OBJECT ORIENTED PROGRAMMING WITH PYTHON

ASHWIN PAJANKAR
SUSHANT GARG

Object Oriented Programming with Python

Learn essentials of OOP with Python 3

Ashwin Pajankar and Sushant Garg

This book is for sale at http://leanpub.com/PythonOOP

This version was published on 2017-07-19

Tweet This Book!

Please help Ashwin Pajankar and Sushant Garg by spreading the word about this book on Twitter!

The suggested hashtag for this book is #PythonOOP.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#PythonOOP

I dedicate this book to my dear wife, Kavitha.

Contents

Acknowledgments

I would like to thank my wife, Kavitha, and elder sister, Savita Gadhave for encouraging me to write this book and share my knowledge with rest of the world. I also thank Leanpub for providing a great platform to authors like me who wish to reach more people. Finally, I thank the co-author of this book, Sushant Garg who helps me in making this book better.

Preface

When I first saw a bunch of kids booting up a strange-looking PCB (Printed Circuit Board) with Linux, I literally jumped in excitement. When I asked them what it was, they said, "Raspberry Pi". This is how I was introduced to Raspberry Pi. The occasion was a Hackathon organized at my former workspace for the specially abled kids. When I saw those lovely and creative kids using a little computer, I immediately decided to explore it and after six months my first book on Raspberry Pi[1] got published.

I have been working since last 10 years with Python prior to the publication of my first book. My first book and all the subsequent books use Python 2 and Python 3 extensively for explaining all the concepts and their demonstrations. I have extensively used Python for Test Automation, Industrial Automation, IoT, Electronics, Robotics, Image Processing, Signal Processing, Computer Vision, Supercomputing, and Parallel Programming. All of my books written on these topics use Python. I always receive feedback on my books where people say that I should write a dedicated book on the basics of Python. However, I am not much into writing books on very basics of Programming. I still want all of my readers to be very comfortable with Python and the best way to learn Basic Python is to start coding in Python. This book covers the basics of OOP for such beginners.

Why am I writing this book?

As I said earlier, many of my readers find it quite difficult to understand the Python's Object Oriented coding Philosophy. So, I decided to write a book on the very basics of OOP with Python. As of now the book has ten chapters which cover all the essentials of the Object Oriented Programming with Python. Thanks to the revolutionary publishing model of Leanpub platform, I can keep on adding the chapters, topics, and code to the book and can also improve the existing text and code examples. I hope that this book helps all the Python enthusiasts to learn basics of OOP with Python. Please do let me know if you want me to add more essential topics to the book. Following the code examples in this book will help everyone to get comfortable with OOP in Python 3.

Who this book is for?

This book is not for someone who is very new to programming, coding, and computer science. If you are planning to learn how to code by reading this book then you will be disappointed. You must know the basics of coding as well as a bit of Python as I am not covering it in this book. You should know how to install Python 3. You should have a working setup of Python 3. Anything as basic as a Raspberry Pi would be good enough to try the coding examples in the book.

[1] https://www.packtpub.com/hardware-and-creative/raspberry-pi-computer-vision-programming

How is this book organized?

This book is and will always be an *In Progress* book (this is the basis of Leanpub). I will often update the book. Also, all the chapters in this book will be reused in the other books I am currently writing. following are the contents of the book:

Chapter 1 introduces readers to the philosophy Classes and Objects in Python.

Chapter 2 explores the members of the class. It covers member variables and methods.

Chapter 3 deals with modules and packages.

Chapter 4 introduces inheritance, related fundamentals, and access modifiers in Python.

Chapter 5 explores polymorphism in detail.

Chapter 6 explores exceptions and exception handling mechanism in Python

Chapter 7 explores built-in data structure like tuples, dictionaries, lists, and sets.

Chapter 8 deals with the basics of strings in Python.

Chapter 9 introduces readers to the Generators and Iterators in Python.

You will also find several asides, each one starting with an icon on the left. Let me explain them.

This a warning box. The text contained explains important aspects or gives important instructions. It is strongly recommended to read the text carefully and follow the instructions.

This is an information box. This may contain link to an external webpage relevant to the topic under discussion.

This is a tip box. It has suggestions to the reader that could simplify the learning process.

This is an exercise box. This will contain an exercise which will aid the readers to explore the topic further.

What this book is not?

This book is not for learning Python programming from scratch. So, if you are completely new to Python programming, you might find it a bit difficult to follow this book. It is also not a book for IoT (Internet of Things), Home Automation, or Robotics applications. I have written other books on these topics. Readers can go through my profile page on Leanpub[2] to check the other books I have written.

[2] https://leanpub.com/u/ashwinpajankar

About the Author

Ashwin Pajankar

When someone asks me about my career, I like to say I am someone who loves to work on Automation, Software Testing, Single Board Computers, and IoT. I am also interested and have worked in the areas of Computer Science like Parallel Programming, Image Processing, and Wireless Communications.

I was raised in a small town in India. Like thousands of other folks of my age, I finished my undergraduate studies in Computer Science and Engineering and joined workforce when I was in my early twenties. After couple of years of stint in a multinational IT services company, I decided to pursue further studies. I took admission to IIIT Hyderabad[3] and graduated with MTech in Computer Science & Engineering. I was lucky enough to able to fund my graduate studies with various grants, loans, and scholarships. After finishing graduate studies I joined workforce again. I am planning to work till my brain is working (which is an awful lot of time :)).

I have work experience in software development and testing till now. Recently, I find myself increasingly involved in Software Design, Architecture, and Innovation in IoT space. Till now, I have trained more than a thousand professionals for technical skills. More than tens of thousands of copies of my books published through Packt and Leanpub have been purchased till now.

I have work experience in various programming platforms like C, C++, Java, Python, and bash scripting. I have also worked with all the major databases like Microsoft SQL Server, Oracle, IBM DB2, Teradata, MySQL, and sqlite. In IoT domain, I have worked on Raspberry Pi, Arduino, Banana Pro, Intel Edison, and Grove sensors.

I have never had privilege of working on web or mobile programming. Yet, I have rudimentary understanding of XML, HTTP protocol, and web-services. Recently, I have started learning PHP by myself to be able to code for the web.

