Generators and closures

Generators - where to find them

Generator - what do you associate this word with? Perhaps it refers to some electronic device. Or perhaps it refers to a heavy and serious machine designed to produce power, electrical or other.

A Python generator is **a piece of specialized code able to produce a series of values, and to control the iteration process**. This is why generators are very often called **iterators**, and although some may find a very subtle distinction between these two, we'll treat them as one.

You may not realize it, but you've encountered generators many, many times before. Take a look at the very simple snippet:

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

The `range()` function is, in fact, a generator, which is (in fact, again) an iterator

What is the difference?

A function returns one, well-defined value - it may be the result of a more or less complex evaluation of, e.g., a polynomial, and is invoked once - only once.

A generator **returns a series of values**, and in general, is (implicitly) invoked more than once.

In the example, the `range()` generator is invoked six times, providing five subsequent values from zero to four, and finally signaling that the series is complete.

The above process is completely transparent. Let's shed some light on it. Let's show you the **iterator protocol**.

The **iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the** `for` **and** `in` **statements**. An object conforming to the iterator protocol is called an **iterator**.

An iterator must provide two methods:

- `__iter__()` which should **return the object itself** and which is invoked once (it's needed for Python to successfully start the iteration)
- `__next__()` which is intended to **return the next value** (first, second, and so on) of the desired series - it will be invoked by the `for`/`in` statements in order to pass through the next iteration; if there are no more values to provide, the method should **raise the** `StopIteration` **exception**.

Does it sound strange? Not at all. Look at the example below.

```
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)
```

We've built a class able to iterate through the first `n` values (where `n` is a constructor parameter) of the Fibonacci numbers.

Let us remind you - the Fibonacci numbers (Fib_i) are defined as follows:

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

In other words:

- the first two Fibonacci numbers are equal to 1;
- any other Fibonacci number is the sum of the two previous ones (e.g., $Fib_3 = 2$, $Fib_4 = 3$, $Fib_5 = 5$, and so on)

Let's dive into the code:

- lines 2 through 6: the class constructor prints a message (we'll use this to trace the class's behavior), prepares some variables (`__n` to store the series limit, `__i` to track the current Fibonacci number to provide, and `__p1` along with `__p2` to save the two previous numbers);
- lines 8 through 10: the `__iter__` method is obliged to return the iterator object itself; its purpose may be a bit ambiguous here, but there's no mystery; try to imagine an object which is not an iterator (e.g., it's a collection of some entities), but one of its components is an iterator able to scan the collection; the `__iter__` method should **extract the iterator and entrust it with the execution of the iteration protocol**; as you can see, the method starts its action by printing a message;
- lines 12 through 21: the `__next__` method is responsible for creating the sequence; it's somewhat wordy, but this should make it more readable; first, it prints a message, then

it updates the number of desired values, and if it reaches the end of the sequence, the method breaks the iteration by raising the StopIteration exception; the rest of the code is simple, and it precisely reflects the definition we showed you earlier;
- lines 23 and 24 make use of the iterator.

The code produces the following output:

```
__init__
__iter__
__next__
1
__next__
1
__next__
2
__next__
3
__next__
5
__next__
8
__next__
13
__next__
21
__next__
34
__next__
55
__next__
```

Look:

- the iterator object is instantiated first;
- next, Python invokes the `__iter__` method to get access to the actual iterator;
- the `__next__` method is invoked eleven times - the first ten times produce useful values, while the eleventh terminates the iteration.

The previous example shows you a solution where the **iterator object is a part of a more complex class**.

The code isn't really sophisticated, but it presents the concept in a clear way.

Take a look at the code.

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

    def __iter__(self):
        print("Fib iter")
        return self

    def __next__(self):
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret

class Class:
    def __init__(self, n):
        self.__iter = Fib(n)

    def __iter__(self):
        print("Class iter")
        return self.__iter;

object = Class(8)

