ГЛАВА 1

Масштабирование с помощью генераторов

Этот цикл `for` кажется довольно простым:

```
for item in items:
    do_something_with(item)
```

И все же в нем происходит волшебство. Как вы, вероятно, знаете, поэлементный перебор коллекции называется *итерированием*. Мало кто понимает принцип работы механизма итерирования в Python и способен по достоинству оценить то, насколько он глубокий и продуманный. Эта глава сделает вас одним из таких людей. Благодаря ей вы научитесь писать *хорошо масштабируемые* приложения на языке Python, способные обрабатывать огромные массивы данных и отличающиеся высокой производительностью и эффективным использованием памяти.

Итерирование лежит и в основе одного из мощнейших инструментов Python — *функции-генератора*. Эти функции позволяют не только создавать полезные итераторы, но и реализовывать некоторые паттерны организации кода, способствующие формированию правильных привычек программирования.

Изучение материала данной главы позволит вам существенно улучшить навыки написания кода *на любом языке*. Именно этому, как правило, способствует освоение генераторов Python. Итак, приступим.

Итерация в Python

В Python есть встроенная функция `iter()`, которая принимает коллекцию и возвращает *объект-итератор*:

```
>>> numbers = [7, 4, 11, 3]
>>> iter(numbers)
<list_iterator object at 0x10219dc50>
```

Итератор — это объект, выдающий значения последовательно, по одному за раз:

```
>>> numbers_iter = iter(numbers)
>>> for num in numbers_iter:
...     print(num)
7
4
11
3
```

Как правило, вам нет необходимости использовать функцию `iter()`. Если вместо нее вы напишете `for num in numbers`, то Python автоматически применит функцию `iter()` к данной коллекции. При этом возвращенный объект, каким бы он ни был, будет применяться в качестве итератора в цикле `for`:

```
# Этот фрагмент...
for num in numbers:
    print(num)

# ...фактически эквивалентен этому:
numbers_iter = iter(numbers)
for num in numbers_iter:
    print(num)
```

Итератор коллекции — это отдельный объект, обладающий собственной идентичностью, которую можно проверить с помощью функции `id()`:

```
>>> # Функция id() возвращает уникальный для каждого объекта номер.
... # Разные объекты всегда имеют разные идентификаторы.
>>> id(numbers)
4330133896
>>> id(numbers_iter)
4330216640
```

Функция `iter()` может получить итератор несколькими способами, один из которых основан на *магическом методе* `__iter__()`. Он может определяться любым классом, в том числе вашим, и вызывается без аргументов. Каждый раз он создает новый объект-итератор. Данный метод предусмотрен, например, для списков:

```
>>> numbers._iter
<method-wrapper '_iter_' of list object at 0x10130e4c8>
>>> numbers._Iter_()
<list_iterator object at 0x1013180f0>
```

В Python различают *итераторы* и *итерируемые объекты*. Объект считается *итерируемым* тогда и только тогда, когда мы можем передать его в функцию

`iter()` и получить обратно готовый к использованию итератор. Если этот объект предусматривает метод `__iter__()`, то функция `iter()` вызовет его для получения итератора. Списки и кортежи Python — это итерируемые объекты, так же как и строки, поэтому вы можете задействовать код `for char in my_str:` для перебора символов `my_str`. Любой контейнер, который можно использовать в цикле `for`, является итерируемым.

Цикл `for` — это самый распространенный способ перебора элементов последовательностей. Если вам требуется более точный контроль, можете применить встроенную функцию `next()`. Обычно она вызывается с одним аргументом, представляющим собой итератор. В результате каждого вызова `next(my_iterator)` возвращает очередной элемент последовательности:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> # Создание нового итератора...
>>> names_it = iter(names)
>>> next(names_it)
'Tom'
>>> next(names_it)
'Shelly'
>>> next(names_it)
'Garth'
```

При очередном вызове `next(names_it)` функция `next()` сгенерирует специальное встроенное исключение `StopIteration`:

```
>>> next(names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Выбрасывая это специфическое исключение, итератор сигнализирует о завершении перебора элементов последовательности. Вам редко придется выбрасывать или перехватывать это исключение самостоятельно, хотя позднее мы рассмотрим несколько паттернов, в рамках реализации которых это бывает полезно. Чтобы понять принцип работы цикла `for`, представьте, что он вызывает функцию `next()` в ходе каждой итерации и завершается при возникновении исключения `StopIteration`.

