



ПРОГРАММИРОВАНИЕ 10 КЛАСС

УРОК №7

**Гимназия
№ 1514**

ИТЕРАТОРЫ

Что такое итерируемый объект?



ИТЕРАТОРЫ

Объект считается **итерируемым**, если он либо физически является последовательностью, либо если он является объектом, который воспроизводит по одному результату за раз в контексте инструментов выполнения итераций, таких как цикл for



ИТЕРАТОРЫ

Во многих современных языках программирования используют такие сущности как **итераторы**.

Основное их назначение – это упрощение навигации по элементам объекта, который, как правило, представляет собой некоторую коллекцию (список, словарь и т.п.) и экономия памяти.

Итератор представляет собой объект-перечислитель, который для данного объекта выдает следующий элемент, либо бросает исключение, если элементов больше нет.



Основное место использования итераторов — это цикл *for*.

```
num_list = [1, 2, 3, 4, 5]
for i in num_list:
    print(i)
```

1
2
3
4
5

Если перебираются элементы в некотором списке или символы в строке с помощью цикла *for*, то, фактически, это означает, что при каждой итерации цикла происходит обращение к итератору, содержащемуся в строке/списке, с требованием выдать следующий элемент, если элементов в объекте больше нет, то итератор генерирует исключение, обрабатываемое в рамках цикла *for* незаметно для пользователя.



ИТЕРАТОРЫ ФАЙЛОВ

```
>>> f = open('script1.py')
>>> f.readline()
'import sys\n'
>>> f.readline()
'print(sys.path)\n'
>>> f.readline()
'x = 2\n'
>>> f.readline()
'print(2 ** 33)\n'
>>> f.readline()
''
```

Каждый раз, вызывая метод *readline*, мы перемещаемся к следующей строке. По достижении конца файла возвращается пустая строка, что может служить сигналом для выхода из цикла.



ИТЕРАТОРЫ ФАЙЛОВ

```
>>> f = open('script1.py')
>>> f.__next__()
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'
>>> f.__next__()
'x = 2\n'
>>> f.__next__()
'print(2 ** 33)\n'
>>> f.__next__()
Traceback (most recent call last):
...текст сообщения об ошибке опущен...
StopIteration
```

Файлы имеют также метод `__next__`, который производит практически тот же эффект, — всякий раз, когда его вызывают, он возвращает следующую строку. По достижении конца файла метод `__next__` возбуждает встроенное исключение ***StopIteration***



Лучший способ построчного чтения текстового файла состоит не в том, чтобы прочитать его целиком, а в том, чтобы позволить циклу `for` автоматически вызывать метод `__next__` для перемещения к следующей строке в каждой итерации:

```
>>> for line in open('script1.py'):
...     print(line.upper(), end="")
```

```
IMPORT SYS
PRINT(SYS.PATH)
X= 2
PRINT(2 ** 33)
```



Такой способ построчного чтения текстовых файлов считается лучшим по трем причинам: программный код выглядит *проще*, он выполняется *быстрее* и более *экономно* *использует память*. **Чем** если бы мы вызвали метод **readlines** для загрузки содержимого файла в память в виде списка строк:

```
>>> for line in open('script1.py').readlines():  
...     print(line.upper(), end="")
```

```
IMPORT SYS  
PRINT SYS.PATH  
X= 2  
PRINT 2 ** 33
```



Так как в этом случае файл загружается целиком, данный способ не позволит работать с файлами, слишком большими, чтобы поместиться в память компьютера. При этом версия, основанная на применении итераторов, не подвержена таким проблемам с памятью, так как содержимое файла считывается по одной строке за раз.



Таким образом, В ЧЕМ ПРЕИМУЩЕСТВО ИТЕРАТОРОВ?

При использовании итераторов 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.



В чем разница кодов?

```
>>> for line in open('script1.py'):
    print(line.upper(), end="")
```

```
>>> f = open('script1.py')
    while True:
        line = f.readline()
        if not line:
            break
        print(line.upper(), end="")
```



В чем разница кодов?

```
>>> for line in open('script1.py'):
    print(line.upper(), end="")

>>> f = open('script1.py')
    while True:
        line = f.readline()
        if not line:
            break
        print(line.upper(), end="")
```

Второй вариант будет работать медленнее версии, основанной на использовании итератора в цикле `for`, потому что итераторы внутри интерпретатора выполняются со скоростью, присущей программам, написанным на языке C, тогда как версия на базе цикла `while` работает со скоростью интерпретации байт-кода виртуальной машиной Python.



Выполнение итераций вручную: `iter` и `next`

```
>>> f = open('script1.py')
>>> f.__next__()
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'
```

```
>>> f = open('script1.py')
>>> next(f)
'import sys\n'
>>> next(f)
'print(sys.path)\n'
```

Встроенная функция `next`, автоматически вызывает метод `__next__` объекта.




