

MCS-201 Programming in C and Python



Block

3

INTRODUCTION TO PYTHON PROGRAMMING**UNIT 9****Introduction to Python** **199****UNIT 10****Data Structures and Control Statements in Python** **213****UNIT 11****Functions and Files Handling In Python** **279****UNIT 12****Modules and Packages** **300**

BLOCK 3 INTRODUCTION

Python programming is widely used in Artificial Intelligence, Machine Learning, Neural Networks and many other advanced fields of Computer Science. Ideally, It is designed for rapid prototyping of complex applications. Python has interfaces with various Operating system calls and libraries, which are extensible to C, C++ or Java. Many large companies like NASA, Google, YouTube, Bit Torrent, etc. uses the Python programming language for the execution of their valuable projects.

This block prepares you about the fundamentals of Python Programming, after going through this block you will be able to understand the basic constructs of Python programming language viz. the data types, functions, packages, scripts and modules. You will learn to apply concepts of Python Programming for solving various problems assigned in check your progress section of the respective units.

Problem-solving skills are recognized as an integral component of computer programming and in this block the primary focus of this course is to teach the basic programming constructs of Python language. Emphasis is placed on developing the student's ability to apply problem-solving strategies and to implement these strategies in Python as a programming language.

Basically one must explore possible avenues to a solution one by one until s/he comes across a right path to an optimized and efficient solution. In general, as one gains experience in solving problems, one develops his/her own techniques and strategies, though they are often intangible.

This block consists of 4 units and is organized as follows:

Unit- 9 provides an overview of Python programming and compares it with C programming

Unit -10 Introduces the Data structures and Control Statements in Python

Unit – 11 provides understanding of functions and file handling in Python.

Unit - 12 introduces you the concept of Modules and Packages in Python.

Happy Programming!!



MCS-201 PROGRAMMING IN C AND PYTHON

Block

3

INTRODUCTION TO PYTHON PROGRAMMING

UNIT 9

Introduction to Python	199
-------------------------------	------------

UNIT 10

Data Structures and Control Statements in Python	213
---	------------

UNIT 11

Functions and Files Handling In Python	279
---	------------

UNIT 12

Modules and Packages	300
-----------------------------	------------

PROGRAMME/COURSE DESIGN COMMITTEE

Prof. (Retd.) S.K. Gupta
IIT, Delhi

Prof. T.V. Vijay Kumar
Dean,
School of Computer & System Sciences,
JNU, New Delhi

Prof. Ela Kumar,
Dean, Computer Science & Engg
IGDTUW, Delhi

Prof. Gayatri Dhingra
GVMITM, Sonapat, Haryana

Mr. Milind Mahajani
Vice President
Impressico Business Solutions
Noida UP

Prof. V.V. Subrahmanyam
Director
SOCIS, IGNOU, New Delhi

Prof. P. Venkata Suresh
SOCIS, IGNOU, New Delhi

Dr. Shashi Bhushan
Associate Professor
SOCIS, IGNOU, New Delhi

Shri Akshay Kumar
Associate Professor
SOCIS, IGNOU, New Delhi

Shri M. P. Mishra
Associate Professor
SOCIS, IGNOU, New Delhi

Dr. Sudhansh Sharma
Asst. Professor
SOCIS, IGNOU, New Delhi

BLOCK PREPARATION TEAM

Dr. Sudhansh Sharma, Assistant Professor
School of Computers and Information Sciences,
IGNOU, New Delhi (*Writer Unit-9*)

Prof. S.R.N. Reddy, (*Content Editor*)
HOD, Dept. of CSE,
IGDTUW, Delhi

Ms. Sarita Bansal Garg, Assistant Professor,
Department of Business Administration,
Maharaja Agrasen Institute of Management
Studies, New Delhi (*Writer Unit-10*)

Prof. Parmod Kumar (*Language Editor*)
School of Humanities, IGNOU, New
Delhi (*Unit-9*)

Ms. Jyoti Bisht, Research Scholar
School of Computers and Information Sciences,
IGNOU, New Delhi (*Writer Unit-11 & Unit-12*)

Dr. Rajesh Kumar, (*Language Editor*)
Associate Professor, Department of
English, Motilal Nehru College Delhi
University (*Unit-10, 11, & 12*)

Course Coordinator: Dr. Sudhansh Sharma

PRINT PRODUCTION

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

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

March, 2021

© Indira Gandhi National Open University, 2021

ISBN-

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

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

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

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

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

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

UNIT 9 INTRODUCTION TO PYTHON

Structure

- 9.0 Introduction
- 9.1 Objectives
- 9.2 History of Python
- 9.3 Need of Python
- 9.4 Packages for Cross platform application of Python
- 9.5 Getting started with Python
- 9.6 Program structure in python
- 9.7 Running the First program
- 9.8 Summary

9.0 INTRODUCTION

Python programming is widely used in Artificial Intelligence, Machine Learning, Neural Networks and many other advanced fields of Computer Science. Ideally, It is designed for rapid prototyping of complex applications. Python has interfaces with various Operating system calls and libraries, which are extensible to C, C++ or Java. Many large companies like NASA, Google, YouTube, Bit Torrent, etc. uses the Python programming language for the execution of their valuable projects.

To build the carrier path the skill of programming can be a fun and profitable way, but before starting the learning of this skill, one should be clear about the choice of programming language. Before learning any programming language, one should figure out which language suits best to the learner. As in our case the comparison of C and Python programming languages may help the learners to analyze and generate a lot of opinions about their choice of programming language. In this unit, I have tried to compile a few of them to give you a clear picture.

Metrics	Python	C
<i>Introduction</i>	It is a high-level, general-purpose & interpreted programming language.	C is general-purpose procedural programming language.
<i>Speed</i>	Being Interpreted programming language its execution speed is slower then that of the compiled programming language (i.e. C).	Being compiled programming language its execution speed is faster then that of the interpreted programming language (i.e. Python).
<i>Usage</i>	Number of lines of code written in Python is quite less in comparison to C	Program syntax of C is quite complicated in comparison to Python.

Declaration of variables	Declaration of Variable type is not required, they are un-typed and a given variable can be stuck different types of values at different instances during the execution of any python program.	In C, declaration of variable type is must, and it is done at the time of its creation, and only values of that declared type must be assigned to the variables.
Error Debugging	Error debugging is simple, as it takes only one instruction at a time. Compilation and execution is performed simultaneously. Errors are instantly shown, and execution stops at that instruction only.	Being compiler dependent language, error debugging is difficult in C i.e. it takes the entire source code, compiles it and finally all errors are shown.
Function renaming mechanism	the same function can be used by two different names i.e. it supports the mechanism of function renaming	Function renaming mechanism is not supported by C i.e. the same function cannot be used by two different names.
Complexity	Syntax of Python programs is Quite simple, easy to read, write and learn	The syntax of a C program is comparatively harder than the syntax of Python.
Memory-management	It supports automatic garbage collection for memory management.	In C, the Programmer has to explicitly do the memory management.
Applications	Python is a General-Purpose programming language can be used for Microcontrollers also.	C is generally used for hardware related applications.
Built-in functions	Library of built-in functions is quite large in Python.	Library of built-in functions is quite limited in C
Implementing Data Structures	Gives ease of implementing data structures with built-in insert, append functions.	Explicit implementation of functions is required for the implementation of datastructures
Pointers	Pointers functionality is not available in Python.	Pointers are frequently used in C.

We learned from the comparison of C and Python, that Python is an high-level, general-purpose, interpreted programming language. It is dynamically typed and garbage-collected, and supports multiple programming paradigms like structured (particularly, procedural,) object-oriented, and functional programming, and due to its comprehensive standard library Python is often described as a "batteries included" language

In this course you are given exposure to both programming languages i.e. C and Python, based on your requirement you can choose your option to build your career in programming.

9.1 OBJECTIVES

After going through this unit you will be able to:

- Understand the need of Python as a programming language

- Describe the cross platform applications
- Understand the structure of python program
- Write and execute your first code in python

9.2 HISTORY OF PYTHON

Python was conceived in the late 1980s as a successor to the ABC language, it was created by Guido van Rossum in 1989 and its first release was in 1991. Later, Python 2.0, released in 2000, this version includes features like list comprehensions and a garbage collection system, which was capable of collecting reference cycles. Python 2 implemented the technical details of Python Enhancement Proposal(PEP), and thus simplified the code development complexities of the earlier versions. But due to some stability issues with Python 2.x versions, the development of Python 2.7 (last version in 2.x, released in 2010) will be discontinued in 2020. The developers were resolving the issues with Python 2 in parallel, and in 2008, Python 3.0 was released. It was a major revision of the language, this version i.e. Python 3 was mainly released to fix problems which exist in Python 2. As python 3 was not completely backward-compatible, thus much of the Python 2 code are not running unmodified on Python 3. To make the migration process easy in Python 3, some features of Python 3 have been backported to Python 2.x versions.

Thus, to migrate a project from python 3 to python 2.x, a lot of changes were required, which are not only related to the projects and its applications but also to the libraries that form the part of the Python environment.

You can freely download the latest version of Python from www.python.org, where various Python interpreters are available for many operating systems. A global community of programmers develops and maintains C-Python, which is an open source reference implementation. A non-profit organization, the Python Software Foundation, manages and directs resources for Python and C-Python development.

So, what is C-python? It is the default or reference implementation of the Python programming language, and as the name suggests C-python is written in C language. Some of the implementations which are based on C-Python runtime core but with extended behavior or features in some aspects are Stack-less Python and Micro-Python; Stack-less Python, relates to C-Python with an emphasis on concurrency using tasklets and channels (used by ds-python for the Nintendo DS), and Micro-Python relates to working with microcontrollers.

C-python compiles the python source code into intermediate bytecode, which is executed by the C-python virtual machine. It is to be noted that Python is dynamic programming language, but it is said to be slow, because the default C-Python implementation compiles the python source code in bytecode which is slow as compared to machine code(native code).C-Python is distributed with a large standard library written in a mixture of C and Python,

C-Python provides the highest level of compatibility with Python packages and C extension modules.

Further, Jython, IronPython and PyPy are also the implementation of Python as a programming language, they also provide a good level of compatibility with Python packages. These implementations are based on Java, .NET and Python, respectively. We will be briefly discussing about these implementations, starting with Jython.

Jython is actually the Java platform based implementation of the Python programming language, so that you can run Python programs on the Java platform. The programs written in Jython use Java classes and not the Python modules. The Jython compiler, compiles the Jython programs into Java byte code, which can then be run by Java virtual machine. Jython enables the use of Java class library functions from the Python program. In comparison to C-Python, Jython is slow, and it is not that much compatible with the libraries of CPython. However, the compatibility in Iron Python and PyPy is quite good.

