GENERATORS AND DECORATORS

Generators

Generators are very easy to implement, but a bit difficult to understand. They are used to create iterators (like lists), but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time). Confused? Good. Examples are an easier way to understand this than theory, so let us see some first:

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n


>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()

>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1
>>> # Once the function yields, the function is paused and the control is
>>> # transferred to the caller.
>>> # Local variables and theirs states are remembered between successive
>>> # calls.
>>> next(a)
This is printed second
2
>>> next(a)
This is printed at last
3
>>> # Finally, when the generator terminates, StopIteration is raised
automatically on further calls.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that, the value of variable *n* is remembered between each call. You can think of `yield` as a way to pause the execution of the function.

Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.

To restart the process we need to create another generator object using something like a = my_gen().

One final thing to note is that we can use generators with for loops directly.

This is because, a for loop takes an iterator and iterates over it using next() function. It automatically ends when StopIteration is raised.

Example using generator with a for loop:

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n

# Using for loop
for item in my_gen():
    print(item)

# When you run the program, the output will be:
This is printed first
1
This is printed second
2
This is printed at last
3
```

The above example is of less use and we studied it just to get an idea of what was happening in the background.

Normally, generator functions are implemented with a loop having a suitable terminating condition.

Let's take an example of a generator that reverses a string.

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

# For loop to reverse the string
# Output:
# o
# l
# l
# e
# h
for char in rev_str("hello"):
     print(char)
```

An example for a generator which yields 7 random integers for a lottery:

```
import random

def lottery():
    # returns 6 numbers between 1 and 40
    for i in range(6):
        yield random.randint(1, 40)

    # returns a 7th number between 1 and 15
    yield random.randint(1, 15)

for number in lottery():
       print("And the next number is...\n", number)
```

Why Generators?

We've taught you what generators are, but you might be thinking, this can be done by manipulating lists and for loops too. Why do the extra steps? Turns out, there is a very big advantage to using generators in some cases. And that is in terms of resources required by them is far lower than other type of object.

```
>>> def my_gen():
...  for i in range(1000000):
...   yield i
...
>>> my_list = []
>>> for i in range(1000000):
...  my_list.append(i)
...
>>> import sys
>>> sys.getsizeof(my_list)
8697464  # size in bytes occupied by the list
>>> sys.getsizeof(my_gen)
136  # size occupied by the generator
```

At any point, a generator doesn't contain ALL of the data. It just remembers the current state of its variables, and the way to generate the next sequence. This can become very time and

memory friendly. Just try adding one more 0 to the range arguments, and you will understand.

Exercise:

Write a generator function which returns the Fibonacci series. They are calculated using the following formula: The first two numbers of the series are equal to 1, and each consecutive number returned is the sum of the last two numbers. Hint: Can you use only two variables in the generator function? Remember that assignments can be done simultaneously.

```
import types

# fill in this function
def fib():
    pass

# testing code
if type(fib()) == types.GeneratorType:
    print("Good, The fib function is a generator.")

    counter = 0
    for n in fib():
        print(n)
        counter += 1
        if counter == 15:
            break
```

Advanced:

Modify the above function to accept an argument `n`, such that it generates all the Fibonacci numbers less than or equal to `n` (and remove the need of the `counter` variable).

Decorators

What are decorators in Python?

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.

Prerequisites for learning decorators

In order to understand about decorators, we must first know a few basic things in Python.

We must be comfortable with the fact that, everything in Python (yes, even classes), are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too (with attributes). Various names can be bound to the same function object.

Here is an example:

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

second = first
second("Hello")
```

When you run the code, both functions first and second give same output. Here, the names first and second refer to the same function object.

Now things start getting weirder.

Functions can be passed as arguments to another function.

If you have used functions like map, filter and reduce in Python, then you already know about this.

Such function that take other functions as arguments are also called higher order functions. Here is an example of such a function:

```
def inc(x):
    return x + 1

def dec(x):
    return x - 1

def operate(func, x):
    result = func(x)
    return result
```

We invoke the function as follows:

```
>>> operate(inc, 3)
4
>>> operate(dec, 3)
2
```

Furthermore, a function can return another function.

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned

new = is_called()
new() #Outputs "Hello"
```

Here, `is_returned()` is a nested function which is defined and returned, each time we call `is_called()`.

Getting back to decorators

Functions and methods are called **callable** as they can be called.
Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")
```

```
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

In the example shown above, `make_pretty()` is a decorator.

In the assignment step.

```
pretty = make_pretty(ordinary)
```

The function `ordinary()` got decorated and the returned function was given the name pretty.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary)
```

This is a common construct and for this reason, Python has a syntax to simplify this.
We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")

is equivalent to

def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just syntactic sugar to implement decorators.

Decorating functions with parameters

The above decorator was simple, and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like below?

```
def divide(a, b):
    return a/b
```

This function has two parameters, a and b. We know that it will raise `ZeroDivisionError` if we pass in b as 0.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

Let us make a decorator to check for this case that will cause the error.

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return
        return func(a,b)
    return inner

@smart_divide
def divide(a,b):
    return a/b
```

This new implementation will return None if the error condition arises.

```
>>> divide(2,5)
I am going to divide 2 and 5
0.4

>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

In this manner we can decorate functions that take parameters.
Notice that parameters of the nested `inner()` function inside the decorator is same as the parameters of functions it decorates.

Chaining decorators

Multiple decorators can be chained in Python.
This is to say, a function can be decorated multiple times with different (or same) decorators.
We simply place the decorators above the desired function.

```
def star(func):
    def inner(text):
        print("*" * 30)
        func(text)
        print("*" * 30)
    return inner

def percent(func):
    def inner(text):
        print("%" * 30)
```

```
        func(text)
        print("%" * 30)
    return inner

@star
@percent
def printer(msg):
    print(msg)
printer("Hello")
```

Output:

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

The above syntax of

```
@star
@percent
def printer(msg):
    print(msg)
```

is equivalent to

```
def printer(msg):
    print(msg)
printer = star(percent(printer))
```

The order in which we chain decorators, obviously, matters. Had we reversed the order as,

```
@percent
@star
def printer(msg):
    print(msg)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```