This is my thenth book and I am planning to write at least three to four books per year. All the books will be free and published on Leanpub. This is link to my LinkedIn[4] profile. Please feel free to contact me for providing your valuable feedback on my writing style.

Sushant Garg

[3] www.iiit.ac.in

[4] https://in.linkedin.com/in/ashwinpajankar

Errata and Suggestions

I am aware of the fact that there are several errors in the text. Unfortunately, English is not my mother language and this is one of the main reasons I prefer lean publishing: being an in-progress book I have all the time to check and correct them. I have decided that once this book reaches the end, I will look for a professional editor that helps me to fix all the errors in my English. However, feel free to contact me to let know every type of mistake. On the other end, I am totally open to suggestions and improvements about book content. So, please feel free to suggest any related topic you want me to add to the book or to let me know book parts that are not clear or well explained.

You can contact me on LinkedIn[5].

[5] https://in.linkedin.com/in/ashwinpajankar

How to Help the Author

As I mentioned earlier, English is not my mother language. I am really pleased when I hear any feedback from my dear readers on any aspect of the book.

If you really want to help me, you may consider to:

- give me your valuable feedback about unclear things or errors in the text and the examples
- write a small review about what you think of this book in the feedback section[6]
- use your favorite social network or blog to spread the word. The suggested hashtag for this book on Twitter is #PythonOOP[7]

Also, if you want to collaborate with me on any technical project, you can reach me on LinkedIn[8].

[6] https://leanpub.com/PythonOOP/feedback
[7] https://twitter.com/search?q=#PythonOOP
[8] https://in.linkedin.com/in/ashwinpajankar

1. Introduction

Object Oriented Programming (shortened as OOP) is a programming paradigm. In this chapter, we will get introduced to the following concepts, - Objects - Classes - What is OOP

Also we will understand following things experimentally, - Simple program with Object and class - Everything is an object in Python

1.1 Objects and Classes

An object in the real world is something we can touch, see, feel, and interact. For example, an apple is an object. A bike is an object. A house is an object. Similarly, in the world of software engineering, we have the objects too. An object is the collection of data and associated behaviours. Object-oriented Programming refers to the style of programming in which the software is designed and developed with objects and related concepts. Let's take an example of GUI style OS. Here, a window of a program is an object. A taskbar is an object. On screen apps are objects. See, how the real-life objects translate flawlessly into the Objects in Software engineering. That's why because the objects in OOP style are analogous to their real-life counterparts, the objects in the physical world. Now, consider that there are two objects, a bike and a car. Both are the objects. However, their types are different. A car belongs to the class of cars and a bike belongs to the class of bikes. In OOP, a class describe an object. In the other words, class is a data type. Objects are variables whose data type is a class.

Let's see an example of simple program with a class and an object in Python.

This is our first Object Oriented Program in Python 3.

prog01.py

```
1  class Point:
2      pass
3
4  p1 = Point()
5
6  print(p1)
```

The output will be as follows,

```
1  <__main__.Point object at 0x03E2EE90>
```

In the program above, we are just creating a simple class `Point` and creating the objecting `p1`. This is a simplest example of a class and an object.

In this chapter, we won't directly dive into very difficult concepts. We are just warming ourselves up. In the next chapter, we will add more details to this class.

Create empty classes in Python for other real-world object.

1.2 Everything is an Object in Python

Pytho is truly object oriented programming language. Almost everything is an object in Python. Everything can have associated documentation with it. Everything has methods and attributes. This is the sign of a true oject-oriented programming language. Let's see a way we can demonstrate this. Open the Python interpreter. Run the following code in that,

```
a = 5
print(type(a))
print(a.__doc__)
```

Run the example above and see the output. It will show that `a` is the object of class `int`. The `__doc__` is used to print its documentation.

We can have custom documentation for our class. It is called **docstrings**. We will see that in the next chapter.

When they say *Almost everything is an object in Python*, they really mean it. For example, `print()` is a built-in method. We can print its type by running `print(type(print))` statement and it returns `<class 'builtin_function_or_method'>`.

Print the documentation of `print()` method by running `print(print.__doc__)` statement.

1.3 Conclusion

In this chapter, we have got started with classes and objects in Python. This is very light chapter with small exercises. In the next chapter, we will study and demonstrate more concepts related to class using the `Point` class.

2. Getting Started with Classes

In the last chapter, we studied the basics of the classes. In this chapter, we will see how to create inline documentation in the Python code file with the docstrings. We will also study the members of a class in detail. We will use the same `Point` class for the demonstration.

2.1 Docstrings

We can provide inline documentation to the Python code with docstrings. Python docstrings stand for the *Python Documentation Strings*. It is the most convinient way to provide documentation for the developers for their Python code. It is the most recommended way. The following is a simple docstring example,

prog01.py

```
'This is the DocString example'


class Point:
    'This class represent the data-structure Point'
    pass

p1 = Point()

print(p1.__doc__)
print(help(p1))
```

The console (shell) output is as follows,

```
This class represent the data-structure Point
Help on Point in module __main__ object:

class Point(builtins.object)
 |  This class represent the data-structure Point
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

None
```

As we have discussed in the earlier chapter and this example, we can use built-in `__doc__` routine to print the docstrings if available. We can also use the `help()` function to display the docstrings if available.

2.2 Adding attributes to class

If you are familiar with Java or C++, we declare the class variable in the definition of the class. In Python, there is no such thing as the variable declaration. We can add the attributes to the class `Point` by adding a variable for each co-ordinate as follows,

prog01.py

```
1  'This is the DocString example'
2
3
4  class Point:
5      'This class represent the data-structure Point'
6      pass
7
8  p1 = Point()
9
10 p1.x = 1
11 p1.y = 2
12 p1.z = 2
13
14 print(p1.x, p1.y, p1.z)
```

Run the code above and see the result.

