

**КАЛУЖСКИЙ ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО
ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ Н.Э. БАУМАНА
(национальный исследовательский университет)»**



Факультет «Информатика и управление»

Кафедра «Программное обеспечение ЭВМ, информационные технологии»

Высокоуровневое программирование

Лекция №14. «Итерируемый объект, итератор и генератор»

Калуга - 2020

Содержание

В этом разделе рассматриваются:

- итерируемые объекты (**iterable**)
- итераторы (**iterator**)
- генераторные выражения (**generator expression**)

Итерируемый объект

- Итерация - это общий термин, который описывает процедуру взятия элементов чего-то по очереди.
- В более общем смысле, это последовательность инструкций, которая повторяется определенное количество раз или до выполнения указанного условия.
- Итерируемый объект (**iterable**) - это объект, который способен возвращать элементы по одному. Кроме того, это объект, из которого можно получить итератор.

Итерируемый объект

- Примеры итерируемых объектов:
 - все последовательности: список, строка, кортеж
 - словари
 - файлы

Итерируемый объект

- В Python за получение итератора отвечает функция **`iter()`**:

```
1 lst = [1, 2, 3]
2 print(iter(lst))
3 tpl = (4, 5, 6)
4 print(iter(tpl))
```

`<list_iterator object at 0x05511688>`

`<tuple_iterator object at 0x05511688>`

Итерируемый объект

- Функция **`iter()`** отработает на любом объекте, у которого есть метод **`__iter__`** или метод **`__getitem__`**.
- Метод **`__iter__`** возвращает итератор. Если этого метода нет, функция **`iter()`** проверяет, нет ли метода **`__getitem__`** - метода, который позволяет получать элементы по индексу.
- Если метод **`__getitem__`** есть, возвращается итератор, который проходится по элементам, используя индекс (начиная с 0).

Итерируемый объект

- На практике использование метода **`__getitem__`** означает, что все последовательности элементов - это итерируемые объекты. Например, список, кортеж, строка. Хотя у этих типов данных есть и метод **`__iter__`**.

```
1 print(lst.__iter__())  
2 print(lst.__getitem__(0))
```

```
<list_iterator object at 0x05511688>
```

```
1
```

Итератор

- Итератор (**iterator**) - это объект, который возвращает свои элементы по одному за раз.
- С точки зрения Python - это любой объект, у которого есть метод **__next__**. Этот метод возвращает следующий элемент, если он есть, или возвращает исключение **StopIteration**, когда элементы закончились.
- Кроме того, итератор запоминает, на каком объекте он остановился в последнюю итерацию.

Итератор

- В Python у каждого итератора присутствует метод `__iter__` - то есть, любой итератор является итерируемым объектом. Этот метод просто возвращает сам итератор.
- Пример создания итератора из списка:

```
1 numbers = [1, 2, 3]
2 i = iter(numbers)
3 print(next(i), end=' ')
4 print(next(i), end=' ')
5 print(next(i), end=' ')
6 print(next(i), end=' ')
```

1 2 3

StopIteration

Итератор

- Для того, чтобы итератор снова начал возвращать элементы, его надо заново создать.
- Аналогичные действия выполняются, когда цикл **for** проходится по списку:

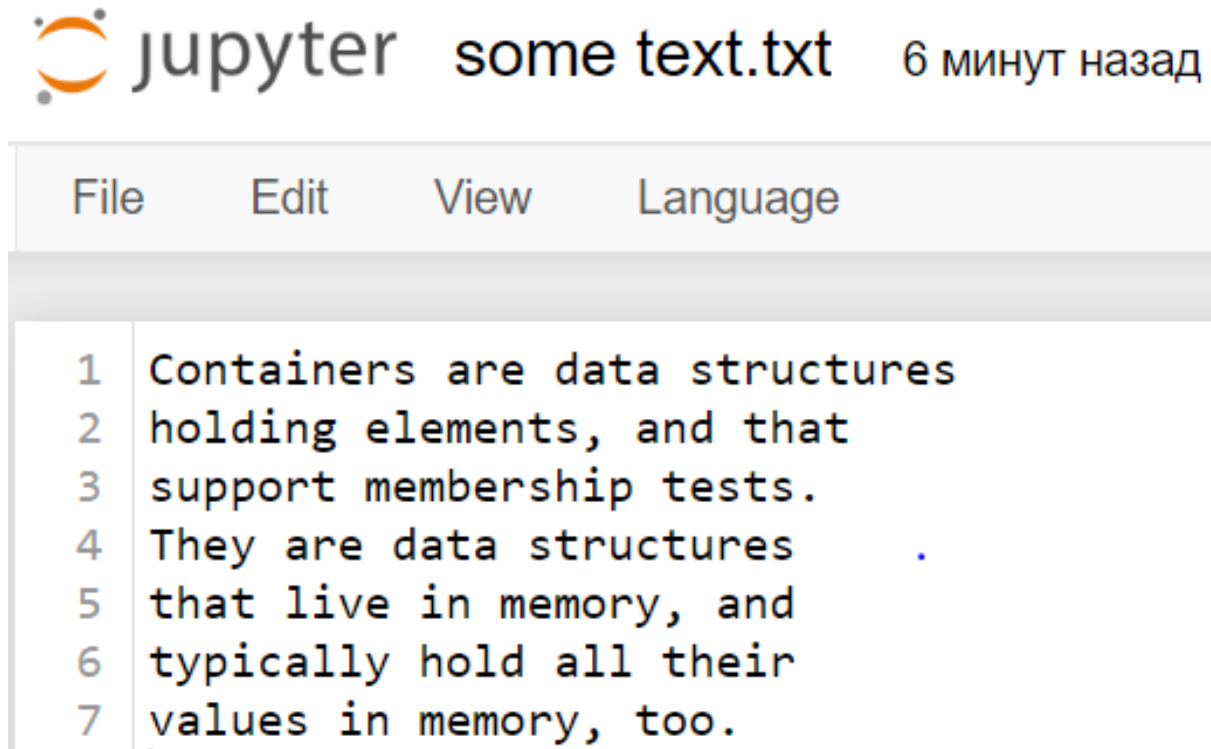
```
1 for item in numbers:  
2     print(item, end=' ')
```

1 2 3

- Когда мы перебираем элементы списка, к списку сначала применяется функция **iter()**, чтобы создать итератор, а затем вызывается его метод **__next__** до тех пор, пока не возникнет исключение **StopIteration**.

Файл как итератор

- Один из самых распространенных примеров итератора - файл.



The image shows a Jupyter Notebook interface. At the top, the Jupyter logo is followed by the filename 'some text.txt' and the text '6 минут назад'. Below this is a menu bar with 'File', 'Edit', 'View', and 'Language'. The main area contains a code cell with the following text:

```
1 Containers are data structures
2 holding elements, and that
3 support membership tests.
4 They are data structures
5 that live in memory, and
6 typically hold all their
7 values in memory, too.
```

Файл как итератор

- Если открыть файл обычной функцией **open**, мы получим объект, который представляет файл:

```
1 f = open('some text.txt')
```

- Этот объект является итератором, что можно проверить, вызвав метод **__next__**:

```
1 next(f)
```

```
'Containers are data structures \n'
```

```
1 f.__next__()
```

```
'holding elements, and that \n'
```

Файл как итератор

- Аналогичным образом можно перебирать строки в цикле **for**:

```
1 for line in f:  
2     print(line.rstrip())
```

support membership tests.
They are data structures
that live in memory, and
typically hold all their
values in memory, too.

- При работе с файлами, использование файла как итератора не просто позволяет перебирать файл построчно - в каждую итерацию загружена только одна строка. Это очень важно при работе с большими файлами на тысячи и сотни тысяч строк

Генератор

- Генераторы - это специальный класс функций, который позволяет легко создавать свои итераторы. В отличие от обычных функций, генератор не просто возвращает значение и завершает работу, а возвращает итератор, который отдает элементы по одному.
- Обычная функция завершает работу, если:
 - встретилось выражение **return**;
 - закончился код функции (это срабатывает как выражение **return None**);
 - возникло исключение.

