top

КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Основы программирования на языке Python

Урок 7

Генераторы и замыкания

Contents

Генераторы и замыкания

Генераторы и где они обитают

Генератор. С чем у вас ассоциируется это слово? Возможно, с каким-то электронным устройством. Или, возможно, с тяжелым и серьезным агрегатом, который производит энергию, скажем, электроэнергию или какую-то другую.

В Python генератор — это специальный участок кода, который генерирует серию значений и управляет процессом итерации. Вот почему генераторы очень часто называют итераторами. Некоторые программисты видят тонкую разницу между этими двумя понятиями, но мы будем считать их тождественными.

Вы уже много раз встречали генераторы, хотя могли не подозревать об этом. Взгляните на этот, очень простой фрагмент кода:

```
for i in range(5):
    print(i)
```

Функция range() — это генератор, который, в свою очередь, является итератором.

В чем разница?

Функция возвращает одно четко определенное значение. Оно может быть результатом более или менее сложного вычисления, например, полинома, и вызывается только один раз.

Генератор возвращает серию значений и, как правило, неявно вызывается более одного раза.

В примере выше генератор range() вызывается шесть раз, выводит пять последовательных значений от нуля до четырех и, наконец, сигнализирует о завершении серии.

Вышеуказанный процесс полностью прозрачен. Давайте разберем его и познакомимся с протоколом итератора.

Протокол итератора — это то, как объект должен себя вести, чтобы соответствовать правилам, налагаемым контекстом выражений for и in. Объект, который соответствует протоколу итератора, называется итератором.

У итератора должны быть два метода:

- __iter__(), который должен вернуть сам объект, и который вызывается один раз (это необходимо, чтобы Python успешно запустил итерацию)
- __next__(), который возвращает следующее значение (первое, второе и т.д.) требуемой последовательности. Этот метод вызывают операторы for/in для следующей итерации; если значений больше нет, метод должен вызвать исключение StopIteration.

Звучит странно? Вовсе нет. Посмотрите на пример ниже.

```
1  class Fib:
2      def __init__(self, nn):
3          print("__init__")
4          self.__n = nn
5          self.__i = 0
6          self.__p1 = self.__p2 = 1
7
8      def __iter__(self):
9          print("__iter__")
```

```
10          return self
11
12      def __next__(self):
13          print("__next__")
14          self.__i += 1
15          if self.__i > self.__n:
16              raise StopIteration
17          if self.__i in [1, 2]:
18              return 1
19          ret = self.__p1 + self.__p2
20          self.__p1, self.__p2 = self.__p2, ret
21          return ret
22
23  for i in Fib(10):
24      print(i)
```

Мы создали класс, который перебирает первые n значений чисел Фибоначчи (здесь n — параметр конструктора).

Напомним, что числа Фибоначчи (*Fibi*) определяются следующим образом:

$$Fib_1 = 1$$
$$Fib_2 = 1$$
$$Fib_i = Fib_{i-1} + Fib_{i-2}$$

Другими словами:

- первые два числа Фибоначчи равны 1;
- любое другое число Фибоначчи является суммой двух предыдущих (например, Fib3 = 2, Fib4 = 3, Fib5 = 5 и т.д.).

Давайте углубимся в код:

- *строки 2-6:* конструктор класса выводит сообщение (мы будем использовать его для отслеживания пове-

дения класса), подготавливает некоторые переменные (__n хранит предел серии, __i отслеживает текущее число Фибоначчи, а __p1 и __p2 сохраняют два предыдущих числа);

- *строки с 8 по 10:* метод __iter__ должен вернуть сам итератор. Чтобы это понять, постарайтесь представить объект, который не является итератором (например, набор некоторых объектов), но один из его компонентов является итератором, который сканирует коллекцию; метод __iter__ должен извлечь итератор и поручить ему выполнение протокола итерации. Как видите, метод начинается с вывода сообщения;
- *строки с 12 по 21:* метод __next__ отвечает за создание последовательности. Сначала он выводит сообщение, затем обновляет количество желаемых значений, и, если он достигает конца последовательности, метод прерывает итерацию, вызывая исключение StopIteration. Остальная часть кода проста и точно отражает определение, которое мы дали ранее;
- *строки 23 и 24* используют итератор.