При использовании функции `next()` вы можете указать второй аргумент в качестве значения по умолчанию. Тогда в самом конце функция `next()` возвратит это значение вместо того, чтобы выбросить исключение `StopIteration`:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> new_names_it = iter(names)
>>> next(new_names_it, "Rick")
'Tom'
```

```
>>> next(new_names_it, "Rick")
'Shelly'
>>> next(new_names_it, "Rick")
'Garth'
>>> next(new_names_it, "Rick")
'Rick'
>>> next(new_names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(new_names_it, "Jane")
'Jane'
```

Рассмотрим другую ситуацию. Что, если вы не просто работаете с готовой последовательностью чисел, а вычисляете, считываете или получаете элементы последовательности каким-то иным способом в реальном времени?

Начнем с простого примера и предположим, что вам нужно написать функцию, создающую список возведенных в квадрат чисел, которые будут обрабатываться другим фрагментом кода:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

Этот код сработает, однако в нем есть потенциальные проблемы. Догадываетесь, в чем они заключаются?

Например, что будет, если значение `MAX` равно не 5, а 10 000 000? Или 10 000 000 000 000? В этом случае объем используемой памяти окажется неоправданно большим, так как данный код создает *огромный* список, использует его *всего один раз*, а затем отбрасывает. Кроме того, цикл `for` не будет запущен вплоть до вычисления всего списка квадратов. При этом пользователь программы может подумать, что она зависла.

Или еще хуже: что, если для получения каждого элемента вместо выполнения быстрых и дешевых арифметических операций вы производите по-настоящему дорогостоящие вычисления? Или вызываете API через сеть? Или считываете значения из базы данных? В этом случае ваша программа будет работать очень медленно, возможно, даже не станет реагировать на запросы и завершит работу сбоем из-за нехватки памяти. Пользователи посчитают вас ужасным программистом.

Решение заключается в том, чтобы создать итератор и «лениво» вычислять каждое значение по мере необходимости. Тогда каждая итерация цикла будет осуществляться по принципу «точно вовремя».

Сделать это можно несколькими способами. Самый лучший из них — использование *функции-генератора*, которая вам наверняка понравится!

Функции-генераторы

В Python есть инструмент под названием *«функция-генератор»*, чьи возможности сложно описать в одном предложении. Начнем с того, что она является *очень* полезным средством создания итераторов.

Функция-генератор во многом похожа на обычную функцию. Однако вместо `return` она использует ключевое слово `yield`, например:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

В цикле `for` ее можно задействовать так:

```
>>> for num in gen_nums():
...     print(num)
0
1
2
3
```

Давайте разберем этот фрагмент. Когда вы вызываете `gen_nums()` как функцию, она сразу же возвращает *объект-генератор*:

```
>>> sequence = gen_nums()
>>> type(sequence)
<class 'generator'>
```

Функция-генератор здесь `gen_nums()` — это то, что мы определяем, а затем вызываем. Функция считается генератором тогда и только тогда, когда она использует ключевое слово `yield` вместо `return`. *Объект-генератор* — это то, что возвращает функция-генератор при ее вызове, в данном случае — `sequence`.

Запомните

Функция-генератор *всегда* возвращает объект-генератор *и ничего другого*.

Этот объект-генератор является итератором, а значит, вы можете перебрать его элементы с помощью функции `next()` или цикла `for`:

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

>>> # Или в цикле for:
... for num in gen_nums():
...     print(num)
0
1
2
3
```

Этот код работает следующим образом: при первом вызове функции `next()` или запуске цикла `for` код в теле `gen_nums()` начинает выполняться с самого начала, возвращая значение, находящееся справа от ключевого слова `yield`.

Расширение функции next()

Пока данный код ведет себя как обычная функция. Однако при следующем вызове `next()` или в рамках следующей итерации цикла `for`, что одно и то же, выполнение кода начинается не с самого начала, а со строки, *следующей после оператора* `yield`. Посмотрите на код функции `gen_nums()` еще раз:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

`gen_nums()` является чем-то более универсальным по сравнению с функцией или подпрограммой. На самом деле она представляет собой *корутину*. Видите ли, у обычной функции может быть несколько точек выхода, иначе называемых операторами возврата `return`, но только одна точка входа: всегда,

когда вы вызываете функцию, ее выполнение начинается с первой строки кода, образующего ее тело.