2.3 Adding a method to the class

We can add behaviours to the class by adding methods to the class. The following is an example of the class `Point` with custom methods `assign()` and `printPoint()`,

prog01.py

```
'This is the DocString example'


class Point:
    'This class represent the data-structure Point'

    def assign(self, x, y, z):
        'This assigns the value to the co-ordinates'
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        'This prints the values 0f co-ordinates'
        print(self.x, self.y, self.z)

p1 = Point()
p1.assign(0, 0, 0)
p1.printPoint()
```

Run the code above and check the output. As we can see the method declaration, it is not much different from a function declaration. The main difference is that, it is mandatory to have a self-reference parameter. In this case, it is named as `self` in both the methods. We can name it anything. However, I have never seen any other name for the variable. The other difference between a function and a method is that a method is always associated with a class. In the code above, the method `assign()` assigns values to the co-ordinates and `printPoint()` prints the values of co-ordinates of the point.

2.4 Initializer method

There is a special in Python method for initializing objects. It is `__init__`. We can use it to assign values to the object attribute variables. The following is an example of that,

prog01.py

```
'This is the DocString example'


class Point:
    'This class represent the data-structure Point'

    def __init__(self, x, y, z):
        'The initializer'
        self.assign(x, y, z)

    def assign(self, x, y, z):
        'This assigns the value to the co-ordinates'
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        'This prints the values 0f co-ordinates'
        print(self.x, self.y, self.z)

p1 = Point(0, 1, 2)
p1.printPoint()
```

Run the code above and check the result.

2.5 Multiline Docstrings in Python

Let's see an example of multiline docstrings in Python. Multiline docstrings are used to spread the docstrings over the multiple lines. The following is an example of a multiline docstring.

prog01.py

```
'This is the DocString example'

class Point:
    """This class represent the data-structure Point
    This is an example of the multiline docstring...
    """



    def __init__(self, x, y, z):
        '''The initializer ---
        This initializes the object with the passed arguments
        '''
        self.assign(x, y, z)

    def assign(self, x, y, z):
        'This assigns the value to the co-ordinates'
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        'This prints the values 0f co-ordinates'
        print(self.x, self.y, self.z)

p1 = Point(0, 1, 2)
p1.printPoint()
```

In the example above, add the code for displaying the docstrings.

2.6 Conclusion

In this chapter, we studied the member variables and the methods of a class. We also studied the initializer method. In the next chapter, we will study modules and packages in Python.

3. Modules and Packages

In this chapter, we will get started with the concept of modularity in Python. We will have a look at modules and packages in Python. We will also write programs to demonstrate these concepts.

3.1 Modules

In the earlier chapter, we saw examples of classes. We had a class `Point` in a python code file `prog01.py`. We can also have multiple classes in a single python code file. Consider following program,

prog01.py

```
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:

    def __init__(self):
        print("Just created an object for class02...")


o1 = Class01()
o2 = Class02()
```

In the example above, we are defining two classes `Class01` and `Class02` in a single python code file. Python code files are also known as python modules. Suppose you have a directory where you have created a bunch of python code files, we can just refer each code file as a module. Run the example above and see the output.

Also, the classes and functions in a python module can be accessed in other module. This is known as modularity in Python. We can group the related classes in a module and can import them when required in other modules. To see this in action, create another python module `prog02` in the same directory. Following is the code for it,

prog02.py

```
import prog01


o1 = prog01.Class01()
o2 = prog01.Class02()
```

When we execute the module `prog02`, we get the following output.

```
Just created an object for Class01...
Just created an object for class02...
Just created an object for Class01...
Just created an object for class02...
```

Can you guess why it is happening?? It is printing the statements twice because we're importing the entire module. So it's importing the part from `prog01` where we're creating the objects. Hence, the objects are created twice. To mitigate this, Python provides a neat way. Make the following changes to `prog01.py`,

prog01.py

```python
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:

    def __init__(self):
        print("Just created an object for class02...")


if __name__ == "__main__":
    o1 = Class01()
    o2 = Class02()
```

Writing our main code under `if __name__ == "__main__":` ensures that it is called only when the module is directly executed. When we import the entire module in other module, the code with main logic is not imported to the other module. Only functions and classes are imported to the other module. If you run both the modules one by one, you'll see that objects are created only once during each run.

But what if we want to import and run the main logic code of the module on demand in the module where it's imported? There is a more **Pythonic** way to organize our code to make this possible. Rewrite the `prog01` module as follows,

prog01.py

```
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:

    def __init__(self):
        print("Just created an object for class02...")


def main():
    o1 = Class01()
    o2 = Class02()

if __name__ == "__main__":
    main()
```

Also, make changes to `prog02` module as follows,

prog02.py

```
import prog01


prog01.main()
```

In the module `prog02`, we're directly importing the `main()` function of `prog01` module. Now, execute the module `prog02` to see the output. It will be same. However, the `prog01` module is now more organized than earlier. Let's make the same changes to `prog02` code as follows,

prog02.py

```
import prog01


def main():
    prog01.main()


if __name__ == "__main__":
    main()
```

Run the code again to see the output. Finally, to add more clarity, we will modify `prog01` as follows,

prog01.py

```
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:

    def __init__(self):
        print("Just created an object for class02...")


def main():
    o1 = Class01()
    o2 = Class02()

if __name__ == "__main__":
    print("Module prog01 is being run directly...")
    main()
else:
    print("Module prog01 has been imported in the current module...")
```

Run both the modules and see the output. What happened here?? We notice that the code before the `else` statement is executed if we directly run the module and the code after the `else` statement is executed when we import an entire module.

As an exercise make the same changes to the module `prog02`.

Let's move on to understand the other way to import the members of a module. In the earliest version of module `prog02` we had the following code,

prog02.py

```
1  import prog01
2
3
4  o1 = prog01.Class01()
5  o2 = prog01.Class02()
```

Here, we addressed the members of module `prog01` with `prog1.member` notation. This is because the statement `import prog01` imports all the members to `prog02` module.

There is another way to import the members of modules. Following is the example of that,

prog03.py

```
1  from prog01 import Class01, Class02
2
3
4  def main():
5      o1 = Class01()
6      o2 = Class02()
7
8
9  if __name__ == "__main__":
10     main()
```

With `from <module> import <member>` syntax, we do not have to use `module>.<member>` convention in the calling module. We can directly address the members of imported module as shown in the example above. Run the program above and see the output.

