02 - Python_Intermediate_Level_Course_Notes

November 25, 2021

1 Python Intermediate Level

By: Himalaya Kakshapati

In this notebook, the following topics will be explored. * Functional Programming * Object-Oriented Programming * Comprehensions * Iterators * Generators * Decorators * Closure

2 Functional Programming in Python

Functional programming is a programming paradigm where programs are constructed by applying and composing functions.

2.1 map() Function

The map() function * iterates through all the items in the given iterable, and * executes the function we passed as an argument on each of them.

The syntax is:

`map(function, iterable)`

[ ]:
```
def starts_with_A(s):
    return s[0] == "A"

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
map_object = map(starts_with_A, fruit)

print(list(map_object))
```

```
[True, False, False, True, False]
```

[ ]:
```
# using lambda function
map_object = map(lambda s: s[0] == "A", fruit)

print(list(map_object))
```

```
[True, False, False, True, False]
```

2.2 filter() function

Similar to the map() function, filter() function takes a function object and an iterable and creates a new list.

As the name suggests, filter() forms a new list that contains only elements that satisfy a certain condition, i.e. the function we passed returns True.

The syntax is:

filter(function, iterable)

```
[ ]: def starts_with_A(s):
         return s[0] == "A"

     fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
     fruits_starting_with_A = filter(starts_with_A, fruit)

     print(list(fruits_starting_with_A))
```

```
['Apple', 'Apricot']
```

```
[ ]: # using lambda function
     fruits_starting_with_A = filter(lambda s: s[0] == 'A', fruit)
     list(fruits_starting_with_A)
```

```
[ ]: ['Apple', 'Apricot']
```

2.3 reduce() function

reduce() works differently than map() and filter(). It does not return a new list based on the function and iterable we've passed. Instead, it returns a single value.

Also, in Python 3 reduce() isn't a built-in function anymore, and it can be found in the functools module.

The syntax is:

reduce(function, sequence[, initial])

reduce() works by calling the function we passed for the first two items in the sequence. The result returned by the function is used in another call to function alongside with the next (third in this case), element.

This process repeats until we've gone through all the elements in the sequence.

The optional argument initial is used, when present, at the beginning of this "loop" with the first element in the first call to function. In a way, the initial element is the 0th element, before the first one, when provided.

```
[16]: from functools import reduce

      def add(x, y):
```

```
    return x + y

lst = [2, 4, 7, 3]
reduce(add, lst)
```

```
[16]: 16
```

```
[17]: # with initial value
reduce(add, lst, 100)
```

```
[17]: 116
```

```
[18]: # using lambda function
reduce(lambda x,y: x+y, lst, 100)
```

```
[18]: 116
```

3 Object-oriented Concepts

Everything in Python is an object. For example, int, str, float, list, tuple, dictionary etc. are all objects. Let's try to understand what objects are.

Class and object instances

A Class is like a blueprint for building objects. Once you have a blueprint, you can make any number of copies of the blueprint. The 'copies' of the blueprint are called object instances.

Let's create a class called `Student`

```
[ ]: class Student:
       pass # pass means do nothing
```

Now, lets create 'copies' known as objects or instances of the class `Student`.

```
[ ]: student1 = Student() # instance of the class Student
student2 = Student() # another instance of the class Student
print(type(student1)) # the type of student1 is Class Student
```

```
<class '__main__.Student'>
```

Let us see the location in memory where the object instances are created. We use the function `id()` for that.

```
[ ]: print(id(student1))
print(id(student2))
student1 is student2
```

```
139682129658192
139682129660304
```

```
[ ]: False
```

```
[ ]: student3 = student1
     student3 is student1
```

```
[ ]: True
```

As you can see above, the location in memory of the two instances are different, meaning that they are two different objects. This is verified by the keyword `is`. Please note that `is` keyword returns `True` if the variables on the left and the right side of the `is` operator are pointing to the same object. If they are pointing to different objects, then it returns `False`.

In the code below, we are assigning the object instance `student1` to the variable `student3`. In this case, we see that the location in memory of both `student1` and `student3` are the same, as verified by the `is` keyword.

This means that when we assign an object to a variable, the variable points to the object.

```
[ ]: student3 = student1
     print(id(student1))
     print(id(student3))
     student1 is student3
```

```
140572972276368
140572972276368
```

```
[ ]: True
```

Class/instance variables and methods

A `class` can have properties and methods. Properties hold data values and methods do some action. Normally, properties are nouns and methods are verbs.