Корутина напоминает функцию, но предусматривает несколько точек *входа*. Как и в случае с обычной функцией, ее выполнение начинается с первой строки кода, однако, обнаружив оператор возврата, корутина не завершает работу, а лишь *приостанавливает* ее. При очередном вызове функции `next()` или выполнении очередной итерации цикла `for`, что одно и то же, работа возобновляется с того места, где она остановилась, то есть точкой повторного входа является строка кода, следующая за оператором `yield`.

В этом и суть: *каждый оператор* `yield` *одновременно определяет точку выхода и точку повторного входа*.

В случае с объектами-генераторами при запросе очередного значения поток управления переходит на строку, следующую после оператора `yield`. В данном случае следующая строка кода инкрементирует переменную `n`, после чего начинается следующая итерация цикла `while`.

Обратите внимание на то, что мы не генерируем исключение `StopIteration` в теле функции `gen_nums()`. Когда работа этой функции завершается (в данном случае после выхода из цикла `while`), объект-генератор выбрасывает исключение `StopIteration` автоматически.

Еще раз: каждый оператор `yield` одновременно определяет точку выхода *и* точку повторного входа. На самом деле генератор может предусматривать несколько таких операторов:

```
def gen_extra_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
    yield 42 # Второй оператор yield
```

Вот вывод, который вы получите при выполнении этого кода:

```
>>> for num in gen_extra_nums():
...     print(num)
0
1
2
3
42
```

Второй оператор `yield` выполняется после выхода из цикла `while`. Когда функция достигает неявного оператора возврата в конце, итерация останавливается. Проанализируйте приведенный ранее код и убедитесь в том, что он имеет смысл.

Преобразование в функцию-генератор

Вернемся к примеру с перебором последовательности квадратов чисел. Вот как мы делали это в прошлый раз:

```
def fetch_squares(max_root):
    squares = []
    for num in range(max_root):
        squares.append(num**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

В качестве упражнения создайте новый файл Python и попробуйте написать функцию-генератор `gen_squares()`, которая решает ту же задачу.

Готово? Отлично. Вот как она выглядит:

```
def gen_squares7(max_root):
    for num in range(max_root):
        yield num ** 2

>>> MAX = 5
>>> for square in gen_squares(MAX):
...     print(square)
0
1
4
9
16
```

Обратите внимание на то, что `gen_squares()` включает встроенную функцию `range()`, которая возвращает итерируемый объект. Это очень важно.

Представьте, что функция `range()` возвращает список. Если при этом значение `MAX` очень велико, то внутри вашей функции-генератора будет создан огромный список, что сведет ее масштабируемость на нет.

Суть в том, что функция-генератор масштабируется ровно в той степени, в какой это допускает ее *наименее* масштабируемая строка. В целом функции-генераторы потребляют довольно мало памяти, но лишь в том случае, если они написаны с умом. При написании функций-генераторов следите за скрытыми узкими местами.

Нужны ли вам генераторы

Строго говоря, нам не *требуются* функции-генераторы для перебора элементов коллекций. Мы просто *предпочитаем* их применять, потому что они значительно упрощают реализацию полезных паттернов масштабирования.

Например, можно ли создать итератор без написания функции-генератора? Да, можно. Для получения списка квадратов чисел можно использовать следующий код:

```
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value

# Вы можете использовать его так:
for square in SquaresIterator(5):
    print(square)
```

Каждое значение получается путем вызова его метода `__next__()` вплоть до выброса исключения `StopIteration`. Этот фрагмент кода дает тот же результат, однако взгляните на код класса `SquaresIterator` и сравните его с приведенным ранее кодом генератора. Какой из них легче читать? Какой легче поддерживать? А какой будет легче модифицировать без ошибок в случае изменения требований? Большинству людей решение с использованием генератора кажется более простым и естественным.

Авторы книг часто применяют слово «генератор» для обозначения функции-генератора или объекта-генератора, возвращаемого в результате ее вызова. При этом они обычно полагают, что смысл слова очевиден из контекста. Иногда это так, а иногда — нет. Порой автор не дает достаточно информации для того, чтобы уловить это важное различие. Как и в случае с функцией и значением, которое она возвращает при вызове, существует большая разница между функцией-генератором и возвращаемым ею объектом-генератором.

Я призываю вас всегда использовать словосочетания «функция-генератор» и «объект-генератор», чтобы однозначно доносить смысл того, что вы имеете в виду. (К тому же это способствует более четкой коммуникации в команде.) Единственное исключение: говоря о данной языковой функции в широком смысле, можно называть ее просто генератором.

В этой книге я постараюсь быть примером и употреблять конкретные названия.

Паттерны использования генераторов и масштабируемая компонуемость

Рассмотрим небольшую функцию-генератор:

```
def matching_lines_from_file(path, pattern):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')
```

Функция `matching_lines_from_file()` демонстрирует несколько важных приемов использования современной версии Python, и ее стоит внимательно изучить. Она сопоставляет подстроки с текстовым файлом и выдает его строки, содержащие искомую подстроку.

Первая строка кода открывает файловый объект `handle`, доступный только для чтения. Если вы еще не открывали файловые объекты с помощью операторов `with`, начните делать это прямо сейчас. Главное преимущество данного подхода заключается в том, что после выхода из блока `with` файловый объект автоматически закрывается, даже если преждевременный выход был спровоцирован исключением. Это аналогично следующему фрагменту кода:

```
try:
    handle = open(path)
    # Чтение из handle
finally:
    handle.close()
```

(Конструкцию `try/finally` мы обсудим в главе 5.) Следующая строка, `for line in handle`, представляет собой полезную для работы с текстовыми файлами идиому, о которой мало кто знает. В ходе каждой итерации цикла `for` очередная строка текста считывается из текстового файла и помещается в переменную `line`.

Иногда люди используют следующий, крайне неэффективный подход:

```
# Не делайте так!!!
for line in handle.readlines():
```

Метод `readlines()` (обратите внимание на множественное число) считывает *весь файл*, разбирает его на строки и возвращает их список. К данному моменту вы уже должны понимать, насколько негативно это может повлиять на масштабируемость.

Напомню, что функция-генератор масштабируется ровно в той степени, в какой это допускает ее *наименее* масштабируемая строка. Поэтому при написании

кода проявляйте осторожность, чтобы не создать узкое место в плане потребления памяти, делающее применение функции-генератора бессмысленным.

Еще один подход, вполне допустимый и масштабируемый, заключается в использовании метода файлового объекта `.readline()` (обратите внимание на единственное число), который возвращает строки по одной за раз:

```
# Метод .readline() считывает и возвращает одну строку текста или пустую
строку при достижении конца файла.
line = handle.readline()
while line != '':
    # Некоторые действия со строкой
    # ...
    # В конце работы цикла считывается следующая строка.
    line = handle.readline()
```

Тем не менее использование выражения `for line in handle` — гораздо более простой и понятный подход.

ВЫРАЖЕНИЯ ПРИСВАИВАНИЯ

Метод `.readline()` лучше применять с *выражением присваивания*:

```
while (line := handle.readline()) != '':
    # Некоторые действия со строкой
    # ...
    # Здесь не нужно снова вызывать метод .readline().
```

Инструкция присваивания наподобие `x = y + 1` сама по себе не представляет ценности. То есть Python не позволяет использовать код вроде `if (x = y + 1) > 2:`. Такой выбор был сделан намеренно еще на этапе разработки языка, чтобы предотвратить ошибки, связанные со случайным применением оператора `=` вместо `==`.

Однако иногда имеет смысл присвоить значение *и* использовать его в той же строке, как в приведенном ранее цикле `while`. Именно для таких случаев в Python был добавлен оператор `:=`.

Благодаря его применению мы можем один раз написать `line := handle.readline()` вместо того, чтобы дважды писать `line = handle.readline()`. Это не просто удобно с синтаксической точки зрения — зачастую это позволяет избавиться от дублирующегося кода (как в приведенном здесь примере), а в некоторых случаях даже избежать повторного выполнения вычислений.

Официально этот оператор называется *выражением присваивания*, но многие называют его моржовым оператором, потому что он напоминает мордочку моржа, если его повернуть.