

# BCS-094 Programming using Python



**OBJECT ORIENTED PROGRAMMING  
IN PYTHON, TESTING & DEBUGGING**

**2**

# MCS-094

## PROGRAMMING USING PYTHON

IGNOU is one of the participants for the development of courses for this Programme.

### Copyright

This course has been developed as part of the collaborative advanced ICT course development project of the Commonwealth of Learning (COL). COL is an intergovernmental organisation created by Commonwealth Heads of Government to promote the development and sharing of open learning and distance education knowledge, resources and technologies.

The Open University of Sri Lanka (OUSL) is the premier Open and Distance learning institution in the country where students can pursue their studies through Open and Distance Learning (ODL) methodologies. Degrees awarded by OUSL are treated as equivalent to the degrees awarded by other national universities in Sri Lanka by the University Grants Commission of Sri Lanka.



The Python Software Foundation is owner of Python. Copyright © 2001-2019 Python Software Foundation; All Rights Reserved. Reference: <https://docs.python.org/>

Kivy is an open source Python library for development of applications. Reference: <https://kivy.org/>

This Programming with Python is a teaching learning resource developed by the Open University of Sri Lanka using various sources duly acknowledged.

© Open University of Sri Lanka and Commonwealth of Learning, 2018. Except where otherwise noted, The Programming with Python course material is made available under Creative Commons AttributionShareAlike 4.0 International (CC BY-SA 4.0) License: <https://creativecommons.org/licenses/by-sa/4.0/legalcode>.

For the avoidance of doubt, by applying this license the Commonwealth of Learning does not waive any privileges or immunities from claims that it may be entitled to assert, nor does the Commonwealth of Learning submit itself to the jurisdiction, courts, legal processes or laws of any jurisdiction. The ideas and opinions expressed in this publication are those of the author/s; they are not necessarily those of Commonwealth of Learning and do not commit the organisation.



The Open University of Sri Lanka  
P. O. Box 10250, Nawala, Nugegoda  
Sri Lanka  
Phone: +94 112881481  
Fax: +94112821285  
Email: [hdelect@ou.ac.lk](mailto:hdelect@ou.ac.lk)  
Website: [www.ou.ac.lk](http://www.ou.ac.lk)



Commonwealth of Learning  
4710 Kingsway, Suite 2500, Burnaby  
V5H 4M2, British Columbia, Canada  
Phone: +1 604 775 8200  
Fax: +1 604 775 8210  
Email: [info@col.org](mailto:info@col.org)  
Website: [www.col.org](http://www.col.org)

---

## ACKNOWLEDGEMENT BY AUTHORS

---

The Open University of Sri Lanka (OUSL), Department of Electrical and Computer Engineering (ECE) wishes to thank those below for their contribution to this course material and accompanying videos and screencasts:

---

## AUTHORS

---

Units 6 and 7      BK Werapitiya (Lecturer, Dept. of ECE, OUSL)

Units 8,9 and 10      S Rajasingham (Lecturer, Dept. of ECE, OUSL)

---

<b>Content Editor</b> DN Koggalahewa (Senior Lecturer, Sri Lanka Institute of IT)	<b>Language Editor:</b> KARD Gunaratne (Lecturer, Dept. of ECE, OUSL)	<b>Reviewer:</b> KARD Gunaratne (Lecturer, Dept. of ECE, OUSL)
--	--	---

---

**Video presenters:**

S Rajasingham (Lecturer, Dept. of ECE, OUSL) M Abeykoon (Software Engineer, Duo Software) SN Muhandiram (Associate Software Engineer, VizuaMatix )

**Screencasters:**

MHMND Herath (Lecturer, Dept. of ECE, OUSL) M Abeykoon (Software Engineer, Duo Software)

---

## COURSE COORDINATOR

---

Dr. Sudhansh Sharma,  
Assistant Professor  
School of Computers and Information Sciences(SOCIS)  
Indira Gandhi National Open University(IGNOU),  
New Delhi-110068

In this Block the Introduction to the Block along with the Check Your Progress in all the units, and Answers to Check Your Progress Questions are written and added in 2021, By

Dr. Sudhansh Sharma, Assistant Professor  
SOCIS, IGNOU, New Delhi-110068

---

## PRINT PRODUCTION

---

Mr. Tilak Raj  
Assistant Registrar (Publication)  
MPDD, IGNOU, New Delhi

Mr. Yashpal  
Assistant Registrar (Publication)  
MPDD, IGNOU, New Delhi

April, 2021

© Indira Gandhi National Open University, 2021

ISBN-

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Indira Gandhi National Open University.

Further information, about the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110 068.

Printed and published on behalf of the Indira Gandhi National Open University by Registrar, MPDD, IGNOU, New Delhi

Laser Composed by Tessa Media & Computers, C-206, Shaheen Bagh, Jamia Nagar, New Delhi

# BLOCK

# 2

## OBJECT ORIENTED PROGRAMMING IN PYTHON, TESTING & DEBUGGING

---

### UNIT 6

Object Orientation as a Programming Paradigm	5
--	---

---

### UNIT 7

Object Oriented Concepts	15
--------------------------	----

---

### UNIT 8

Error and Exception Handling	23
------------------------------	----

---

### UNIT 9

Testing	32
---------	----

---

### UNIT 10

Debugging and Profiling	40
-------------------------	----

---

---

## BLOCK 2 INTRODUCTION

---

There are three widely used programming paradigms named as procedural programming, functional programming, and object-oriented programming. Python supports both procedural and Object Oriented Programming (OOP).

OOP is a programming paradigm that uses objects and their interactions to design applications and computer programs. Python is primarily designed as an object-oriented programming language. This block will focus on Object orientation as a programming paradigm with creating and using classes, objects, attributes and methods. So, learning outcomes of this block will help you to write programs using Python's object-oriented programming support.

Understanding the concepts delivered in this block helps you to understand and manage different types of errors that can occur in a program, i.e. how to identify them and how to correct them. The coverage of content relates to three types of errors, and they are: Syntax errors, Runtime errors and Semantic errors. Subsequently there will be a discussion on the definition of exception and the importance of handling exceptions by using appropriate techniques.

There after there is a coverage of the different testing methods that can be used with any programming language as well as Python. The two main methods we discuss here are black box testing and white box testing. The content coverage gives an overview of the test framework called unit-test for Python. Along with it, we examine few examples on how to write a simple test case.

Finally, there is a session that introduces you to the important modules that handle the basic debugger functions such as analyzing stack frames and set breakpoints etc. It also gives a brief description of what profilers are and how profiling is performed on a Python applications.

This Block comprises of 5 Units, their content coverage relates to following:

Unit-6 Introduces object orientation as a programming paradigm

Unit-7 Covers the Object Oriented Concepts

Unit-8 Provides understanding of Errors and Exception Handling

Unit-9 Introduces the concept Testing

Unit-10 Introduces the concept of Debugging and Profiling

---

## UNIT 6 OBJECT ORIENTATION AS A PROGRAMMING PARADIGM

---

### Structure

- 6.1 Introduction
- 6.2 Overview
- 6.3 Terminologies
- 6.4 Overview of object-oriented programming (OOP)
- 6.5 Basic concepts of objects and classes
  - 6.5.1 User-defined types
  - 6.5.2 Defining Classes in Python
  - 6.5.3 Attributes
- 6.6 Methods in Python
  - Video 8: Object Oriented Basics
  - 6.6.1 Defining methods
  - 6.6.2 Constructor or `__init__` method
    - Video 9: Classes and Object Declaration
  - 6.6.3 The `__str__` method
- 6.7 Operator overloading
- 6.8 Summary
- 6.9 References and Further Reading
- 6.10 Answer to check your progress

---

### 6.1 INTRODUCTION

---

Python is primarily designed as an object-oriented programming language. This unit will focus on Object orientation as a programming paradigm with creating and using classes, objects, attributes and methods. So, learning this unit will help you to write programs using Python's object-oriented programming support.

---

### 6.2 OVERVIEW

---

Upon completion of this unit, you will be able to:

- *Identify* the object oriented nature of Python.
- *Define* class, object and method in Python.
- *Differentiate* functions and methods in Python.