Результат выполнения кода будет следующим:

```
__init__
__iter__
__next__
1
__next__
1
__next__
2
__next__
```

```
3
__next__
5
__next__
8
__next__
13
__next__
21
__next__
34
__next__
55
__next__
```

Итак:

- итератор создается первым;
- затем Python вызывает метод __iter__, чтобы получить доступ к актуальному итератору;
- метод __next__ вызывается одиннадцать раз — первые десять раз дают полезные значения, а одиннадцатый завершает итерацию.

В предыдущем примере показано решение, в котором *итератор является частью более сложного класса*.

Код не очень сложный, но он представляет эту концепцию в понятном виде.

Посмотрите на код.

```
1 class Fib:
2     def __init__(self, nn):
3         self.__n = nn
4         self.__i = 0
5         self.__p1 = self.__p2 = 1
```

```
 6
 7      def __iter__(self):
 8          print("Fib iter")
 9          return self
10
11      def __next__(self):
12          self.__i += 1
13          if self.__i > self.__n:
14              raise StopIteration
15          if self.__i in [1, 2]:
16              return 1
17          ret = self.__p1 + self.__p2
18          self.__p1, self.__p2 = self.__p2, ret
19          return ret
20
21  class Class:
22      def __init__(self, n):
23          self.__iter = Fib(n)
24
25      def __iter__(self):
26          print("Class iter")
27          return self.__iter;
28
29  object = Class(8)
30
31  for i in object:
32      print(i)
```

Мы встроили итератор Fib в другой класс (можно сказать, что мы объединили его в классе Class). Он создан вместе с объектом Class.

Объект класса может использоваться в качестве итератора только в том случае, если он положительно отвечает на вызов __iter__. И если этот класс был вызван таким образом, он предоставляет объект, который соблюдает протокол итерации.

Вот почему вывод кода такой же, как и ранее, хотя объект класса Fib не используется явно внутри цикла for.

Оператор yield

Протокол итератора не особенно трудно понять и использовать, но, несмотря на это, *протокол довольно неудобен*.

Основной дискомфорт состоит в *необходимости сохранять состояния итерации между последовательными вызовами* __iter__.

Например, итератор Fib вынужден хранить точное место, в котором был остановлен последний вызов (т.е. результирующее число и значения двух предыдущих элементов). Из-за этого код вырастает в размерах и становится менее понятным.

Поэтому Python предлагает гораздо более эффективный, удобный и элегантный способ написания итераторов.

Концепция основана на очень специфическом и мощном механизме, который реализует ключевое слово yield.

Ключевое слово yield можно воспринимать как умного брата оператора return, с одним существенным отличием.

Посмотрите на эту функцию:

```
def fun(n):
    for i in range(n):
        return i
```

Выглядит странно, правда? Понятно, что цикл for не может завершить свое первое выполнение, так как оператор return прервет его.

Более того, вызов функции ничего не изменит. Цикл for начнется с нуля и будет немедленно прерван.

Можно сказать, что такая функция не может сохранять и восстанавливать свое состояние между последующими вызовами.

Это также означает, что такую функцию *нельзя использовать в качестве генератора*.

Мы заменили ровно одно слово в коде.

```
def fun(n):
    for i in range(n):
        yield i
```

Мы вставили ключевое слово yield вместо return. Эта маленькое изменение *превращает функцию в генератор*, а сам оператор yield выполняется с очень интересными эффектами.

Прежде всего, он выводит значение выражения, которое указано после ключевого слова yield, точно так же, как return, но не теряет состояния функции.

Все значения переменных заморожены и ждут следующего вызова, когда выполнение будет возобновлено (а не взяты с нуля, как после оператора return).

Есть одно важное ограничение: такая *функция не должна вызываться явно*, поскольку это больше не функция, а *генератор*.

Вызов *вернет идентификатор объекта*, а не последовательности, которую мы ожидаем от генератора.

По тем же причинам предыдущая функция (та, что с оператором return) может быть вызвана только явно, и не должна использоваться в качестве генератора.

Как построить генератор

Давайте покажем вам новый генератор в действии. Вот как его можно использовать:

```
def fun(n):
    for i in range(n):
        yield i

for v in fun(5):
    print(v)
```

Вы сможете угадать, что появится на экране?

