

Углубленный курс информатики

Лекция 6
Функциональные возможности Python.
Итераторы

14 марта 2022 г.

Долганов Игорь Михайлович,
к.т.н., доцент ОХИ ИШПР

Содержание

1. Функциональные возможности Python
 - Анонимные (`lambda`) функции
 - Функция `map`
 - Функция `filter`
 - Функция `reduce`
2. Итерируемые объекты и итераторы
 - Итерируемые объекты
 - Итераторы
 - Функция `sorted`
 - Функция `reversed`
 - Функция `zip`
 - Функция `enumerate`
3. Функции-генераторы и генераторные выражения
 - Функции-генераторы
 - Генераторные выражения

Функциональные возможности Python

Анонимные (`lambda`) функции

- Помимо инструкции `def` в языке Python имеется возможность создавать объекты функций в форме выражений.
- Подобно инструкции `def` это выражение создает функцию, которая будет вызвана позднее, но в отличие от инструкции `def`, выражение возвращает функцию, а не связывает ее с именем.
- Именно поэтому `lambda`-выражения иногда называют анонимными (т.е. безымянными) функциями.
- На практике они часто используются как способ получить встроенную функцию или отложить выполнение фрагмента программного кода.
- Общая форма `lambda`-функций выглядит как ключевое слово `lambda`, за которым следует один или больше аргументов (очень похоже на список аргументов, заключенный в круглые скобки в заголовке `def`) и далее выражение после двоеточия:

`lambda` аргумент1, аргумент2 ... аргументN: выражение, использующее аргументы

Анонимные (`lambda`) функции

- Тело `lambda` является одиночным выражением, а не блоком операторов и похоже на то, что было бы указано в операторе `return` внутри тела `def`.
- Из-за ограничения только выражением, тело `lambda` менее универсально, чем `def` – можно помещать в тело `lambda` лишь определенную логику, не использующую операторы вроде `if`.
- Так было задумано, чтобы ограничить вложенность в программе: выражение `lambda` предназначено для записи простых функций, а оператор `def` можно использовать для решения более крупных задач.

```
1 >>> def f(x, y):
2     ...     return x + y
3
4 >>> f(1, 2)
5 3
6
```

```
1 >>> f = lambda x, y: x + y
2
3 >>> f(1, 2)
4 3
5
```

Использование `lambda`-выражений

- Выражение `lambda` полезно как своего рода краткое условное обозначение функции, которое позволяет встраивать определение функции внутрь кода, где оно применяется.
- В сценариях, где нужно всего лишь встраивать небольшие порции кода, выражения `lambda` окажутся более простыми кодовыми конструкциями, чем операторы `def`.

```

1  >>> from random import randint
2
3  >>> x = [randint(-5, 5) for _ in range(10)]
4
5  >>> x
6  [-4, -2, -4, 2, -5, 0, -4, 4, -5, -3]
7
8  >>> min(x, key=lambda x: x ** 2 - 9 * x + 5)
9  4
10
11 >>> y = [el ** 2 - 9 * el + 5 for el in x]
12
13 >>> y #фактически, поиск минимума происходил в этом списке,
14      #но результат - соответствующий элемент из списка x
15 [57, 27, 57, -9, 75, 5, 57, -15, 75, 41]
16

```

Использование `lambda`-выражений

- Выражения `lambda` широко используются при реализации таблиц переходов, которые представляют собой списки или словари действий, подлежащих выполнению по запросу.

```
1 >>> function_list = [lambda x: x ** 2, #Список из трех вызываемых
2 ...                 lambda x: x ** 3, #функций
3 ...                 lambda x: x ** 4]
4 >>> for f in function_list:
5 ...     print(f(3))
6 ...
7 9
8 27
9 81
10 >>> function_list[0](5)
11 25
12
```

