Python Iterators, Generators, Decorators

BY

AMAR PANCHAL

What are iterators in Python?

- **Iterators are objects that can be iterated upon.**
- Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
- Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

CODE

```
my_list = [10, 20, 30, 40]
my_iter = iter(my_list)
print(next(my_iter))
print(next(my_iter))
print(my_iter.__next__())
print(my_iter.__next__())
```


Building Your Own Iterator in Python

- We just have to implement the methods __iter__() and __next__().

CODE

```
class PowTwo:

        def __init__(self, max = 0):
                self.max = max
                self.n = 0

        def __next__(self):
                if self.n <= self.max:
                        result = 2 ** self.n
                        self.n += 1
                        return result
                else:
                    raise StopIteration
```

Prof.Amar Panchal | 9821601163 | www.amarpanchal.com

What are generators in Python?

- Python generators are a simple way of creating iterators.
- Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).
- It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement.
- If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.
- The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Prof.Amar Panchal | 9821601163 | www.amarpanchal.com

Differences between Generator function and a Normal function

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

CODE

```
# A simple generator functiondef
def my_gen():
        n = 1
        print('This is printed first')
        yield n
        n += 1
        print('This is printed second')
        yield n
        n += 1
        print('This is printed at last')
        yield n
```

Srting

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
```


Python Generator Expression

- my_list = [1, 3, 6, 10]
- a = (x**2 for x in my_list)

Why generators are used in Python?

- 1. Easy to Implement
- 2. Memory Efficient
- 3. Represent Infinite Stream

Python Decorators

- Python has an interesting feature called **decorators** to add functionality to an existing code.
- This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

Functions are Objects

```
def first(msg):
        print(msg)
first("Hello")
second = first
second("Hello")
```


Functions inside Functions

```
def f():

    def g():
        print("Hi, it's me 'g'")#4
        print("Thanks for calling me")#5

    print("This is the function 'f'")#1
    print("I am calling 'g' now:")#2
    g()#3


f()
```

Functions as Parameters

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)
```



```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()
    print("func's real name is " + func.__name__)


f(g)
```

Functions returning Functions

```
def f(x):
    def g(y):
        return y + x + 3
    return g

nf1 = f(1)
nf2 = f(3)

print(nf1(1))
print(nf2(1))
```


A Simple Decorator

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo("Hi")
```

The Usual Syntax for Decorators in Python

- The decoration in Python is usually not performed in the way we did it in our previous example, even though the notation foo = our_decorator(foo) is catchy and easy to grasp. This is the reason, why we used it! You can also see a design problem in our previous approach. "foo" existed in the same program in two versions, before decoration and after decoration.
- We will do a proper decoration now. The decoration occurrs in the line before the function header. The "@" is followed by the decorator function name.
- We will rewrite now our initial example. Instead of writing the statement
- foo = our_decorator(foo)
- we can write
- @our_decorator


```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```