IronPython is the implementation of Python written in C# (Microsoft's .NET framework). As Jython used Java Virtual Machine (JVM), IronPython uses .Net Virtual Machine i.e. Common Language Runtime. The .NET Framework and Python libraries are used efficiently by the IronPython. Further, due to the availability of Just In Time (JIT) compiler and absence of Global Interpreter Lock, the IronPython performs better for the Python programs where use of threads or multiple cores is highly required.

Guido van Rossum (creator of Python) said "If you want your code to run faster, you should probably just use PyPy." —PyPy is an implementation of the Python programming language written in Python itself. The PyPy Interpreter is written in RPython, which is a subset of Python. The PyPy interpreter uses JIT compiler to compile the source code into native machine code which makes it very fast and efficient. PyPy also comes with default support for stackless mode, providing micro-threads for massive concurrency.

☛ Check Your Progress 1

1. Compare between C and Python programming languages

.....

.....

.....

.....

2. Discuss different variants of Python viz. C-Python, Jython, IronPython, PyPy

.....

.....

.....

.....

9.3 NEED OF PYTHON

We learned from past sections 9.0 and 9.1, that Python is free and simple to learn. The primary features of Python are, that it is an high-level, Interpreted and dynamically typed programming language, This encourages the rapid development of application prototypes, makes debugging of errors easy and identifying itself as the language to code with.

Now, you might be in the position to appreciate the simplicity and capability of Python as a programming Language, it is a general purpose language and have applicability in almost every domain of software development, be it web development or Scientific or Business or any other application, you can ever think off. It has wider coverage of applications, and it is complying with the current needs of software industry, Thus it assures a promising future to the learners.

The learners will understand the need of Python, once they understand the application areas of Python. You might not be knowing that various applications like YouTube, BitTorrent, DropBox etc. uses Python to achieve their functionality. The reason behind the choice of python, for these applications is the ability of python to be compatible for cross-platform operating systems (a cross-platform application may run on any Operating System, be it Microsoft Windows or Linux, or macOS or any other.), which simplifies the development of applications. Now, its time to start our discussion for the various types of applications that can be developed by using Python, below are few, there may be many more.

- Artificial Intelligence(AI) and Machine Learning(ML) are need of the times, as they yield the most promising careers for the learners. The field of AI and ML relates to the incorporation of intelligence in to machines by the process of self-learning from the data stored in to the system, various algorithms are available for this purpose. But, to bring AI and ML into action, Python is the first choice. Why?, because various well equipped libraries like Pandas, Scikit-Learn, NumPy etc. are available, to facilitate the engineers and scientists. Learn the algorithm, use the library and you have your solution to the problem. It is that simple.
- Data Science and Data Visualization: Data Scientists is just another promising career, if you know how to extract relevant information from the data source then world is yours. But, its not easy, various algorithms and techniques are required to work with. To simplify your work, again Python emerges to help you to study the data you have, perform operations and extract the required information. Libraries such as Pandas, NumPy help you in extracting information. Further, Matplotlib and Seaborn are the data visualization libraries, which are quite helpful in plotting and visualization of data. This is how Python helps you to become a Data Scientist.
- Mobile Applications: In general people think that Android and iOS are for mobile development, but what about using Python for mobile app

development ? Historically, Python didn't have a strong story when it came to writing mobile GUI applications, and that was quite questionable, because Python is projected as one solution for a variety of applications. But, with the passage of time, lot of development in Python has happened and now we are having Python as one of the option for Mobile App development. Now, we'll take a look at two frameworks i.e. Kivy and BeeWare, as options for mobile application development with Python.

- **Embedded Applications:** We learned from earlier sections of this unit that Python is based on C. Thus, it can be used to create software for embedded applications. The Micro-Python is a software implementation of Python 3, written in C. In fact, Micro-Python is a Python compiler that runs on the hardware of micro-controller, it includes modules to provide access to the low-level hardware. One of the renowned embedded application is Raspberry Pi which uses Python for its computing. It can be used as a computer or like a simple embedded board to perform high-level computations, you can use it for IoT or Mobile applications, thus may produce your smart gadgets.
- *Web Development:* Python is frequently used for web development, the reason behind is the availability of the full-stack frameworks for web development. There are many frameworks but Django, Flask and Pyramid are quite famous, among the frameworks for web development. They it includes common-backend logic and libraries to integrate protocols like HTTPS,FTP,SSL etc., and also supports processing of JSON,XML, E-Mail etc.
- **Game Development :** Gaming is one of the most upcoming software industry, user can build various interactive games by using various modules of Python, like Pygame and Pyglet, also the Python libraries like PySoy are available, to develop 3D games. Some of the renowned games like Civilization-IV, Disney's Toontown Online, Vega Strike etc. are developed by using Python.

Python has a variety of applications where it can be used. No matter what field you take up, Python is rewarding. So I hope you have understood the Python Applications and what sets Python apart from every other programming language.

9.4 PACKAGES FOR CROSS PLATFORM APPLICATION OF PYTHON

Python is the most popular programming language, and it marked its presence by contributing actively to every emerging field in computer science. It has vast set of libraries for almost all fields such as Machine Learning (Numpy, Pandas, Matplotlib), Artificial intelligence (Pytorch, TensorFlow), and Game development (Pygame,Pyglet), and many more.

After going through the section 9.2 of this unit, you understood the meaning of Cross-platform applications and you are now aware of the potential of Python as a cross-platform programming language, you also came to know

that the development of different type of applications require different types of packages, libraries, modules, frameworks etc. Now you might be confused that what is the difference between these terms, are they same or different. Let's Clear your confusion first and then we will briefly discuss about the functionality of different packages, used for the development of various Python applications.

Python: Framework Vs Library Vs Package Vs Module

Framework is a collection of libraries. This is the architecture of the program.

Library is a collection of packages. (*We may understand that, Python library or framework is a pre-written program that is ready to use on common coding tasks.*)

Package is a collection of modules. It must contains an `__init__.py` file as a flag so that the python interpreter processes it as such. The `__init__.py` could be an empty file without causing issues. A package, in essence, is like a directory holding subpackages and modules. While we can create our own packages, we can also use one from the Python Package Index (PyPI) to use for our projects. To import a package, we type `import Game.Sound.load` Or We can also import it giving it an alias: `import Game.Sound.load as load game`.

Module is a file which contains various Python functions and global variables. It is simply just .py extension file which has python executable code. We put similar code together in one module. This helps us modularize our code, and make it much easier to deal with. And not only that, a module grants us reusability. With a module, we don't need to write the same code again for a new project that we take up.

let's see how Module and Package differ:

- 1) A module is a file containing Python code. A package, however, is like a directory that holds sub-packages and modules.
- 2) A package must hold the file `__init__.py`. This does not apply to modules.
- 3) To import everything from a module, we use the wildcard `*`. But this does not work with packages.

We will learn more about them, as we proceed in this course, don't worry. To start with we will discuss about various Frameworks and Libraries first, you will be learning about their usage and also the usage of methods and packages, later.

TensorFlow: TensorFlow is an end-to-end python machine learning library for performing high-end numerical computations. TensorFlow can handle deep neural networks for image recognition, handwritten digit classification, recurrent neural networks, NLP (Natural Language Processing)

Keras : is a leading open-source Python library used for development of neural networks and machine learning projects. It simplifies the process of designing and development of neural networks for the beginners of machine learning. Keras also deals with convolution neural networks(CNN) and Recurrent Neural Networks (RNN), highly required in the field of Deep Learning. It includes algorithms for normalization, optimization, and activation layers. Instead of being an end-to-end Python machine learning library, Keras acts as a user-friendly, extensible interface that enhances modularity & total expressiveness.

Theano : is a library for scientific computation, it allows you to define, optimize as well as evaluate the complex mathematical expressions, which deals with multidimensional arrays. The repetitive computation of a tricky mathematical expression is the basis of several ML and AI applications. Theano quickly performs the data-intensive calculation, the rate of execution is almost hundred times faster than when executing on our CPU alone. It aims to reduce the development and execution time of ML apps, particularly in deep learning algorithms. Only one drawback of Theano in front of TensorFlow is that its syntax is quite hard for the beginners.

Scikit-learn : it is a prominent open-source library for machine learning through Python, it includes a wide range of algorithms like DBSCAN, gradient boosting, random forests, vector machines, and k-means etc., which are generally used to implement various concepts of Machine Learning i.e. Classification, Clustering, Regression etc. It can interoperate with numeric and scientific libraries of Python like NumPy and SciPy. Scikit-learn supports both supervised as well as unsupervised ML.

PyTorch : it is a production-ready library for machine-learning, supported with excellent examples, applications and use cases, also supported by a strong community. It supports GPU and CPU computations, thus provides performance optimization and scalability in research as well as production. The two high-end features of the PyTorch are Deep neural networks and Tensor computation, which are boosted because of the machine learning compiler “Glow”, specially designed to improve the performance of deep learning frameworks.

NumPy : also known as Numerical Python, a library to perform scientific computations. Almost all Python machine-learning tools like Matplotlib, SciPy, Scikit-learn, etc rely on this library to a reasonable extent. It comes with functions for dealing with complex mathematical operations like linear algebra, Fourier transformation, random number and features that work with matrices and n-arrays in Python. It is widely used in handling sound waves, images, and other binary functions.

SciPy : It is library, that works for all type of scientific programming projects. Its main functionality is built upon NumPy, and it includes modules for linear algebra, Integration, optimization, and statistics too. SciPy is supported with extensive documentation, which makes its working really easy.

Pandas : it is an open-source library of python that offers a wide range of tools for data analysis & manipulation, with Pandas, the data from a broad range of sources like CSV, SQL databases, JSON files, and Excel, can be read. The data analysis & manipulation is a pre-requisite for most of the machine learning projects, where a significant amount of time is spent to analyse the trends and patterns hidden in the datasets. Using Pandas one can manage complex data operations with just one or two commands, it comes with several inbuilt methods for data handling, and it also serves as the starting point to create more focused and powerful data tools.

Matplotlib : this library is specifically meant for Data Science, it helps to generate data visualization like 2D diagrams and graphs like histograms, scatter plots, and even graphs for non-cartesian coordinates. It is equipped with the object oriented API to embed plots into the applications, directly. It is because of this library, the Python is able to compete with scientific tools like MATLAB or Mathematica. It is quite compatible with the popular plotting libraries, but developers are required to write more code than usual, while using this library for generating advanced visualizations.

Seaborn : it is an unparalleled visualization library, based on Matplotlib's foundations. Seaborn offers high-level dataset based interface to make amazing statistical graphics. With Seaborn, it is simple to create certain types of plots like time series, heat maps, and violin plots. The functionalities of Seaborn go beyond Pandas and matplotlib with the features to perform statistical estimation at the time of combining data across observations, plotting and visualizing the suitability of statistical models to strengthen dataset patterns.

Scrapy : It is the most widely used library of python, for the purpose of data sciences, specifically data mining. It is used to build crawling programs i.e spider bots, which are used to retrieve structured data (example URLs or Contact Information) from the web. Developers use it for gathering data from APIs.