Ответ:

```
0
1
2
3
4
```

Как построить свой собственный генератор

Что делать, если вам нужно, чтобы генератор выполнял первые n чисел в степени 2?

Нет ничего проще. Просто посмотрите на код ниже.

```
1 def powersOf2(n):
2     pow = 1
3     for i in range(n):
4         yield pow
5         pow *= 2
6
7 for v in powersOf2(8):
8     print(v)
```

Вы сможете угадать, что появится на экране? Запустите код, чтобы проверить свои догадки.

Генераторы также можно использовать в списковом выражении (list comprehensions), как здесь:

```
def powersOf2(n):
    pow = 1
    for i in range(n):
        yield pow
        pow *= 2

t = [x for x in powersOf2(5)]

print(t)
```

Запустите код и проверьте результат.

Функция list() может преобразовать серию последующих вызовов генератора в реальный список:

```
def powersOf2(n):
    pow = 1
    for i in range(n):
        yield pow
        pow *= 2

t = list(powersOf2(3))

print(t)
```

Попытайтесь предсказать результат и выполните код, чтобы проверить, были ли вы правы.

Кроме того, контекст, созданный оператором in, тоже позволяет использовать генератор.

В примере показано, как это сделать:

```
def powersOf2(n):
    pow = 1
    for i in range(n):
        yield pow
        pow *= 2

for i in range(20):
    if i in powersOf2(4):
        print(i)
```

Каким будет результат кода? Запустите программу и проверьте.

Теперь давайте рассмотрим генератор чисел Фибоначчи и убедимся, что он выглядит намного лучше, чем объектная версия, основанная на реализации протокола прямого итератора.

Вот так:

```
def Fib(n):
    p = pp = 1
    for i in range(n):
        if i in [0, 1]:
            yield 1
        else:
            n = p + pp
            pp, p = p, n
            yield n

fibs = list(Fib(10))

print(fibs)
```

Угадайте вывод (список), созданный генератором, и запустите код, чтобы проверить, были ли вы правы.

Подробнее о списковом выражении

Стоит запомнить правила, регулирующие создание и использование особого явления в Python, которое называется списковым выражением. Это простой и эффективный способ создания списков и их содержимого.

На всякий случай мы дали краткую информацию ниже.

```
1  listOne = []
2
3  for ex in range(6):
4      listOne.append(10 ** ex)
5
6
7  listTwo = [10 ** ex for ex in range(6)]
8
9  print(listOne)
10 print(listTwo)
```

Внутри кода есть два фрагмента, каждый из которых создает список, содержащий несколько первых естественных чисел, возведенных в десятую степень.

Первый использует обычный цикл for, а второй — списковое выражение, которое генерирует список на месте, без необходимости уходить в цикл или использовать какой-то расширенный код.

Может показаться, что список создается внутри себя, но это конечно, не так. Программа выполняет практически те же операции, что и в первом фрагменте, но второй формализм, несомненно, более элегантный и позволяет программисту, который читает этот код, игнорировать лишние детали.

В результате выполнения кода выводятся две одинаковые строки, которые содержат следующий текст:

```
[1, 10, 100, 1000, 10000, 100000]
```

Запустите код, чтобы проверить, правы ли мы.

Мы хотим показать вам очень интересный синтаксис. Списковое выражение здесь не только удобно, оно еще и идеальная среда для этого синтаксиса.

Мы говорим об *условном операторе (conditional expression) — это способ выбора одного из двух значений на основе результата логического выражения*.

Итак:

```
expression_one if condition else expression_two
```

На первый взгляд это может показаться немного странным, но вы должны помнить, что это *не условная инструкция*. Более того, это совсем не инструкция. Это оператор.

Значение, которое оно выводит, равно expression_one, если условие True, в противном случае оно выводит expression_two.

Перейдем к примеру, который наглядно покажет, как это работает. Посмотрите на код.

```
1  lst = []
2
3  for x in range(10):
4      lst.append(1 if x % 2 == 0 else 0)
5
6  print(lst)
```

Код заполняет список числами 1 и 0. Если индекс определенного элемента нечетный, элемент устанавливается в 0, в противном случае в 1.