Использование `lambda`-выражений

- Эквивалентный код с `def` потребовал бы временных имен функций (они могли бы конфликтовать с остальными именами) и определений функций за пределами контекста их планируемого применения (который может находиться на сотни строк дальше):

```
1  >>> def f1(x):
2  ...     return x ** 2
3  ...
4  >>> def f2(x):
5  ...     return x ** 3
6  ...
7  >>> def f3(x):
8  ...     return x ** 4
9  ...
10 >>> function_list = [f1, f2, f3]
11 >>> for f in function_list:
12 ...     print(f(3))
13 ...
14 9
15 27
16 81
17 >>> print(function_list[0](5))
18 25
19
```


Использование `lambda`-выражений

- В действительности можно использовать словари и другие структуры данных в Python для построения более универсальных разновидностей таблиц действий:

```
1 >>> key = "got"
2 >>> actions = {"already": lambda: 2 + 2,
3 ...           "got": lambda: 5 * 5,
4 ...           "one": lambda: 2 ** 6}
5
6 >>> actions[key]()
7 25
8
```

- Когда Python создает словарь `actions`, каждое вложенное выражение `lambda` генерирует функцию, подлежащую вызову в будущем.
- Индексация по ключу извлекает одну из функций, а круглые скобки приводят к вызову извлеченной функции.
- При такой реализации словарь становится более универсальным инструментом для множественного ветвления.

Использование `lambda`-выражений

- Версия без `lambda`, потребует три оператора `def` где-то в файле за пределами словаря, в котором функции будут применяться, и ссылаться на функции по их именам:

```

1  >>> def f1():
2  ...     return 2 + 2
3  ...
4  >>> def f2():
5  ...     return 5 * 5
6  ...
7  >>> def f3():
8  ...     return 2 ** 6
9  ...
10 >>> key = "one"
11 >>> actions = {"already": f1, "got": f2, "one": f3}
12 >>> actions[key]()
13 64
14
```

- Операторы `def` могут находиться произвольно далеко в файле, даже если они представляют собой лишь небольшие порции кода. Выражения `lambda` особенно полезны для функций, которые будут использоваться только в одном контексте.
- Выражение `lambda` создано для небольших частей внутристрочного кода; при необходимости написания более сложной логики используйте оператор `def`.

Функция `map`

- Одним из наиболее часто встречающихся действий, выполняемых в программах со списками и другими последовательностями, является применение какой-то операции к каждому элементу и накопление результатов.
- В Python есть много инструментов, которые облегчают написание кода таких операций в масштабах целой коллекции. Например, обновление всех счетчиков в списке несложно обеспечить в цикле `for`:

```
1 >>> counters = [1, 2, 3, 4, 5]
2
3 >>> updated = []
4
5 >>> for x in counters:
6     ...     updated.append(x + 10)
7     ...
8 >>> updated
9 [11, 12, 13, 14, 15]
10
```

Функция `map`

- Встроенная функция `map` применяет переданную функцию к каждому элементу в итерируемом объекте и возвращает список, содержащий все результаты вызовов функции.

```

1  >>> def inc(x): #Функция, подлежащая выполнению
2  ...     return x + 10
3  ...
4  >>> list(map(inc, counters)) #Накапливание результатов
5  [11, 12, 13, 14, 15]
6
    
```

- Здесь функция `map` вызывает функцию `inc` для каждого элемента списка и собирает все возвращаемые значения в новый список.
- Поскольку встроенная функция `map` ожидает передачи функции, применяемой к элементам, она также относится к тем местам, где обычно появляется выражение `lambda`:

```

1  >>> list(map(lambda x: x * 2, counters))
2  [2, 4, 6, 8, 10]
3
    
```

- Подлежащая применению функция умножает на 2 каждый элемент в списке `counters`; так как эта небольшая функция далее нигде не нужна, она была записана как внутрискочная посредством выражения `lambda`.