Python class properties can be in the form of

- class variables
- instance variables

Class variables are common to all the instances, that is they are accessible from any instance of the class. Class variables are created when the class is defined.

Instance variables, on the other hand, are specific to instances. The values of instance variables may differ from instance to instance.

Here is an example of a class variable.

```
[ ]: class Student:
       # class variable
       school = "School of Data Science"
```

Below, we create two instances of `Student` class, and we print the value of `school` variable on both instances. We find that both print statements give us the same result, since we are printing the class variable.

```
[ ]: student1 = Student()
     student2 = Student()
     print(student1.school)
     print(student2.school)
```

```
School of Data Science
School of Data Science
```

Let's now look at examples of instance variables.

Instance variables are normally defined inside the `__init__` method.

Let's first understand what 'methods' are. Basically, 'methods' in object-oriented programming are functions defined witnin a class. Just like a normal function, they normally take in some input in the form of arguments, do some processing on the inputs and return the result.

The `__init__` method in Python is a special method, also called a 'constructor', which gets invoked (called) automatically when an instance of that class in created (instantiated).

There are 3 types of methods in Python.

1. instance methods
2. class methods
3. static methods

The most widely used methods are instance methods. Instance methods are defined with `self` as the first parameter.

In the code below, we have defined the `__init__` method with 3 parameters. Inside the `__init__` method, we have defined two instance variables - `self.name` and `self.level`. Instance variables start with `self`.

Please note that instance variables `self.name` and `self.level` may have different values depending on the instance they are invoked on, unlike class variables which have the same value regardless of the instance they are invoked on.

Below, we define the `Student` class with the constructor defined.

```
[ ]: class Student:
       #constructor
       def __init__(self, name, level):
         # instance variables
         self.name = name
         self.level = level
```

Now, we instantiate two instances of the class. Please note that we are passing two arguments, which are passed to the constructor (`__init__` method). Those two values passed as arguments are what the instance variables inside the constructor are initialized to when creating the instance.

```
[ ]: student1 = Student("Katie Holmes", 1)
     student2 = Student("Mike Tyson", 3)
```

Let's verify!

```
[ ]: print(student1.name)
     print(student1.level)
     print(student2.name)
     print(student2.level)
```

```
Katie Holmes
1
Mike Tyson
3
```

We see that when we printed the instance variables we got the values unique to the instances. Also, we see that the instance variables were initialized with the values passed during the instantiation.

`__str__` method

The `__str__` method is a special method in Python which gets invoked when the object is printed.

```
[ ]: class Student:
       #constructor
       def __init__(self, name, level):
         self.name = name
         self.level = level

       # __str__ method
       def __str__(self):
         return '{} studies in level {}'.format(self.name, self.level)
```

```
[ ]: student1 = Student("Katie Holmes", 1)
     student2 = Student("Mike Tyson", 2)

     print(student1)
     print(student2)
```

```
Katie Holmes studies in level 1
Mike Tyson studies in level 2
```

Example of method

Let's look at one example of a regular method (not special method).

```
[ ]: class Student:
       #constructor
       def __init__(self, name, level):
         self.name = name
         self.level = level

       def __str__(self):
         return '{} studies at level {}'.format(self.name, self.level)
```

```
  # regular method
  def level_up(self):
    self.level += 1
```

```
[ ]: sam = Student("Sam Smith", 1)
     tom = Student("Tom Cruise", 3)
     print(sam)
     print(tom)
```

```
Sam Smith studies at level 1
Tom Cruise studies at level 3
```

Now, when we call the `level_up()` method on the instance `sam`, we see that the level increased by 1. Since we did not call the `level_up()` method on `tom`, his level does not change.

```
[ ]: # invoking method
     sam.level_up()
     print(sam)
     print(tom)
```

```
Sam Smith studies at level 2
Tom Cruise studies at level 3
```

3.1 Inheritence

Inheritance is a mechanism which allows us to create a new class (known as child class) that is based upon an existing class (the parent class), by adding new attributes and methods on top of the existing class. When we do so, the child class inherits attributes and methods of the parent class.

Inheritance really shines when you want to create classes that are very similar. All you need to do is to write the code for the things that they have common in one class - the parent class. And then write code for things that are very specific in a different class - the child class. This saves you from duplicating a lot of code.