In the program above we're importing both the members of `prog01`. Modify the program above to import only a single member of the module. Also add the `else` part after `main()` function call.

3.2 Packages

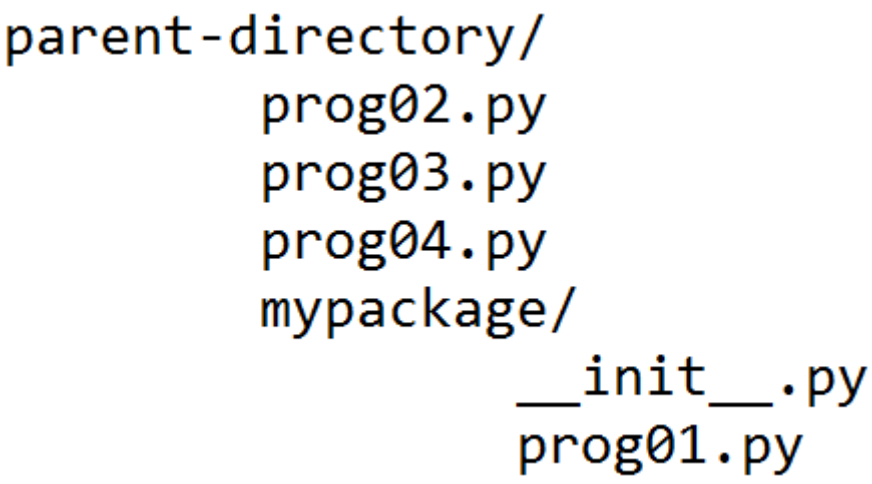
Till now, we have seen how to organize the functions and the classes in Python in separate modules. We can organize the modules into packages. Let's create a package of the code we have written till now. Create a subdirectory in the directory where you're saving all the python modules. Name the subdirectory as `mymodule`. Move the module `prog01` into `mymodule` directory. Create an empty file `__init__.py` and save it inside `mymodule`. And we just created our very own Python package! That's correct. Creating a package is easy. We just need to put the modules in a directory and create an empty `__init__.py` file in the the same directory. The name of the directory containing the modules is the package name.

Now, in the original parent directory, create a python module `prog04` as follows,

prog04.py

```
1  import mypackage.prog01
2
3
4  def main():
5      mypackage.prog01.main()
6
7
8  if __name__ == "__main__":
9      main()
```

Now the directory structure looks as follows,



Now run the `prog01` module and see the output. You must have noticed that we are using `<package>.<module>.<member>` notation to access a member of a package. We do not have to do that if we use `from <module> import <class>` syntax for importing.

Rewrite the program by importing the `prog01` module with `from prog01 import Class01, Class02` syntax and run the code.

We can even have a package within a package. To do that we just have to create another directory within the package directory and add the desired module file and an empty `__init__.py` file to that.

Create a subpackage in `mypackage`. Import and use it in a module in the parent directory.

3.3 Conclusion

In this chapter, we have learned how to introduce the modularity in the code with modules and packages. In the next chapter, we will get started with the concept of inheritance.

4. Inheritance

Modern programming languages have different mechanisms for code reuse. Procedures give an excellent way to reuse the code. Functions in Python is a great example of how code re-usability can be achieved by procedure.

Object oriented programming languages like Python also have another approach for the code reuse. It is know as **Inheritance**. Python has mechanism for *Class-based Inheritance*. A class in Python can inherit the attributes and the behaviours of the other class. The base class from which we derive other class(es) is also known as *Parent Class* or *SuperClass*. The class which inherits (or *derives* or *extends*, keep in mind that these words are used interchangeably in the world of object oriented programming) the attributes and the behaviours is known as *Child class* or *Sub Class*.

4.1 Basic Inheritance in Python

Technically all the Python classes are subclasses of a special built-in class known as `object`. This mechanism allows Python to treat objects alike. So, if you are creating any class in Python, it means that you're using inheritance implicitly. It's implicit because you do not have to make any special provision in the code for it.

You can also explicitly derive your custom class from the class `object` as follows,

prog01.py

```
1  class MyClass(object):
2      pass
```

The code above has the same meaning and functionality as the code follows,

prog01.py

```
1  class MyClass():
2      pass
```

Let's define a custom class and derive another class from it as follows,

prog02.py

```
1  class Person:
2      pass
3
4
5  class Employee:
6      pass
7
8
9  def main():
10     print(issubclass(Employee, Person))
11
12
13 if __name__ == "__main__":
14     main()
```

Run the code above. `issubclass()` returns true if the class mentioned in the first argument is a subclass of the class mentioned in the second argument. The above program prints `false`. OOPS! We have forgotten to derive `Employee` from `Person`. Let's fo that by modifying the code as follows,

prog02.py

```
class Person:
    pass


class Employee(Person):
    pass


def main():
    print(issubclass(Employee, Person))
    print(issubclass(Person, object))
    print(issubclass(Employee, object))


if __name__ == "__main__":
    main()
```

You must have noticed that we added two more statements in the `main()` function. Run the code. It must print `True` thrice.

4.2 Method Overriding

Let's add more functionality to the class `Person`. Modify the module `prog01.py` as follows,

prog02.py

```
class Person:

    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age

    def __str__(self):
        return self.firstname + " " + self.lastname + ", " + str(self.age)


class Employee(Person):
    pass


def main():
    x = Person("Ashwin", "Pajankar", 31)
    print(x)

if __name__ == "__main__":
    main()
```

Now, we want to define the behaviour of `Employee` class. As we have derived it from `Person`, we can reuse the methods in the class `Person` to define behaviour of `Employee` as follows,

prog02.py

```
class Person:

    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age

    def __str__(self):
        return self.firstname + " " + self.lastname + ", " + str(self.age)

```

```
11
12 class Employee(Person):
13     pass
14
15
16 def main():
17     x = Person("Ashwin", "Pajankar", 31)
18     print(x)
19
20     y = Employee("James", "Bond", 32)
21     print(y)
22
23 if __name__ == "__main__":
24     main()
```