Генератор

- После выполнения функции управление возвращается, и программа выполняется дальше. Все аргументы, которые передавались в функцию, локальные переменные, все это теряется. Остается только результат, который вернула функция.
- Функция может возвращать список элементов, несколько объектов или возвращать разные результаты в зависимости от аргументов, но она всегда возвращает какой-то один результат.

Генератор

- Генератор же генерирует значения. При этом значения возвращаются по запросу, и после возврата одного значения выполнение функции-генератора приостанавливается до запроса следующего значения. Между запросами генератор сохраняет свое состояние.
- Python позволяет создавать генераторы двумя способами:
 - генераторное выражение;
 - функция-генератор.

Генераторное выражение

- Генераторное выражение использует такой же синтаксис, как **list comprehensions**, но возвращает итератор, а не список.
- Генераторное выражение выглядит точно так же, как **list comprehensions**, но используются круглые скобки:

Обратите внимание, что это не **tuple comprehensions**, а генераторное выражение.

```
1 genexpr = (x**2 for x in range(10,10000))
```

```
1 genexpr
```

```
<generator object <genexpr> at 0x00630BF8>
```

```
1 next(genexpr)
```

```
100
```

```
1 next(genexpr)
```

```
121
```

List comprehensions (генераторы списков)

```
1 numbers = ['n: {}'.format(num) for num in range(10,16)]  
2 numbers
```

```
['n: 10 ', 'n: 11 ', 'n: 12 ', 'n: 13 ', 'n: 14 ', 'n: 15 ']
```

```
1 items = ['10', '20', 'a', '30', 'b', '40']  
2 only_digits = [int(item) for item in items if item.isdigit()]  
3 only_digits
```

```
[10, 20, 30, 40]
```

```
[expression for item1 in iterable1 if condition1  
             for item2 in iterable2 if condition2  
             ...  
             for itemN in iterableN if conditionN ]
```

Dict comprehensions (генераторы словарей)

```
1 d = {num: num**2 for num in range(1,11)}  
2 print(d)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

```
1 d = {  
2     'Moscow': 'Russian Federation',  
3     'London': 'England',  
4     'Madrid': 'Spain',  
5     'Paris': 'France'  
6 }  
7  
8 d_upper = {str.upper(key): str.upper(value) for key, value in d.items()}  
9 print(d_upper)
```

```
{'MOSCOW': 'RUSSIAN FEDERATION', 'LONDON': 'ENGLAND', 'MADRID': 'SPAIN', 'PARIS': 'FRANCE'}
```

Set comprehensions (генераторы множеств)

```
1 numbers = [10, '30', 30, 10, '56']  
2 unique_numbers = {int(num) for num in numbers}  
3 print(unique_numbers)
```

{56, 10, 30}

Функция-генератор

- Генераторы - это специальный класс функций, который позволяет легко создавать свои итераторы. В отличии от обычных функций, генератор не просто возвращает значение и завершает работу, а возвращает итератор, который отдает элементы по одному.
- Более корректное определение: функция-генератор - это функция, в которой присутствует ключевое слово **yield**. При вызове, эта функция возвращает объект генератор. Так как и сама функция и объект, который она возвращает, называется генератор, возникает путаница, о чем идет речь.

Функция-генератор

- С точки зрения синтаксиса, генератор выглядит как обычная функция. Но, вместо **return**, используется оператор **yield**.
- Каждый раз, когда внутри функции встречается **yield**, генератор приостанавливается и возвращает значение. При следующем запросе, генератор начинает работать с того же места, где он завершил работу в прошлый раз.
- Так как **yield** не завершает работу генератора, он может использоваться несколько раз.

Функция-генератор

- Если вызвать генератор и присвоить результат в переменную, его код еще не будет выполняться:

```
1  def generate_nums(number):  
2      print('Start of generation')  
3      yield number  
4      print('Next number')  
5      yield number+1  
6      print('The end')  
7  
8  result = generate_nums(100)  
9  result
```

<generator object generate_nums at 0x0625C3E0>

Функция-генератор

- Теперь в переменной **result** находится итератор.
- Раз **result** это итератор, можно вызвать функцию **next**, чтобы получить значение:

```
1 next(result)
```

Start of generation

100

```
1 next(result)
```

Next number

101

```
1 next(result)
```

The end

StopIteration

Функция-генератор

- Раз функция-генератор возвращает итератор, его можно использовать в цикле:

```
1 for num in generate_nums(100):  
2     print('Number:', num)
```

Start of generation

Number: 100

Next number

Number: 101

The end

Обычная функция и аналогичный генератор

- С помощью генераторов зачастую можно написать ту же функцию с меньшим количеством промежуточных переменных.