Let's take a more concrete example to illustrate this concept.

Suppose we are creating a program which deals with various shapes. Each shape has some common properties. For example, color of the shape, whether it is filled or not and so on. In addition to that, there are some properties which vary from shape to shape. For example, area and perimeter. The area of rectangle is `width * length` whereas the area of circle is `r²` . At first, it might be tempting to create classes for different shapes like this:

```
[ ]: # Rectangle class
     class Rectangle:
         def __init__(self, color, filled, width, length): # constructor
             self.__color = color
             self.__filled = filled
             self.__width = width
```

```
        self.__length = length

    # methods
    def get_color(self):
        return self.__color

    def set_color(self, color):
        self.__color = color

    def is_filled(self):
        return self.__filled

    def set_filled(self, filled):
        self.__filled = filled

    def get_area(self):
        return self.__width * self.__length

# Circle class
class Circle:
    def __init__(self, color, filled, radius):
        self.__color = color
        self.__filled = filled
        self.__radius = radius

    def get_color(self):
        return self.__color

    def set_color(self, color):
        self.__color = color

    def is_filled(self):
        return self.__filled

    def set_filled(self, filled):
        self.__filled = filled

    def get_area(self):
        return math.pi * self.__radius ** 2
```

Did you notice the amount of duplicate code we are writing?

Both classes share the same `__color` and `__filled` attribute as well as their getter and setter methods.

To make the situation worse, If we want to update how any of these methods work, then we would have to visit each classes one by one to make the necessary changes.

By using inheritance, we can abstract out common properties to a general Shape class (parent class) and then we can create child classes such as `Rectangle`, `Triangle` and `Circle` that inherits from the `Shape` class.

A child class class inherits all the attributes and methods from it's parent class, but it can also add attributes and methods of it's own.

To create a child class based upon the parent class we use the following syntax:

[ ]:
```
class ParentClass:
    # body of ParentClass
    # method1
    # method2

class ChildClass(ParentClass): # ChildClass inherits from ParentClass
    # body of ChildClass
    # method 1
    # method 2
```

In Object Oriented lingo, When a class `c2` inherits from a class `c1`, we say * class c2 extends class c1, or * class c2 is derived from class c1.

The following program demonstrate inheritance in action. It creates a class named `Shape`, which contains attributes and methods common to all shapes. Then it creates two child classes `Rectangle` and `Triangle` which contains attributes and methods specific to them only.

[ ]:
```
import math

class Shape: # parent class
    def __init__(self, color='black', filled=False):
        self.__color = color
        self.__filled = filled

    def get_color(self):
        return self.__color

    def set_color(self, color):
        self.__color = color

    def get_filled(self):
        return self.__filled

    def set_filled(self, filled):
        self.__filled = filled


class Rectangle(Shape): # Rectangle is derived from 'Shape' class; so it is a␣
 ↪child of 'Shape' class
    def __init__(self, length, breadth):
        super().__init__() # calls the constructor of the parent (Shape) class
        self.__length = length
        self.__breadth = breadth

    def get_length(self):
        return self.__length

    def set_length(self, length):
        self.__length = length

    def get_breadth(self):
        return self.__breadth

    def set_breadth(self, breadth):
        self.__breadth = breadth

    def get_area(self):
        return self.__length * self.__breadth

    def get_perimeter(self):
        return 2 * (self.__length + self.__breadth)


class Circle(Shape):
    def __init__(self, radius):
        super().__init__() # calls the constructor of the parent (Shape) class
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def set_radius(self, radius):
        self.__radius = radius

    def get_area(self):
        return math.pi * self.__radius ** 2

    def get_perimeter(self):
        return 2 * math.pi * self.__radius


r1 = Rectangle(10.5, 2.5) # instantiation of 'Rectangle' class

print("Area of rectangle r1:", r1.get_area())
print("Perimeter of rectangle r1:", r1.get_perimeter())
print("Color of rectangle r1:", r1.get_color())
print("Is rectangle r1 filled ? ", r1.get_filled())
r1.set_filled(True)
```

```python
print("Is rectangle r1 filled ? ", r1.get_filled())
r1.set_color("orange")
print("Color of rectangle r1:", r1.get_color())

c1 = Circle(12) # instantiation of 'Circle' class

print("\nArea of circle c1:", format(c1.get_area(), "0.2f"))
print("Perimeter of circle c1:", format(c1.get_perimeter(), "0.2f"))
print("Color of circle c1:", c1.get_color())
print("Is circle c1 filled ? ", c1.get_filled())
c1.set_filled(True)
print("Is circle c1 filled ? ", c1.get_filled())
c1.set_color("blue")
print("Color of circle c1:", c1.get_color())
```

```
Area of rectangle r1: 26.25
Perimeter of rectangle r1: 26.0
Color of rectangle r1: black
Is rectangle r1 filled ?  False
Is rectangle r1 filled ?  True
Color of rectangle r1: orange

Area of circle c1: 452.39
Perimeter of circle c1: 75.40
Color of circle c1: black
Is circle c1 filled ?  False
Is circle c1 filled ?  True
Color of circle c1: blue
```