for i in object:
    print(i)
```

We've built the `Fib` iterator into another class (we can say that we've composed it into the `Class` class). It's instantiated along with `Class`'s object.

The object of the class may be used as an iterator when (and only when) it positively answers to the `__iter__` invocation - this class can do it, and if it's invoked in this way, it provides an object able to obey the iteration protocol.

This is why the output of the code is the same as previously, although the object of the `Fib` class isn't used explicitly inside the `for` loop's context.

The `yield` statement

The iterator protocol isn't particularly difficult to understand and use, but it is also indisputable that the **protocol is rather inconvenient**.

The main discomfort it brings is **the need to save the state of the iteration between subsequent `__iter__` invocations**.

For example, the `Fib` iterator is forced to precisely store the place in which the last invocation has been stopped (i.e., the evaluated number and the values of the two previous elements). This makes the code larger and less comprehensible.

This is why Python offers a much more effective, convenient, and elegant way of writing iterators.

The concept is fundamentally based on a very specific and powerful mechanism provided by the `yield` keyword.

You may think of the `yield` keyword as a smarter sibling of the `return` statement, with one essential difference.

Take a look at this function:

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

It looks strange, doesn't it? It's clear that the `for` loop has no chance to finish its first execution, as the `return` will break it irrevocably.

Moreover, invoking the function won't change anything - the `for` loop will start from scratch and will be broken immediately.

We can say that such a function is not able to save and restore its state between subsequent invocations.

This also means that a function like this **cannot be used as a generator**.

We've replaced exactly one word in the code - can you see it?

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

We've added `yield` instead of `return`. This little amendment **turns the function into a generator**, and executing the `yield` statement has some very interesting effects.

First of all, it provides the value of the expression specified after the `yield` keyword, just like `return`, but doesn't lose the state of the function.

All the variables' values are frozen, and wait for the next invocation, when the execution is resumed (not taken from scratch, like after `return`).

There is one important limitation: such a **function should not be invoked explicitly** as - in fact - it isn't a function anymore; **it's a generator object**.

The invocation will **return the object's identifier**, not the series we expect from the generator.

Due to the same reasons, the previous function (the one with the `return` statement) may only be invoked explicitly, and must not be used as a generator.

How to build a generator

Let us show you the new generator in action.

This is how we can use it:

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

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

Can you guess the output?

Answer:

```
0
1
2
3
4
```

How to build your own generator

What if you need a **generator to produce the first *n* powers of *2***?

Nothing easier. Just look at the code below.

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

for v in powersOf2(8):
    print(v)
```

Can you guess the output? Run the code to check your guesses.

Generators may also be used within **list comprehensions**, just like here:

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

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

print(t)
```

Run the example and check the output.

The `list()` function can transform a series of subsequent generator invocations into **a real list**:

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

```
t = list(powersOf2(3))

print(t)
```

Again, try to predict the output and run the code to check your predictions.

Moreover, the context created by the `in` operator allows you to use a generator, too.

The example shows how to do it:

```
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)
```

What's the code's output? Run the program and check.

Now let's see a **Fibonacci number generator**, and ensure that it looks much better than the objective version based on the direct iterator protocol implementation.

Here it is:

```
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)
```

Guess the output (a list) produced by the generator, and run the code to check if you were right.

More about list comprehensions

You should be able to remember the rules governing the creation and use of a very special Python phenomenon named **list comprehension - a simple and very impressive way of creating lists and their contents**.

In case you need it, we've provided a quick reminder below.

```
listOne = []

for ex in range(6):
    listOne.append(10 ** ex)


listTwo = [10 ** ex for ex in range(6)]

print(listOne)
print(listTwo)
```

There are two parts inside the code, both creating a list containing a few of the first natural powers of ten.

The former uses a routine way of utilizing the `for` loop, while the latter makes use of the list comprehension and builds the list in situ, without needing a loop, or any other extended code.

It looks like the list is created inside itself - it's not true, of course, as Python has to perform nearly the same operations as in the first snippet, but it is indisputable that the second formalism is simply more elegant, and lets the reader avoid any unnecessary details.

The example outputs two identical lines containing the following text:

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

Run the code to check if we're right.

There is a very interesting syntax we want to show you now. Its usability is not limited to list comprehensions, but we have to admit that comprehensions are the ideal environment for it.

It's a **conditional expression - a way of selecting one of two different values based on the result of a Boolean expression**.

Look:

```
expression_one if condition else expression_two
```

It may look a bit surprising at first glance, but you have to keep in mind that it is **not a conditional instruction**. Moreover, it's not an instruction at all. It's an operator.

The value it provides is equal to `expression_one` when the condition is `True`, and `expression_two` otherwise.

A good example will tell you more. Look at the code.

