers"""

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        num = self.num
        self.num += 2
        return num
```

```
>>> a = iter(InfIter())
>>> next(a)
1
>>> next(a)
3
>>> next(a)
5
>>> next(a)
7
```



…поместили бесконечный (теоретически) набор в конечную память

Экономия ресурсов!
Profit!

Section XII: Итераторы и генераторы

Лайт практика

Напишите класс `reverse_iter`, который принимает на вход лист (`list`) и итерируется по нему в обратном направлении.

```
>>> it = reverse_iter([1, 2, 3, 4])
>>> next(it)
4
>>> next(it)
3
>>> next(it)
2
>>> next(it)
1
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Section XII: Итераторы и генераторы

Итераторы, неужели нельзя проще?

Для создания итератора нужно:

- 🤯 Написать класс с методами `__iter__()` и `__next__()`
- 🤯 Следить за внутренними состояниями
- 🤯 Прописать исключение для случая, когда значения закончатся

🤯🤯🤯 и т.д.

Ну это же рутина, давайте автоматизируем!

Section XII: Итераторы и генераторы

Генераторы – вернемся к простому

Def. Генераторы являются простым способ создания итератора. Генератор - это функция, которая возвращает объект (итератор), по которому возможно итерироваться (по одному).

Для создания генератора нужно:

Написать ключевое слово `yield` вместо `return`

Все.

Слишком просто, чтобы быть правдой? Заглянем под капот…

Section XII: Итераторы и генераторы

Генераторы. Зайдем с другой стороны – функции, `return` и `yield`

Разница между генераторной функцией и обычной функцией:

- Генераторная функция может содержать более одного `yield` (в отличие от единственного `return`).
- Во время вызова генераторная функция возвращает объект (итератор), но не начинает выполнение.
- Методы типа `__iter__()` и `__next__()` реализованы автоматически, поэтому можем итерироваться с помощью `next()`.
- После срабатывания `yield`, генераторная функция встает на паузу, и управление передается функции-вызывателю.
- Локальные переменные и их состояния хранятся между успешными вызовами
- После завершения генераторной функции `StopIteration` вызывается автоматически.

Section XII: Итераторы

Генераторы. Зайдем с другой с yield

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

```
>>> # It returns an object but does not start execution
>>> a = my_gen()                                immediately.

>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1
>>> # Once the function yields, the function is paused
          and the control is transferred to the caller.
>>> # Local variables and theirs states are remembered
>>> next(a)                              between successive calls.
This is printed second
2


>>> next(a)
This is printed at last
3


>>> # Finally, when the function terminates, StopIteration
>>> next(a)        is raised automatically on further calls.
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

Для повторения процесса нужно создать новый генератор

Section XII: Итераторы и генераторы

Генераторы. Использование циклов

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n

# Using for loop
for item in my_gen():
    print(item)
```

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Section XII: Итераторы и генераторы

Генераторы. Использование циклов

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]

# For loop to reverse the string
# Output:
# o
# l
# l
# e
# h
for char in rev_str("hello"):
    print(char)
```

Вызов `yield` прямо во время цикла – основной кейс для использования генераторов

Все просто!

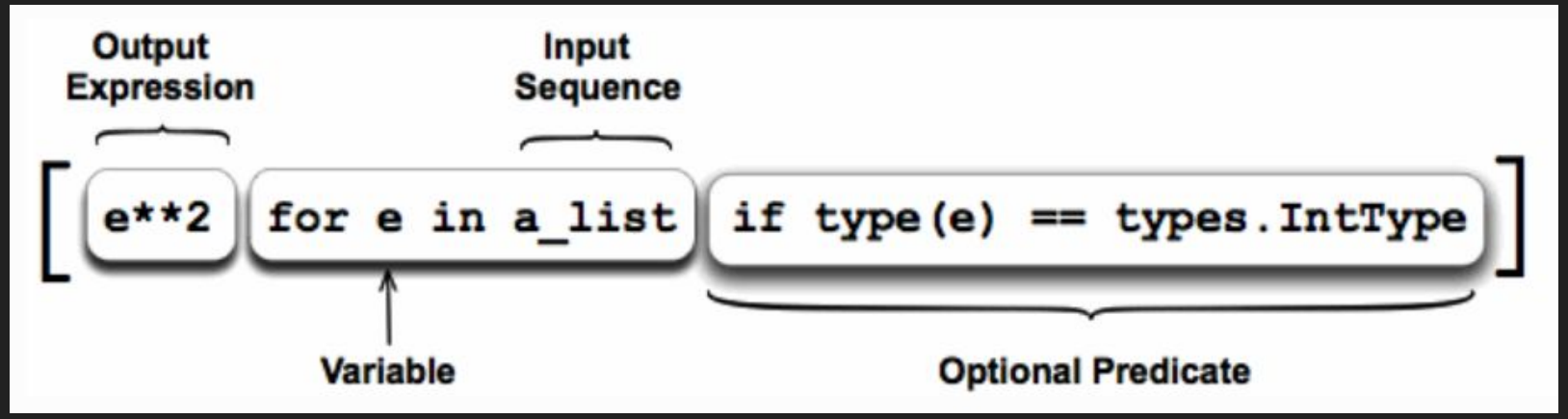
Сделаем еще проще!

Section XII: Итераторы и генераторы

List Comprehension (постижение)

```
pow2 = [2 ** x for x in range(10)]

# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
print(pow2)
```

```
pow2 = []
for x in range(10):
    pow2.append(2 ** x)
```



Section XII: Итераторы и генераторы

List Comprehension 

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]
>>> pow2
[64, 128, 256, 512]
>>> odd = [x for x in range(20) if x % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

Такой же прием работает со словарями и множествами

А как же кортежи?

Section XII: Итераторы и генераторы

Генераторные выражения

```
# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
# Output: [1, 9, 36, 100]
[x**2 for x in my_list]

# same thing can be done using generator expression
# Output: <generator object <genexpr> at 0x0000000002EBDAF8>
(x**2 for x in my_list)
```

Ленивые вычисления – выделяем память по запросу

Экономия ресурсов!

Profit!

Section XII: Итераторы и генераторы

Генераторные выражения

```
# Intialize the list
my_list = [1, 3, 6, 10]

a = (x**2 for x in my_list)
# Output: 1
print(next(a))

# Output: 9
print(next(a))

# Output: 36
print(next(a))

# Output: 100
print(next(a))

# Output: StopIteration
next(a)
```

Полезности

```
>>> sum(x**2 for x in my_list)
146

>>> max(x**2 for x in my_list)
100
```

Section XII: Итераторы и генераторы

Генераторные выражения – а вообще, зачем?

1. Легко реализовать
2. Эффективны по памяти
3. Непрерывный поток
4. Пайплайны

```
def PowTwoGen(max = 0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

```
class PowTwo:
    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

Section XII: Итераторы и генераторы

Еще немного лайта

- Напишите программу, которая принимает одно или несколько имен файлов в качестве аргументов и печатает все имена длиной более 40 символов (с помощью выражений генератора).
- Напишите функцию `findfiles`, которая рекурсивно спускается по дереву каталогов для указанного каталога и генерирует пути ко всем файлам в дереве.
- Напишите программу `split.py`, который принимает целое число `n` и имя файла и разбивает файл на несколько небольших файлов, каждый из которых содержит `n` строк.
- Напишите функцию для рекурсивного вычисления общего количества строк кода, игнорируя пустые строки и строки комментариев, во всех файлах `python` в указанном каталоге.

Q&A