In lines `3-19`, we have defined a `Shape` class. It is a parent class and only contains attributes and methods common to all shapes. This class defines two private attributes `__color` and `__filled`, then it provides getter and setter methods for those attributes.

In lines `22-45`, we have defined a `Rectangle` class which inherits from `Shape` class. Pay close attention to the syntax we are using.

This lines tells us that Rectangle class extends the `Shape` class or `Rectangle` class is a child class of Shape class. Thus the `Rectangle` class inherits attributes and methods defined in the `Shape` class. In addition to that, the `Rectangle` class adds two private attributes, getter and setter methods for private attributes, as well as methods to calculate area and perimeter of the rectangle.

Notice the code in line 25.

`super().__init__()`

We use `super()` function to call the parent class methods. So the above code calls `Shape` class's `__init__()` method. This is required to set the values of attributes in the parent class. Otherwise, when you try to access values of attributes defined in parent class using getter or setter methods, you will get an error.

Similarly, In lines `48-63` we have defined a `Circle` class. Just like `Rectangle`, it extends the `Shape` class and adds few attributes and methods of its own.

The code in lines `66-86`, creates `Rectangle` and `Circle` object and then calls `get_area()`, `get_perimeter()`, `get_filled()`, `get_color()`, `set_color()` and `set_filled()` methods on these objects one by one. Notice how we are able to call methods which are defined in the same class, as well as methods which are defined on the parent class.

3.2 Polymorphism and Method Overriding

In a literal sense, Polymorphism means the ability to take various forms. In Python, Polymorphism allows us to define methods in the child class with the same name as defined in their parent class.

As we know, a child class inherits all the methods from the parent class. However, you will encounter situations where the method inherited from the parent class doesn't quite fit into the child class. In such cases, you will have to re-implement method in the child class. This process is known as *Method Overriding*.

In you have overridden a method in child class, then the version of the method will be called based upon the the type of the object used to call it. * If a child class object is used to call an overridden method then the child class version of the method is called. * On the other hand, if parent class object is used to call an overridden method, then the parent class version of the method is called.

The following program demonstrates method overriding in action:

[ ]:
```
class A:
    def explore(self):
        print("explore() method from class A")

class B(A):
    def explore(self): # override
        print("explore() method from class B") # method overriding

b_obj = B() # instance of class B
a_obj = A() # instance of class A

b_obj.explore()
a_obj.explore()
```

```
explore() method from class A
explore() method from class A
```

Here `b_obj` is an object of class `B` (child class), as a result, class `B` version of the explore() method is called.

However, the variable `a_obj` is an object of class `A` (parent class), as a result, class `A` version of the explore() method is called.

If for some reason you still want to access the overridden method of the parent class in the child class, you can call it using the `super()` function as follows:

```
[ ]: class A:
         def explore(self):
             print("explore() method from class A")

     class B(A):
         def explore(self):
             super().explore()  # calling the parent class explore() method
             print("explore() method from class B")


     b_obj = B()
     b_obj.explore()
```

```
explore() method from class A
explore() method from class B
```

You just saw Polymorphism in action - i.e. different invokations (and consequently different behaviors) of the same method, depending on the object the method is called from.

4 Comprehensions

There are 3 types of comprehensions in Python.

