5. OOP in Python – Object Oriented Shortcuts

This chapter talks in detail about various built-in functions in Python, file I/O operations and overloading concepts.

Python Built-in Functions

The Python interpreter has a number of functions called built-in functions that are readily available for use. In its latest version, Python contains 68 built-in functions as listed in the table given below:

BUILT-IN FUNCTIONS				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

This section discusses some of the important functions in brief:

len() function

The len() function gets the length of strings, list or collections. It returns the length or number of items of an object, where object can be a string, list or a collection.

```
>>> len(['hello', 9 , 45.0, 24])
4
```

len() function internally works like **list.__len__()** or **tuple.__len__()**. Thus, note that len() works only on objects that has a **__len__()** method.

```
>>> set1
{1, 2, 3, 4}
>>> set1.__len__()
4
```

However, in practice, we prefer **len()** instead of the **__len__()** function because of the following reasons:

- It is more efficient. And it is not necessary that a particular method is written to refuse access to special methods such as __len__.
- It is easy to maintain.
- It supports backward compatibility.

Reversed(seq)

It returns the reverse iterator. seq must be an object which has __reversed__() method or supports the sequence protocol (the __len__() method and the __getitem__() method). It is generally used in **for** loops when we want to loop over items from back to front.

```
>>> normal_list = [2, 4, 5, 7, 9]
>>>
>>> class CustomSequence():
     def __len__(self):
           return 5
     def __getitem__(self,index):
           return "x{0}".format(index)
>>> class funkyback():
     def __reversed__(self):
           return 'backwards!'
>>> for seq in normal_list, CustomSequence(), funkyback():
           print('\n{}: '.format(seq.__class__.__name__), end="")
           for item in reversed(seq):
                 print(item, end=", ")
```

The for loop at the end prints the reversed list of a normal list, and instances of the two custom sequences. The output shows that **reversed()** works on all the three of them, but has a very different results when we define **__reversed__**.

Output

You can observe the following output when you execute the code given above:

```
list: 9, 7, 5, 4, 2,
CustomSequence: x4, x3, x2, x1, x0,
funkyback: b, a, c, k, w, a, r, d, s, !,
```

Enumerate

The **enumerate ()** method adds a counter to an iterable and returns the enumerate object.

The syntax of enumerate () is:

```
enumerate(iterable, start=0)
```

Here the second argument **start** is optional, and by default index starts with **zero (0)**.

```
>>> # Enumerate
>>> names = ['Rajesh', 'Rahul', 'Aarav', 'Sahil', 'Trevor']
>>> enumerate(names)
<enumerate object at 0x031D9F80>
>>> list(enumerate(names))
[(0, 'Rajesh'), (1, 'Rahul'), (2, 'Aarav'), (3, 'Sahil'), (4, 'Trevor')]
>>>
```

So **enumerate()** returns an iterator which yields a tuple that keeps count of the elements in the sequence passed. Since the return value is an iterator, directly accessing it is not much useful. A better approach for enumerate() is keeping count within a for loop.

```
>>> for i, n in enumerate(names):
      print('Names number: ' + str(i))
      print(n)
Names number: 0
Rajesh
Names number: 1
Rahul
Names number: 2
Aarav
Names number: 3
Sahil
Names number: 4
           Trevor
```

There are many other functions in the standard library, and here is another list of some more widely used functions:

- **hasattr**, **getattr**, **setattr** and **delattr**, which allows attributes of an object to be manipulated by their string names.

- **all** and **any**, which accept an iterable object and return **True** if all, or any, of the items evaluate to be true.

- **nzip**, which takes two or more sequences and returns a new sequence of tuples, where each tuple contains a single value from each sequence.

File I/O

The concept of files is associated with the term object-oriented programming. Python has wrapped the interface that operating systems provided in abstraction that allows us to work with file objects.

The **open()** built-in function is used to open a file and return a file object. It is the most commonly used function with two arguments:

```
open(filename, mode)
```

The open() function calls two argument, first is the filename and second is the mode. Here mode can be 'r' for read only mode, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending, any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The default mode is read only.

On windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb' and 'r+b'.

```
>>> text = 'This is the first line'
>>> file = open('datawork','w')
>>> file.write(text)
22
>>> file.close()
```

In some cases, we just want to append to the existing file rather then over-writing it, for that we could supply the value 'a' as a mode argument, to append to the end of the file, rather than completely overwriting existing file contents.

```
>>> f = open('datawork','a')
>>> text1 = ' This is second line'
>>> f.write(text1)
20
>>> f.close()
```

Once a file is opened for reading, we can call the read, readline, or readlines method to get the contents of the file. The read method returns the entire contents of the file as a str or bytes object, depending on whether the second argument is 'b'.

For readability, and to avoid reading a large file in one go, it is often better to use a for loop directly on a file object. For text files, it will read each line, one at a time, and we can process it inside the loop body. For binary files however it's better to read fixed-sized chunks of data using the read() method, passing a parameter for the maximum number of bytes to read.

```
>>> f = open('fileone','r+')
>>> f.readline()
'This is the first line. \n'
>>> f.readline()
'This is the second line. \n'
```

Writing to a file, through write method on file objects will writes a string (bytes for binary data) object to the file. The writelines method accepts a sequence of strings and write each of the iterated values to the file. The writelines method does not append a new line after each item in the sequence.

Finally the close() method should be called when we are finished reading or writing the file, to ensure any buffered writes are written to the disk, that the file has been properly cleaned up and that all resources tied with the file are released back to the operating system. It's a better approach to call the close() method but technically this will happen automatically when the script exists.

An alternative to method overloading

Method overloading refers to having multiple methods with the same name that accept different sets of arguments.

Given a single method or function, we can specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters.

```
class Human:
    def sayHello(self, name=None):
        if name is not None:
            print('Hello ' + name)
        else:
            print('Hello ')

#Create Instance
```

```
obj = Human()

#Call the method, else part will be executed
obj.sayHello()

#Call the method with a parameter, if part will be executed
obj.sayHello('Rahul')
```

Output

```
Hello
Hello Rahul
```

Default Arguments

Functions Are Objects Too

A callable object is an object can accept some arguments and possibly will return an object. A function is the simplest callable object in Python, but there are others too like classes or certain class instances.

Every function in a Python is an object. Objects can contain methods or functions but object is not necessary a function.

```
def my_func():
    print('My function was called')
my_func.description = 'A silly function'
def second_func():

    print('Second function was called')

second_func.description = 'One more sillier function'

def another_func(func):
    print("The description:", end=" ")
    print(func.description)
    print('The name: ', end=' ')
    print(func.__name__)
                print('The class:', end=' ')
```

```
    print(func.__class__)
    print("Now I'll call the function passed in")
    func()



another_func(my_func)
another_func(second_func)
```

In the above code, we are able to pass two different functions as argument into our third function, and get different output for each one:

```
The description: A silly function
The name:  my_func
The class: <class 'function'>
Now I'll call the function passed in
My function was called
The description: One more sillier function
The name:  second_func
The class: <class 'function'>
Now I'll call the function passed in
Second function was called
```

callable objects

Just as functions are objects that can have attributes set on them, it is possible to create an object that can be called as though it were a function.

In Python any object with a __call__() method can be called using function-call syntax.