Выполнение итераций вручную: `iter` и `next`

```
>>> L = [1,2,3]
      for elem in L:
          print(elem, end = ' ')
1 2 3

>>> L = [1, 2, 3]
>>> I = iter(L)
>>> I.__next__()
1
>>> I.__next__()
2
>>> I.__next__()
3
>>> I.__next__()
Traceback (most recent call last):
...текст сообщения об ошибке опущен...
StopIteration
```

С технической точки зрения итерационный протокол имеет еще одну сторону. В самом начале цикл `for` получает итератор из итерируемого объекта, передавая его встроенной функции ***iter***, которая возвращает объект, имеющий требуемый метод **`__next__`**



Выполнение итераций вручную: `iter` и `next`

```
>>> f = open('script1.py')
>>> iter(f) is f
True
>>> f.__next__()
'import sys\n'
```

При работе с файлами этот начальный этап не нужен, потому что объект файла имеет собственный итератор. То есть объекты файлов имеют собственный метод `__next__`, и потому для работы с файлами не требуется получать другой объект.



ЧТО ИЗУЧИЛИ?



ЧТО ИЗУЧИЛИ?

Есть итераторы, они позволяют получить элемент коллекции\последовательности в каждый конкретный момент времени. Так не надо хранить весь объект в памяти. И скорость доступа на уровне C.

Для того, чтобы создать итератор объекта, необходимо вызвать *iter()*. Для получения следующего объекта используется *next()*.

Для файлов создавать итератор не надо, он имеет собственный итератор



ПРИМЕРЫ ИТЕРАТОРОВ

Классический способ выполнить обход всех ключей *словаря*, например, состоит в том, чтобы явно запросить список ключей.

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D.keys():
...     print(key, D[key])
```

```
a1
c3
b2
```



ПРИМЕРЫ ИТЕРАТОРОВ

Или можно проитерировать:

```
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'c'
>>> next(I)
'b'
>>> next(I)
Traceback (most recent call last):
...текст сообщения об ошибке опущен...
StopIteration
```



ПРИМЕРЫ ИТЕРАТОРОВ

Или еще:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D:
...     print(key, D[key])
a1
c3
b2
```



ПРИМЕРЫ ИТЕРАТОРОВ

Итератор `range`:

Функция `range` возвращает итератор, который генерирует список целых чисел в заданном диапазоне по требованию.

```
>>> R = range(10)
>>> R
range(0, 10)
>>> I = iter(R)    # Вернет итератор для диапазона
>>> next(I)        # Переход к следующему результату
0
>>> next(I)
1
>>> list(range(10)) # Получили список с результатом
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



Функция **enumerate** создает итератор, нумерующий элементы другого итератора. Результирующий итератор выдает кортежи, в которых первый элемент – номер (начиная с нуля), а второй – элемент исходной последовательности

```
print [x for x in enumerate("abcd")]  
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

```
words = 'Мама мыла раму'.split()  
ordered = enumerate(words)  
print(*ordered)
```



ГЕНЕРАТОРЫ СПИСКОВ

```
>>> L = [1, 2, 3, 4, 5]
```

```
>>> for i in range(len(L)):
    L[i] += 10
```

```
>>> L
[11, 12, 13, 14, 15]
```

VS

```
>>> L = [x + 10 for x in L] >>> L
[21, 22, 23, 24, 25]
```

В чем разница?



Выражения генераторов списков нельзя считать равнозначной заменой инструкции цикла `for`, потому что они ***создают новые объекты списков*** (что может иметь значение при наличии нескольких ссылок на первоначальный список).

Более того, генераторы списков могут выполняться значительно ***быстрее*** (зачастую почти в два раза), чем инструкции циклов `for`, потому что итерации выполняются со скоростью языка C, а не со скоростью программного кода на языке Python. Такое преимущество в скорости особенно важно для больших объемов данных.



Использование генераторов списков для работы с файлами

```
>>> f = open('script1.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```



Использование генераторов списков для работы с файлами

```
>>> f = open('script1.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

VS

```
>>> lines = [line.rstrip() for line in open('script1.py')]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```



Итераторы map, zip и filter:

Встроенные функции map, zip и filter возвращают итераторы.

```
>>> M = map(abs, (-1, 0, 1)) # возвращает итератор
```

```
>>> M
```

```
<map object at 0x0276B890>
```

```
>>> next(M) # использование итератора
```

```
1 # результаты исчерпываются безвозвратно.
```

```
>>> next(M)
```

```
0
```

```
>>> next(M)
```

```
1
```

```
>>> next(M)
```

```
StopIteration
```



После однократного получения отдельного результата итератора, этот результат исчезает.