---

## 6.3 TERMINOLOGIES

---

<b>Object:</b>	A real world entity or a concept
<b>class:</b>	A user-defined type.
<b>field:</b>	A variable that belong to a class.
<b>Method</b>	Functions that belong to a class
<b>attribute:</b>	A field or a method associated with a class

---

## 6.4 OVERVIEW OF OBJECT-ORIENTED PROGRAMMING (OOP)

---

In the "real" world, objects are the entities of which the world is encompassed. Everything that happens in the world is considered to be the interactions between the objects in the world. Real-world objects share two characteristics, attributes and behaviours.

For an example if we take the real world object "Dog", Dogs have attributes (name, colour, breed, hungry) and behaviours (barking, fetching, wagging tail). Bicycles also have attributes (current gear, current pedal cadence, current speed) and behaviours (changing gear, changing pedal cadence, applying brakes). Dogs and Bicycles are physical objects. Similarly, we can consider conceptual objects such as a course that a student will be registered for. Courses have state (course code, course title, number of credits, the department offering the course) and behaviour (offer course, exempt course, drop course).

In OOP, programs are made up of objects and functions that are required to work with those objects. So, programs contain object definitions and function definitions. An object definition directly corresponds to some entity or a concept in the real world, whereas the functions correspond to the methods those entities interact.

In general, a **class** is a blueprint where **objects** are **instances** of a particular class. For example, the vehicle class may include objects like bicycle or a car.

Object oriented programming concept is a model organised around "class/object" concept rather than on functions. Objects are modeled after real-world entities.

---

## 6.5 BASIC CONCEPTS OF OBJECTS AND CLASSES

---

In Python, a class is defined as a type. A class creates a new type where objects are instances of that class. Even variables that store integers are treated as instances of class int. Using type function, we can find out the type of any object.

Functions are used to reuse the code in different places in a program. In Python there are inbuilt functions like `print()` and also enables you to create your own functions.

### 6.5.1 User-defined types

In Python new types can be defined and manipulated from Python code in a similar way that strings are defined manipulated. To define a new type, an extension module must be defined and supported by Python. It is easy as all extension types follow a general pattern. During the execution of a Python program, all Python objects are treated as variables of type `PyObject*`, which contain a pointer to the object's "type object". Here, the type object determines which (C) functions get called when an attribute of an object is accessed. These C functions are called "type methods".

As an example, let us consider that we need to write a program which has a point in two-dimensional space. How do we represent a point in Python?

In mathematics, points are represented within two parentheses like this: (2,3) which means x coordinate is 2 and y coordinate is 3.

There are several ways we can represent a point in Python:

- store the coordinates in two separate variables
- store the coordinates as elements of a list or a tuple.
- create a new type called `Point` represent points as objects.

### 6.5.2 Defining Classes in Python

In general terms, 'Class' is a blueprint to create objects. In Python, class is a special data type. A Class contains some data items and methods that operate on those data items. Objects created as instances of such a defined class will be having all those data items and methods.

Class statement followed by the class name creates a new class. An example of a simple Python class is given below.

```
class Bike:
    '''This the name space for a new class'''
    pass
```

'class' keyword defines the template that specify what data and functions will be included in any object of type `Bike`.

With the class named '`Bike`', we can make an object called '`myBike`'.

```
>>> myBike = Bike()
>>> print(myBike)
output
< main_.Bike object at 0x033F7970>
```



Since Bike is defined at the top level, its name appears as ” \_\_ main \_\_.Bike”.

An object is an instance of a particular class. To create an object, we call the class using the class name and pass the arguments its `__init__` method accepts. (passing arguments and the `__init__` method will be discussed later).

Creating a new object is called instantiation of a class. When an object is printed, Python displays to which class this object belongs to and the memory address it is stored (Memory address is given in hexadecimal as evident from prefix 0x).

### 6.5.3 Attributes

The attributes are data members such as class variables or instance variables and methods. There are two types of attributes.

- Data attributes - Owned by an instance of a class
- Class attributes - Owned by the class as a whole these attributes can share with all the instances of a class.

```
class Bike:  
    gear = 1  
    speed = 0
```

Attributes are accessed via dot notation. Similarly, values also are assigned to an instance also using dot notation:

```
>>> myBike.gear = 2  
>>> myBike.speed = 45
```

Class methods have only one specific difference from ordinary functions. That is, they must have prefix that has to be added to the beginning of the parameter list which does not need a value when the method is called. This parameter is called **self**.

Here is an example to see how Python gives a value for ‘self’.

When you call a method of this object as `myBike.method(arg1, arg2)`, this is automatically converted by Python into `myBike.method(blank, arg1, arg2)`.

That is, even if you have a method without arguments, then you still have to have one argument which is ‘self’.

To read the value of an attribute we have to use the same syntax as we assigned values:

```
>>> myGear = myBike.gear  
>>> print(myGear)  
output  
1
```

Here the variable name `myGear` can also be `gear` and that will not get mixed up with the attribute `gear`.

```
>>> print(myBike.gear, myBike.speed)
(1, 0)
```

Dot notation can be used as part of any expression. For example,  
An object can be passed as an argument to a function. For example,

```
def print_bike(k):
    print(k.gear, k.speed)
```

`print_bike(k)` function takes a `myBike` as an argument and invokes `print` function,

```
>>> print_bike(myBike)
output
(1, 0)
```

Inside the function, `k` will be substituted by the argument `myBike`. If `k` is changed inside the function, the values of `myBike` changes.

```
>>>def set_gear(newValue):
    gear = newValue
    print("Gear is at ", %d, gear)
>>>set_gear(4)

Output
Gear is at 4
```

---

## 6.6 METHODS IN PYTHON

---

A method, sometimes called a function, which defines a behaviour, is created by `def` statement. Each class has several methods that are associated with it. You may recall we used methods when manipulating strings. We need methods to work with user-defined types also.

Methods are similar to functions except for two differences. Since Methods are needed to work with a class, they are defined inside a class definition. Secondly, invoking a method is different from that of invoking a function.

### Video 8: Object Oriented Basics

You may watch this video first and then attempt Activity 6.1.

URL: <https://youtu.be/2cB528Ww5F8>



### Activity 6.1:

Write a `Student` class which contains `studentID`, `lastName`, `courseID`. Input values to one object of type `student` and print the values.

### 6.6.1 Defining methods

Let us define a class named Student and create a function named `display_student()`.

```
class Student:
    regNo = ' '
    name = ' '
    fee = 0

def display_student(student):
    print(student.regNo, student.name, student.fee)
```

To call this function, need to pass student object as an argument.

```
>>> s1 = student()
>>> s1.regNo = AL205
>>> s1.name = 'Jayawan'
>>> s1.fee = 5000
>>> display_student(s1)
Output
AL205, Jayawan, 5000
```

To make `display_student` a method in this class, the function definition has to be moved inside the class definition. It is important proper indentation is done so that it is clear that `display_student` is a method in class Student.

```
class Student:
    def display_student (student):
        print(student.regNo, student.name, student.fee)
```

Same method can be written with **self** as follows also,

```
class Student:
    def display_student(self):
        print(self.regNo, self.name, self.fee)
```

Now you know few basic details about how to define a class, how to initialize attributes, how to print attributes and how to define methods and invoke them. There are two important methods in Python called *init* and *str*. Now, let us see why they are important.

### 6.6.2 Constructor or `__init__` method

The `init` method is the constructor of a class. It is invoked when an object is created and also called the 'initialization' of the object.

As in any other language where variables are initialized at the beginning, it is a good practice to write `__init__` method first when a new class is written to instantiate objects.

An example for an init method would look like the following,

```
# inside class Student that include student_display
method:
def __init__(self, regNo = '0', name = ' ', fee = 0):
    self.regNo = regNo
    self.name = name
    self.fee = fee
```

Parameters of `__init__` method usually have the same names as the attributes. The following expression, means, that the value of the parameter `regNo` is assigned as the value of an attribute of `self`.

```
self.regNo = regNo
```

Parameters are not a must. If `Student` is called with no arguments, you will get the default values.

```
>>> s2 = Student()
>>> Student.display_student(s2)
0,0
```

Since the program is getting bigger now, it is much easier to type it in an editor and run the program.

The complete program would look like following:

```
Class Student (object):
    def __init__(self, regNo = '0', name = ' ', fee = 0):
        self.regNo = regNo
        self.name = name
        self.fee = fee
    def display_student(self):
        print(self.regNo, self.name, self.fee)
```

Please note the indentation. Proper indentation is a must in Python.

### Video 9: Classes and Object Declaration

You may watch this video first and then attempt Activity 6.2.