Pygame : is a python programming language library for making multimedia applications like video games, animations etc. It includes libraries related to computer graphics and sound, which are designed to be used with the Python programming language. Pygame is suitable to create client-side applications that can be potentially wrapped in a standalone executable.

Django : it a web framework build using Python, it encourages rapid development of web applications. Developers can focus on writing app only, It's primary goal is to ease the creation of complex, database-driven websites. Django emphasizes on code reusability and pluggability of components, thus it supports less code, low coupling, and rapid development of websites.

MicroPython : is a software implementation of Python 3, written in C. Infact, MicroPython is a Python compiler that runs on the hardware of micro-controller, it includes modules to provide access to the low-level hardware. One of the renowned embedded application is Raspberry Pi which uses Python for its computing. It can be used as a computer or like a simple

embedded board to perform high-level computations, you can use it for IoT or Mobile applications, thus may produce your smart gadgets.

BeeWare – is a Python GUI and mobile development framework, to develop native Python Mobile Apps. “BeeWare” project, provides a set of tools and an abstraction layer, which can be used to write native-looking mobile and desktop applications using Python.

Kivy – is an opensource Python Framework used as a Cross-platform Python GUI for Mobile App development, it allows you to write pure-Python graphical applications that run on both i.e. the main desktop platforms (Windows, Linux, macOS and Raspberry Pi) and also on on iOS & Android. Both Kivy and BeeWare are worth considering. So far as maturity goes, Kivy seems to be the more mature platform right now

☛ Check Your Progress 2

1. Compare Framework, Library, Package and Modules in python
2. Identify the prominent tools of python for following:
 - a. Artificial Intelligence application
 - b. Mobile application development
 - c. Embedded programming
 - d. Web development
 - e. Game development
3. Discuss the utility of following:
 - a. Tensor Flow
 - b. Keras
 - c. Theano
 - d. Pytorch
 - e. NumPy
 - f. SciPy
 - g. Pandas
 - h. Matplotlib
 - i. Kivy
 - j. Django
 - k. Pygame

9.6 PROGRAM STRUCTURE IN PYTHON

This section relates to the discussion over the programming structure of Python programming, i.e. the way you divide a program in your source files and mix parts from various libraries or modules or packages in to a single program, i.e. directly or indirectly, a single program includes multiple

files from different modules. Modules are in fact the library tools, which are implemented to make a collection of top-level files i.e. the high level files may use tools, defined in modules, which may use files, defined in other modules. Coming to our point, in python a file takes a module to get access to the tools it defines, and also to the tools defined in other modules included in programme. In python high level file has important path of control of your program, where from you can start your application. Just refer to the figure given below

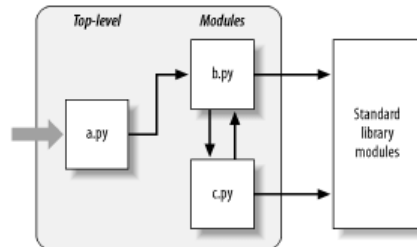


Figure : Structure Python Program

To understand the structure of Python program, say there exist three files a.py, b.py and c.py. The file model a.py is chosen for high level file . it is known as a simple text file of statements. Files b.py and c.py are modules. They are also text files of statements but they are generally not started directly, but they are invoked by a.py i.e high level file.

This is the general discussion over the structure of Python program, although Python includes all the components as they are in any other language such as **data types, conditional statements, looping constructs**, functions, file handling, Classes, Exception handling, Libraries, Modules, packages etc. We will discuss a few of them over here and the rest will be discussed in the subsequent units in this course

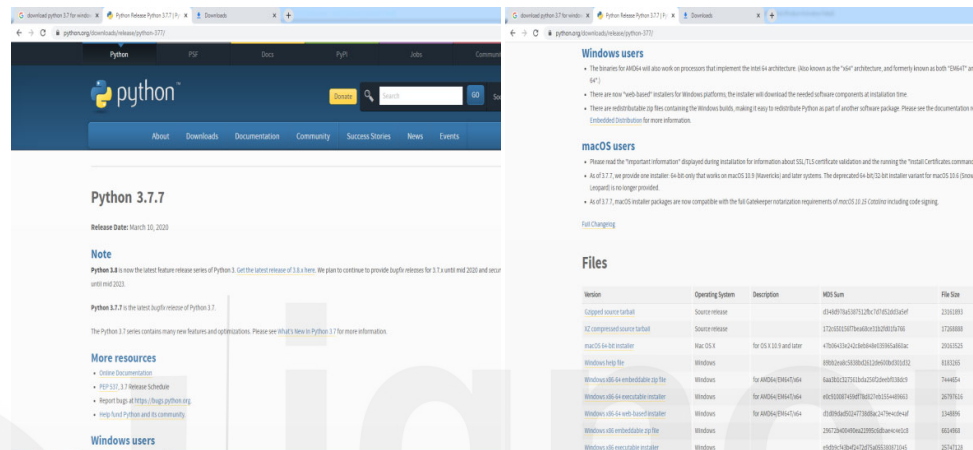
9.5 GETTING STARTED WITH PYTHON

Now, we are having a bit of clarity about the Python as a programming language, from the past sections we learned about the various libraries and frameworks of Python. Now, we need to work on the IDEs (Integrated Development Environments) of Python, there are many IDEs like Jupyter Notebook, Spyder, VS Code, R Programming etc., all are collectively available in Anaconda (Anaconda is a free and open-source distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.)), or you may also go for the cloud versions Like Google Colab Notebook, where you need not to have high configuration hardware, only internet is required and through your gmail account you can work on Python using Jupyter notebook.

We will discuss in brief, some of the ways to work with Python, you may choose any or try all and other options too.

- 1) Just browse for <https://www.python.org> and perform following steps :

- a. Download the latest version of Python for the operating system installed on your computer, as in my case its windows 64 bit, so I downloaded python-3.7.7 (python-3.7.7-amd64.exe) from <https://www.python.org/downloads/release/python-377/>
- b. Now Run this exe file and install the Python, just click next and go ahead, till the setup installation is finished
- c. Finally you are having an interface for Python programming



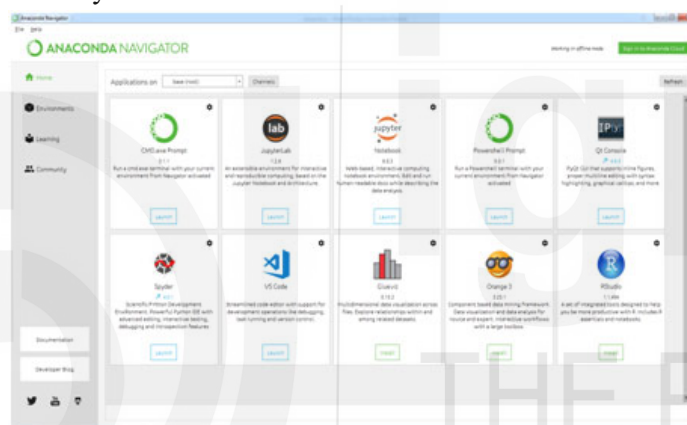
There is a variety of IDEs for python, Like Jupyter Notebook, Spyder, VS Code etc, and all are available at Anaconda (a free and open-source distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.)). To start with Anaconda Just perform following steps.

- 2) Just browse <https://www.anaconda.com/> and perform following steps:
 - a. Click the Download button on the webpage of <https://www.anaconda.com/>
 - b. the distribution section <https://www.anaconda.com/distribution/> will open,
 - c. click the download option given on this page <https://www.anaconda.com/distribution/>
 - d. <https://www.anaconda.com/distribution/#download-section> will open, the option of 64bit and 32bit graphic installer for Python 3.x (currently 3.7) and 2.x (currently 2.7) are given under Anaconda 2020.02 for windows installer.
 - e. It is recommended to download 64bit version of Python 3.x (currently 3.7),
 - f. Anaconda3-2020.02-Windows-x86_64.exe will be downloaded.
 - g. Now, just run the setup of this Anaconda3-2020.02-Windows-x86_64.exe , and click next-next, till the installation is completed.
 - h. Finally, you will find Anaconda Navigators shortcut on your desktop, click on it and you can start working with any IDE be it

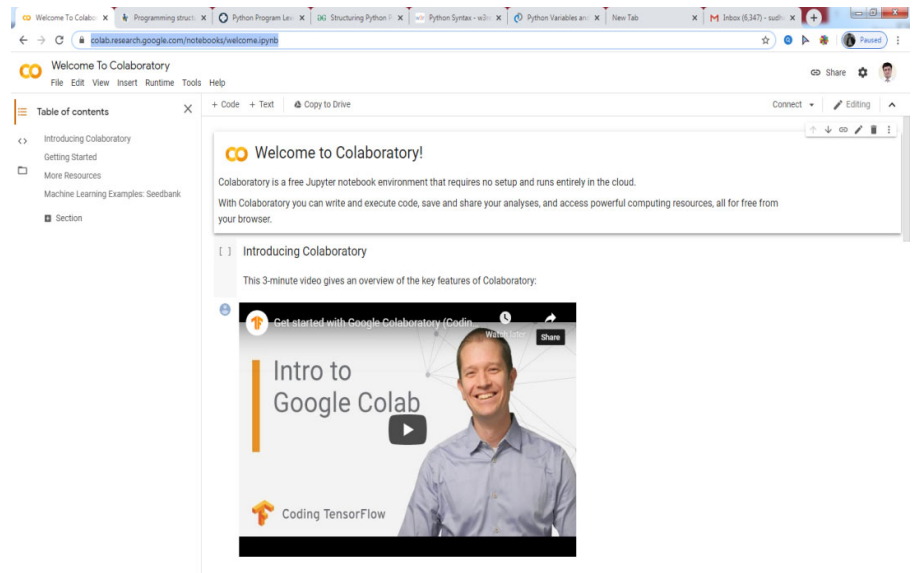
Jupyter Notebook, Spyder, VS Code etc., even you can work with R-Programming.

Important: Before working with IDEs you need to understand how to work with them and which one is suitable, following are observations, currently:

- a) When you start Jupyter Notebook on windows (by clicking on the jupyter section, given in the anaconda Navigator), a browser will open in internet explorer, many a times Jupyter Notebook won't work here, then just copy the URL from the Internet Explorer and paste it in the Google Chrome, you will find that Jupyter Notebook starts working, other details regarding the writing, saving, execution of program, will be discussed later.
- b) Program execution in Jupyter is line by line and in VS Code and Spyder it goes sideways, even errors can also be seen sideways. Any ways all are good to work with Python.