```
>>> M = map(abs, (-1, 0, 1))
```

```
# Для выполнения второго прохода, необходимо создать итератор
```

```
>>> for x in M: print(x) #next() вызывается автоматически
```

```
1
```

```
0
```

```
1
```

```
>>> list(map(abs, (-1, 0, 1)))
```

```
#получаем список с результатами
```



zip – составление пар

Найдем скалярное произведение двух векторов в пространстве

```
a = (2, 3, -1)
b = (3, 0, 5)
mul = 0
for i in range(3):
    mul += a[i] * b[i]
print(mul)
```

```
a = (2, 3, -1)
b = (3, 0, 5)
def mul(i):
    return a[i] * b[i]
print(sum(mul(i) for i in range(3)))
```



Найдем скалярное произведение двух векторов в пространстве (Python-вариант)

```
a = (2, 3, -1)
b = (3, 0, 5)
print(sum(p[0] * p[1] for p in zip(a, b)))
```

Пример на строках

```
>>> list(zip('abc', 'xyz')) # список результатов
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```



filter – фильтрация коллекции по условию

filter(function, iterable) – возвращает итератор из тех элементов, для которых function возвращает истину.

```
a = [1, -4, 6, 8, -10]
```

```
def func(x):
```

```
    if x > 0:
```

```
        return 1
```

```
    else:
```

```
        return 0
```

```
b = filter(func, a)
```

```
b = list(b)
```

```
print(b)
```

```
[1, 6, 8]
```

Функция func()
возвращает 1,
если ей передан
аргумент больше
нуля, и 0 во всех
остальных
случаях.




```
import random
```

```
def odd(n):
```

```
    return n%2 != 0
```

```
seq = [random.randrange(1, 100) for i in range(10)]
```

```
print(seq)
```

```
odd_seq = filter(odd, seq)
```

```
print(list(odd_seq))
```

```
a = [-1,0,1,0,0,1,0,-1]
```

```
b = list(filter(None,a))
```

```
print(b)
```

```
[-1, 1, 1, -1]
```

Если вместо функции в качестве первого аргумента `filter()` передается значение `None`, то в отфильтрованном объекте окажутся те значения, которые сами по себе являются `true`.



```
s = ['a', ',', 'd', 'cc', ' ']  
ss = list(filter(None, s))  
print(ss)  
['a', 'd', 'cc', ' ']
```

```
def numbs(x):  
    if '0' <= x <= '9':  
        return 1  
    else:  
        return 0  
s = "5a 3 k 99 d00"  
for i in filter(numbs,s):  
    print(i)
```



Найдем номера слов в строке, начинающихся на «ма»

```
# получили список слов в строке
words = input().split()
# перенумеровали его
ordered = enumerate(words)

# отфильтровали список
def feature(s):
    return s[1][:2].lower() == "ma"

filtered = filter(feature, ordered)

# вывели на печать
print(*[pair[0] for pair in filtered])
```



ИСПОЛЬЗОВАНИЕ ИТЕРАТОРОВ

```
>>> map(str.upper, open('script1.py'))
```

```
<map object at 0x02660710>
```

```
>>> 'x = 2\n' in open('script1.py')
```

```
True
```

```
>>> sorted(open('script1.py'))
```

```
['import sys\n', 'print(2 ** 33)\n', 'print(sys.path)\n', 'x = 2\n']
```

```
>>> list(zip(open('script1.py'), open('script1.py')))
```

```
[('import sys\n', 'import sys\n'), ('print(sys.path)\n',  
, 'print(sys.path)\n'), ('x = 2\n', 'x = 2\n'), ('print(2 ** 33)\n', 'print(  
2 ** 33)\n')]
```

```
>>> list(enumerate(open('script1.py')))
```

```
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'), (3, 'print(  
2 ** 33)\n')]
```

```
>>> list(filter(bool, open('script1.py')))
```

```
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
```



ЧТО ИЗУЧИЛИ?



ЧТО ИЗУЧИЛИ?

Итераторы:

Сильно. Выгодно. Надежно!

Более подробные условия узнавайте по телефону:
+7916138****

Внимание! Номер телефона был скрыт в соответствии с политикой безопасности компании



ПРАКТИКА

1. Написать функцию, принимающую итерируемый объект и выводящий его элементы. Без использования **for**. Вызвать ее для списка, списка словарей, значений словаря.
2. Написать свою функцию **enumerate**, занимающую как можно меньше памяти.
3. Написать пример генератора списка, его же переписать через итераторы.
4. В одну строку (!) удалить из файла лишние пробелы в начале и конце строки и перевести все буквы uppercase.
(Генератором)



ПРАКТИКА

5. Получить список пар номеров и значений первых n чисел Фибоначчи:
 $((1,1), (2,1), (3,2), (4,3), \dots)$
6. Написать генератор, создающий список элементов перевернутой введенной строки.
7. Написать генератор последовательности факториалов натуральных чисел
8. Написать генератор последовательности простых чисел