URL: <https://youtu.be/NYC34dE-XY8>



### Activity 6.2:

For the bicycle class, write an init method that initialize gear and speed to 1 and 0 respectively.

### 6.6.3 The `__str__` method

`__str__` is a special method in Python that would print a string which would describe an object.

Following is an example for str method for Student class.

```
class Student:
# inside class Student that include student_display
method and __init__ method:
def __str__(self):
    return('Student Reg No %s, Name %s, Fee %d'
%(self.regNo,self.name, self.fee))
>>>s1 = Student(13,Mala,3500)
>>>print(s1)
output
Student Reg No 13, Name Mala, Fee 3500
```

That is, when we print an object, str method is invoked. This method is very useful for debugging and print log files.

Next, we will see behaviours of operators in Python with add method and discuss operator overloading.

---

## 6.7 OPERATOR OVERLOADING

---

When specially named methods are defined in a class, Python will automatically call them when instances of the class appear in the expressions. That is, by defining special methods, we can specify the behaviour of operators on programmer-defined types.

For example, if you define a method named `__add__` for the Student class, you can use the `+` operator on Student objects.

Here is an example for operator overloading:

```
class Student():
# inside class Student that include student_display,
__init__, __str__ methods write this method and run:
    def __add__(self, fee_increment): total =
        self.fee + fee_increment return(total)

>>> start_fee = Student('13', 'Mala', 5000)
>>> print(start_fee+300) 5300
```

As you can see, when you apply the `+` operator to Student objects, Python invokes `__add__`. To print the result, Python invokes `__str__`.

Thus, for each and every operator in Python, a corresponding special method exists similar to `__add__`. Here we are changing the behavior of an operator to work with programmer-defined types.

---

**Activity 6.3:**

Write a `__str__` method for the bicycle class and print it.

---

**Check Your Progress**

- Q-1 What are Constructors? Discuss the implementation of constructors with suitable example code in Python.
- Q-2 Discuss the concept of Operator Overloading with suitable example
- Q-3 Define a class employee to display the records of employee (make suitable assumptions for record entries for employees)

---

## 6.8 SUMMARY

---

In this unit we had a brief overview of Object Orientation as a programming paradigm. We studied the syntax of the class statement, and how to define objects, classes and methods in Python. We also discussed method definitions, `__init__`, `__str__` methods and how to define operator overloading methods.

---

## 6.9 FURTHER READING

---

- 1) Python for Everybody, Object oriented programming, accessed web (2018), <https://www.py4e.com/html3/14-objects> Copyright CC-BY 3.0 - Charles R. Severance
- 2) Object-Oriented Programming in Python 1.0, <http://python-textbok.readthedocs.io/en/1.0/Classes.html>
- 3) Allen B. Downey (2012). Think Python. <http://greenteapress.com/wp/think-python/>

---

## 6.10 ANSWER TO CHECK YOUR PROGRESS

---

Ans-1      Refer section 6.6.2

Ans-2      Refer section 6.7

Ans-3      Refer section 6.6



---

## UNIT 7 OBJECT ORIENTED CONCEPTS

---

### Structure

- 7.1 Introduction
- 7.2 Objectives
- 7.3 Terminologies
- 7.4 Overview of Object Oriented concepts
- 7.5 Inheritance
- 7.6 Multiple inheritance
- 7.7 Data Encapsulation
- 7.8 Polymorphism
  - 7.8.1 Overriding and Overloading
- Video 10: Multiple Inheritance and Encapsulation
- 7.9 Summary
- 7.10 References and Further Reading
- 7.11 Answer to check your progress

---

### 7.1 INTRODUCTION

---

There are three widely used programming paradigms named as procedural programming, functional programming, and object-oriented programming. Python supports both procedural and Object Oriented Programming (OOP).

OOP is a programming paradigm that uses objects and their interactions to design applications and computer programs.

---

### 7.2 OBJECTIVES

---

Upon completion of this unit, you will be able to:

- *Identify* object oriented concepts such as inheritance and polymorphism
- *Apply* object oriented concepts such as inheritance and polymorphism to solve real world problems

---

### 7.3 TERMINOLOGIES

---

**Inheritance:** A way of defining a new class that is a subtype of the previously defined class which is called a super class. Subtype has all attributes of the super class and its behaviours.

**IS-A relationship:** The relationship between a child class and its parent class.



---

## 7.4 OVERVIEW OF OBJECT ORIENTED CONCEPTS

---

There are some more basic programming concepts in OOP such as Inheritance, Encapsulation, and Abstraction. In this unit we will discuss object oriented concepts in Python with Inheritance, Encapsulation and Polymorphism.

---

## 7.5 INHERITANCE

---

Concept of inheritance enables to represent objects according to their similarities and differences. All similar attributes and functions can be defined in a base class which is called the super class. Inheritance can also be seen as a way of arranging objects in a hierarchy from the most general to the most specific.

An object which *inherits* from another object is considered to be a *subtype* of that object. When we can describe the relationship between two objects using the phrase *is-a*, that relationship is inheritance. We also say that a class is a *subclass* or a *child class* of a class from which it inherits, or that the previous class is its *superclass* or *parent class*.

We can refer the most generic class at the base of a hierarchy as a *base class*. We can put all the functionality that the objects have in common in a base class, and then define one or more subclasses with their own custom functionality.

Here is a simple example of inheritance:

```
class Person:
    def __init__(self, name, surname, idno):
        self.name = name
        self.surname = surname
        self.idno = idno

class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)
    def enrol(self, course):
        self.classes.append(course)

class StaffMember(Person):
    PERMANENT, TEMPORARY = range(2)

    def __init__(self, employment_type, *args, **kwargs):
        self.employment_type = employment_type
        super(StaffMember, self).__init__(*args, **kwargs)

class Lecturer(StaffMember):

    def __init__(self, *args, **kwargs):
        self.courses_taught = []

        super(Lecturer, self).__init__(*args, **kwargs)
    def assign_teaching(self, course):

        self.courses_taught.append(course)

>>>Mali = Student(Student.POSTGRADUATE,"Mali", "Silva",
"S045")
Mali.enrol(a_postgrad_course)
saman = Lecturer(StaffMember.PERMANENT,"saman", "perera",
"077")
saman.assign_teaching(an_undergrad_course)
```

Base class is Person, which represents any person associated with a university and one subclass represents students where as another represents staff members. One more subclass represents Staff Members who are lecturers.

This example represents both student numbers and staff numbers by a single attribute, number, which is defined in the base class. It uses different attributes for student (undergraduate or postgraduate) and also for staff members (permanent or a temporary employee).

This example contains a method in Student for enrolling a student in a course, and a method in Lecturer for assigning a course to be taught by a lecturer.

The `__init__` method of the base class initialises all the instance variables that are common to all subclasses. Each subclass **override** the `__init__` method so that we can use it to initialise that class's attributes. Overriding Methods can override parent class methods to add special or different functionality in to the subclass.

When the current class and object are passed as parameters, super function return a proxy object with the correct `__init__` method, which we can then call. In each of the overridden `__init__` methods we use the method's

parameters which are specific to our class inside the method, and then pass the remaining parameters to the parent class's `__init__` method.

A common convention is to add the specific parameters for each successive subclass to the *beginning* of the parameter list, and define all the other parameters using `*args` and `**kwargs` then the subclass doesn't need to know the details about the parent class's parameters. Because of this, if we add a new parameter to the superclass's `__init__`, we will only need to add it to all the places where we create that class or one of its subclasses.

---

## 7.6 MULTIPLE INHERITANCE

---

Python programming allows multiple inheritance. It means you can inherit from multiple classes at the same time.

```
class SuperClass1():
    def method_s1(self):
        print("method_s1 called")

class SuperClass2():
    def method_s2(self):
        print("method_s2 called")

class ChildClass(SuperClass1, SuperClass2):
    def child_method(self):
        print("child method")

c = ChildClass()
c.method_s1()
c.method_s2()
```

In this example `ChildClass` inherited `SuperClass1` and `SuperClass2`. And also object of Child Class is now able to access `method_s1` and `method_s2`.

### Activity 7.1:

- 1) Write a Person Class. Make another class called Student that inherits it from Person class.
- 2) Define few attributes that relate with Student class, such as school they are associated with, graduation year, GPA etc.
- 3) Create an object called student Set some attribute values for the student, that are only coded in the Person class and another set of attribute values for the student, that are only in the Student class.
- 4) Print the values for all of these attributes.