- List Comprehension
- Set Comprehension
- Dictionary Comprehension

4.1 List Comprehension

The syntax for List Comprehension is as follows:

[expression `for` item `in` iterable]

```
[ ]: # Examples of List Comprehension
     squares_list = [i*i for i in range(1,10)] # square of numbers from 1 to 9
     squares_odd_list = [x*x for x in range(1,20) if x%2!=0] # squares of odd␣
      ↪numbers from 1 to 19
     print(squares_list)
     print(squares_odd_list)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

List comprehension with if-else

The syntax for List Comprehension with if-else is:

[expression1 `if` conditional `else` expression2 `for` item `in` iterable]

An example is shown below.

```
[ ]: # square a number if it is even; leave it alone if it is odd
     lst = [i*i if i%2==0 else i for i in range(10)]
     lst
```

```
[ ]: [0, 1, 4, 3, 16, 5, 36, 7, 64, 9]
```

Multiple if conditions in List Comprehension

Suppose, if you want to get a list of numbers that are multiples of both 2 and 3, then you would normally use a for loop with 2 if statements as follows.

```
[ ]: lst = []
     for i in range(20):
       if i%2==0:
         if i%3==0:
           lst.append(i)
     print(lst)
```

```
[0, 6, 12, 18]
```

You can do the same thing with just one line of code using list comprehension, as shown below.

```
[ ]: lst = [i for i in range(20) if i%2==0 if i%3==0]
     print(lst)
```

```
[0, 6, 12, 18]
```

Call a method on each element using List Comprehension

You can call a method on each element of an iterable using the list comprehension. Example follows.

```
[ ]: lst = [name.capitalize() for name in ['STEVE', 'KaRa', 'SAIF']]
     print(lst)
```

```
['Steve', 'Kara', 'Saif']
```

```
[ ]: # for loop equivalent
     lst = []
     for name in ['STEVE', 'KaRa', 'SAIF']:
       lst.append(name.capitalize())

     print(lst)
```

```
['Steve', 'Kara', 'Saif']
```

4.2 Set Comprehension

Set Comprehension is very much like the List Comprehension, with the exception that we use the curly braces `{}` instead of square brackets `[]`, and comprehension returns a set, which do not allow duplicates.

Let's look at an example of a set comprehension to come up with a set of Pythagorean Triplets. Pythagorean Triplets are tuples of the form (x, y, z) where x squared plus y squared is equal to z squared.

```
[ ]: pythagorean_triplets = {(x, y, z) for x in range(1,20) for y in range(1,20) for␣
     ↪z in range(1,20) if (x**2 + y**2) == z**2}
     print(pythagorean_triplets) # no duplicates since it is a set
```

```
{(5, 12, 13), (4, 3, 5), (8, 15, 17), (15, 8, 17), (9, 12, 15), (12, 5, 13),
(12, 9, 15), (3, 4, 5), (8, 6, 10), (6, 8, 10)}
```

4.3 Dictionary Comprehension

Dictionary comprehensions are enclosed by curly braces `{}`.

The basic syntax for dictionary comprehension is: {key:value `for` var `in` iterable}

For example, you want to create a dictionary of numbers in the range 1 to 5 and their corresponding square roots.

You can create a dictionary comprehension as follows for the purpose.

```
[ ]: # dict of numbers from 1 to 5 and their corresponding square roots
     square_roots_dict = {num: num**0.5 for num in range(1,6)} #␣
     ↪range(1,6)=>[1,2,3,4,5]
     print(square_roots_dict)
```

```
{1: 1.0, 2: 1.4142135623730951, 3: 1.7320508075688772, 4: 2.0, 5:
2.23606797749979}
```

Another example: Suppose, you want to provide a 10% discount for items with amount more than $10. You can do that using the dictionary comprehension in the following way.

```
[ ]: bill = {"Apple":12, "Orange":7, "Kiwi":25, "Banana":6}
     discounted_bill = {k:(v*0.9 if v>10 else v)  for k,v in bill.items()} # 10%␣
     ↪discount for amount greater than $10
     discounted_bill
```

```
[ ]: {'Apple': 10.8, 'Banana': 6, 'Kiwi': 22.5, 'Orange': 7}
```

5 Iterators

Iterators in Python are objects that can be iterated upon (i.e. all the values can be traversed through one at a time). Iterator object returns data one element at a time.

Iterator object implements two special methods: `__iter__()` and `__next__()`

An object from which we can get an iterator is called an **iterable**. Examples of built-in iterables are: List, Tuple, String, Set etc.

The `iter()` function, which calls the `__iter__()` method of the iterable, returns an iterator from the iterable passed as argument.

Iterating through an Iterator

The `next()` function is used to manually iterate through the items of an iterator. When the end is reached, and there is not any more data to be returned, an `StopIteration` exception is raised.

[ ]:
```
lst = ["hello", "how", "are", "you"] # lst is an iterable
# build an iterator (my_iterator) using the iter() function
my_iterator = iter(lst) # equivalent to lst.__iter__()
# my_iterator = lst.__iter__()