When we run the code above, we get the following output,

```
1 Ashwin Pajankar, 31
2 James Bond, 32
```

As you can see, class `Employee` inherits the initializer and the `__str__()` method from class `Person`. However, we want to add another additional attribute to `Employee`, `empno`. We also have to change the behaviour of the `__str__()` method accordingly. The following code does it,

prog02.py

```
1  class Person:
2
3      def __init__(self, first, last, age):
4          self.firstname = first
5          self.lastname = last
6          self.age = age
7
8      def __str__(self):
9          return self.firstname + " " + self.lastname + ", " + str(self.age)
10
11
12 class Employee(Person):
13     def __init__(self, first, last, age, empno):
14         self.firstname = first
15         self.lastname = last
16         self.age = age
```

```
17          self.empno = empno
18
19      def __str__(self):
20          return self.firstname + " " + self.lastname + ", " + str(self.age) + ", \
21  " + str(self.empno)
22
23
24  def main():
25      x = Person("Ashwin", "Pajankar", 31)
26      print(x)
27
28      y = Employee("James", "Bond", 32, 1007)
29      print(y)
30
31  if __name__ == "__main__":
32      main()
```

In the code above, we're overriding the definitions of `__init__()` and `__str__()` methods from `Person` in `Employee` to suit our needs. Run the code and see the output. This way we can override any method of the base class in the subclass.

4.3 super()

In the last example, we have seen how to override base class's methods for in the subclass. There is other way of doing that. See the code example below,

prog02.py

```
class Person:

    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age

    def __str__(self):
        return self.firstname + " " + self.lastname + ", " + str(self.age)


class Employee(Person):
    def __init__(self, first, last, age, empno):
        super().__init__(first, last, age)
        self.empno = empno

    def __str__(self):
        return super().__str__() + ", " + str(self.empno)


def main():
    x = Person("Ashwin", "Pajankar", 31)
    print(x)

    y = Employee("James", "Bond", 32, 1007)
    print(y)

if __name__ == "__main__":
    main()
```

As you can see, we have used `super()`. It is a special method which returns the object as an instance of the base class. We can use `super()` in any method. As super returns the object as an instance of the base class, the code `super().<method()>` calls the respective method of the base class. `super()` can be used inside any method of a subclass to invoke an instance of parent class as an object. Also it does not have to be called in the first line of subclass method definition. It can be called at any place in the body. Thus, we can also write the `__init__()` method of `Employee` as follows,

```
class Employee(Person):
    def __init__(self, first, last, age, empno):
        self.empno = empno
        super().__init__(first, last, age)

    def __str__(self):
        return super().__str__() + ", " + str(self.empno)
```

It produces exactly the same output.

4.4 Conclusion

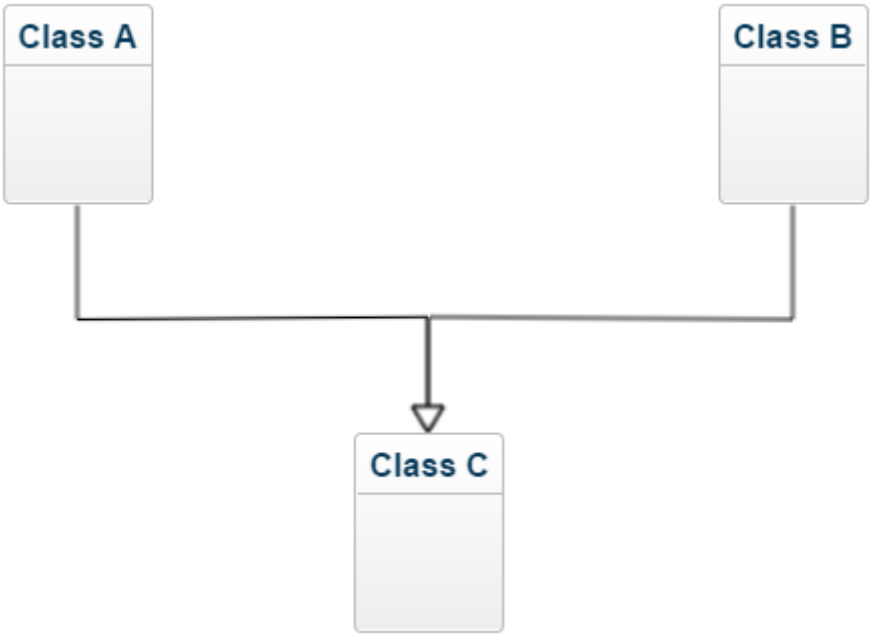
In this long and extensive chapter, we studied the inheritance and overriding. In the next chapter, we will study multiple inheritance, the diamond problem, and abstract classes and methods.

5. More Inheritance

In the last chapter we studied basic inheritance, overriding, and `super()`. In this chapter, we will study multiple inheritance, the diamond problem, access modifiers, and abstract classes and methods.

5.1 Multiple Inheritance

When a class is derived from more than one classes, then the mechanism is known as *Multiple Inheritance*. The following diagram depicts an example of it,



Let's try to write a simple code for the same,

prog01.py

```
1  class A:
2      pass
3
4
5  class B:
6      pass
7
8
9  class C(A, B):
10     pass
11
12
13 def main():
14     pass
```

```
15
16
17  if __name__ == "__main__":
18      main()
```

Multiple inheritance is used in the cases where we need the attributes and behaviours of more than one class to be derived in a single class.