---

## 7.7 Data Encapsulation

---

Encapsulation hides the implementation details of a class from other objects. The aim of using encapsulation is that the data inside the object should only be accessed through a public interface. If we want to use the data stored in an

object to perform an action or calculate a derived value, we define a method associated with the object which does this.

Encapsulation is a good idea for several reasons:

- The functionality is defined in one place and not in multiple places.
- It is defined in a logical place - the place where the data is kept.
- Data inside our object is not modified unexpectedly by external code in a completely different part of our program.
- Encapsulation encourages the concept of data hiding

In the previous unit we discussed few more concepts in object oriented design. There, we identified objects like student and bicycle, and later defined classes to represent them. It was clear that the objects and the real world entities have close associations.

However, there will be no clear association between objects and real world concepts all the time. So, as a programmer, you need to find out how they should interact. In such a case, we can discover class interfaces by data encapsulation as we discovered function interfaces by encapsulation and generalisation.

**Protected members can access** only within its class and subclasses. By prefixing the name of the variable with **a single underscore**, you can define protected variables in Python.

And also by **using** double underscore ( `__` ) in front of the variable or a function name you will be able to define private members.

```
class Person:
    def __init__(self): self.fname = 'saman'
        self._lname = 'perera'

    def PrintPersonName(self):
        return self.fname + ' ' + self.__lname

#Outside class
P = Person()
print(P.fname)

print(P.PrintPersonName())
print(P.__lname)

#AttributeError: 'Person' object has no attribute '__lname'
```

In this example accessing public variables outside the class definition is possible. But cannot access private variable outside the class.

## 7.8 POLYMORPHISM

In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived

classes. For example, given a base class *shape*, polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the area method to it will return the correct result.

For example, `sum` is a built-in function which adds the elements of a sequence. As long as the elements in this particular sequence support addition, `sum` function will work.

The `Student` class provides an `add` method. Therefore, we can use it to work with built-in method '`sum`'.

```
>>> s1 = Student(99, 200) >>> s2 = Student(99, 300) >>> s3 = Student(99, 500) >>>
full_fee = sum([s1, s2, s3]) >>> print(full_fee) 1000
```

You can define functions to work with any type of data. Sometimes these may even operate on types not intended originally.

Functions which can work with different types are called polymorphic. They facilitate code reuse.

### 7.8.1 Overriding and Overloading

Overriding is replacing a method of the superclass with a new method in the subclass. The new method in the subclass is automatically called instead of the superclass method.

To override a method in the base class, subclass needs to define a method with same method name and same number of parameters as method in base class.

```
class one ( ):
    def __init__(self):
        self._a = 1
    def method_one(self):
        print("method_one from class one")

class two (one):
    def __init__(self):
        self._b = 1
    def method_one(self):
        print("method_one from class two")

c = two()

c. method_one ( )
```

In this example we are overriding `method_one()` from the base class.

Try to comment `method_one()` in class `two` and

`method_one()` from Base class which is class `one` will run.

Method overloading is having multiple methods with the same name but with different sets of arguments.

For example, method overloading is useful if we want to have a method that accepts either an integer or a string, For this situation with OOP concepts, we can have two methods, but both method name is **add**. One method accepts string values and one other method used for the integer values.

### Video 10: Multiple Inheritance and Encapsulation

You may watch this video first and then attempt Activity 7.2.

URL: <https://youtu.be/sXKrnYdDJ8w>



---

#### Activity 7.2:

A CEO buys a car. Later on the CEO buys two new cars BMW and a Mercedes. There is a driver for the CEO who chooses a car to drive to the office.

- 1) Identify the classes involved in this scenario.
- 2) Select appropriate superclass and subclasses
- 3) Implement move method inside the superclass.
- 4) Invoke the move method in superclass by creating instances of subclasses from a sub class.
- 5) Implement the move method inside the subclasses.
- 6) Override the move methods by creating instances of sub class.

---

#### Check Your Progress

- Q-1 What do you understand by data encapsulation? How data access specifiers are taken care in Python? Discuss.
- Q-2 Compare Overloading and Overriding

---

## 7.9 SUMMARY

In this unit you learned many fundamentals concepts of Object Oriented Programming. We discussed inheritance, customization with subclasses, Polymorphism and encapsulation in Python with real world examples. In this unit you have also learned ways to reuse programming codes with inheritance.

---

## 7.10 FURTHER READING

- 1) Python for Everybody, Object oriented programming, accessed web (2018), <https://www.py4e.com/html3/14-objects> Copyright CC-BY 3.0 - Charles R. Severance

- 2) Allen B. Downey (2012). Think Python.  
<http://greenteapress.com/wp/think-python/>

**Download this book for free at**

<http://greenteapress.com/thinkpython/thinkpython.pdf>

---

## 7.11 ANSWER TO CHECK YOUR PROGRESS

---

Ans-1 Refer section 7.7

Ans-2 Refer section 7.8



---

## UNIT 8 ERROR AND EXCEPTION HANDLING

---

### Structure

- 8.1 Introduction
- 8.2 Objectives
- 8.3 Terminologies
- 8.4 Errors
  - 8.4.1 Syntax Errors
  - 8.4.2 Runtime Errors
- 8.5 Semantic Errors
- 8.6 Exceptions
  - 8.6.1 User-Defined Exceptions
- 8.7 Exception Handling
- 8.8 Unit summary
- 8.9 References and Further Reading
- 8.10 Answer to Check your progress

---

### 8.1 INTRODUCTION

---

In this unit we discuss about different types of errors that can occur in a program, how to identify them and how to correct them. The three types of errors that will be discussed in this unit are: Syntax errors, Runtime errors and Semantic errors. The latter part of this unit will discuss the definition of exception and the importance of handling it using appropriate techniques.

---

### 8.2 OBJECTIVES

---

Upon completion of this unit, you will be able to:

- *Identify* the possibilities where programs can lead to unexpected outcomes
- *Describe* the techniques used to distinguish among the type of errors
- *Describe* the errors and exceptions handling mechanisms in Python
- *Write* robust code using appropriate error handling techniques

---

### 8.3 TERMINOLOGIES

---

- Error:** Mistake done while writing the program
- Fault:** Manifestation of an error
- Recursion:** Process of defining a function or calculating a number by repeated application of an algorithm.



---

## 8.4 ERRORS

---

Since there are many kinds of errors that can occur in a program, it is highly important to distinguish them and solve them. The following are the types of errors that occurs in a Python program. These errors can be found in the process of debugging:

- Syntax errors occur when translating the source code into byte code. These errors indicate the mistakes in the syntax of a program such as using a keyword as a variable or misspelling a keyword. For example `SyntaxError: invalid syntax`.
- Runtime errors appear when the program is run. They are produced by the interpreter and are also called exceptions as they mean something exceptional has happened. Runtime error messages display where the error occurred after execution of which function. For example:
- Semantic errors will not make the program stop immediately. Instead they will let the program run without displaying any error messages but produce an incorrect result. For example, one statement may not be executed in the program but will produce a result which is incorrect.

### 8.4.1 Syntax Errors

Syntax errors are the ones which is easiest to solve when you find them. Sometimes, syntax error messages are vague and it is not easy to understand what has happened. For example, messages like `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, do not give any useful information to fix the problem.

However, the good thing about this is that the message informs there is an error at one particular point in the program. The location pointed may not be very accurate but it will be closer to where the error is and is usually the preceding line.

#### Example 8.1:

```
>>> while True print 'Welcome to Python'  
SyntaxError: invalid syntax
```

Here are some guidelines to avoid some common syntax errors:

- 1) Do not use Python keywords as variable names.
- 2) Check whether Python keywords are misspelled.
- 3) Check whether strings matching quotation marks ( ' ' ).
- 4) Make sure that ":" is there at the end of the header of every compound statement like class, for, while, if, and def statements.
- 5) Check whether all strings in the code have matching quotation marks.

- 6) Any unclosed parenthesis such as (, {, or [ makes Python continue with the next line as part of the current statement.

Proper indentation makes sure that statements are written in the way they are supposed to be executed. It enables programmers to detect mistakes before running the program. Since, there could be issues in mixing spaces and tabs, use a text editor that generates consistent indentation.

Especially, it is useful to give serious consideration when writing and using variables. It is advisable to write functions that use global variables first. In a new class, always encapsulate related variables as attributes and transform the associated functions into methods.