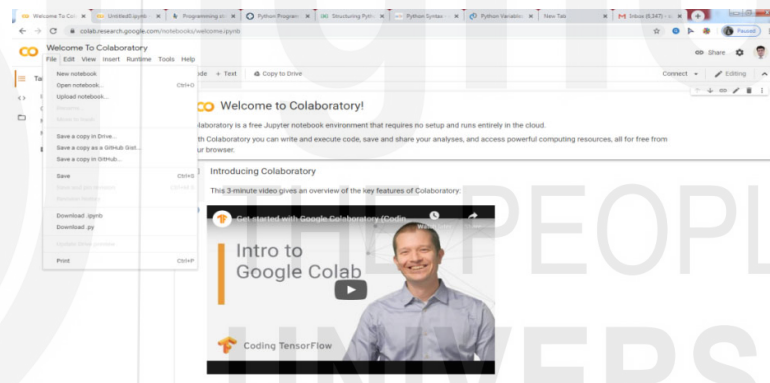


- 3) Many a times the learners may not be equipped with the systems having latest hardware configurations, as desired for the installation of Python, or their might be compatibility issues with operating system or may be due to any reason you are not able to install and start your work with Python. Under such circumstances the solution is Google Colab Notebook (<https://colab.research.google.com/notebooks/welcome.ipynb>), use this and just login with your gmail account and start your work on Jupyter Notebook, as simple as that.

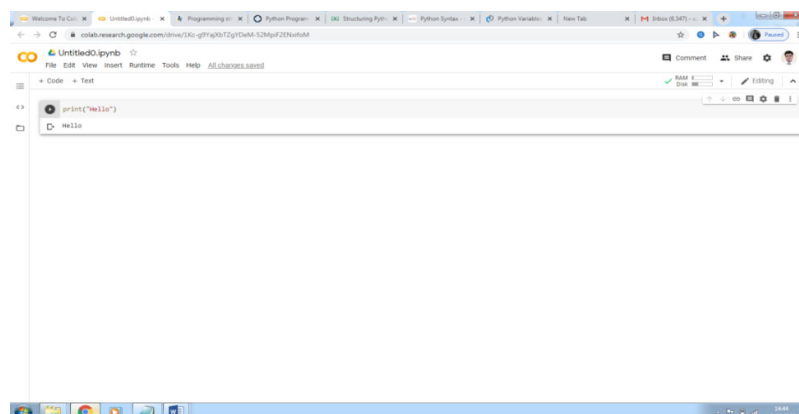


9.7 RUNNING THE FIRST PROGRAM

Just click file option and select new workbook, and new Jupyter notebook will open in Google Colab, now you may start your work



As here, I wrote my first program (`print("hello")`) to say "hello" to all of you , and executed it by simply pressing the play button, you may see just before the print command I wrote, and the output comes just beneath the statement `print("hello")`.



9.8 SUMMARY

The conceptual fundamentals of python programming language were discussed in this unit, after going through this unit the learner will be equipped not only with the with the historical understanding of python, but also with the various application areas and the tools relevant to explore the concerned application area.

SOLUTION TO CHECK YOUR PROGRESS

Check your progress 1

- 1) refer to section 9.0
- 2) refer to section 9.2

Check your progress 2

- 1) refer to section 9.4
- 2) refer to section 9.3
- 3) refer to section 9.4



ignou
THE PEOPLE'S
UNIVERSITY

UNIT 10 DATA STRUCTURES AND CONTROL STATEMENTS IN PYTHON

Data Structures and
Control Statements
in Python

Structure

- 10.1 Introduction
 - 10.2 Objectives
 - 10.3 Identifiers and Keywords
 - 10.4 Statements and Expressions
 - 10.4 Variables
 - 10.5 Operators
 - 10.6 Data Types
 - 10.7 Data Structures
 - 10.8 Control Flow Statements
 - 10.9 Summary
-

10.1 INTRODUCTION

A **Python** is a general-purpose, high level programming language which is widely used by programmers. It was created by Guido Van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code. To learn how to code in any language, it is, therefore, very important to learn its basic components. This is what is the objective of this chapter. In this chapter, we will learn about the basic constituents of Python starting from its identifiers, variables, operators and also about how to combine them to form expressions and statements. We will also study about the data types available in Python along with the control flow statements.

10.2 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic building blocks of Python programming Language
- Understand various Data Structures
- Understand usage of control flow statements

10.3 IDENTIFIERS AND KEYWORDS

An **identifier** is a name given to a variable, function, class or module. Identifiers may be one or more characters in the following format:

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myIndia, other_1 and mca_course, all are valid examples. A Python identifier can begin with an uppercase or a lowercase alphabet or (_).
- An identifier cannot start with a digit but is allowed everywhere else. 1plus is invalid, but plus1 is perfectly fine.
- Keywords (listed in TABLE1) cannot be used as identifiers.
- One cannot use spaces and special symbols like !, @, #, \$, % etc. as identifiers.
- Identifier can be of any length. (However, it is preferred to keep it short and meaningful).

Examples of valid identifiers are: marksPython, Course, MCA301 etc.

Keywords are a list of reserved words that have predefined meaning to the Python interpreter. These are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name. Attempting to use a keyword as an identifier name will cause an error. As Python is case sensitive, keywords must be written exactly as given in TABLE1.

TABLE 1 :List of keywords in Python

and	async	not
assert	finally	or
break	for	pass
class	from	nonlocal
continue	global	raise
def	import	try
elif	in	while
else	is	with
except	lambda	yield
False	True	None
del	if	Return
As	await	

Check your progress - 1:

Q1. Is Python case sensitive when dealing with Identifiers?

- Yes
- No

Q2. What is the maximum possible length of an identifier?

- 32 characters
- 64 characters
- 76 characters
- None of the above

Q3. All keywords in Python are in _____

- a) lower case
- b) UPPER CASE
- c) Capitalized
- d) None of the above

10.4 STATEMENTS AND EXPRESSIONS

A **statement** is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment. When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statement executes.

For example, the script

```
Print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

An **expression** is an arrangement of values and operators which are evaluated to make a new value. Expressions are statements as well. A value is the representation of some entity like a letter or a number that can be manipulated by a program. A single value **25** or a single variable **x** or a combination of a variable, operator and value **z + 25** are all examples of expressions. An expression, when used in interactive mode is evaluated by the interpreter and result is displayed instantly. For example,

```
In[: 7 + 3
Out[: 10
```

But in script, an expression all by itself doesn't show any output altogether. You need to explicitly print the result.

Check your progress-2 :

Q4. What is the output of the following Python expression?: 36 / 4

- a) 9.0
- b) 9

Q5. What is the output of following statement? : $y = 3.14$

- a) 3.14
- b) y
- c) None of the above

Q6. An expression can contain:

- a) Values
- b) Variables
- c) Operators
- d) All of the above
- e) None of the above

Q7. In the Python statement $x = a + 5 - b$:

a and b are _____

$a + 5 - b$ is _____

- a) Operands, an equation
- b) Operands, an expression
- c) Terms, a group
- d) Operators, a statement

10.5 VARIABLES

Variable is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution. Python is a Dynamically Typed Language. There is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type. To define a new variable in Python, we simply assign a value to a name.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letter; but it is a good idea to begin variable names with a lowercase letter. Variable names are case-sensitive. E.g., IGNOU and Ignou are different variables.

The underscore character can appear in a name. it is often used in names with multiple words, such as `my_name` or `marks_in_maths`. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

The general format for assigning values to variables is as follows:

variable_name = expression

The equal sign (=) also known as simple assignment operator is used to assign values to variables. In the general format, the operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the expression which can be a value or any code snippet that results in a value. That value is stored in the variable on the execution of the assignment statement. Assignment operator should not be confused with the = used in algebra to denote equality. E.g.,

```
In[: number = 100          #integer type value is assigned to a
variable number
```

```
In[: name = "Python"      #string type value is assigned to
variable name
```

In Python, not only the value of a variable may change during program execution but also the type of data that is assigned. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable. A new assignment overrides any previous assignments.

```
In[: year = 2020          #an integer value is assigned to year
variable
```

```
In[: year
```

```
Out[: 2020
```

```
In[:year="twenty twenty" #then an string value is assigned to year
variable
```

```
In[: year
```

```
Out[: 'twenty twenty'
```

Python allows you to assign a single value to several variables simultaneously. E.g.

```
In[: x = y = z =5        #an integer value is assigned to
variables x, y, z simultaneously.
```

Check your progress -3 :

Q8. Which of the following is a valid variable name in Python?

- a) do it
- b) do+1
- c) 1do
- d) All of the above
- e) None of the above

Q9. What is the value of 'a' here?

```
a = 25
```

```
b = 35
```

```
a = b
```

- a) 25
- b) 35

- c) It will print error
- d) None of the above

Q10. Which of the following is true for variable names in Python?

- a) unlimited length
- b) variable names are not case sensitive
- c) underscore and ampersand are the only two special characters allowed
- d) none of the mentioned

Q11. What error occurs when you execute the following Python Code snippet?