5.2 Method Resolution order

Consider the following example,

prog02.py

```
class A:

    def m(self):
        print('This is m() from Class A.')


class B:

    def m(self):
        print('This is m() from Class B.')


class C(A, B):
    pass


def main():
    x = C()
    x.m()


if __name__ == "__main__":
    main()
```

When we run the code, the output is as follows,

```
This is m() from Class A.
```

This is because the order in which class C is derived from the parent classes. If we change the class C code as follows,

```
class C(B, A):
    pass
```

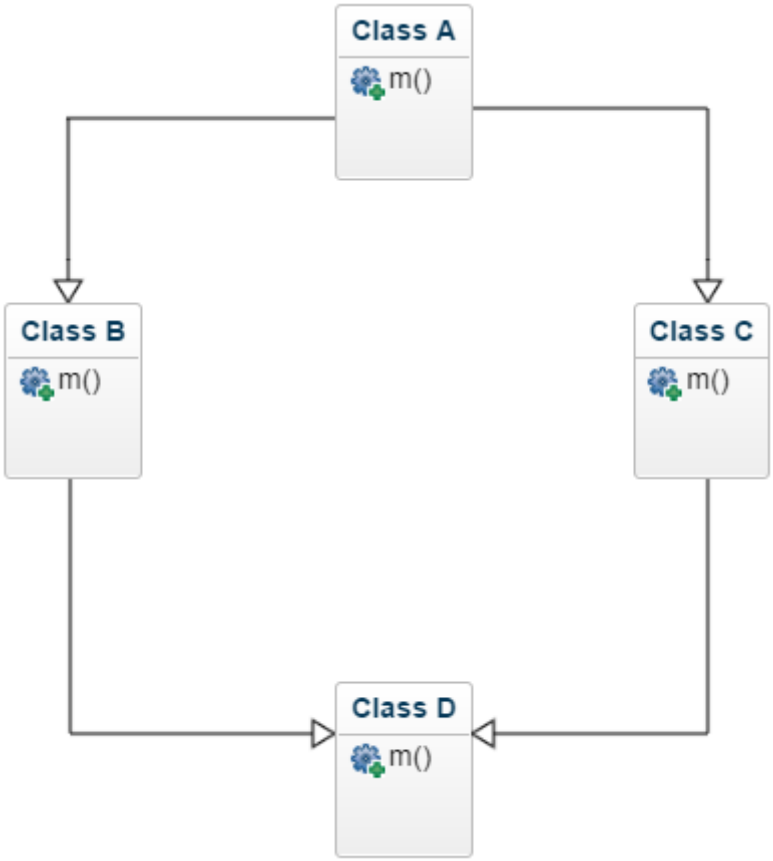
The output is,

```
This is m() from Class B.
```

The mechanism with which the derived methods of a subclass are resolved is known as *Method Resolution Order*.

5.3 The Diamond Problem

Consider the following example,



Let's implement the code as follows,

prog03.py

```
class A:
    def m(self):
        print("m of A called")


class B(A):
    def m(self):
        print("m of B called")


class C(A):
```

```
12      def m(self):
13          print("m of C called")
14
15
16  class D(B, C):
17      def m(self):
18          print("m of D called")
19          B.m(self)
20          C.m(self)
21          A.m(self)
22
23
24  def main():
25      x = D()
26      x.m()
27
28
29  if __name__ == "__main__":
30      main()
```

It works flawlessly and the MRO works!

```
1  m of D called
2  m of B called
3  m of C called
4  m of A called
```

However, if we want to invoke class A's method `m()` from method `m()` of the classes B and C as follows, then it is an issue,

prog04.py

```
1  class A:
2      def m(self):
3          print("m of A called")
4
5
6  class B(A):
7      def m(self):
8          print("m of B called")
9
10
```

```
class C(A):
    def m(self):
        print("m of C called")


class D(B, C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)
        A.m(self)


def main():
    x = D()
    x.m()


if __name__ == "__main__":
    main()
```

The output is as follows,

```
m of D called
m of B called
m of A called
m of C called
m of A called
m of A called
```

This is known as the diamond problem.

We want `m()` of the class A to be called only once. We need to change the code as follows,

prog04.py

```
class A:
    def m(self):
        print("m of A called")


class B(A):
    def m(self):
        print("m of B called")
        super().m()


class C(A):
    def m(self):
        print("m of C called")
        super().m()


class D(B, C):
    def m(self):
        print("m of D called")
        super().m()


def main():
    x = D()
    x.m()


if __name__ == "__main__":
    main()
```

The output is as follows,

```
m of D called
m of B called
m of C called
m of A called
```

This way if we use `super()` the diamond problem is resolved in Pythonic way.

5.4 Abstract class and method

A class whose object is not created is known as an abstract class. And a method which is declared without implementation is known as an abstract method. An abstract class may or may not have an abstract method. In Python we an explicitly make a class and a method abstract. Abstract class can only be subclassed. In many situations we need to have a class and methods only to be derived and overridden respectively. The following is a perfect real-life example,

prog05.py

```
from abc import ABC, abstractmethod


class Animal(ABC):

    @abstractmethod
    def move(self):
        pass


class Human(Animal):

    def move(self):
        print("I Walk.")


class Snake(Animal):

    def move(self):
        print("I Crawl.")


def main():
    a = Human()
    b = Snake()
    a.move()
    b.move()


if __name__ == "__main__":
    main()
```

In order to make a class abstract explicitly, we need to derive it from a built-in class `ABC`. If we want

to make a class methods abstract explicitly, then use `@abstractmethod` decorator with the method. Run the code above and see the output.

5.5 Access Modifiers in Python

In many programming languages like C++ and Java, there is the concept of **Access Control** for the members of classes. In Python, there is no strict measure for controlling the access of a member from outside the class.

So, if you do not want a method or a class variable to be accessed from outside, you can mention that in the docstrings of the class. The other way to let others know not to access a variable or a method made for internal use is to prefix the variable by underscore. The other person who is modifying the code or using it by importing is will understand that the variable or method is for the internal use only. However, (s)he still can access it from outside. The another way to strongly suggest others not to access the variable or method from the outside is to use the mechanism of *name mangling*. To do that, we have to prefix the method or variable with a double underscore. Then it can only be accessed with a special syntax which is demonstrated in the program below,

prog06.py

```
class A:

    def __init__(self):
        self.a = "Public"
        self._b = "Internal use"
        self.__c = "Name Mangling in action"


def main():
    x = A()
    print(x.a)
    print(x._b)
    print(x.__c)

if __name__ == "__main__":
    main()
```

The output displays the values for the first two attributes and for the third attribute, it displays an error which contains the message,

```
AttributeError: 'A' object has no attribute '__c'
```

In order to see the value of the attribute __c, make the following changes in the last `print()`,

```
    print(x._A__c)
```

The output is as follows,

```
Public
Internal use
Name Mangling in action
```

5.6 Conclusion

In this chapter, we studied and implemented the advanced concepts related to the inheritance in Python. In the next chapter, we will study Polymorphism.