## 8.4.2 Runtime Errors

Once the program is syntactically correct, it will start running. Following are a few factors that would occur when runtime errors occur.

- 1) If the program does nothing, it could be because the functions and classes in the program do not actually invoke anything to start execution.
- 2) If your program is caught in an infinite loop or infinite recursion, it would also not do anything.

### Infinite loop

To make sure that the loop is executed, we can write a few print statements immediately before the loop and after the loop that says “entering the loop” and “exit loop” respectively. If only first message is displayed and not the second, that means the loop is not ending. Never ending loops are also called ‘Infinite Loops’.

### Example 8.2:

```
1. Print('Entering the loop')
2. x = 14
3. while x < 15:
4.     # write something to do with x
    but do not decrement
5.     print "x: ", x
6.     print('Exit loop')
```

When the program is run you will see value of x being oriented without stopping. Then you can stop the program and add,

```
x = x - 1
```

Then you will see the last print statement.

### Infinite recursion

If there is an infinite recursion, it will cause the program run for a while and then produce a “RuntimeError: Maximum recursion depth exceeded” error. That is, there should be some condition that causes the function or the method to

return without making a recursive invocation. Then you must identify the base case. Like with the infinite loop, you can add print statements at the beginning of the function and in the middle so that an output is printed every time the function or the method is invoked.

Even if you do not get this error but suspect there would be an issue with a recursive method or function, can verify it by adding print statements as discussed above.

When something goes wrong during runtime, a message will be printed including the name of the exception, (this would be discussed under the section 'Exceptions') and the line of the program where the problem occurred. It can be traced back to the place where the error occurred.

---

## 8.5 SEMANTIC ERRORS

---

Semantic errors are the logical errors or problems in the algorithm. It is hard to debug these as the interpreter provides no information about the problem. Only the programmers know what the program is expected to do.

It would be easy if you can make a connection between source code and the output and then start debugging. One way to do this is to add a few well-placed print statements and the other is to setup the debugger, inserting and removing breakpoints, and execute the program step by step.

To verify the correctness of an algorithm, break it into smaller programs or modules and test each module separately. Then test the integrated system.

For example, let's look at a scenario for calculating the average of three numbers.

### Example 8.3:

```
num1 = float(input('Enter a number 1: '))
num2 = float(input('Enter a number 2: '))
num3 = float(input('Enter a number 3: '))
average = num1+num2+num3/3
print ('The average of the three numbers is:', average)
```

Say that inputs are given as 5,6,8, then the output of the above example is:

```
('The average of the three numbers is:',
13.666666666666666)
```

But the output of the actual calculation, i.e. average of 3 numbers should be:

```
('The average of the three numbers is:',
6.333333333333333)
```

Therefore the output of the program stated above is incorrect because of the order of operations in arithmetic.

In order to rectify this problem, the following parenthesis should be added:

```
average= (num1+num2+num3)/3
```

Hence the program should be as Example 8.4.

#### Example 8.4:

```
num1 = float(input('Enter a number: '))
num2 = float(input('Enter a number: '))
num3 = float(input('Enter a number: '))
average= (num1+num2+num3)/3
print ('The average of the the three numbers
is:',average)
```

The above example would give you an idea of how a program may not give you an error, but logically be incorrect.

#### Activity 8.1

List the different types of errors and explain how you can identify them separately.

## 8.6 EXCEPTIONS

Programming errors detected during the execution are called exceptions. An exception modifies the flow of the program due to a fault. These exceptions are generally not handled by programs, and Python interpreter display error messages as shown below.

#### Example 8.5:

```
>>> 12 *3/0
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module> 12 *3/0
ZeroDivisionError: division by zero

>>> 111 + num
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
NameError: name 'spam' is not defined

>>> 111 + '222'

Traceback (most recent call last):

  File "<pyshell#14>", line 1, in <module>
TypeError: unsupported operand type(s) for +: int and
str
```

Usually, the last line of the error message indicates the type of the exception such as ZeroDivisionError, NameError and TypeError. For built-in exceptions there are standard exception names. However, for user-defined exceptions names must be provided and it is good to adhere to standard practices.

Following are a set of exceptions used as base classes for many other built-in exceptions.

- *exception* **BaseException**

In Python all exceptions are considered to be instances of a class derived from this exception. But it is not meant to be directly inherited by user-defined classes. If `_str_` is called on an instance of this class, the set of the argument(s) to the instance is returned, or the empty string if there are no arguments.

**args** - some built-in exceptions expect a number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

- *exception* **Exception**

All exceptions that are built-in but not exiting the system, are derived from this class. All user-defined exceptions should also be derived from this class.

- *exception* **ArithmeticError**

This is the base class for built-in exceptions which are raised for various arithmetic errors: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

- *exception* **BufferError**

This is raised when a buffer related operation cannot be performed.

- *exception* **LookupError**

This is the base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: `IndexError`, `KeyError`. This can be raised directly by `codecs.lookup()`.

There is a set of exceptions called 'Concrete exceptions' which are raised frequently. They are, `exception AssertionError`, `exception AttributeError`, `exception NameError`, `exception TypeError` and `exception IndexError`.

Other than these there are many other built-in exceptions.

### **Activity 8.2**

What are exceptions and why is it important to handle them appropriately? State with examples.

---

## **8.7 EXCEPTION HANDLING**

---

Python exceptions are written in a hierarchical structure. Figure 8.1 from the Python Library Reference depicts how an exception starts at the lowest level possible (a child) and travels upward (through the parents), waiting to be caught.

When an exception is met, the Python program would terminate. Then you have to catch the error to see what went wrong. When programming, if you do not know what types of exceptions may occur, you can always just catch a higher level exception.

For example, if you did not know that `ZeroDivisionError` from the previous example was a “stand-alone” exception, you could have used the `ArithmeticError` for the exception and caught that. As the Figure 8.1 shows, `ZeroDivisionError` is a child of `ArithmeticError`, which in turn is a child of `StandardError`, etc.

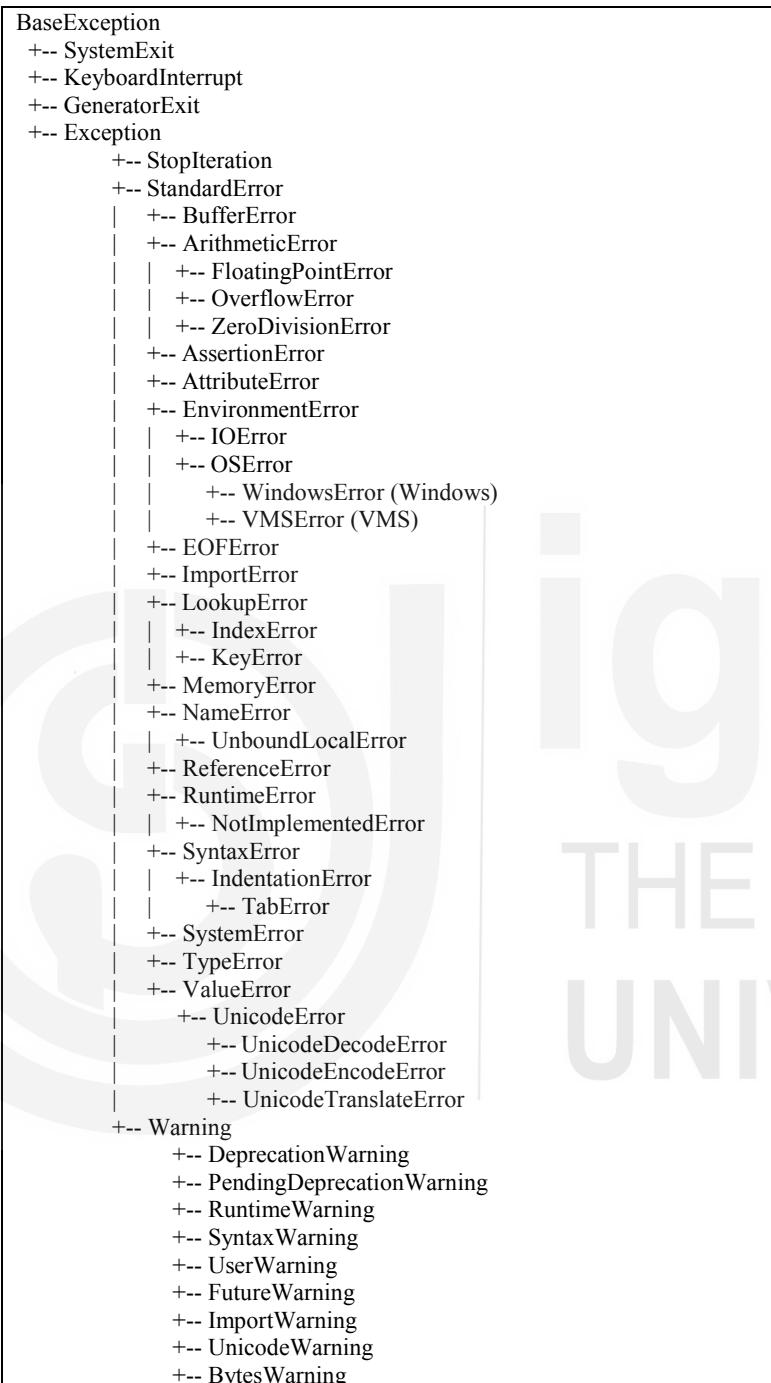


Figure 8.1: Exception Hierarchy

Reference: <https://docs.python.org/2/library/exceptions.html#builtin-exceptions>