# iterate through the iterator using the next() method
print(next(my_iterator)) # output "hello"
print(next(my_iterator)) # output "how"
print(next(my_iterator)) # output "are"
print(next(my_iterator)) # output "you"
print(my_iterator)
```

```
hello
how
are
you
<list_iterator object at 0x7f775d74c790>
```

[ ]:
```
print(next(my_iterator)) # raises StopIteration exception (no more data)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-45-1d6b77208b9b> in <module>()
----> 1 print(next(my_iterator)) # raises StopIteration exception (no more data

StopIteration:
```

6 Generators

Python generators are a simple way to create iterators. A generator is a function which returns an iterator object. The iterator object can then be iterated over one item at a time.

We can create generator function in Python using the `yield` keyword. A generator function can contain one or more `yield` statements, and it may contain `return` statement as well.

The difference between `return` and `yield` is as follows.

`return` : returns the value and terminates the function.

`yield` : pauses the function saving all its states and continues on from there on subsequent calls. Does not terminate the function.

```
[ ]: # generator function definition
     def my_gen():
         yield 2
         yield 3
         yield 4
```

```
[ ]: # creating generator object
     my_gen_obj = my_gen()
```

```
[ ]: print(type(my_gen_obj)) # my_gen is a generator object
```

```
<class 'generator'>
```

```
[ ]: print(next(my_gen_obj)) # output 2
     print(next(my_gen_obj)) # output 3
     print(next(my_gen_obj)) # output 4
```

```
2
3
4
```

```
[ ]: print(next(my_gen)) # raises StopIteration (no more values to return)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-8297e7aea314> in <module>()
----> 1 print(next(my_gen)) # raises StopIteration (no more values to return)

TypeError: 'function' object is not an iterator
```

Let's see how we can use generator to reverse a string.

```
[ ]: def rev_str(input_str):
       length = len(input_str) # get the length of the string
       for i in range(length-1, -1, -1): # for example, if the string has 4␣
      ↪characters, then i goes from 3 to 0
         yield  input_str[i] # return one character at a time, starting from the end␣
      ↪of the string

     for i in rev_str("generator"):
       print(i)
```

```
r
o
t
a
r
```

```
e
n
e
g
```

6.1 Generator Expression

Simple generator objects can be created using generator expressions. Generator expressions create anonymous generator functions, much like lambda functions create anonymous functions.

Generator expresssions are very similar to list comprehension. Generator expressions use parentheses whereas list comprehensions use square brackets. Also, generator expressions return the data one at a time, whereas list comprehension returns all the data at once.

[ ]:
```
# creating a generator expression
my_gen = (2**i for i in range(10)) # returns 2 to the power i, i going from 0␣
↪to 9, one value at a time
print(my_gen)
```

```
<generator object <genexpr> at 0x7fd9b205fbd0>
```

[ ]:
```
next(my_gen)
```

[ ]: 1

[ ]:
```
for i in my_gen:
  print(i)
```

```
2
4
8
16
32
64
128
256
512
```

[ ]:
```
next(my_gen)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-41-78dc05d55b31> in <module>()
----> 1 next(my_gen)

StopIteration:
```

7 Decorators

A decorator in Python takes in a function, adds some functionality to it and returns it.

Functions are first-class citizens in Python, which means that functions can be treated just like any other regular variables, i.e. functions can be

- passed as arguments to a another function
- returned from a function
- modified
- assigned to a variable

This concept is fundamental to understanding Python decorators.

[ ]:
```
# Assigning functions to variables
def increment_one(num):
  return num+1

add_one = increment_one # function increment_one assigned to variable add_one
print(add_one(7))
print(increment_one(7))
```

```
8
8
```

[ ]:
```
# Functions passed as arguments to another function
def increment_one(num):
  return num+1

def decrement_one(num):
  return num-1

def my_function(func, num): # here the first argument is a function
  return func(num)

print(my_function(increment_one, 5)) # increment_one function being passed as␣
 ↪argument
print(my_function(decrement_one, 5))
```