6. Polymorphism

In the last chapter we studied advanced inheritance and the access modifiers in Python. In this chapter, we will study polymorphism. *Polymorphism* literally means the ability to assume various forms. In terms of the programming languages, it refers to the provision of a single interface to entities of different types. Most of the object oriented programming languages allow various degrees of polymorphism in the code. If you have worked with the programming languages like C++ or Java, then you must have a fair idea of this.

We studied *Overriding* in the last chapter. It is a form of Polymorphism. So, we already studied a type of Polymorphism in Python 3. In this chapter, we will first study **Method Overloading** and then **Operator Overloading** which fall under the concept of Polymorphism.

6.1 Method Overloading

When a method can be invoked with different number of arguments, it is known as method overloading. In the programming languages like C++, we can have multiple definitions of member functions of a class. However, Python does not allow it as we know that *Everything is an object in Python*. So, to make this possible, we use methods with default arguments. The example is as follows,

prog01.py

```
class A:

    def method01(self, i=None):
        if i is None:
            print("Sequence 01")
        else:
            print("Sequence 02")


def main():
    obj1 = A()
    obj1.method01()
    obj1.method01(5)


if __name__ == "__main__":
    main()
```

Run the code above and see the output.

Try to define a class with two or multiple methods with exactly same name but different number of parameters.

6.2 Operator overloading

Operators operate on operands and perform various operations. As we know that everything is an object in Python, the operands the operators operate on in Python are all the objects. The operations and the outcomes of operations of operators on built-in objects in Python are already well defined in Python. We can assign additional responsibility to the operators for the objects of user-defined classes. This concept is known as *Operator Overloading*. Following is a simple example for the addition operator,

prog02.py

```
class Point:

    def __init__(self, x, y, z):
        self.assign(x, y, z)

    def assign(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        print(self.x, self.y, self.z)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Point(x, y, z)

    def __str__(self):
        return("({0},{1},{2})".format(self.x, self.y, self.z))


def main():
    p1 = Point(1, 2, 3)
    p2 = Point(4, 5, 6)
    print(p1 + p2)


if __name__ == "__main__":
    main()
```

Run the code and check the output. When we do `p1 + p2` operation in the code, Python will call `p1.__add__(p2)` which in turn call `Point.__add__(p1,p2)`. Similarly, we can overload other operators as well. The special function that we need to implement for binary operations are tabulated below,

Operator	Special Function
+	object.**add**(self, other)
-	object.**sub**(self, other)
*	object.**mul**(self, other)
//	object.**floordiv**(self, other)
/	object.**truediv**(self, other)
%	object.**mod**(self, other)
**	object.**pow**(self, other[, modulo])
<<	object.**lshift**(self, other)
>>	object.**rshift**(self, other)
&	object.**and**(self, other)
^	object.**xor**(self, other)
|	object.**or**(self, other)

Following is the table for the extended operations,

Operator	Special Function
+=	object.**add**(self, other)
-=	object.**sub**(self, other)
*=	object.**mul**(self, other)
//=	object.**floordiv**(self, other)
/=	object.**truediv**(self, other)
%=	object.**mod**(self, other)
**=	object.**pow**(self, other[, modulo])
<<=	object.**lshift**(self, other)
>>=	object.**rshift**(self, other)
&=	object.**and**(self, other)
^=	object.**xor**(self, other)
|=	object.**or**(self, other)

Following is the table for the unary operators,

Operator	Special Function
+	object.**pos**(self)
-	object.**neg**(self)
abs()	object.**abs**(self)
~	object.**invert**(self)
complex()	object.**complex**(self)
int()	object.**int**(self)
long()	object.**long**(self)
float()	object.**float**(self)
oct()	object.**oct**(self)
hex()	object.**hex**(self)

Following is the table for the comparison operators,

Operator	Special Function
<	object.**lt**(self, other)
<=	object.**le**(self, other)
==	object.**eq**(self, other)
!=	object.**ne**(self, other)
>=	object.**ge**(self, other)
>	object.**gt**(self, other)

Try to implement the program for operator overloading for other operators.

6.3 Conclusion

In this chapter, we studied method overloading and operator overloading. In the next chapter, we will study exception handling.

7. Exceptions

When we write the programs in Python (or any programming language, for that matter), we usually do not get it correct the very first time. It's where the terms *Errors* and *Exceptions* come into the discussion. In this chapter, we will get started with Errors and Exceptions in Python.

7.1 Syntax Errors

When we make mistakes in the Syntax of the code, those are known as Syntax/Parsing Errors. Consider the result of execution of the statement `print("Hello)` in the interpreter mode,

```
>>> print("Hello)



SyntaxError: EOL while scanning string literal
```

In the `print()` statement above, we forgot to add `"` after the string `Hello`. This is a Syntax error. So, the Python interpreter highlighted it why throwing `SyntaxError`.

7.2 Exceptions

We say that the syntax/parsing errors are handled with `SyntaxError`. Even though a statement is correct in Syntax, it may encounter an error (which is not related to syntax) during execution. Errors detected during execution are called *exceptions*.

Consider the following statements and their executions in the interpreter,

```
>>> 1/0

Traceback (most recent call last):

  File "<pyshell#0>", line 1, in <module>

    1/0

ZeroDivisionError: division by zero

>>> '1' + 1

Traceback (most recent call last):

  File "<pyshell#1>", line 1, in <module>

    '1' + 1

TypeError: must be str, not int

>>> a = 8 + b

Traceback (most recent call last):

  File "<pyshell#2>", line 1, in <module>

    a = 8 + b

NameError: name 'b' is not defined
```

The last line of the output of the execution of each statement shows what is wrong in the statement. This demonstrates that an exception is implicitly raised whenever there is an error during execution.

The exceptions defined in the Python library are known as *Built-in exceptions*. This link[1] has a list

[1] https://docs.python.org/3/library/exceptions.html#bltin-exceptions

of all of them.