```
apple = mango
```

- a) SyntaxError
- b) NameError
- c) ValueError
- d) TypeError

10.6 OPERATORS

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

Python language supports a wide range of operators. They are:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators

TABLE 2 : List of Arithmetic Operators

Operator	Operator Name	Description	Example (try in lab) n1 = 5, n2 = 6
+	Addition operator	Adds two operands, producing their sum	In[]: n1 + n2 Out[]: 11
-	Subtraction operator	Subtracts the two operands, producing their difference	In[]: n1 - n2 Out[]: -1
*	Multiplication operator	Produces the product of the operands	In[]: n1 * n2 Out[]: 30
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor	In[]: n1 / n2 Out[]: 0.833333

%	Modulus operator	Divides left hand operand by the right hand operand and returns a remainder	In[]: n1 % n2 Out[]: 5
**	Exponent operator	Performs exponential (power) calculation on operators	In[]: n1 ** n2 Out[]: 15625
//	Floor division operator	Returns the integral part of the quotient	In[]: n1 // n2 Out[]: 0

TABLE 3 : List of Assignment Operators

Operator	Operator Name	Description	Example (try in lab) (‘~’ stand for is equivalent to)
=	Assignment	Assigns values from right side operands to left side operand	m = n1 + n2 In[]: m = n1 + n2 In[]: m Out[]: 11
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand	m += n1 ~ m = m + n1 In[]: m += n1 In[]: m Out[]: 16
-=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand	m -= n1 ~ m = m - n1 In[]: m -= n1 In[]: m Out[]: 11
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand	m *= n1 ~ m = m * n In[]: m *= n1 In[]: m Out[]: 55
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand	m /= n1 ~ m = m / n1 In[]: m /= n1 In[]: m Out[]: 11.0
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand	m ** n1 ~ m = m ** n1 In[]: m **= n1 In[]: m Out[]: 161051.0
//=	Floor Division	Produces the integral part	m //= n1 ~ m =

	Assignment	of the quotient of its operands where the left operand is the dividend and the right operand is the divisor	<code>m // n1</code> <code>In[]: m //= n1</code> <code>In[]: m</code> <code>Out[]: 32210.0</code>
<code>%=</code>	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	<code>m %= n1 ~ m = m % n1</code> <code>In[]: m %= n1</code> <code>In[]: m</code> <code>Out[]: 0.0</code>

TABLE 4 : List of Relational Operators

Operator	Operation	Description	Example (Try in Lab) <code>n1 = 10, n2 = 0,</code> <code>s1 = "Hello",</code> <code>s2="World"</code>
<code>==</code>	Equals to	If values of two operands are equal, then the condition is True, otherwise it is False.	<code>In[]: n1 == n2</code> <code>Out[]: False</code> <code>In[] : s1 == s2</code> <code>Out[]: False</code>
<code>!=</code>	Not equal to	If values of two operands are not equal, then condition is True, otherwise it is False.	<code>In[] : n1 != n2</code> <code>Out[]: True</code> <code>In[]: s1 != s2</code> <code>Out[]: False</code>
<code>></code>	Greater than	If the value of the left operand is greater than the value of the right operand, then the condition is True, otherwise it is False.	<code>In[]: n1 > n2</code> <code>Out[]: True</code> <code>In[]: s1 > s2</code> <code>Out[]: True</code>
<code><</code>	Less than	If the value of the left operand is less than the value of the right operand, the condition is True otherwise it is False.	<code>In[]: n1 < n2</code> <code>Out[]: False</code>
<code>>=</code>	Greater than or equal to	If the value of the left operand is greater than or equal to the right operand, the condition is True otherwise it is False.	<code>In[]: n1 >= n2</code> <code>Out[]: True</code>
<code><=</code>	Less than or equal to	If the value of the left operand is less than or equal to the right operand, the condition is True otherwise it is False.	<code>In []: n1 <= n2</code> <code>Out[]: False</code>

Note :For strings, the characters in both the strings are compared one by one based on their Unicode value. The character with the lower Unicode value is considered to be smaller.

TABLE 5 : List of Logical Operator

Operator	Operation	Description	Example (Try in Lab) n1 = 10, n2 = -20
And	Logical AND	If both operands are True, then condition becomes True.	In[]: n1 == 10 and n2 == -20 Out[]: True
Or	Logical OR	If any of the two operands are True, then condition becomes True.	In[]: n1 >= 10 or n2 < -20 Out[]: True
Not	Logical NOT	Used to reverse the logical state of its operand.	In[]: not(n1 == 20) Out[]: True

TABLE 6 : Boolean Logic Truth Table

n1	n2	n1 and n2	n1 or n2	not n1
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

TABLE 7 : List of Bitwise Operators

Operator	Operator Name	Description	Example (try in Lab) n1 = 60, n2 = 13
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands are 1s.	In[]: n1 & n2 Out[]: 12 (means 0000 1100)
	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.	In[]: n1 n2 Out[]: 61 (means 0011 1101)
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.	In[]: n1 ^ n2 Out[]: 49 (means 0011 0001)

~	Binary Ones Complement	Inverts the bits of its operand.	In[]: ~n1 Out[]: -61 (means 1100 0011 in 2's complement form due to a signed binary number)
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	In[]: n1 << 2 Out[]: 240 (means 1111 0000)
>>	Binary Right Shift	The left operand value is moved right by the number of bits specified by the right operand.	In[]: n1 >> 2 Out[]: 15 (means 0000 1111)

TABLE 8 : Bitwise Truth Table

n1	n2	n1 & n2	n1 n2	n1 ^ n2	~n1
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

TABLE 9 : Operator Precedence in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor Division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=	Comparisons
Not	Logical NOT
And	Logical AND
Or	Logical OR

Check your progress – 4 :

Q12. 4 is 100 in binary and 11 is 1011. What is the output of the following bitwise operators?

a = 4

b = 11

```
print(a | b)  
print(a >> 2)
```

- a) 15, 1
- b) 14, 1
- c) 15, 2
- d) 14, 2

Q13. What is the output of `print(2 ** 3 ** 2)`?

- a) 64
- b) 128
- c) 256
- d) 512

Q14. What is the output of the following assignment operator?

```
y = 10  
x = y += 2
```

- ```
print(x)
```
- a) 12
  - b) 10
  - c) `SyntaxError`

Q15. What is the output of following code?

```
a = 100
b = 200
print(a and b)
```

- a) 100
- b) 200
- c) `True`
- d) `False`

Q16. Which of the following operators has the lowest precedence?

- a) `%`
- b) `+`
- c) `**`
- d) `not`
- e) `and`

---

## 10.7 DATA TYPES

---

Data types specify the type of data like numbers and characters to be stored and manipulated within a program. Basic data types of Python are:

- 10.7.1 Number
- 10.7.2 String
- 10.7.3 None

- 10.7.4 List
- 10.7.5 Tuple
- 10.7.6 Dictionary
- 10.7.7 Set

### 10.7.1 Number

In Numbers or Numerical Data Types, we mainly have numbers. These numbers are also of four types : Integer, Float, Complex, Boolean.

**TABLE 10 : Data Types in Python**

| Type / Class | Description            | Examples           |
|--------------|------------------------|--------------------|
| Int          | Integer numbers        | -12, -3, 0, 123, 2 |
| float        | Floating point numbers | -2.04, 4.0, 14.23  |
| complex      | Complex numbers        | 3+4j, 2-2j         |
| bool         | Boolean numbers        | True, false        |

#### In Interactive mode:

**Note :** `type()` function can be used to check the data type of the variables

#### Examples:

```
In[]: i = 23 #variable i assigned with integer value
```

```
In[]: type(i)
```

```
Out[]: int
```

```
In[]: c = 2+5j #variable c assigned with a complex number
```

```
In[]: type(c)
```

```
Out[]: complex
```

```
In[]: b = 10>7
```

```
In[]: type(b) #to display the data type of variable b
```

```
Out[]: bool
```

```
In[]: b #to display the value of b
```

```
Out[]: True
```

### 10.7.2 String



A **string** is an ordered sequence of characters. These characters may be alphabets, digits or special characters including spaces. String values are enclosed in single quotation marks (e.g. "hello") or in double quotation marks (e.g., "python course"). The quotes are not a part of the string, they are used to mark the beginning and end of the string for the interpreter. In Python, there is no character data type, a character is a string of length one. It is represented by **str** class. Few of the characteristics of strings are:

- Numerical operations can not be performed on strings, even when the string contains a numeric value like str2 defined below.
- A string has a length. Get the length with the len() built-in function.
- A string is indexable. Get a single character at a position in a string with the square bracket operator, for example str1[4]. Indexing always start with 0.
- One can retrieve a slice (sub-string) of a string with a slice operation, for example str1[4:8].
- Strings are immutable, which means you can't change an existing string.

```
In[]: str1 = 'Hello Friend' #variable str1 of string type is declared
```

```
In[]: str2 = "452" #variable str2 of string type is declared
```

```
In[]: len(str1) #length of string str1
```

```
Out[]: 12
```

```
In[]: str1[4] #to access 5th character in the string
```

```
Out[]: 'o'
```

```
In[]: str1[-1] #negative indices can be used, which
 count backward from the end of string.
 [-1] yields the last letter.
```

```
Out[]: 'd'
```

```
In []: str[4:8] # Operator returns the part of the string from
 the first index to the second index, including
 the first but excluding the last. So, sub-string
 starting from 5 character to 8 character is
 extracted.
```

```
Out[]: 'o Fr'
```

If one omits the first index (before the colon), the slice starts at the beginning of the string and if second index is omitted, the slice goes to the end of the string.

```
In[]: str1[:3]
```

```
Out[: 'hel'
```

```
In[: str1[3:]
```

```
Out[: 'lo Friend'
```

```
In [: str1[2] = 'a'
```

**TypeError: 'str' object does not support item assignment.**

```
In[: str2 = ' Narender'
```

```
In[: str1 + str2 #for concatenating two strings, "+" operator is
used
```

```
Out[: 'hello Friend Narender'
```

```
In[: str1 * 3 #for creating multiple concatenated copies of a
string, "*" operator is used
```

```
Out[: 'hello FriendhelloFriendhello Friend'
```

### 10.7.3 None

A **none** is another special data type in Python. A none is frequently used to represent the absence of a value. For example,

```
In[: money = None #None value is assigned to variable
money
```

Some literatures on Python consider additional data types like List, Tuple, Set & Dictionary whereas some others consider them as the built-in data structures. We will put them in the category of data structures and discuss considering them in same but considering them as data types is also not wrong.

### 10.7.4 List

A **list** is a sequence of items separated by commas and items enclosed in square brackets [ ]. These are similar to arrays available with other programming languages but unlike arrays items in the list may be of different data types. Lists are mutable, and hence, they can be altered even after their creation. Lists in Python are ordered and have a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and creditability. It is represented by **list** class.

```
In[: list1 = [5, 3.4, 'IGNOU', 'Part 3', 45] #to create a list
```

```
In[: list1
list1 #to print the elements of the list
```

```
Out[]: [5, 3.4, 'IGNOU', 'Part 3', 45]

In[]: list1[0] #accessing first element of the
list list

Out[]: 5

In[]: list1[3] #accessing 4th element of the list

Out[]: Part 3

In[]: list1[-1] #accessing last element of the
list list

Out[]: 45

In[]: list1[-3] #accessing last 3rd element of the
list list

Out[]: 'IGNOU'
```

### 10.7.5 Tuple

A **tuple** is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma and enclosed in parentheses.

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple and it is immutable.

```
In[]: t = ('hello', 'world', 2) #creating tuple t

In[]: t # printing tuple t

Out[]: ('hello', 'world', 2)

In[]: t[1] #accessing specific element of the tuple
say second element

Out[]: 'world'
```

### 10.7.6 Dictionary

A **dictionary** in Python holds data items in key-value pairs of items. The items in the dictionary are separated with the comma and enclosed in the curly brackets {}. Dictionaries permit faster access to data. Every key is separated from its value using a colon (:) sign. The key value pairs of a dictionary can be accessed using the key. Keys are usually of string type and their values can be of any data type. In order to access any value in the dictionary, we have to specify its key in square brackets [].

```

#Creating dictionary dict1

In[: dict1 = {'Fruit':'Apple','Climate':'Cold','Price(Kg)':120}

In[: dict1 #printing the contents of
dictionary dict1

Out[: {'Fruit': 'Apple', 'Climate': 'Cold', 'Price(Kg)': 120}

In[: dict1['Climate'] #Getting value by specifying the
key

Out[:Cold

In[: dict1.keys() #printing all the Keys in the
dictionary

Out[: dict_keys(['Fruit', 'Climate', 'Price(Kg)'])

In[: dict1.values() #printing all the values in the
dictionary

Out[: dict_values(['Apple', 'Cold', 120])

```

### 10.7.7Set

In Python, **aset** is an orderd collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

Sets can be created by using the built-in `set()` function with an iterable object or a sequence by placing the sequence inside curly braces, separated by 'comma'. A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. The order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```

In[: set1 = set("IGNOU MCA BATCH") #creating
set set1

In[: set1
#displaying contents of set1

Out[: {' ', 'A', 'B', 'C', 'G', 'H', 'I', 'M', 'N', 'O', 'T', 'U'}

In[: set2 = set([2, 3, 2, 'MCA', 'IGNOU', 1, 'IGNOU']) #set with
the use of mixed values

In[: set2

Out[: {1, 2, 3, 'IGNOU', 'MCA'}

```

**Check your progress - 5:**

Q17. Which of these is not a core data type?