Просто? Не особенно, к этому надо привыкнуть. Элегантно? Несомненно.

А можно ли использовать этот прием в списковом выражении? Можно.

Посмотрите на пример.

```
1  lst = [1 if x % 2 == 0 else 0 for x in range(10)]
2
3  print(lst)
```

Компактность и элегантность — вот те два слова, которые приходят на ум при взгляде на этот код.

Итак, что общего между генераторами и списковыми выражениями? Есть ли между ними какая-то связь? Да. Довольно слабая связь, но она однозначно есть.

Всего лишь одно изменение *превращает любое списковое выражение в генератор*.

Теперь посмотрите на код ниже и постарайтесь найти то, что превращает списковое выражение в генератор:

```
lst = [1 if x % 2 == 0 else 0 for x in range(10)]
genr = (1 if x % 2 == 0 else 0 for x in range(10))

for v in lst:
    print(v, end=" ")
print()

for v in genr:
    print(v, end=" ")
print()
```

Это *круглые скобки*. В квадратных скобках получается списковое выражение, а в круглых — генератор.

Но при запуске код выдает две одинаковые строки:

```
1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0
```

Откуда мы знаем, что второе присваивание создает генератор, а не список?

Тому есть несколько доказательств. Примените функцию len() к обеим сущностям.

len(lst) даст 10. Это понятно и предсказуемо. len(genr) вызовет исключение, и вы увидите следующее сообщение:

```
TypeError: object of type 'generator' has no len()
```

Конечно, ни список, ни генератор сохранять не нужно, ведь их можно создать именно там, где они вам нужны. Как здесь:

```
for v in [1 if x % 2 == 0 else 0 for x in range(10)]:
    print(v, end=" ")
print()

for v in (1 if x % 2 == 0 else 0 for x in range(10)):
    print(v, end=» «)
print()
```

Примечание: *несмотря на то, что результаты обоих циклов выглядят одинаково, это совершенно не означает, что они и работают одинаково. В первом цикле список создается (и повторяется)*

целиком. На самом деле, он существует во время выполнения цикла.

Во втором цикле никакого списка вообще нет. Есть только последовательность значений, которые генератор создает одно за другим.

Поэкспериментируйте с этими циклами сами.

Функция lambda

Понятие lambda-функции пришло в программирование из математики, если точнее, то из той ее части, которая называется *лямбда-исчислением*. Но они все-таки разные.

Математики используют *лямбда-исчисление* во многих формальных системах, связанных с логикой, рекурсией или доказуемостью теорем. Программисты используют лямбда-функцию, чтобы упростить код, сделать его чистым и понятным.

Лямбда-функция — это функция без имени (ее можно назвать *анонимной функцией*). У вас может возникнуть естественный вопрос: «Как же использовать то, что нельзя идентифицировать?»

К счастью, это не проблема, так как такой функции всегда можно дать имя, если это действительно нужно, но чаще всего лямбда-функция может спокойно существовать и работать инкогнито.

Объявление лямбда-функции совершенно не похоже на обычное объявление функции. Убедитесь в этом сами:

```
lambda parameters : expression
```

Такое выражение *возвращает значение выражения при учете текущего значения текущего лямбда-аргумента*.

Как обычно, рассмотрим это все на примере. В нем используются три лямбда-функции с разными именами. Посмотрите внимательно:

```
two = lambda : 2
sqr = lambda x : x * x
pwr = lambda x, y : x ** y

for a in range(-2, 3):
    print(sqr(a), end=" ")
    print(pwr(a, two()))
```

Давайте проанализируем код:

- первая lambda является анонимной *функцией без параметров*, которая всегда возвращает 2. Раз мы *присвоили ей переменную с именем* two, мы можем сказать, что функция больше не является анонимной, и теперь ее можно вызвать, обратившись к ней по имени.
- вторая — это *анонимная функция с одним параметром*. Она возвращает значение своего аргумента в квадрате. Название у нее соответственное.
- третья lambda-функция *принимает два параметра* и возвращает значение первого, возведенного в степень второго. Имя переменной внутри лямбда-функции говорит само за себя. Мы не использовали имя pow, чтобы избежать путаницы со встроенной функцией с тем же именем и теми же задачами.