7.3 Handling exceptions

Now we know that the interpreter automatically raises an exception when an error is encountered during the runtime. Consider the following piece of code,

```
def main():

    a = 1/0

    print("DEBUG:  We are here...")

if __name__ == "__main__":

    main()
```

When we execute it, we can notice that the interpreter encounters an exception at the line `a = 1/0`. Once this exception is encountered, it does not run the statements which follow the statement where the exception is encountered. This is how exceptions are handled by default in Python.

However, Python has a better provision of handling the exceptions. We can enclose the code in a `try` block where we are likely to encounter an exception and the logic to handle it in `except` block as follows,

prog02.py

```
def main():

    try:

        a = 1/0

        print("DEBUG:  We are here...")

    except Exception:

        print("Exception Occured")

if __name__ == "__main__":

    main()
```

Run the code and it will invoke the `except` block when an exception is encountered rather than terminating abruptly. Note that the code after the statement where exception occurred is not executed.

In the `except` block, `Exception` is the base class for all the built-in exceptions in Python. In later parts of the chapter, we will also study user-defined exceptions which will be derived from the class `Exception`.

Let's modify the code a little bit and add some more code to `main()` function which is not the part of `try` or `except` blocks.

prog02.py

```
1  def main():
2
3      try:
4
5          a = 1/0
6
7          print("DEBUG:  This line will not be executed...")
8
9      except Exception:
10
11         print("Exception Occured")
12
13     print("This line will be executed...")
14
15 if __name__ == "__main__":
16
17     main()
```

When run, you will notice that the line outside the `try` and `except` block is executed in spite the exception is encountered.

7.3.1 Handling exceptions by types

When we run a program, we might encounter exceptions of multiple types. We can have the provision of handling various different types of exceptions. A simple example is as follows,

prog02.py

```
def main():

    try:

        a = 1/0

        print("DEBUG:  This line will not be executed...")

    except ZeroDivisionError as err:

        print("Error: {0}".format(err))

    except TypeError as err:

        print("Error: {0}".format(err))

    except Exception as err:

        print("Error: {0}".format(err))

    print("This line will be executed...")

if __name__ == "__main__":

    main()
```

In the code above, we have a `except` blocks for handling ZeroDivisionError and TypeError. If any other unexpected exception is encountered then it is handled by the last `except` block which is a generic exception handler block. Note that, the generic block should always be the last `except` block (as shown in the code above). If it is the very first `except` block then whenever any exception is encountered, the generic exception handling block will be executed always. This is because class `Exception` is base class for all the exceptions and it takes precedence.

Rewrite the above program with `except Exception` as the first exception handling block.

As we know everything is an object in python. `SyntaxError` is also a type of exception. This is special type of exception which cannot be handled in the `except` block.

7.4 `else` block

We can add an `else` block to the code after the `except` block. If any error is not encountered in the `try` block then `else` block is executed. The following program demonstrates this,

prog02.py

```
def main():

    try:

        a = 1/1

    except ZeroDivisionError as err:

        print("Error: {0}".format(err))

    except TypeError as err:

        print("Error: {0}".format(err))

    except Exception as err:

        print("Error: {0}".format(err))

    else:

        print("This line will be executed...")

if __name__ == "__main__":

    main()
```

7.5 Raising an exception

We know that an exception is automatically raised when there is a runtime error. We can raise an exception explicitly and deliberately using `raise` statement. The following code demonstrates that,

prog03.py

```
def main():

    try:

        raise Exception("Exception has been raised!")

    except Exception as err:

        print("Error: {0}".format(err))

    else:

        print("This line will be executed...")

if __name__ == "__main__":

    main()
```

The output is as follows,

```
Error: Exception has been raised!
```

7.6 `finally` clause

`finally` is a clause for the `try` statement, which is always executed *On the way out*. It means that it is essentially the *Clean Up Clause*. It is always executed in the end of the try clause irrespective whether an exception occurred in the `try` statement. If any exception is not handled in the `except` block, then it is re raised in `finally`. Following is the example of the same,

prog04.py

```
def divide(x, y):

    try:

        result = x / y

    except ZeroDivisionError:

        print("division by zero!")

    else:

        print("result is", result)

    finally:

        print("executing finally clause")

def main():

    divide(2, 1)

    divide("2", "1")

if __name__ == "__main__":

    main()
```

Following is the output,

```
result is 2.0

executing finally clause

executing finally clause

Traceback (most recent call last):

  File "C:/Users/Ashwin/Dropbox/PythonOOP/manuscript/Code/Chapter07/prog04.py", \
line 17, in

    main()

  File "C:/Users/Ashwin/Dropbox/PythonOOP/manuscript/Code/Chapter07/prog04.py", \
line 13, in main

    divide("2", "1")

  File "C:/Users/Ashwin/Dropbox/PythonOOP/manuscript/Code/Chapter07/prog04.py", \
line 3, in divide

    result = x / y

TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As we can see, in the code above there is no provision for handling `TypeError` exception. So the `finally` clause raises it once again.

7.7 User Defined Exceptions

We can define our own exception derived from the `Exception` class. Usually in a module, we define a single base class derived from `Exception` and derive all the other exception classed from that. The example of this is shown below,

prog05.py

```
class Error(Exception):

   pass

class ValueTooSmallError(Error):

   pass

class ValueTooLargeError(Error):

   pass

def main():

    number = 10

    try:

        i_num = int(input("Enter a number: "))

        if i_num < number:

            raise ValueTooSmallError

        elif i_num > number:

            raise ValueTooLargeError

        else:

            print("Perfect!")

    except ValueTooSmallError:

        print("This value is too small!")
```

```
36
37      except ValueTooLargeError:
38
39          print("This value is too large!")
40
41  if __name__ == "__main__":
42
43      main()
```

Please run the program above and see the output. In the program above, the class `Error` is inherited from the built-in class `Exception`. The we inherit more sub-classes from the class `Error`.

7.8 Conclusion

In this chapter, we studied the Exception in Python 3.

7.9 Summary

the entire book is dedicated to get the novice Python 3 programmers to get started with the *Object Oriented Programming in Python 3*. Please do let me know your feedback.