```
1 def work_with_items(items):  
2     result = []  
3     for item in items:  
4         result.append('n: {}'.format(item))  
5     return result  
6  
7 for i in work_with_items(range(10)):  
8     print(i, end = '; ')
```

n: 0; n: 1; n: 2; n: 3; n: 4; n: 5; n: 6; n: 7; n: 8; n: 9;

Обычная функция и аналогичный генератор

- Можно заменить таким генератором:

```
1 def yield_items(items):  
2     for item in items:  
3         yield 'n: {}'.format(item)  
4  
5 for i in yield_items(range(10)):  
6     print(i, end = '; ')
```

n: 0; n: 1; n: 2; n: 3; n: 4; n: 5; n: 6; n: 7; n: 8; n: 9;

Генератор

```
1 def fibonacci():
2     prev, cur = 0, 1
3     while True:
4         yield prev
5         prev, cur = cur, prev + cur
6
7 fib = fibonacci()
8 print(next(fib))
9 print(next(fib))
```

0

1

```
1 for num, fib in enumerate(fibonacci()):
2     print('{0}: {1}'.format(num, fib), end='; ')
3     if num > 9:
4         break
```

0: 0; 1: 1; 2: 1; 3: 2; 4: 3; 5: 5; 6: 8; 7: 13; 8: 21; 9: 34; 10: 55;

Генератор

- В результате вызова функции или вычисления выражения, получаем объект-генератор типа **`types.GeneratorType`**.
- В объекте-генераторе определены методы **`__next__`** и **`__iter__`**, то есть реализован протокол итератора, с этой точки зрения, в Python любой генератор является итератором.
- Концептуально, итератор — это механизм поэлементного обхода данных, а генератор позволяет отложено создавать результат при итерации. Генератор может создавать результат на основе какого то алгоритма или брать элементы из источника данных(коллекция, файлы, сетевое подключения и пр) и изменять их.
- Ярким пример являются функции **`range`** и **`enumerate`**:

Генератор

- **range** генерирует ограниченную арифметическую прогрессию целых чисел, не используя никакой источник данных.
- **enumerate** генерирует двухэлементные кортежи с индексом и одним элементом из итерируемого объекта.
- Любая функция в Python, в теле которой встречается ключевое слово **yield**, называется генераторной функцией – при вызове она возвращает объект-генератор.
- Объект-генератор реализует интерфейс итератора, соответственно с этим объектом можно работать, как с любым другим итерируемым объектом.

Резюме

a generator
expression



is

a generator



always is

an iterator



next()

*lazily produce
next value*

is

a generator
function



always is

iter()



(an) iterable

typically is

a container



produces



{list, set, dict}
comprehension

Задачи для самостоятельного решения

- Даны n предложений. Определите, сколько из них содержат хотя бы одну цифру.
- Дана строка s и символ k . Реализуйте функцию, рисующую рамку из символа k вокруг данной строки, например:

```
*****  
*Текст в рамке*  
*****
```
- Для введенного предложения выведите статистику *символ=количество*. Регистр букв не учитывается.
- Дата характеризуется тремя натуральными числами: день, месяц и год. Учитывая, что год может быть високосным, реализуйте две функции, которые определяют вчерашнюю и завтрашнюю дату.

Задачи для самостоятельного решения

- Напишите функцию, которая принимает неограниченное количество числовых аргументов и возвращает кортеж из двух списков:
 - отрицательных значений (отсортирован по убыванию);
 - неотрицательных значений (отсортирован по возрастанию).
- Составьте две функции для возведения числа в степень: один из вариантов реализуйте в рекурсивном стиле.
- Дано натуральное число. Напишите рекурсивные функции для определения:
 - суммы цифр числа;
 - количества цифр в числе.