```
6
4
```

[ ]:
```
# Function being returned from a function
def double():
  def mult_by_2(num):
    return 2*num
  return mult_by_2 # function being returned

num2 = double() # num2 is a function => returned by double()
```

```
result = num2(3)
print(result)
```

```
6
```

Now, let's look at decorators.

[ ]:
```
# Decorator takes in a function, adds some functionality and returns it

# The decorate() function will decorate the function given to it as argument.
# So, it can be considered a decorator.
def decorate(func):
  def inner_func():
    print("function has been decorated")
    return func()
  return inner_func

# we will decorate this function
def regular_func():
  return "Just a regular function"
```

[ ]:
```
# regular_func() being invoked (called)
regular_func()
```

[ ]:
```
'Just a regular function'
```

[ ]:
```
# decorating the regular_func() function
decorated = decorate(regular_func)
print(decorated())
```

```
function has been decorated
Just a regular function
```

As you can see above, the decorator function added some functionality (printing the statement "function has been decorated") to the `regular_func` function.

In Python, we have an easier way of decorating a function by using the @ symbol and the name of the decorator just above the function to be decorated as shown below.

[ ]:
```
@decorate
def regular_func():
  return "Just a regular function"
```

[ ]:
```
regular_func()
```

```
function has been decorated
```

[ ]:
```
'Just a regular function'
```

Now, let's see another useful example of decorator.

Supppose, you have the following function.

```
[ ]: def div(x, y):
       return x/y
```

```
[ ]: div(2,0) # throws division by zero error
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-52-05e0cf01e79e> in <module>()
----> 1 div(2,0)

<ipython-input-50-4d1ad4367829> in div(x, y)
      1 def div(x, y):
----> 2   return x/y

ZeroDivisionError: division by zero
```

The above function call resulted in a division by zero error.

We can avoid such errors by using a decorator to the `div()` function.

```
[ ]: # decorator function
     def validate_div(func):
       def inner_func(x, y):
         print("Dividing {} by {} ...".format(x, y))
         if y == 0:
           print("Sorry! Can't divide by zero!")
           return
         return func(x, y)
       return inner_func

     @validate_div   # decorating the div function
     def div(x, y):
       return x/y
```

```
[ ]: div(2,0)
```

```
Dividing 2 by 0 …
Sorry! Can't divide by zero!
```

As you saw above, the division by zero error was averted by the use of a decorator function - `validate_div()`.

```
[ ]: div(3,2)
```

```
Dividing 3 by 2 …
```

[ ]: 1.5

8 Closure

When we define a function inside another function, the inner function is called a nested function. Nested functions are able to access variables defined in the enclosing (outer) function.

[ ]:
```
def greet(): # outer (enclosing) function
  msg = "Welcome to Python world!" # variable defined in the outer (enclosing)␣
 ↪function
  def print_msg(): # inner (nested) function
    print(msg) # trying to access a variable defined in the scope of outer␣
 ↪function
  print_msg() # calling the inner() function

# invoke outer() function
greet()
```

```
Welcome to Python world!
```

As you can see above, nested functions can access variables defined in the enclosing function's scope.

Now let's return the `print_msg` function above instead of calling it.

[ ]:
```
def greet(): # outer (enclosing) function
  msg = "Welcome to Python world!" # variable defined in the outer (enclosing)␣
 ↪function
  def print_msg(): # inner (nested) function
    print(msg) # trying to access a variable defined in the scope of outer␣
 ↪function
  return print_msg # returning the inner() function

# invoke outer() function
greet_func = greet() # greet() returns a function which is being assigned to␣
 ↪greet_func variable
greet_func()
```

```
Welcome to Python world!
```

This is a bit unusual. The `greet()` function was called and the returned function was assigned to the variable `greet_func`. This means that the function execution completed after the execution of that line, still the variable (`msg`) defined in the function was accessible when greet_func() is called.

Now, let's delete the `greet()` function.

[ ]:
```
del greet
```

[ ]:
```
greet_func()
```

```
Welcome to Python world!
```

Even if we deleted the original function (`greet()`), the `msg` variable defined in the original function was still availble.

This is called Closure.

As seen above, the properties of closure are:

- There must be a nested function (function inside a function)
- The nested function should refer to a variable defined in the scope of enclosing function
- The enclosing function should return the nested function