We can handle multiple exceptions in a similar way. For example, suppose we plan to use `ZeroDivisionError` and include `FloatingPointError` as well. If we want to have the same action taken for both errors, can simply use the parent exception `ArithmeticError` as the exception to catch. That way, when either a floating point or zero division error occurs, we don't need to have a separate case for each one.

### 8.7.1 User-Defined Exceptions

You can create new exceptions by creating a new exception class. They are defined similar to other classes usually offering a number of attributes that allow information about the error to be extracted by handlers for the exception. For different error conditions, specific subclasses should be created as shown in Example 8.6.

#### Example 8.6

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in input.
    Attributes:
        Expression - input expression in which the error
        occurred
        message - explanation of the error """
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state
    transition that's not allowed.
    Attributes:
        previous - state at beginning of transition
        next - attempted new state
        message - explanation of why the specific
        transition is not allowed """
    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Reference: <https://docs.python.org/3/tutorial/errors.html>

Exceptions are usually defined with names that end in “Error,” as per the naming standard for exceptions. Many standard modules define their own exceptions to report errors that may occur in functions they define.

---

### **Activity 8.3**

Describe what user-defined exceptions are.

---

### **Check Your Progress**

- Q-1     Differentiate between Errors and Exceptions, give suitable example of each.
- Q-2     How Built-in exceptions differ from the User defined exceptions, give suitable example of each.
- 

## **8.8 SUMMARY**

---

The objective of this study unit is to introduce the types of errors and the techniques to distinguish among them in order to resolve these errors. In the first part of this unit, we discuss definitions of errors and the techniques commonly used to identify such errors. In the latter part of the unit we discuss error handling which gives an overview of what exceptions are and the techniques to handle them efficiently. You also get to know about user-defined exceptions that are supported by Python.

---

## **8.9 REFERENCES AND FURTHER READING**

---

- 1) Errors and Exceptions, Retrieved 10<sup>th</sup> June 2017,  
<https://docs.python.org/2/library/exceptions.html#builtin-exceptions>  
<https://docs.python.org/3/tutorial/errors.html>
  - 2) Downey, A. (2013). Think Python. Chapter 8. Needham, MA: Green Tea Press. Retrieved March 18, 2017, from green tea press,  
<http://greenteapress.com/wp/think-python/>
- 

## **8.10 ANSWER TO CHECK YOUR PROGRESS**

---

Ans-1     Refer section 8.4 and 8.6

Ans-2     Refer section 8.6 and 8.7.1



---

## UNIT 9 TESTING

---

### Structure

- 9.1 Introduction
- 9.2 Objectives
- 9.3 Terminologies
- 9.4 Software Testing
- 9.5 Testing Methods: Black box and White box
  - 9.5.1 Boundary Value Analysis
    - Video 11: Software Testing Types Check Your Progress
- 9.6 Testing Frameworks for Python
- 9.7 Example of Python Unit Testing
  - Video 12: Unit Testing in Python
- 9.8 Summary
- 9.9 References and Further Reading
- 9.10 Answer to check your progress

---

### 9.1 INTRODUCTION

---

This session introduces you to the different testing methods that can be used with any programming language as well as Python. The two main methods we discuss here are black box testing and white box testing. This unit also gives an overview of the test framework called unit-test for Python. Along with it, we examine few examples on how to write a simple test case.

---

### 9.2 OBJECTIVES

---

Upon completion of this unit you will be able to:

- *Explain* testing methods used in Python
- *Differentiate* black box testing and white box testing
- *Select* a suitable test framework for Python
- *Write* simple test cases in Python

---

### 9.3 TERMINOLOGIES

---

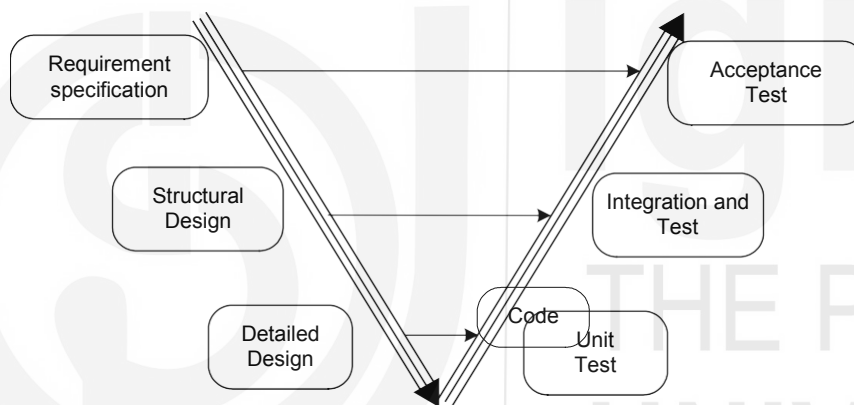
**Recursion:** A function applied within its own definition

**Framework** An abstraction in software providing a generic functionality that can be modified.

## 9.4 SOFTWARE TESTING

Testing can be defined as execution of a software for the purpose of verification and validation. Verification is the process of making sure that the product is build the right way whereas validation is the process of making the right product. Testing strategies vary in their effectiveness at finding faults. There is no direct relationship between testing and reliability. However, more stringent the level of testing the system survives, the more reliable it is likely to be.

The popular representation of the software life cycle shown in Figure 9.1 depicts the role of testing towards the end of development. It also illustrates how activities in the first half of the life cycle contribute to testing, by providing test specifications, at various levels. For example, the detailed design stage produces a test specification for the unit (or module) test, which determines how the code should be tested for conformance to the design.



**Figure 9.1 Testing in various stages of the life cycle**

In terms of intermediate products, there are two inputs to any testing stage: the product to be tested, and the relevant test specification.

In general, the principal stages of testing during software development can be summarised as follows.

### Testing during the development

- **Unit or modules testing** - Completed modules Vs module designs
- **Integration testing** - Integrated program Vs system design
- **Functional testing** - Completed system Vs system specification
- **System testing** - Completed system Vs objectives

First, module tests are carried out as a check on the coding of module designs.

Next, the integrated software is tested against its higher level design. Then, the functionality of the system it is tested against the specification.

Finally, the system test addresses non-functional aspects of the specification, like performance and reliability.

These stages of testing are used by the developer with the aim of finding and eliminating faults before the system is delivered to the customer or the marketplace. Following are the stages of testing after development:

#### Testing after development

- **Acceptance testing** - Completed system Vs requirements of real users
- **Alpha test** - User and developer test the system using real data
- **Beta test** - Release of product to a section of the market for real use and fault reporting
- **Installation testing** - Tests to check the installation process
- **During use** - Using spare capacity to do additional automatic testing

The precise selection of which of these post-development test stages to use depends on the kind of system and whether it is critical. Often, these decisions are driven by market considerations rather than technical reasons.

#### Functional and Non-functional Testing

Functional testing verifies that all the specified requirements have been incorporated. These are designed to determine that the developed software system operates the way it should. Non-functional testing is designed to find out whether your system will provide a good user experience.

Examples of non-functional tests include:

- Load/Performance testing
- Compatibility testing
- Localisation testing
- Security testing
- Reliability testing
- Stress testing
- Usability testing
- Compliance testing

---

## 9.5 TESTING METHODS: BLACK BOX AND WHITE BOX

---

There are many testing methods. In this unit we will consider two main types called Black Box Testing and White Box Testing.