- a) Lists
- b) Dictionary
- c) Tuples
- d) Class

Q18. What data type is the object below?

L = [1, 23, 'hello', 1]

- a) List
- b) Dictionary
- c) Tuple
- d) Array

Q19. What will be the output of the following Python code?

```
In[:str="hello"
```

```
In[: str[:2]
```

- a) he
- b) lo
- c) olleh
- d) hello

Q20. In python we do not specify types, it is directly interpreted by the compiler. So consider the following operation to be performed:

x = 13 ? 2

objective is to make sure x has a integer value, select all that apply

- a) x = 13 // 2
- b) x = int(13 / 2)
- c) x = 13 % 2
- d) all of the above

Q21. What is the result of print(type({})) is set)

- a) True
- b) False

Q22. What is the data type of print (type(10))?

- a) float
- b) integer
- c) int

Q23. If we convert one data type to another, then it is called

- a) Type Conversion
- b) Type Casting
- c) Both of the above

d) None of the above

Q24. In which data type, indexing is not valid?

- a) list
- b) string
- c) distionary
- d) None of the above

Q25. In which of the following data type duplicate items are not allowed?

- a) list
- b) set
- c) dictionary
- d) None of the above

Q26. Which statement is correct?

- a) List is immutable && Tuple is mutable
- b) List is mutable && Tuple is immutable
- c) Both are mutable
- d) Both are immutable

---

## 10.8 DATA STRUCTURES

---

Data Structures are used to organize, store and manage data for efficient access and modification. Some literatures on Python categorize List, Tuple, Dictionary and Set as data types and some categorize these as primitive data structures. We had already introduced the above mentioned structures in the data type. Lets discuss them in detail here:

### 10.8.1 List

A **list** a container data type that acts as a dynamic array. That is to say, a list is a sequence that can be indexed into and that can grow and shrink. A few characteristics of lists are:

- A list has a (current) length – Get the length of a lsit with len() function.
- A list has an order – the items in a list are ordered, order going from left to right.
- A list is heterogeneous –different types of objects can be inserted into the same list.
- Lists are mutable and one can extend, delete, insert or modify items in the list.

The syntax for creating list is,

**list\_name = [item\_1, item\_2, item\_3, ....., item\_n]**

Any item can be stored in a list like string, number, object, another variable and even another list. In a list, we can have a mix of different item types and these item types need not have to be homogeneous. For example, we can have a list which is a mix of type numbers, strings and another list itself. The contents of the list variable are displayed by executing the list variable name.

Examples :

```
In[: list_name = [] #empty list
without any items
```

```
#when each item in the list is string
```

```
In[: stringlist = ["mahesh", "ramesh", "suresh", "vishnu",
 "ganesh"]
```

```
#when each item in the list is number
```

```
In[: numlist = [4, 6, 7, 10, 15, 13]
```

```
#when list contains mixed items
```

```
In[: mixed_list = ['cat', 87.23, 65, [9, 8, 1]]
```

#### 10.8.1.1 Indexing and Slicing in Lists

As an ordered sequence of elements, each item in a list can be individually, through indexing. The expression inside the bracket is called the index. Lists use square brackets [ ] to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed.

The syntax for accessing an item in a list is,

```
list_name [index]
```

where index should always be an integer value and indicates the item to be selected.

```
In[: stringlist[0] #to access first item in the stringlist
```

```
Out[: 'mahesh'
```

```
In[: numlist[4] #to access fifth item in the
numlist
```

```
Out[: 15
```

```
In[: numlist[6] #if index value is more than the number
of items in the list, it will raise an error
```

Output would be :

**Traceback (most recent call last):**

**File “<stdin>”, line 1, in <module>**

**numlist[6]**

**IndexError : List index out of range**

Here, in numlist index numbers range from 0 to 5 as it contains 6 items.

In addition to positive index numbers, one can also access items from the list with a negative index number, by counting backwards from the end of the list, starting at -1. It is useful if the list is long and we want to locate an item towards the end of a list.

```
In[]: stringlist[-2] #to access second list item from
the list
```

```
Out[]: 'vishnu'
```

The slice operator also works on lists:

```
In[]: numlist[1:3]
```

```
Out[]: [6, 7]
```

```
In[]: mixed_list[:3]
```

```
Out[]: ['cat', 87.23, 65]
```

```
In[]: mixed_list[2:]
```

```
Out[]: [65, [9, 8, 1]]
```

```
In[]: numlist[:]
```

```
Out[]: [4, 6, 7, 10, 15, 13]
```

If the first index is omitted, the slice starts at the beginning and if the second index is omitted, the slice goes to the end. So, if both indexes are omitted, the slice is a copy of the whole list.

### 10.8.1.2 List Operations

The + operator concatenates lists:

```
In[]: a = [1, 2, 3]
```

```
In[]: b = [4, 5, 6]
```

```
In[]: a + b
```

```
Out[]: [1, 2, 3, 4, 5, 6]
```

Similarly, the \* operator repeats a list given a number of times:



```
In[]: c = ['pass']
```

```
In[]: c * 3
```

```
Out[]: ['pass', 'pass', 'pass']
```

**==** operator is used to compare the two lists.

```
In[]: a == b
```

```
Out[]: False
```

**in** and **not in** membership operator are used to check for the presence of an item in the list.

```
In[]: 2 in a
```

```
Out[]: True
```

```
In[]: 4 not in [a]
```

```
Out[]: True
```

### 10.8.1.3 The list() function

The built-in-list() function is used to create a list. The syntax for list() function is,

```
list([sequence])
```

where the sequence can be a string, tuple or list itself. If the optional sequence is not specified then an empty list is created.

```
In[]: greet = "How are you doing?"
```

```
#string declaration
```

```
In[]: greet
```

```
Out[]: 'How are you doing?'
```

```
In[]: str_to_list = list(greet)
using list()
```

```
#string converted to list
```

```
In[]: str_to_list
```

```
Out[]: ['H','o','w',' ','a','r','e',' ','y','o','u',' ','d','o','i','n','g','?']
```

```
In[]: friend = "nidhi"
```

```
#string declaration
```

```
In[]: str_to_list + friend
string
```

```
#list concatenated with
```

```
TypeError: can only concatenate list (not "str") to list
```

```
In[]: str_to_list + list(friend)
list
```

```
#list concatenated with
```

```
Out[:]['H','o','w',' ','a','r','e',' ','y','o','u',' ','d','o','i','n','g',' ','n','i','d','h','i']
```

Data  
Cont

#### 10.8.1.4 Modifying items in Lists

Lists are mutable in nature as the list items can be modified after a list has been created. Also, a list can be modified by replacing the older item with a newer item in its place and without assigning the list to a completely new variable.

```
In[: stringlist[1] = "umesh"
```

```
In[: stringlist
```

```
Out[: ['mahesh', 'umesh', 'suresh', 'vishnu', 'ganesh']
```

When an existing list variable is assigned to a new variable, an assignment (=) on lists does not make a new copy. Instead, assignment makes both the variables names point to the same list in memory. That's why any change in one list will reflect in other list also.

```
In[: listofstrings = stringlist
```

```
In[: listofstrings[1] = "ramesh"
```

```
In[: listofstrings
```

```
Out[: ['mahesh', 'ramesh', 'suresh', 'vishnu', 'ganesh']
```

```
In[: stringlist
```

```
Out[: ['mahesh', 'ramesh', 'suresh', 'vishnu', 'ganesh']
```

#### 10.8.1.5 Traversing a list

The most common way to traverse the elements of a list is with a for loop.

```
In[: for names in stringlist :
```

```
 print(names)
```

```
Out[: mahesh
```

```
 ramesh
```

```
 suresh
```

```
 vishnu
```

```
 ganesh
```

This works well if one needs to read the elements of the list. But if one wants to write or update the elements, one needs the indices. For this one needs to combine the functions range and len:

```
In[]: for i in range(len(numlist)) :
```

```
 numlist[i] = numlist[i] * 2
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 30, 26]
```

This loop traverses the list and updates each element. The command `len` returns the number of elements in the list. The command `range` returns a list of indices from 0 to `n-1`, where `n` is the length of the list. Each time through the loop, the variable `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

#### 10.8.1.6 List Methods

Python provides methods that operate on lists. For example, **append** adds a new element to the end of a list:

```
In[]: numlist.append(50)
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 30, 26, 50]
```

**extend** takes a list as an argument and appends all of the elements:

```
In[]: nlist = [25, 35, 45]
```

```
In[]: numlist.extend(nlist)
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 30, 26, 50, 25, 35, 45]
```

**sort** arranges the elements of the list from low to high:

```
In[]: numlist.sort()
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 25, 26, 30, 35, 45, 50]
```

**pop** is used to delete elements from a list if the index of the element is known:

```
In[]: numlist.pop(5)
```

```
Out[]: 26
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 25, 30, 35, 45, 50]
```

**pop** modifies the list and returns the element that was removed. If the removed element is not required then one can use the **del** operator:

```
In[]: del numlist[0]
```

```
In[]: numlist
```

```
Out[]: [12, 14, 20, 25, 30, 35, 45, 50]
```

To remove more than one element, **del** can be used with a slice index

```
In[]: del numlist[0:2]
```

```
In[]: numlist
```

```
Out[]: [20, 25, 30, 35, 45, 50]
```

If element from the list is known which needs to be removed and not the index, **remove** can be used:

```
In[]: numlist.remove(30)
```

```
In[]: numlist
```

```
Out[]: [20, 25, 35, 45, 50]
```

The return value for remove is none.

To get a list of all the methods associated with the list, pass the list function to **dir()**

```
In[]: dir(list)
```

**TABLE 11 : Various List Methods**

| List Methods | Syntax                   | Description                                                                                                                            |
|--------------|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| append()     | list.append(item)        | The append() method adds a single item to the end of the list. This method does not return new list and it just modifies the original. |
| count()      | list.count(item)         | The count() method counts the number of times the item has occurred in the list and returns it.                                        |
| insert()     | list.insert(index, item) | The insert() method inserts the item at the given index, shifting items to the right.                                                  |
| extend()     | list.extend(list2)       | The extend() method adds the items in list2 to the end of the list.                                                                    |

|           |                   |                                                                                                                                                                                                                                                                 |
|-----------|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| index()   | list.index(item)  | The index() method searches for the given item from the start of the list and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the list then ValueError is thrown by this method. |
| remove()  | list.remove(item) | The remove() method searches for the first instance of the given item in the list and removes it. If the item is not present in the list then ValueError is thrown by this method.                                                                              |
| sort()    | list.sort()       | The sort() method sorts the items in place in the list. This method modifies the original list and it does not return a new list.                                                                                                                               |
| reverse() | list.reverse()    | The reverse() method reverses the items in place in the list. This method modifies the original list and it does not return a new list.                                                                                                                         |
| pop()     | list.pop([index]) | The pop() method removes and returns the item at the given index. This method returns the rightmost item if the index is omitted.                                                                                                                               |

Note: Replace the word "list" mentioned in the syntax with your actual list name in your code.