Выполнение кода даст следующий результат:

```
4 4
1 1
0 0
1 1
4 4
```

Этот пример достаточно хорошо показывает, как объявлять лямбда-функции и как они себя ведут, но ничего не говорит о том, зачем они нужны, и для чего они используются, поскольку их можно спокойно заменить обычными функциями Python.

В чем их польза?

Как и для чего использовать лямбда-функции?

Лямбды интереснее всего использовать в чистом виде, *как анонимные части кода, которые считают результат*.

Представьте, что нам нужна функция (назовем ее printfunction), которая выводит значения данной (другой) функции для набора выбранных аргументов.

Функция printfunction должна быть универсальной — в качестве аргументов она должна принимать набор аргументов, помещенных в список, и функцию, которую нужно посчитать. Мы хотим избежать жесткого кодирования.

Посмотрите на пример кода.

```
1 def printfunction(args, fun):
2     for x in args:
3         print('f(', x,')=', fun(x), sep='')
4
```

```
5  def poly(x):
6      return 2 * x**2 - 4 * x + 2
7
8  printfunction([x for x in range(-2, 3)], poly)
```

Вот как мы реализовали эту идею.

Давайте проанализируем код. Функция printfunction() принимает два параметра:

- *первый* — список аргументов, для которых нужно вывести результаты;
- *второй* — функция, которая должна вызываться столько раз, сколько значений соберется в первом параметре.

Примечание: *мы также определили функцию с именем poly() — это функция, значения которой нужно вывести. Вычисление, которое выполняет функция, не очень сложное — это многочлен, или полином (отсюда и название):*

$$f(x) = 2x^2 - 4x + 2$$

Имя функции затем передается в printfunction() наряду с набором из пяти различных аргументов. Этот набор создается с помощью спискового выражения.

Результат вывода будет следующим:

```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

Можно ли не определять функцию poly(), раз мы не собираемся использовать ее больше одного раза? Можно — в этом и заключается преимущество лямбда-функции.

Посмотрите на пример ниже. Есть ли разница?

```
def printfunction(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')
printfunction([x for x in range(-2, 3)],
              lambda x: 2 * x**2 - 4 * x + 2)
```

Функция printfunction() осталась точно такой же, но нет функции poly(). Нам это больше не нужно, так как многочлен теперь находится прямо внутри вызова printfunction() в виде лямбды, которая определяется следующим образом:

```
lambda x: 2 * x**2 - 4 * x + 2.
```

Код стал короче, понятнее и читабельнее.

Давайте рассмотрим еще один случай, когда лямбды могут быть полезны. Начнем с описания встроенной функции map(). По одному имени функции сложно сказать, что она делает, но ее идея проста, а сама функция действительно полезна.

Лямбда-функции и функция map()

В самом простом случае функция map() принимает два аргумента:

- функцию;
- список.

```
map(function, list)
```

Приведенное выше описание чрезвычайно упрощено, так как:

- второй аргумент функции map() может быть любым объектом, который может быть повторен (например, кортеж или просто генератор).
- функция map()может принимать более двух аргументов.

Функция map() *применяет функцию, которую первый аргумент передал всем элементам второго аргумента, и возвращает итератор, который выводит все последующие результаты функции*. Полученный итератор можно использовать в цикле или преобразовать в список, используя функцию list().

Есть ли здесь место для лямбда-функции?

Посмотрите на код ниже — мы использовали две лямбды.

```
1  list1 = [x for x in range(5)]
2  list2 = list(map(lambda x: 2 ** x, list1))
3  print(list2)
4  for x in map(lambda x: x * x, list2):
5      print(x, end=' ')
6  print()
```

Хитрость вот в чем:

- создайте список list1 со значениями от 0 до 4;
- затем используйте функцию map и первую лямбда-функцию, чтобы создать новый список, в котором все элементы равны 2 в степени, полученной из соответствующего элемента в списке list1;
- затем выводится list2;

- на следующем шаге снова используйте функцию map(), чтобы воспользоваться генератором, который она возвращает, и напрямую вывести все значения, которые он выдает. Как видите, мы задействовали и вторую лямбда-функцию, которая просто возводит в квадрат каждый элемент из списка list2.

Попробуйте представить тот же код без лямбд. Как думаете, он был бы лучше? Вряд ли.

Лямбда и функция filter()

Еще одна функция в Python, которую можно значительно улучшить с помощью лямбды, это filter().

Она принимает те же аргументы, что и функция map(), но делает с ними кое-что другое. Она фильтрует второй аргумент, руководствуясь указаниями, которые получает в функции, указанной в качестве первого аргумента (функция вызывается для каждого элемента списка, так же как и в функции map()).

Элементы, которые возвращают True, успешно проходят фильтр, а остальные отклоняются.

Пример ниже показывает, как работает функция filter().

