Generators and Lambda Function

~~Fantastic Beasts~~ Generators and where to find them

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 might now have realized, but you've encountered generators many, many time before.

```
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.

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**.

Here is an example of Fibonacci numbers (Fib_i) (Fibonacci number - Wikipedia).

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

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

    def __next__(self):
        print("__next__")
```

```
        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

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

Let's analyze the code:

- Lines 2 to 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 to 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 to 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.

Expected output:

```
__init__
__iter__
__next__
1
__next__
1
__next__
2
__next__
3
__next__
5
__next__
```

The iterator object is instantiated first.

Next, Python invokes the __$iter$__ method to get access to the actual iterator.

The __$next$__ method is invoked six times - the first five times produce useful values, while the eleventh terminates the iteration.

```
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)
```

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

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.

Expected output:

```
Class iter
1
1
2
```

```
3
5
8
13
21
```

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.

```
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**.

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2

for v in powers_of_2(8):
    print(v)
```

This can print the first 8 powers of 2.

Generators can also be used within list comprehensions.

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2

t = [x for x in powers_of_2(5)]
```

```
print(t) # [1, 2, 4, 8, 16]
```

The *list()* functions can transform a series of subsequent generator invocations into a real list.

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2

t = list(powers_of_2(3))
print(t) # [1, 2, 4]
```

The context created by the *in* operator allows you to use a generator too.

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2

for i in powers_of_2(4):
    print(i)
```

Here is the upgraded version of the Fibonacci number generator, it looks much better than the objective version.

```
def fibonacci(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(fibonacci(10))
print(fibs) # [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

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.

```
list_1 = []
for ex in range(6):
    list_1.append(10 ** ex)

list_2 = [10 ** ex for ex in range(6)]
print(list_1) # [1, 10, 100, 1000, 10000, 100000]
```

```
print(list_2) # [1, 10, 100, 1000, 10000, 100000]
```

Here is a quick recap.

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_list = [1 if x % 2 == 0 else 0 for x in range(10)]
print(the_list) # [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]

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

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

It's the **parentheses**.

The brackets make a comprehension, the parentheses make a generator.

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

```
print(len(the_list)) # 10
print(len(the_generator)) # TypeError
```

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()
# 1 0 1 0 1 0 1 0 1 0
# 1 0 1 0 1 0 1 0 1 0
```

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.

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:

```
lambda parameters: experssion
```

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

```
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()))
# 4 4
# 1 1
# 0 0
# 1 1
# 4 4
```

Here is an example that uses three $lambda$ functions.

- 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.

However, you may ask.

Where is the benefit of using such?

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 $print\_function$) which prints the values of a given (other) function for a set of selected arguments.

We want $print\_function$ 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.

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

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

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

Here is the implemented code.

The $print\_function()$ function takes two parameters:

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

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 $print\_function()$ along with a set of five different arguments - the set is built with a list comprehension clause.

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

This is the expected output.

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.

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

print_function([x for x in range(-2, 3)], lambda x: 2 * x ** 2 - 4 * x + 2)
```

This code here gives the same output and result as the previous one.

The $print\_function()$ 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 $print\_function()$ 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.

Lambdas and the $map()$ function [Map and Filter Explained - YT]

Let me 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.

In the simplest of all possible cases, the $map()$ function:

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

Takes two arguments:

1. A function
2. A 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.

```
list_1 = [x for x in range(5)]
list_2 = list(map(lambda x: 2 ** x, list_1))
print(list_2) # [1, 2, 4, 8, 16]

for x in map(lambda x: x * x, list_2):
    print(x, end=' ') # 1 4 16 64 256
print()
```

We've used two lambdas in the code here.

1. Build the $list\_1$ with values from 0 to 4.
2. 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 $list\_1$.
3. $list\_2$ is printed.
4. 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 $list\_2$.

Lambdas and the $filter()$ function

$filter()$ 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.

```
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) # [2, -5, 10, 6, -6]
print(filtered) # [2, 10, 6]
```

Here is an example, and your output might be different from mine, as we used the $randint()$ function.

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

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?

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

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

The code here 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.

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

    def inner():
        return loc
    return inner

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

I've changed 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.

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.

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.

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

    def power(p):
        return p ** loc
    return power

fsqr = make_closure(2)
fcub = make_closure(3)
```

```
for i in range(5):
    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 $make\_closure()$. Note:

- The first closure obtained from $make\_closure()$ defines a tool squaring its argument.
- The second one is designed to cube the argument.

Expected output:

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