Функция `map`

- Функция `map` – встроенная функция, она всегда доступна, всегда работает одинаково и дает преимущества в плане производительности (в некоторых режимах применения она быстрее вручную написанного цикла `for`).
- Функцию `map` можно использовать более развитыми способами. Например, получив в качестве аргументов несколько последовательностей, `map` передает извлеченные из последовательностей элементы как индивидуальные аргументы функции `pow`:

```
1 >>> pow(2, 3) #2 ** 3
2 8
3 >>> list(map(pow, [1, 2, 3], [4, 5, 6])) #1 ** 4, 2 ** 5, 3 ** 6
4 [1, 32, 729]
5
```

- В случае множества последовательностей `map` ожидает функцию с `n` аргументами для `n` последовательностей. Функция `pow` принимает в каждом вызове два аргумента – по одному из каждой последовательности, переданной `map`.

- Вызов `map` похож на выражения генераторов списков:

```
1 >>> list(map(inc, [10, 20, 30, 40]))
2 [20, 30, 40, 50]
3
4 >>> [inc(x) for x in [10, 20, 30, 40]]
5 [20, 30, 40, 50]
6
```

- Поскольку `map` применяет к каждому элементу вызов функции, а не произвольное выражение, она является менее универсальным инструментом и часто требует добавочных вспомогательных функций или `lambda`-выражений.

Функция `filter`

- Встроенная функция `filter` выбирает элементы из итерируемого объекта на основе проверочной функции.
- Из-за возвращения итерируемого объекта функция `filter` (подобно `range`) требует вызова `list` при отображении всех ее результатов.
- Например, следующий вызов `filter` выбирает из последовательности элементы больше нуля.

```
1 >>> list(range(-5, 5))
2 [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
3
4 >>> list(filter(lambda x: x > 0, range(-5, 5)))
5 [1, 2, 3, 4]
6
```

- Элементы из последовательности либо итерируемого объекта, для которых проверочная функция возвращает истинное значение, добавляются в результирующий список.

Функция `filter`

- Как и `map`, функция `filter` приблизительно эквивалентна циклу `for`, но является встроенной, лаконичной и часто более быстрой:

```
1 >>> res = []
2 >>> for x in range(-5, 5):
3     ...     if x > 0:
4     ...         res.append(x)
5     ...
6 >>> res
7 [1, 2, 3, 4]
8
```

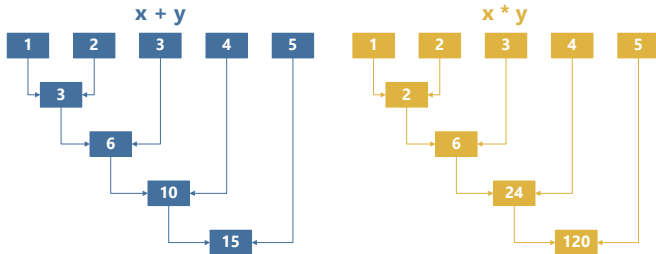
- Также подобно `map` функцию `filter` можно эмулировать посредством синтаксиса генератора списка с часто более простыми результатами (особенно если удастся избежать создания новой функции):

```
1 >>> [x for x in range(-5, 5) if x > 0]
2 [1, 2, 3, 4]
3
```


Функция `reduce`

- Функция `reduce`, которая находится в модуле `functools`, принимает итерируемый объект, подлежащий обработке, но сама возвращает одиночный результат.

```
1 >>> from functools import reduce
2
3 >>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
4 15
5
6 >>> reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
7 120
8
```



Функция `reduce`

- На каждом шаге `reduce` передает текущую сумму или произведение вместе с очередным элементом из списка указанной функции `lambda`.
- По умолчанию начальное значение инициализируется первым элементом последовательности.
- В целях демонстрации рассмотрим эквивалент первого из двух вызовов в виде цикла `for` с жестко закодированным сложением внутри цикла:

```

1  >>> arr = [1, 2, 3, 4, 5]
2
3  >>> res = arr[0]
4
5  >>> for x in arr[1:]:
6  ...     res += x
7  ...
8  >>> res
9  15
10

```

- Встроенная функция `reduce` позволяет передавать необязательный третий аргумент, который помещается перед элементами последовательности и служит начальным значением, а также стандартным результатом, когда последовательность пуста.