```
1  from random import seed, randint
2
3  seed()
4  data = [ randint(-10,10) for x in range(5) ]
5  filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))
6  print(data)
7  print(filtered)
```

Примечание: *мы использовали модуль random для инициализации генератора случайных чисел (не*

путать с генераторами, о которых мы только что говорили) с функцией seed() и для получения пяти случайных целых значений между -10 и 10, используя функцию randint().

Затем список фильтруется, и принимаются только четные числа, которые больше нуля.

Вряд ли вы получите точно такой же результат, но у нас получилось следующее:

```
[6, 3, 3, 2, -7]
[6, 2]
```

Краткий обзор замыканий

Давайте начнем с определения: *замыкание (closure) — это метод, который позволяет хранить значения, несмотря на то, что контекста, в котором они были созданы, больше не существует.* Путано? Немного.

Давайте проанализируем простой пример:

```
def outer(par):
    loc = par

var = 1
outer(var)

print(var)
print(loc)
```

Пример явно ошибочный.

Последние две строки вызовут исключение NameError, то есть за пределами функции не доступны ни par, ни loc.

Обе переменные существуют только при выполнении функции outer().

Посмотрите на пример.

```
def outer(par):
    loc = par
    def inner():
        return loc
    return inner

var = 1
fun = outer(var)
print(fun())
```

Мы внесли серьезные изменения в код.

Теперь в нем есть новый элемент — функция, которая называется inner, внутри другой функции, которая называется outer.

Как она работает? Как и любая другая функция, за исключением того, что функция inner() может быть вызвана только из функции outer(). Можно сказать, что функция inner() является приватным инструментом функции outer(), потому что только она может получить к ней доступ.

Смотрите внимательно:

- *функция inner()* возвращает значение переменной, которая доступна внутри своей области видимости, так как функция inner() может использовать любой объект, который принадлежит функции outer().
- *Функция outer ()* возвращает саму функцию inner(), точнее, ее копию, которая была заморожена в момент вызова функции outer(). Замороженная функция содержит полное окружение, включая состояние всех

локальных переменных, что также означает, что значение loc успешно сохраняется, несмотря на то, что функция outer() давно прекратила свое существование.

То есть код действителен и выводит следующее: 1.

Функция, которая возвращается во время вызова функции outer(), называется *замыканием*.

Замыкание должно вызываться точно так же, как оно было объявлено.

В предыдущем примере (см. код ниже):

```
def outer(par):
    loc = par
    def inner():
        return loc
    return inner v

ar = 1
fun = outer(var)
print(fun())
```

у функции inner() не было параметров, поэтому нам пришлось вызывать ее без аргументов.

А теперь посмотрите на код ниже.

```
1  def makeclosure(par):
2      loc = par
3      def power(p):
4          return p ** loc
5      return power
6
7  fsqr = makeclosure(2)
8  fcub = makeclosure(3)
9  for i in range(5):
10     print(i, fsqr(i), fcub(i))
```

Вполне возможно *объявить замыкание с произвольным числом параметров*, например, как с функцией power().

Это означает, что замыкание не только использует замороженное окружение, но также может *изменить свое поведение, используя значения, взятые извне*.

Этот пример показывает еще одно интересное обстоятельство — можно *создать сколько угодно замыканий в одном и том же фрагменте кода*. Это делается с помощью функции makeclosure().

Примечание:

- *первое замыкание, полученное из функции make-closure(), определяет инструмент, возводящий его аргумент в квадрат;*
- *второе возводит аргумент в куб.*

Поэтому у нас получается такой результат вывода:

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

Теперь поэкспериментируйте сами.