CERTIFIED ASSOCIATE IN PYTHON PROGRAMMING

BY: IMRAN

Generators

- 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.**
- **Below is a simple example of generators**

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

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

Generators

- 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**.

Generators

- 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)

Generators

```
class Fib:
    def __init__(self, nn):
        print("__init__")
        self.__n = nn
        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(10):
    print(i)
```

Generators

- 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;

Generators

- 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.

Generators

```
class Fib:
    def __init__(self, nn):
        print("__init__")
        self.__n = nn
        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(10):
    print(i)
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 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.

The yield statement

- 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**.

The yield statement

- 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).

The yield statement

- 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.

The yield statement

- **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?

The yield statement

```
• 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)
```

The yield statement

- **List comprehensions**
- Generators may also be used within **list comprehensions**, just like here:
- `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)`

The yield statement

- **The `list()` function**
- The list() function 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)
```

The yield statement

- **The `in` operator**
- Moreover, the context created by the in operator allows you to use a generator, too.
- The example shows how to do it:

```
def powers_of_2(n):
power = 1
for i in range(n):
yield power
power *= 2
for i in range(20):
if i in powers_of_2(4):
print(i)
```

The yield statement

- **The Fibanacci number generator**
- 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 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)
```

list comprehensions

- list_1 = []
- for ex in range(6):
-     list_1.append(10 ** ex)
- list_2 = [10 ** ex for ex in range(6)]
- print(list_1)
- print(list_2)

list comprehensions

```
the_list = []

for x in range(10):

    the_list.append(1 if x % 2 == 0 else 0)

print(the_list)
```

list comprehensions

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

print(the_list)
```

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 lambda function

- 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:
-

The lambda function

- 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()))
```

Lambdas

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

Lambdas and the map() function

- In the simplest of all possible cases, the map() function:
- `map(function, list)`
- takes two arguments:
  - a function;
  - 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.

Module 4: MISCELLANEOUS
Part 1: Generators, Iterators, and Closures

Lambdas and the map () functi on

```
list_1 = [x for x in range(5)]
list_2 = list(map(lambda x: 2 ** x, list_1))
print(list_2)
for x in map(lambda x: x * x, list_2):
    print(x, end=' ')
print()
```

ImranNust imran_muet muhammad-imran-b7865495

Module 4: MISCELLANEOUS
Part 1: Generators, Iterators, and Closures

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.

ImranNust imran_muet muhammad-imran-b7865495

Lambdas and the filter() function

```
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)
```

Lambdas and the filter() function

```
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)
```

Key Takeaways

- 1. An **iterator** is an object of a class providing at least **two** methods (not counting the constructor!):
- __iter__() is invoked once when the iterator is created and returns the iterator's object **itself**;
- __next__() is invoked to provide the **next iteration's value** and raises the StopIteration exception when the iteration **comes to and end**.
-

  2. The yield statement can be used only inside functions. The yield statement suspends function execution and causes the function to return the yield's argument as a result. Such a function cannot be invoked in a regular way – its only purpose is to be used as a **generator** (i.e. in a context that requires a series of values, like a for loop.)

Key Takeaways

- 3. A **conditional expression** is an expression built using the if-else operator. For example:
- `print(True if 0 >=0 else False)`
- outputs True.
- 4. A **list comprehension** becomes a **generator** when used inside **parentheses** (used inside brackets, it produces a regular list). For example:
- `for x in (el * 2 for el in range(5)):`
- `print(x)`
- outputs 02468.

Key Takeaways

- 4. A **lambda function** is a tool for creating **anonymous functions**. For example:
- `def foo(x,f):`
- `return f(x)`
- `print(foo(9, lambda x: x ** 0.5))`
  outputs 3.0.
- 5. The map(fun, list) function creates a **copy** of a list argument, and applies the fun function to all of its elements, returning a **generator** that provides the new list content element by element. For example:
- `short_list = ['mython', 'python', 'fell', 'on', 'the', 'floor']`
- `new_list = list(map(lambda s: s.title(), short_list))`
- `print(new_list)`
- outputs ['Mython', 'Python', 'Fell', 'On', 'The', 'Floor'].

Module 4: MISCELLANEOUS Part 1: Generators, Iterators, and Closures

Key Takeaways

- 6. The filter(fun, list) function creates a **copy** of those list elements, which cause the fun function to return True. The function's result is a **generator** providing the new list content element by element. For example:

```
short_list = [1, "Python", -1, "Monty"]
new_list = list(filter(lambda s: isinstance(s, str), short_list))
print(new_list)
```

outputs ['Python', 'Monty'].

ImranNust imran_muet muhammad-imran-b7865495

Key Takeaways

- 7. A 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**. For example:

```
def tag(tg):
tg2 = tg
tg2 = tg[0] + '/' + tg[1:]
def inner(str):
return tg + str + tg2
return inner
b_tag = tag('<b>')
print(b_tag('Monty Python'))
```

outputs <b>Monty Python</b>