Итерируемые объекты и итераторы

Итерируемые объекты

- **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.

Примеры итерируемых объектов

- Списки
- Строки
- Кортежи
- Словари
- Множества

- **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.
- В Python у каждого итератора присутствует метод `__iter__` – то есть, любой итератор является **итерируемым объектом**. Этот метод просто возвращает сам итератор.
- В Python за получение итератора отвечает функция `iter`:

```
1 | >>> some_numbers = [1, 2, 3]
2 |
3 | >>> iter(some_numbers)
4 | <list_iterator at 0xb4ede28c>
5 |
```

- Функция `iter` работает с любым объектом, 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.

Итераторы

- Пример создания итератора из списка:

```

1  >>> numbers = [1, 2, 3]
2  >>> i = iter(numbers)
3

```

- Теперь можно использовать функцию `next`, которая вызывает метод `__next__`, чтобы взять следующий элемент:

```

1  >>> next(i)
2  1
3  >>> next(i)
4  2
5  >>> next(i)
6  3
7  >>> next(i)
8  -----
9  StopIteration                                Traceback (most recent call last)
10 <ipython-input-8-bed2471d02c1> in <module>()
11 -----> 1 next(i)
12

```

- После того, как элементы закончились, возвращается исключение `StopIteration`.
- Для того, чтобы итератор снова начал возвращать элементы, его нужно создать заново.

- Аналогичные действия выполняются, когда цикл `for` проходит по списку:

```
1  >>> for item in numbers:
2      ...     print(item)
3      ...
4      1
5      2
6      3
7
```

- 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`.
- Итераторы полезны тем, что они отдают элементы по одному.
- Например, при работе с большими коллекциями в памяти будет находиться не все ее элементы, а только один.

Функция `sorted`

- Функция `sorted` возвращает **новый отсортированный список**, который получен из *итерируемого* объекта, который был передан как аргумент. Функция также поддерживает дополнительные параметры, которые позволяют управлять сортировкой.
- Если сортировать **список** элементов, то возвращается **новый список**:

```
1 >>> list_of_words = ["one", "two", "list", "", "dict"]
2
3 >>> sorted(list_of_words)
4 ["", "dict", "list", "one", "two"]
5
```

- При сортировке **кортежа** также возвращается список:

```
1 >>> tuple_of_words = ("one", "two", "list", "", "dict")
2
3 >>> sorted(tuple_of_words)
4 ["", "dict", "list", "one", "two"]
5
```


Функция `sorted`

■ Сортировка множества:

```
1 >>> set_of_words = {"one", "two", "list", "", "dict"}
2
3 >>> sorted(set_of_words)
4 ["", "dict", "list", "one", "two"]
5
```

■ Сортировка строки:

```
1 >>> string_to_sort = "long string"
2
3 >>> sorted(string_to_sort)
4 >>> [" ", "g", "g", "i", "l", "n", "n", "o", "r", "s", "t"]
5
```

Функция `sorted`

- Если передать `sorted` словарь, функция вернет отсортированный список ключей:

```
1  >>> dict_for_sort = {
2      ...             "id": 1,
3      ...             "name": "London",
4      ...             "IT_VLAN": 320,
5      ...             "User_VLAN": 1010,
6      ...             "Mngmt_VLAN": 99,
7      ...             "to_name": None,
8      ...             "to_id": None,
9      ...             "port": "G1/0/11"
10     ... }
11
12 >>> sorted(dict_for_sort)
13 ["IT_VLAN",
14  "Mngmt_VLAN",
15  "User_VLAN",
16  "id",
17  "name",
18  "port",
19  "to_id",
20  "to_name"]
21
```