In black box testing, test cases are derived from the requirements without reference to the code itself or its structure. Here, the program is treated as a

black box and testers check whether expected output is given for a selected set of inputs.



In white box testing, test data are derived from the internal logic of the program such as endless loops and wrong branching statements. The structure of a program is said to be tested exhaustively if every possible path through the software has been executed at least once. However even this 'exhaustive' testing is not guaranteed to activate every possible fault. Faults that depend on specific data values could still remain: e.g. if the correct line of code which find the absolute value of  $x-y$  and compare it with `constant_p`,

If  $\text{abs}(x-y) < \text{constant\_P}$ ,

were to be erroneously replaced by

`if(x-y) < constant_P`

The error would remain undetected if all test cases exercising this part of the code had  $x \geq y$ .

Moreover, in most real-life cases, the number of paths through a program is very large. Therefore, something less than full path coverage is usually chosen.

Next, we will discuss an important black box testing technique called boundary value analysis.

### 9.5.1 Boundary Value Analysis

Here we choose test cases directly on, above and beneath the boundary of the inputs.

Example:

For  $x$  valid if  $1.0 \leq x \leq 1.0$

test the boundary values such as  $x \in \{1.0, 1.0, 1.001, 1.001\}$

i.e. in general  $\text{min}$ ,  $\text{min}-1$ ,  $\text{max}$ , and  $\text{max}+1$

It is known that many software faults occur at the boundaries of input domains.

#### Video 11: Software Testing Types

You may watch this video and then attempt Activity 9.1.

URL: <https://youtu.be/GiHIZPps0TY>

**Activity 9.1:**



Explain the difference between white box testing and black box testing.

---

## 9.6 TESTING FRAMEWORKS FOR PYTHON

---

Python has a unit testing framework called **unittest** which is **sometimes** referred to as “PyUnit”. It had been inspired by Junit for Java.

**unittest** is important as it supports test automation. It provides other facilities such as sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. It also provides classes that make it easy to support these features.

**test fixture** - represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

**test case** - is the smallest unit of testing and checks for a specific response to a particular set of inputs. **unittest** provides a **base class** called **TestCase** which can be used to create new test cases.

**test suite** - is a collection of test cases that should be executed together.

**test runner** - is a component which coordinates the execution of tests and provides the outcome to the user. The runner may use a graphical user interface, a text interface, or return a special value to indicate the results of executed tests.

**TestCase** and **FunctionTestCase** classes support the concepts test case and test fixture, respectively. The former should be used when creating new tests, and the latter can be used when integrating existing test code with a **unittest**-driven framework. When building test fixtures

using **TestCase**, the **setUp()** and **tearDown()** methods can be overridden to provide initialization and cleanup for the fixture.

With **FunctionTestCase**, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it is successful, the cleanup method is run whatever the outcome of the test. Each instance of the **TestCase** will only be used to run a single test method, so a new fixture is created for each test.

**TestSuite** class facilitates implementation of test suits. It allows individual tests and test suites to be aggregated; when the suite is executed, all tests are added directly to the suite and in “child” test suites are run.

### *Activity 9.2:*

Explain the concepts used in unittest framework.

## 9.7 EXAMPLE OF PYTHON UNIT TESTING

The **unittest** module provides a rich set of tools for constructing and running tests. Here, we will discuss a small subset of those tools sufficient to meet the needs of most users.

Let us examine how we can test few methods used in string objects. Example 9.1:

```
import unittest
class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('kala'.upper(), 'KALA')

    def test_isupper(self):
        self.assertTrue('KALA'.isupper())
        self.assertFalse('Kala'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
    #check that s.split fails when the separator is not a string
    with self.assertRaises(TypeError):
        s.split(2)

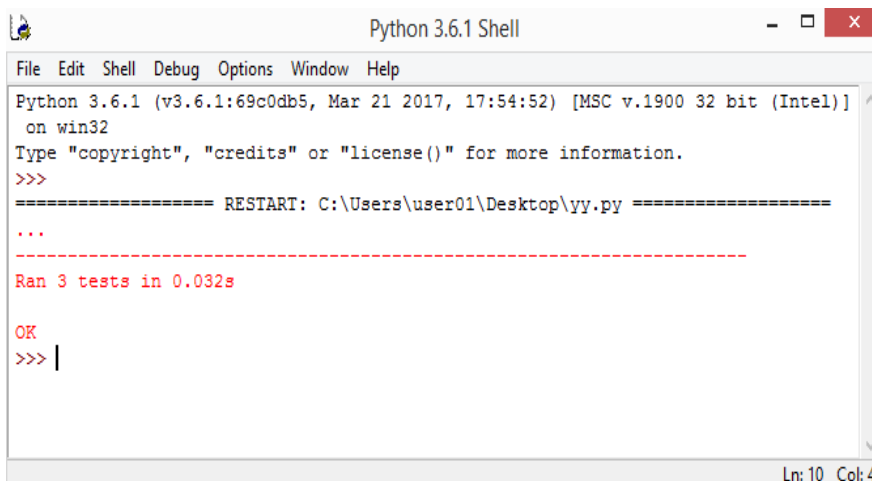
if __name__ == '__main__':
    unittest.main()
```

A **testcase** is created as a subclass of **unittest.TestCase**. Here, the three individual tests are named with 'test' prefix so that these names indicate to the test runner that they represent tests.

At the beginning of each method, **assertEqual()** is called to check for an expected result; **assertTrue()** or **assertFalse()** to verify a condition; or **assertRaises()** to verify that a specific exception is raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

The **setUp()** and **tearDown()** methods enable to define instructions that will be executed before and after each test method.

The final block shows a simple way to run the tests. **unittest.main()** provides



```
Python 3.6.1 Shell
File Edit Shell Debug Options Window Help
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\user01\Desktop\yy.py =====
...
Ran 3 tests in 0.032s
OK
>>> |
```

a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

Instead of **unittest.main()**, there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with the following lines as shown below:

```
suite
=unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

Running the revised script from the interpreter or another script produces the following output:

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\user01\Desktop\yy.py =====
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok
-----
Ran 3 tests in 0.000s

OK
>>>
```

The above examples show some commonly used **unittest** features, which are sufficient to meet many of everyday testing needs.

### Video 12: Unit Testing in Python 0

You may watch this video and then attempt Activity 9.3.

URL: <https://youtu.be/agMq0XVkfmk>



---

### Activity 9.3

Write a test case for a string method that test for a “FOOD”.

---

### Check Your Progress

- Q-1 What is debugging? Discuss functions of bdb and pdb debugging modules of Python.
  - Q-2 Compare profilers and debuggers, give example of each.
  - Q-3 Discuss the implementation of profiling interface provided by Python standard library.
-

---

## 9.8 SUMMARY

---

The objective of this study unit is to introduce you to the testing methods used in Python and identify the differences between black box testing and white box testing. It also enables you to familiarise with a suitable framework for Python along with writing simple test cases. Later we discussed individual tests that are defined with methods and a few examples on the ways to run the tests with a finer level of control.

---

## 9.9 FURTHER READING

---

- 1) Jackson, C., & Jackson, C. (2014). Chapter 14. In Learning to program using Python. Utgivningsort okänd: Createspace, Retrieved March 18, 2017, [www.sandal.tw/upload/Python\\_programming\\_2nd\\_Edition.pdf](http://www.sandal.tw/upload/Python_programming_2nd_Edition.pdf)

**Download this book for free at**

[www.sandal.tw/upload/Python\\_programming\\_2nd\\_Edition.pdf](http://www.sandal.tw/upload/Python_programming_2nd_Edition.pdf)

- 2) Downey, A. (2013). Think Python. Debugging. Needham, MA: Green Tea Press. Retrieved March 18, 2017, from green tea press, <http://greenteapress.com/wp/think-python/>

**Download this book for free at** <http://greenteapress.com/wp/think-python/>

unittest – Unit testing framework

<https://docs.python.org/2/library/unittest.html>

---

## 9.10 ANSWER TO CHECK YOUR PROGRESS

---

Ans-1 Refer section 10.4

Ans-2 Refer section 10.5

Ans-3 Refer section 10.5



---

## UNIT 10 DEBUGGING AND PROFILING

---

### Structure

- 10.1 Introduction
- 10.2 Objectives
- 10.3 Terminologies
- 10.4 Finding and removing programming errors
- 10.5 Introduction to the profilers
- 10.6 Summary
- 10.7 References and Further Reading
- 10.8 Answer to check your progress

---

### 10.1 INTRODUCTION

---

This session introduces you to the important modules that handle the basic debugger functions such as analysing stack frames and set breakpoints etc. It also gives a brief description of what profilers are and how profiling is performed on a Python applications.