```
lst = []

for x in range(10):
    lst.append(1 if x % 2 == 0 else 0)

print(lst)
```

The code fills a list with `1`'s and `0`s - if the index of a particular element is odd, the element is set to `0`, and to `1` otherwise.

Simple? Maybe not at first glance. Elegant? Indisputably.

Can you use the same trick within a list comprehension? Yes, you can.

Look at the example.

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

print(lst)
```

Compactness and elegance - these two words come to mind when looking at the code.

So, what do they have in common, generators and list comprehensions? Is there any connection between them? Yes. A rather loose connection, but an unequivocal one.

Just one change can **turn any comprehension into a generator**.

Now look at the code below and see if you can find the detail that turns a list comprehension into a generator:

```
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()
```

It's the **parentheses**. The brackets make a comprehension, the parentheses make a generator.

The code, however, when run, produces two identical lines:

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

How can you know that the second assignment creates a generator, not a list?

There is some proof we can show you. Apply the `len()` function to both these entities.

`len(lst)` will evaluate to `10`. Clear and predictable. `len(genr)` will raise an exception, and you will see the following message:

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

Of course, saving either the list or the generator is not necessary - you can create them exactly in the place where you need them - just like here:

```
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()
```

Note: the same appearance of the output doesn't mean that both loops work in the same way. In the first loop, the list is created (and iterated through) as a whole - it actually exists when the loop is being executed.

In the second loop, there is no list at all - there are only subsequent values produced by the generator, one by one.

Carry out your own experiments.

The `lambda` function

The `lambda` function is a concept borrowed from mathematics, more specifically, from a part called *the Lambda calculus*, but these two phenomena are not the same.

Mathematicians use *the Lambda calculus* in many formal systems connected with logic, recursion, or theorem provability. Programmers use the `lambda` function to simplify the code, to make it clearer and easier to understand.

A `lambda` function is a function without a name (you can also call it **an anonymous function**). Of course, such a statement immediately raises the question: how do you use anything that cannot be identified?

Fortunately, it's not a problem, as you can name such a function if you really need, but, in fact, in many cases the `lambda` function can exist and work while remaining fully incognito.

The declaration of the `lambda` function doesn't resemble a normal function declaration in any way - see for yourself:

```
lambda parameters : expression
```

Such a clause **returns the value of the expression when taking into account the current value of the current `lambda` argument**.

As usual, an example will be helpful. Our example uses three `lambda` functions, but gives them names. Look at it carefully:

```
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()))
```

Let's analzye it:

- the first `lambda` is an anonymous **parameterless function** that always returns `2`. As we've **assigned it to a variable named** `two`, we can say that the function is not anonymous anymore, and we can use the name to invoke it.
- the second one is a **one-parameter anonymous function** that returns the value of its squared argument. We've named it as such, too.
- the third `lambda` **takes two parameters** and returns the value of the first one raised to the power of the second one. The name of the variable which carries the `lambda` speaks for itself. We don't use `pow` to avoid confusion with the built-in function of the same name and the same purpose.

The program produces the following output:

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

This example is clear enough to show how `lambda`s are declared and how they behave, but it says nothing about why they're necessary, and what they're used for, since they can all be replaced with routine Python functions.

Where is the benefit?

How to use lambdas and what for?

The most interesting part of using lambdas appears when you can use them in their pure form - **as anonymous parts of code intended to evaluate a result**.

Imagine that we need a function (we'll name it `printfunction`) which prints the values of a given (other) function for a set of selected arguments.

We want `printfunction` to be universal - it should accept a set of arguments put in a list and a function to be evaluated, both as arguments - we don't want to hardcode anything.

Look at the example code.

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

def poly(x):
    return 2 * x**2 - 4 * x + 2

printfunction([x for x in range(-2, 3)], poly)
```

This is how we've implemented the idea.

Let's analyze it. The `printfunction()` function takes two parameters:

- the first, a list of arguments for which we want to print the results;
- the second, a function which should be invoked as many times as the number of values that are collected inside the first parameter.

Note: we've also defined a function named `poly()` - this is the function whose values we're going to print. The calculation the function performs isn't very sophisticated - it's the polynomial (hence its name) of a form:

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

The name of the function is then passed to the `printfunction()` along with a set of five different arguments - the set is built with a list comprehension clause.

The code prints the following lines:

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

Can we avoid defining the `poly()` function, as we're not going to use it more than once? Yes, we can - this is the benefit a lambda can bring.

Look at the example below. Can you see the difference?