Параметр `reverse`

- Флаг `reverse` позволяет управлять порядком сортировки. По умолчанию сортировка будет по возрастанию элементов.
- Указав флаг `reverse`, можно поменять порядок:

```
1 >>> list_of_words = ["one", "two", "list", "", "dict"]
2
3 >>> sorted(list_of_words)
4 ["", "dict", "list", "one", "two"]
5
6 >>> sorted(list_of_words, reverse=True)
7 ["two", "one", "list", "dict", ""]
8
```

Дополнительные аргументы функции `sorted`

Параметр `key`

- С помощью параметра `key` можно указывать, как именно выполнять сортировку. Параметр `key` ожидает функцию с одним параметром, с помощью которой должно быть выполнено сравнение.
- Например, таким образом можно отсортировать список строк по длине строки:

```
1 >>> list_of_words = ["one", "two", "list", "", "dict"]
2
3 >>> sorted(list_of_words, key=len)
4 ["", "one", "two", "list", "dict"]
5
```

Дополнительные аргументы функции `sorted`

Параметр `key`

- Если нужно отсортировать ключи словаря, но при этом игнорировать регистр строк:

```
1  >>> dict_for_sort = {
2      ...             "id": 1,
3      ...             "name": "London",
4      ...             "IT_VLAN": 320,
5      ...             "User_VLAN": 1010,
6      ...             "Mngmt_VLAN": 99,
7      ...             "to_name": None,
8      ...             "to_id": None,
9      ...             "port": "G1/0/11"
10     ... }
11
12 >>> sorted(dict_for_sort, key=str.lower)
13 ["id",
14  "IT_VLAN",
15  "Mngmt_VLAN",
16  "name",
17  "port",
18  "to_id",
19  "to_name",
20  "User_VLAN"]
21
```

Дополнительные аргументы функции `sorted`

Параметр `key`

- Параметру `key` можно передавать любые функции, не только встроенные. Также тут удобно использовать анонимную функцию `lambda`.
- С помощью параметра `key` можно сортировать объекты не по первому элементу, а по любому другому.
- Например, чтобы отсортировать список кортежей из двух элементов по второму элементу, надо использовать такой прием:

```
1 >>> list_of_tuples = [("IT_VLAN", 320),
2 ...   ("Mngmt_VLAN", 99),
3 ...   ("User_VLAN", 1010),
4 ...   ("DB_VLAN", 11)]
5
6 >>> sorted(list_of_tuples, key=lambda x: x[1])
7 [("DB_VLAN", 11), ("Mngmt_VLAN", 99), ("IT_VLAN", 320),
8  ("User_VLAN", 1010)]
9
```

Функция `reversed`

- Функция `reversed` возвращает обратный итератор, то есть возвращает итератор, который перебирает элементы оригинала в обратном порядке.
- Функция `reversed` не создает копию и не изменяет оригинал последовательности.

```

1  >>> seq_string = "Python" #для строки
2  >>> list(reversed(seq_string))
3  ["n", "o", "h", "t", "y", "P"]
4
5  >>> seq_tuple = ("P", "y", "t", "h", "o", "n") #для кортежа
6  >>> list(reversed(seq_tuple))
7  ["n", "o", "h", "t", "y", "P"]
8
9  >>> seq_list = [1, 2, 4, 3, 5] #для списка
10 >>> list(reversed(seq_list))
11 [5, 3, 4, 2, 1]
12
13 >>> seq_range = range(5, 9) #для итерируемого объекта
14 >>> list(reversed(seq_range))
15 [8, 7, 6, 5]
16

```

- Так как функция `reversed` возвращает итерируемый объект, для получения всех элементов использовалась функция `list`.

Функция `zip`

- Функция `zip` принимает последовательности, а возвращает итератор с кортежами, в котором n -ый кортеж состоит из n -ых элементов последовательностей, которые были переданы как аргументы.
- Например, пятый кортеж будет содержать пятый элемент каждой из переданных последовательностей.
- Если на вход были переданы последовательности разной длины, то все они будут отрезаны по самой короткой последовательности.
- Порядок элементов сохраняется.