---

### 10.2 OBJECTIVES

---

Upon completion of this unit you will be able to:

- *explain* the importance of performing Python profiling
- *identify* the modules that handles the basic debugger functions
- *use* these modules appropriately in the Python program
- *perform* profiling on an existing application

---

### 10.3 TERMINOLOGIES

---

**Debugging:** The process of finding and removing programming errors

**Framework** An abstraction in software providing a generic functionality that can be modified.

---

### 10.4 FINDING AND REMOVING PROGRAMMING ERRORS

---

Finding and removing programming errors is called debugging in shorten form. It is an important skill that all programmers should acquire since it is an integral part of programming.

In Python, there are two important modules that play a major role in debugging. They are `bdb`, the debugger framework and `pdb`, the Python debugger.

The following sections give a brief description on both the modules.

The `bdb` module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following syntax is used to define the exception, which is raised by the `bdb` class for quitting the debugger.

```
exception bdb.BdbQuit
```

The following class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

```
class bdb.Breakpoint(self, file, line, temporary=0, cond=None, funcname=None)
```

Breakpoints are indexed by number through a list called `bdbnumber` and by (file, line) pairs through `bplist`. The former points to a single instance of class `Breakpoint`. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a `funcname` is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

**Breakpoint** instances have the following methods:

- `deleteMe()`  
Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.
- `enable()`  
Mark the breakpoint as enabled.
- `disable()`  
Mark the breakpoint as disabled.
- `pprint([out])`

Print all the information about the breakpoint:

- The breakpoint number.
- If it is temporary or not.
- Its file, line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

- The other module that is important for debugging is called Python Debugger also known as (pdb). Use of this debugger to track down exceptions will allow you to examine the state of the program just before the error.
- The module pdb defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.
- The debugger is extensible and it is defined as the class pdb. The extension interface uses the modules bdb and cmd.

### **Activity 10.1:**

What is a debugger framework (bdb)? State each function it handles with examples.

The debugger's prompt is (Pdb). Typical usage to run a program under control of the debugger is shown in Example 10.1

### **Example 10.1**

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

pdb.py can also be invoked as a script to debug other scripts. For example:

```
Python -m pdb myscript.py
```

When invoked as a script, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After normal exit of the program, pdb will restart the program. Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.

The typical usage to break into the debugger from a running program is to insert,

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the c command.

The typical usage to inspect a crashed program is shown in Example 10.2.

**Example 10.2:**

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

`pdb.run(statement[, globals[, locals]])`

executes the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type `continue`, or you can step through the statement using `step` or `next` (all these commands are explained below). The optional global and local arguments specify the environment in which the code is executed; by default the dictionary of the module `__main__` is used.

`pdb.runeval(expression[, globals[, locals]])`

Evaluate the expression (given as a string) under debugger control. When **`runeval()`** returns, it returns the value of the expression. Otherwise this function is similar to **`run()`**.

`pdb.runcall(function[, argument, ...])`

Call the function (a function or method object, not a string) with the given arguments. When **`runcall()`** returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

`pdb.set_trace()`

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

`pdb.post_mortem([traceback])`

Enter post-mortem debugging of the given traceback object. If no traceback is given, it uses one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

```
pdb.pm()
```

Enter post-mortem debugging of the trace back found in `sys.last_traceback`.

The `run*` functions and `set_trace()` are aliases for instantiating the **Pdb** class and calling the method of the same name. If you want to access further features, you have to do this yourself:

```
class pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip=None)
```

Pdb is the debugger class.

The *completekey*, *stdin* and *stdout* arguments are passed to the underlying `cmd.Cmd` class;

The *skip* argument, if given, must be an iterable of glob-style module name patterns. (Glob module finds all path names matching a specified pattern) The debugger will not step into frames that originate in a module that matches one of these patterns.

Example call to enable tracing with *skip*:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

---

## 10.5 INTRODUCTION TO THE PROFILERS

---

A *profile* is a set of statistics that describes how often and for how long various parts of the program are executed. These statistics can be formatted into reports via the `pstats` module.

`cProfile` and `profile` provide deterministic profiling of Python programs.

The Python standard library provides three different implementations of the same profiling interface:

- 1) **cProfile** is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.
- 2) Available in Python version 2.5.
- 3) **profile**, a Python only module whose interface is imitated by **cProfile**, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

Changed in Python version 2.4: Now also reports the time spent in calls to built-in functions and methods.

- 4) **hotshot** was an experimental C module that focuses on minimising the overhead of profiling, at the expense of longer data post-processing times.

It is no longer maintained and may be dropped in a future version of Python.

The **profile** and **cProfile** modules export the same interface, so they are mostly interchangeable; **cProfile** has a much lower overhead but is newer and might not be available on all systems. **cProfile** is really a compatibility layer on top of the internal `_lsprof` module. The `hotshot` module is reserved for specialized usage.

Note: The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes. This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python code.

### Activity 10.2:

What is a Python debugger (pdb) and what are the debugging functionalities it supports?

### How to Perform Profiling

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("test|bar")')
```

Use **profile** instead of **cProfile** if the latter is not available on your system.

The above action would run `re.compile()` and print profile results like the following figure:

```
197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   1   0.000    0.000    0.001    0.001 <string>:1(<module>)
   1   0.000    0.000    0.001    0.001 re.py:212(compile)
   1   0.000    0.000    0.001    0.001 re.py:268(_compile)
   1   0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
   1   0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
   4   0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
 3/1   0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

Figure 10.1: Profile Results

The first line indicates that 197 calls were monitored. Of those calls, 192 were *primitive*, meaning that the call was not induced via recursion. The next

line: 'Orderedby: standard name', indicates that the text string in the far right column was used to sort the output. The column headings include:

- **ncalls**  
for the number of calls,
- **tottime**  
for the total time spent in the given function (and excluding time made in calls to sub-functions)
- **percall**  
is the quotient of tottime divided by ncalls
- **cumtime**  
is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.
- **percall**  
is the quotient of cumtime divided by primitive calls
- **filename:lineno(function)**  
provides the respective data for each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the run() function:

The `pstats.Stats` class reads profile results from a file and formats them in various ways.

The file `cProfile` can also be invoked as a script to profile another script. For example:

```
Python -m cProfile [-o output_file] [-s sort_order]  
myscript.py
```

`-o` writes the profile results to a file instead of to stdout

`-s` specifies one of the `sort_stats()` sort values to sort the output by. This only applies when `-o` is not supplied.

The **pstats** module's **Stats** class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats  
p = pstats.Stats('restats')  
p.strip_dirs().sort_stats(-1).print_stats()
```

The `strip_dirs()` method removed the extraneous path from all the module names. The `sort_stats()` method sorted all the entries according to the standard module/line/name string that is printed. The `print_stats()` method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand which algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class `__init__` methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats('time', 'cum').print_stats(.5, '__init__')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `__init__` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

```
p.print_callers(.5, '__init__')
```

and you would get a list of callers for each of the listed functions.

If more functionality is needed, you will have to read the manual, or guess what the following functions do:



```
p.print_callees()  
p.add('restats')
```

---

### **Activity 10.3:**

What is profiling?

Referring to the examples above, profile a function that takes in a single argument.

---

### **Check Your Progress**

- Q-1 What is debugging? Discuss functions of bdb and pdb debugging modules of Python.
- Q-2 Compare profilers and debuggers, give example of each.
- Q-3 Discuss the implementation of profiling interface provided by Python standard library.
- 

## **10.6 SUMMARY**

---

The objective of this unit is to introduce you to important modules that handle the basic debugger functions and it also provides few examples in order to use these debugger functions appropriately. Further it provides an introduction on how to perform profiling in Python along with examples that will help you achieve it.

---

## **10.7 REFERENCES AND FURTHER READING**

---

- 1) pdb – The Python Debugger  
<https://docs.python.org/2/library/pdb.html>
- Profiling –  
<https://developer.android.com/studio/profile/battery-historian>
- 

## **10.8 ANSWER TO CHECK YOUR PROGRESS**

---

- Ans-1 Refer section 10.4
- Ans-2 Refer section 10.5
- Ans-3 Refer section 10.5