#### 10.8.1.7 Built-In Functions Used on Lists

There are many built-in functions for which a list can be passed as an argument.

**TABLE 12 : Built – In Functions Used on Lists**

| Built-In Functions | Description                                                                       |
|--------------------|-----------------------------------------------------------------------------------|
| len()              | The len() function returns the numbers of items in a list.                        |
| sum()              | The sum() function returns the sum of numbers in the list.                        |
| any()              | The any() function returns True if any of the Boolean values in the list is True. |
| all()              | The all() function returns True if all the Boolean values                         |

|          |                                                                                                      |
|----------|------------------------------------------------------------------------------------------------------|
|          | in the list are True, else returns False.                                                            |
| sorted() | The sorted() function returns a modified copy of the list while leaving the original list untouched. |

```
In[: len(stringlist)
```

```
Out[: 5
```

```
In[: sum(numlist)
```

```
Out[: 175
```

```
In[: max(numlist)
```

```
Out[: 50
```

```
In[: min(numlist)
```

```
Out[: 20
```

```
In[: any([1, 1, 0, 0, 1, 0])
```

```
Out[: True
```

```
In[: all([1, 1, 1, 1])
```

```
Out[: True
```

```
In[: sorted_stringlist = sorted(stringlist)
```

```
In[: sorted_stringlist
```

```
Out[: ['ganesh', 'mahesh', 'ramesh', 'suresh', 'vishnu']
```

#### 10.8.1.8 Nested List

A list inside another list is called a **nested list** and you can get the behavior of nested lists in Python by storing lists within the elements of another list. The syntax for nested lists is:

```
nested_list_name = [[item_1, item_2, item_3],
 [item_4, item_5, item_6],
 [item_7, item_8, item_9]]
```

Each list inside another list is separated by a comma. One can traverse through the items of nested lists using the for loop.

```
In[: asia = ["India", "Japan", "Korea"],
 ["Srilanka", "Myanmar", "Thailand"],
```

**[“Cambodia”, “Vietnam”, “Israel”]**

In[]: **asia[0]**

Out[]: **['India', 'Japan', 'Korea']**

In[]: **asia[0][1]**

Out[]: **'Japan'**

In[]: **asia[1][2]**

Out[]: **'Thailand'**

**Program 10.1 : Write a Program to Find the Transpose of a Matrix**

```
1. matrix = [[10, 20],
2. [30, 40],
3. [50, 60]]
4. matrix_transpose = [[0, 0, 0],
5. [0, 0, 0]]
6. for rows in range(len(matrix)):
7. for columns in range(len(matrix[0])):
8. matrix_transpose[columns][rows] = matrix[rows][columns]
9. print("Transposed Matrix is")
10. for items in matrix_transpose:
11. print(items)
```

Fig 1 : Screen Shot of execution of Program 10.1

```
[2] matrix = [[10, 20],
 [30, 40],
 [50, 60]]
 matrix_transpose = [[0, 0, 0],
 [0, 0, 0]]
 for rows in range(len(matrix)):
 for columns in range(len(matrix[0])):
 matrix_transpose[columns][rows] = matrix[rows][columns]
 print("Transposed Matrix is")
 for items in matrix_transpose:
 print(items)
```

```
➞ Transposed Matrix is
[10, 30, 50]
[20, 40, 60]
```

**Check your progress – 6 :**

Q27. Empty list in python is made by?

- a) `l = []`
- b) `l = list()`

- c) Both of the above
- d) None of the above

Q28. If we try to access the item outside the list index, then what type of error it may give?

- a) List is not defined
- b) List index out of range
- c) List index out of bound
- d) No error

Q29. l = [1,2,3,4], then l[-2] is?

- a) 1
- b) 2
- c) 3
- d) 4

Q30. The marks of a student on 6 subjects are stored in a list, list1 = [80, 66, 94, 87, 99, 95]. How can the students average marks be calculated?

- a) print(avg(list1))
- b) print(sum(list1)/len(list1))
- c) print(sum(list1)/sizeof(list1))
- d) print(total(list1)/len(list1))

Q31. What will be the output of the following Python code?

```
list1=["Python", "Java", "c", "C", "C++"]
print(min(list1))
```

- a) c
- b) C++
- c) C
- d) Min function can't be used on string elements

Q32. The elements of a list are arranged in descending order:

Which of the following two will give same outputs?

- a) print(list\_name.sort())
- b) print(max(list\_name))
- c) print(list\_name.reverse())
- d) print(list\_name[-1])

i, ii

i, iii

ii, iii

iii, iv

Q33. What will be the output of below python code?

```
list1=["tom", "mary", "simon"]
list1.insert(5,8)
print(list1)
```



- a) ["tom", "mary", "simon", 5]
- b) ["tom", "mary", "simon", 8]
- c) [8, "tom", "marry", "simon"]
- d) Error

Q34. What will be the output of below python code?

```
list1=[1,3,5,2,4,6,2]
```

```
list1.remove(2)
```

```
print(sum(list1))
```

- a) 18
- b) 19
- c) 21
- d) 22

Q35. What will be the output of below python code?

```
list1=[3,2,5,7,3,6]
```

```
list1.pop(3)
```

```
print(list1)
```

- a) [3,2,5,3,6]
- b) [2,5,7,3,6]
- c) [2,5,7,6]
- d) [3,2,5,7,3,6]

## 10.8.2 Tuple

In mathematics, a tuple is a finite ordered list (sequence) of elements. A **tuple** is defined as a data structure that comprises an ordered, finite sequence of immutable, heterogeneous elements that are of fixed sizes. Often, we may want to return more than one value from a function. Tuples solve this problem. They can also be used to pass multiple values through one function parameter.

### 10.8.2.1 Creating Tuples

A tuple is a finite ordered list of values of possibly different types which is used to bundle related values together without having to create a specific type to hold them. Tuples are immutable. Once a tuple is created, one cannot change its values. A tuple is defined by putting a comma-separated list of values inside parantheses(). Each value inside a tuple is called an item. The syntax for creating tuples is,

```
tuple_name = (item_1, item_2, item_3,, item_n)
```

Syntactically, a tuple is a comma-separated list of values:

```
In[]: t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when Python code is looked at:

```
In[]: t = ('a', 'b', 'c', 'd', 'e')
```

It is actually the comma that forms a tuple making the commas significant and not the parentheses.

To create a tuple with a single element, one must include the final comma:

```
In[]: t1 = ('a',)
```

```
In[]: type(t1)
```

```
Out[]: tuple
```

In Python, the tuple type is tuple. Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

```
In[]: t2 = ('a')
```

```
In[]: type(t2)
```

```
Out[]: str
```

One can create an empty tuple without any values. The syntax is,

```
tuple_name = ()
```

```
In[]: empty_tuple = ()
```

```
In[]: empty_tuple
```

```
Out[]: ()
```

One can store any type of type string, number, object, another variable, and even another tuple itself. One can have a mix of different types of items in tuples, and they need not be homogeneous.

```
In[]: socis = ('mca','6 sem','3-6 yrs',54000)
```

```
In[]: ignou = ('soe','soh',socis,'soms')
```

```
In[]: ignou
```

```
Out[]: ('soe', 'soh', ('mca', '6 sem', '3 - 6 yrs', 54000), 'soms')
```

### 10.8.2.2 Basic Tuple Operations

Most list operators also work on tuples. The bracket operator indexes an element:

```
In[]: t[0]
```

```
Out[]: 'a'
```

And the slice operator selects a range of elements.

```
In[: t[1:3]
```

```
Out[: ('b', 'c')
```

But if you try to modify one of the elements of the tuples, one will get an error:

```
In[: t[0] = 'A'
```

**TypeError: 'tuple' object does not support item assignment**

We can't modify the elements of a tuple, but can replace one tuple with another:

```
In[: t = ('A',) + t[1:]
```

```
In[: t
```

```
Out[: ('A', 'b', 'c', 'd', 'e')
```

Here, + operator is used to concatenate two tuples together. Similarly, \* operator can be used to repeat a sequence of tuple items.

```
In[: t * 2
```

```
Out[: ('A', 'b', 'c', 'd', 'e', 'A', 'b', 'c', 'd', 'e')
```

**in** and **not in** membership operator are used to check for the presence of an item in a tuple.

Comparison operators like <, <=, >, >=, == and != are used to compare tuples.

Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered even if they are really big.

```
In[: (0, 1, 2) < (0, 3, 4)
```

```
Out[: True
```

```
In[: (0, 1, 500000) < (0, 3, 4)
```

```
Out[: True
```

### 10.8.2.3 The tuple() Function

The built-in tuple() function is used to create a tuple. The syntax for the tuple() function is:

```
tuple([sequence])
```

where sequence can be a number, string or tuple itself. If the optional sequence is not specified, then an empty tuple is created.

```
In[]: t3 = tuple()
```

```
In[]: t3
```

```
Out[]: ()
```

If the argument is a sequence (string, list or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
In[]: t3 = tuple('IGNOU')
```

```
In[]: t3
```

```
Out[]: ('I', 'G', 'N', 'O', 'U')
```

```
In[]: t4 = (1, 2, 3, 4)
```

```
In[]: nested_t = (t3, t4)
```

```
In[]: nested_t
```

```
Out[]: (('I', 'G', 'N', 'O', 'U'), (1, 2, 3, 4))
```

#### 10.8.2.4 Built-In Functions Used on Tuples

There are many built-in functions as listed in TABLE for which a tuple can be passed as an argument.