Примечание

Так как `zip` – это итератор, для отображения его содержимого используется `list`

```
1  >>> a = [1, 2, 3]
2
3  >>> b = [100, 200, 300]
4
5  >>> list(zip(a, b))
6  [(1, 100), (2, 200), (3, 300)]
7
```


Функция `zip`

- Использование `zip` со списками разной длины:

```
1 >>> a = [1, 2, 3, 4, 5]
2
3 >>> b = [10, 20, 30, 40, 50]
4
5 >>> c = [100, 200, 300]
6
7 >>> list(zip(a, b, c))
8 [(1, 10, 100), (2, 20, 200), (3, 30, 300)]
9
```

- Использование `zip` для создания словаря:

```
1 >>> d_keys = ["hostname", "location", "vendor", "model",
2             "IOS", "IP"]
3 >>> d_values = ["london_r1", "21 New Globe Walk", "Cisco",
4              "4451", "15.4", "10.255.0.1"]
5
6 >>> dict(zip(d_keys, d_values))
7 {"IOS": "15.4",
8  "IP": "10.255.0.1",
9  "hostname": "london_r1",
10 "location": "21 New Globe Walk",
11 "model": "4451",
12 "vendor": "Cisco"}
13
```

Функция `enumerate`

- Иногда, при переборе объектов в цикле `for`, нужно не только получить сам объект, но и его порядковый номер.
- Это можно сделать, создав дополнительную переменную, которая будет расти на единицу с каждым прохождением цикла.
- Однако, гораздо удобнее это делать с помощью итератора `enumerate`.

```
1 >>> list1 = ["str1", "str2", "str3"]
2
3 >>> for position, string in enumerate(list1):
4     ...     print(position, string)
5     ...
6 0 str1
7 1 str2
8 2 str3
9
```

Функция `enumerate`

- Функция `enumerate` умеет считать не только с нуля, но и с любого значения, которое ему указали после объекта:

```
1 >>> list1 = ["str1", "str2", "str3"]
2
3 >>> for position, string in enumerate(list1, 100):
4     ...     print(position, string)
5     ...
6 100 str1
7 101 str2
8 102 str3
9
```

- Если необходимо увидеть содержимое, которое сгенерирует итератор, полностью, можно воспользоваться функцией `list`:

```
1 >>> list1 = ["str1", "str2", "str3"]
2
3 >>> list(enumerate(list1, 100))
4 [(100, "str1"), (101, "str2"), (102, "str3")]
5
```

Функции-генераторы и генераторные выражения

- **Функция-генератор** – это функция, которая может возвращать значение, а позднее продолжить свою работу с того места, где она была приостановлена.
- Такая функция генерирует последовательность значений с течением времени, а также автоматически поддерживает протокол итераций и может использоваться в контексте итераций.
- Функции-генераторы записываются как обычные операторы `def`, но в них применяются операторы `yield`, вместо операторов `return`.
- Главное отличие функций-генераторов от обычных функций состоит в том, что они **поставляют значение**, а не возвращают его – оператор `yield` приостанавливает работу функции и передает значение вызывающей программе, при этом сохраняется информация о состоянии, необходимая для возобновления работы с того места, где она была приостановлена.
- Функция-генератор возобновляет работу, ее выполнение продолжается с первой инструкции, следующей после оператора `yield`. Это позволяет функциям воспроизводить последовательности значений в течение долгого времени, вместо того, чтобы создавать всю последовательность сразу и возвращать ее в виде некоторой конструкции, такой как список.