```
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)
```

The `printfunction()` has remained exactly the same, but there is no `poly()` function. We don't need it anymore, as the polynomial is now directly inside the `printfunction()` invocation in the form of a lambda defined in the following way: `lambda x: 2 * x**2 - 4 * x + 2`.

The code has become shorter, clearer, and more legible.

Let us show you another place where lambdas can be useful. We'll start with a description of `map()`, a built-in Python function. Its name isn't too descriptive, its idea is simple, and the function itself is really usable.

Lambdas and the `map()` function

In the simplest of all possible cases, the `map()` function takes two arguments:

- a function;
- a list.

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

The above description is extremely simplified, as:

- the second `map()` argument may be any entity that can be iterated (e.g., a tuple, or just a generator)

- `map()` can accept more than two arguments.

The `map()` **function applies the function passed by its first argument to all its second argument's elements, and returns an iterator delivering all subsequent function results**. You can use the resulting iterator in a loop, or convert it into a list using the `list()` function.

Can you see a role for any lambda here?

Look at the code below - we've used two lambdas in it.

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

This is the intrigue:

- build the `list1` with values from `0` to `4`;
- next, use `map` along with the first `lambda` to create a new list in which all elements have been evaluated as `2` raised to the power taken from the corresponding element from `list1`;
- `list2` is printed then;
- in the next step, use the `map()` function again to make use of the generator it returns and to directly print all the values it delivers; as you can see, we've engaged the second `lambda` here - it just squares each element from `list2`.

Try to imagine the same code without lambdas. Would it be any better? It's unlikely.

Lambdas and the `filter()` function

Another Python function which can be significantly beautified by the application of a lambda is `filter()`.

It expects the same kind of arguments as `map()`, but does something different - it **filters its second argument while being guided by directions flowing from the function specified as the first argument** (the function is invoked for each list element, just like in `map()`).

The elements which return `True` from the function **pass the filter** - the others are rejected.

The example below shows the `filter()` function in action.

```
from random import seed, randint

seed()
data = [ randint(-10,10) for x in range(5) ]
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))
print(data)
print(filtered)
```

Note: we've made use of the `random` module to initialize the random number generator (not to be confused with the generators we've just talked about) with the `seed()` function, and to produce five random integer values from `-10` to `10` using the `randint()` function.

The list is then filtered, and only the numbers which are even and greater than zero are accepted.

Of course, it's not likely that you'll receive the same results, but this is what our results looked like:

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

A brief look at closures

Let's start with a definition: **closure is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore**. Intricate? A bit.

Let's analyze a simple example:

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

var = 1
outer(var)

print(var)
print(loc)
```

The example is obviously erroneous.

The last two lines will cause a `NameError` exception - neither `par` nor `loc` is accessible outside the function. Both the variables exist when and only when the `outer()` function is being executed.

Look at the example.

```
1 def outer(par):
2     loc = par
3     def inner():
4         return loc
5     return inner
6
7 var = 1
8 fun = outer(var)
9 print(fun())
```

We've modified the code significantly.

There is a brand new element in it - a function (named `inner`) inside another function (named `outer`).

How does it work? Just like any other function except for the fact that `inner()` may be invoked only from within `outer()`. We can say that `inner()` is `outer()`'s private tool - no other part of the code can access it.

Look carefully:

- the `inner()` function returns the value of the variable accessible inside its scope, as `inner()` can use any of the entities at the disposal of `outer()`
- the `outer()` function returns the `inner()` function itself; more precisely, it returns a copy of the `inner()` function, the one which was frozen at the moment of `outer()`'s invocation; the frozen function contains its full environment, including the state of all local variables, which also means that the value of `loc` is successfully retained, although `outer()` ceased to exist a long time ago.

In effect, the code is fully valid, and outputs:

```
1
```

The function returned during the `outer()` invocation is a **closure**.

A closure has to be invoked in exactly the same way in which it has been declared.

In the previous example (see the code below):

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

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

the `inner()` function was parameterless, so we had to invoke it without arguments.

Now look at the below code.

```
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))
```

It is fully possible to **declare a closure equipped with an arbitrary number of parameters**, e.g., one, just like the `power()` function.

This means that the closure not only makes use of the frozen environment, but it can also **modify its behavior by using values taken from the outside**.

This example shows one more interesting circumstance - you can **create as many closures as you want using one and the same piece of code**. This is done with a function named `makeclosure()`. Note:

- the first closure obtained from `makeclosure()` defines a tool squaring its argument;
- the second one is designed to cube the argument.

This is why the code produces the following output:

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

Carry out your own tests.