Функции-генераторы

- Определим функцию-генератор, которую можно применять для получения квадратов серии чисел с течением времени:

```

1 | >>> def generate_squares(n):
2 |     ...     for x in range(n):
3 |             ...     yield x ** 2
4 |     ...
5 |

```

- Функция `generate_squares` выдает значение и потому возвращает управление вызывающему коду на каждой итерации цикла; при возобновлении ее выполнения восстанавливается предыдущее состояние, включая последние значения переменных `x` и `n`, а управление снова подхватывается непосредственно после оператора `yield`.

```

1 | >>> for i in generate_squares(5):
2 |     ...     print(i, end=" | ")
3 |     ...
4 | 0 | 1 | 4 | 9 | 16 |
5 |

```

- При вызове функции-генератора напрямую будет получен объект генератора:

```

1 | >>> fg = generate_squares(5)
2 | >>> fg
3 | <generator object generate_squares at 0x00000217B6B79E40>
4 |

```

Функции-генераторы

- Возвращенный объект генератора имеет метод `__next__`, который запускает функцию или возобновляет ее выполнение с места, откуда она последний раз выдала значение, и иницирует исключение `StopIteration`, когда достигнут конец серии значений и происходит возврат управления из функции.
- Встроенная функция `next` вызывает метод `__next__` итерируемого объекта:

```
1  >>> next(fg)
2  0
3  >>> next(fg)
4  1
5  >>> next(fg)
6  4
7  >>> next(fg)
8  9
9  >>> next(fg)
10 16
11 >>> next(fg)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14   StopIteration
15
```

- После использования итерации по генератору, он **останется пустым**. Для повторной итерации придется создать новый объект генератора.

- Синтаксически генераторные выражения похожи на генераторы списков и поддерживают весь их синтаксис, в том числе фильтры `if` и вложение циклов, но они помещаются в круглые скобки.

```
1 >>> [x ** 2 for x in range(5)] #Генератор списка
2 [0, 1, 4, 9, 16]
3
4 >>> (x ** 2 for x in range(5)) #Генераторное выражение
5 <generator object <genexpr> at 0x00000217B6E4CCF0>
6
```

- Генераторные выражения возвращают объект генератора – автоматически созданный итерируемый объект.
- Этот итерируемый объект поддерживает протокол итерации, чтобы выдавать по одной порции результирующего списка за раз в любом итерационном контексте.
- Итерируемый объект также сохраняет состояние генератора, пока он действует – переменную `x` в предшествующих выражениях наряду с местоположением в коде генератора.

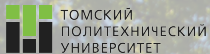
- Совокупный эффект во многом подобен функциям-генераторам, но в контексте выражения: мы получаем обратно объект, который запоминает оставленное им место после возвращения каждой части результата.

```
1 >>> g = (x ** 2 for x in range(5))
2 >>> iter(g) is g
3 True
4 >>> next(g)
5 0
6 >>> next(g)
7 1
8 >>> next(g)
9 4
10 >>> next(g)
11 9
12 >>> next(g)
13 16
14 >>> next(g)
15 Traceback (most recent call last):
16 File "<stdin>", line 1, in <module>
17 StopIteration
18
```

- Внутренне генераторные выражения имеют механизм итерации, в большинстве случаев скрытый от пользователей, т.к. циклы `for` запускают его автоматически:

```
1 >>> for number in (x ** 2 for x in range(5)):  
2 ...     print(f"{number}, {number / 2.0}")  
3 ...  
4 0, 0.0  
5 1, 0.5  
6 4, 2.0  
7 9, 4.5  
8 16, 8.0  
9
```

- В точности как функции-генераторы генераторные выражения обеспечивают оптимизацию *расхода памяти* – они не требуют создания сразу всего результирующего списка, что происходит в случае генератора списка.
- Также подобно функциям-генераторам они разделяют работу по выпуску результатов на небольшие временные интервалы – результаты выдаются постепенно вместо того, чтобы заставлять вызывающий код ожидать создания полного набора в единственном вызове.
- С другой стороны, на практике генераторные выражения могут выполняться несколько медленнее генераторов списков, а потому их лучше всего применять для очень крупных результирующих наборов или в приложениях, которые не могут ожидать генерации полных результатов.



Контакты

Долганов Игорь Михайлович,
к.т.н., доцент ОХИ ИШПР



Учебный корпус №2, ауд. 136



chuva@tpu.ru



+7-960-978-43-07

Благодарю за внимание!