**TABLE 13: Built-In functions Used on Tuples**

| Built-In Functions | Description                                                                                                    |
|--------------------|----------------------------------------------------------------------------------------------------------------|
| len()              | The len() function returns the number of items in a tuple.                                                     |
| sum()              | The sum() function returns the sum of numbers in the tuple.                                                    |
| sorted()           | The sorted() function returns a sorted copy of the tuple as a list while leaving the original tuple untouched. |

```
In[]: len(t3)
```

```
Out[]: 5
```

```
In[]: sum(t4)
```

```
Out[]: 10
```

```
In[]: t5 = sorted(t3)
```

```
In[]: t5
```

```
Out[]: ['G', 'I', 'N', 'O', 'U']
```

#### 10.8.2.5 Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows one to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables `x` and `y` in a single statement.

```
In[]: m = ['good', 'luck']
```

```
In[]: x, y = m
```

```
In[]: x
```

```
Out[]: 'good'
```

```
In[]: y
```

```
Out[]: 'luck'
```

Python roughly translates the tuple assignment syntax to be the following:

```
In[]: m = ['good', 'luck']
```

```
In[]: x = m[0]
```

```
In[]: y = m[1]
```

Stylistically, when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
In[]: (x, y) = m
```

#### 10.8.2.6 Relation between Tuples and Lists

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing. Lists are mutable, and their items are accessed via indexing. Items cannot be added, removed or replaced in a tuple.

If an item within a tuple is mutable, then you can change it. Consider the presence of a list as an item in a tuple, then any changes to the list get reflected on the overall items in the tuple.

```
In[]: ignou = ['soe', 'soms', 'socis']
```

```
In[]: univ = ('du', 'ggsipu', 'jnu', ignou)
```

```
In[]: univ
```

```
Out[]: ('du', 'ggsipu', 'jnu', ['soe', 'soms', 'socis'])
```

```
In[]: univ[3].append('soss')
```

```
In[]: univ
```

```
Out[]: ('du', 'ggsipu', 'jnu', ['soe', 'soms', 'socis', 'soss'])
```

### 10.8.2.7 Tuple Methods

To get a list of all the methods associated with the tuple, pass the tuple function to `dir()`.

Various methods associated with tuple are listed in the TABLE.

**TABLE 14: Various Tuple Methods**

| Tuple Methods        | Syntax                              | Description                                                                                                                                                                                                                                                                                  |
|----------------------|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>count()</code> | <code>tuple_name.count(item)</code> | The <code>count()</code> method counts the number of times the item has occurred in the tuple and returns it.                                                                                                                                                                                |
| <code>index()</code> | <code>tuple-name.index(item)</code> | The <code>index()</code> method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple, then <code>ValueError</code> is thrown by this method. |

Note: Replace the word “`tuple_name`” mentioned in the syntax with your actual tuple name in the code.

```
In[]: channels = ("dd", "star", "sony", "zee", "dd", "sab")
```

```
In[]: channels.count("dd")
```

```
Out[]: 2
```

```
In[]: channels.index("dd")
```

```
Out[]: 0
```

```
In[]: channels.index("zee")
```

```
Out[]: 3
```

```
In[]: channels.count("sab")
```

```
Out[]: 1
```

**PROGRM 10.2 : Program to Populate with User-Entered Items**

```
1. tuple_items = ()
2. total_items = int(input("Enter the total number of items: "))
3. for i in range(total_items):
4. user_input = int(input("Enter a number: "))
5. tuple_items += (user_input,)
6. print(f"Items added to tuple are {tuple_items}")
7. list_items = []
8. total_items = int(input("Enter the total number of items: "))
9. for i in range(total_items):
10. item = input("Enter an item to add: ")
11. list_items.append(item)
12. items_of_tuple = tuple(list_items)
13. print(f"Tuple items are {items_of_tuple}")
```

Fig 2 : Screen Shot of execution of Program 10.2

```

tuple_items = ()
total_items = int(input("Enter the total number of items: "))
for i in range(total_items):
 user_input = int(input("Enter a number: "))
 tuple_items += (user_input,)
print(f"Items added to tuple are {tuple_items}")
list_items = []
total_items = int(input("Enter the total number of items: "))
for i in range(total_items):
 item = input("Enter an item to add: ")
 list_items.append(item)
items_of_tuple = tuple(list_items)
print(f"Tuple items are {items_of_tuple}")

```

```

Enter the total number of items: 5
Enter a number: 8
Enter a number: 2
Enter a number: 9
Enter a number: 1
Enter a number: 6
Items added to tuple are (8, 2, 9, 1, 6)
Enter the total number of items: 4
Enter an item to add: 2
Enter an item to add: 4
Enter an item to add: 6
Enter an item to add: 8
Tuple items are ('2', '4', '6', '8')

```

Items are inserted into the tuple using two methods: using continuous concatenation += operator and by converting list items to tuple items. In the code, tuple\_items is of tuple type. In both the methods, the total number of items are specified which will be inserted to the tuple beforehand. Based on this number, the for loop is iterated using the range() function. In the first method, the user entered items are continuously concatenated to the tuple using += operator. Tuples are immutable and are not supposed to be changed. During each iteration, each original\_tuple is replaced by original\_tuple + (new\_element), thus creating a new tuple. Notice a comma after new\_element. In the second method, a list is created. For each iteration, the user entered value is appended to the list\_variable. This list is then converted to tuple type using tuple() function.

**Program 10.3 : Program to Swap Two Numbers without Using Intermediate/Temporary Variable. Prompt the User for Input.**

1. a = int(input("Enter a value for the first number "))
2. b = int(input("Enter a value for the second number "))
3. b, a = a, b
4. print("After Swapping")
5. print(f"Value for first number {a}")
6. print(f"Value for second number {b}")



Fig. 3 : Screen Shot of Program 10.3

```
a = int(input("Enter a value for the first number "))
b = int(input("Enter a value for the second number "))
b, a = a, b
print("After Swapping")
print(f"Value for first number {a}")
print(f"Value for second number {b}")
```

```
Enter a value for the first number 5
Enter a value for the second number 9
After Swapping
Value for first number 9
Value for second number 5
```

The contents of variables a and b are reversed. The tuple variables are on the left side of the assignment operator and, on the right side, are the tuple values. The number of variables on the left and the number of values on the right has to be the same. Each value is assigned to its respective variable.

**Check your progress – 7 :**

Q36. A python tuple can also be created without using parentheses

- a) False
- b) True

Q37. What is the output of the following?

```
aTuple = "Yellow", 20, "Red"
```

```
a, b, c = aTuple
```

```
print(a)
```

- a) ('Yellow', 20, 'Red')
- b) TypeError
- c) Yellow

Q38. Choose the correct way to access value 20 from the following tuple

```
aTuple = ("Orange", [10, 20, 30], (5, 15, 25))
```

- a) aTuple[1:2][1]
- b) aTuple[1:2](1)
- c) aTuple[1:2][1]
- d) aTuple[1][1]

Q39. Select which is true for python tuple

- a) A tuple maintains the order of items
- b) A tuple is unordered
- c) we cannot change the tuple once created
- d) we can change the tuple once created

Q40. What will be the output of below python code:

```
tuple1=(2, 4, 3)
```

```
tuple2=tuple1*2
```

```
print(tuple2)
```

- a) (4, 8, 6)
- b) (2, 4, 3, 2, 4, 3)
- c) (2, 2, 4, 4, 3, 3)
- d) Error

Q41. What will be the output of below python code:

```
tupl = ("annie", "hena", "sid")
```

```
print(tupl[-3:0])
```

- a) ("annie")
- b) ()
- c) None
- d) Error as slicing is not possible in tuple

Q42. Which of the following options will not result in an error when performed on tuples in python where tupl = (5, 2, 7, 0, 3)?

- a) tupl[1] = 2
- b) tupl.append(2)
- c) tupl1 = tupl + tupl
- d) tupl.sort()

### 10.8.3 Dictionaries

In the real world, you have seen your Contact-list in your phone. It is practically impossible to memorize the mobile number of everyone you come across. In the Contact-list, you store the name of the person as well as his number. This allows you to identify the mobile number based on a person's name. One can think of a person's name as the key that retrieves his mobile number, which is the value associated with the key. So, dictionary can be thought of :

- an un-ordered collection of key-value pairs, with the requirement that the keys are unique within a dictionary.
- as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.
- has a length, specifically the number of key-value pairs.
- provides fast look up by key.

The keys of the dictionary must be immutable object types and are case sensitive. Keys can be either a string or a number. But lists can not be used as keys. A value in the dictionary can be of any data type including string, number, list or dictionary itself.

Dictionaries are constructed using curly braces {}, wherein a list of key:value pairs get separated by commas. There is a colon(:) separating each of these keys and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values.

The syntax for creating a dictionary is :

```
dictionary_name = {key_1:value_1, key_2:value_2, key_3:value_3,
, key_n : value_n}
```

```
In[]: eng2sp = {'one' : 'uno', 'two' : 'dos', 'three' : 'tres'}
```

```
In[]: eng2sp
```

```
Out[]: {'one' : 'uno', 'two' : 'dos', 'three' : 'tres'}
```

With Python 3.6 version, the output of dictionary statements is ordered key:value pairs. Here, ordered means “insertion ordered”, i.e, dictionaries remember the order in which the key:value pairs were inserted. The elements of a dictionary are never indexed with integer indices. Instead, the keys are used to look up the corresponding values:

```
In[]: eng2sp['two']
```

```
Out[]: 'dos'
```

The key ‘two’ always maps to the value ‘dos’ so the order of the items doesn’t matter. If the key isn’t in the dictionary, one get an exception:

```
In[]: eng2sp['four']
```

```
Out[]: KeyError: 'four'
```

Slicing in dictionaries is not allowed since they are not ordered like lists.

### 10.8.3.1 Built –In Functions Used on Dictionaries

There are many built-in functions for which a dictionary can be passed as an argument. The main operations on a dictionary are storing a value with some key and extracting the value for a given key.

**TABLE 15: Built-In Functions Used on Dictionaries**

| Built – in Functions | Description                                                                                                  |
|----------------------|--------------------------------------------------------------------------------------------------------------|
| len()                | The len() function returns the number of items (key:value pairs) in a dictionary.                            |
| all()                | The all() function returns Boolean True value if all the keys in the dictionary are True else returns False. |
| any()                | The any() function returns Boolean True value if any                                                         |

|                       |                                                                                                                   |
|-----------------------|-------------------------------------------------------------------------------------------------------------------|
|                       | of the key in the dictionary is True else returns False.                                                          |
| <code>sorted()</code> | The <code>sorted()</code> function by default returns a list of items, which are sorted based on dictionary keys. |

```
In[: len(eng2sp)
```

```
Out[: 3
```

`len()` function can be used to find the number of key:value pairs in the dictionary `eng2sp`. In Python, any non-zero integer value is True, and zero is interpreted as False. With `all()` function, if all the keys are Boolean True values, then the output is True else it is False.

```
In[: dict_func = {0 : True, 1 : False, 4 : True}
```

```
In[: all(dict_func)
```

```
Out[: False
```

For `any()` function, if any one of the keys is True then it results in a True Boolean value else False Boolean value.

```
In[: any(dict_func)
```

```
Out[: True
```

The `sorted()` function returns the sorted list of keys by default in ascending order without modifying the original key:value pairs. For list of keys sorted in descending order by passing the second argument as `reverse = True`.

```
In[: sorted(eng2sp)
```

```
Out[: ['one', 'three', 'two']
```

```
In[: sorted(eng2sp, reverse=True)
```

```
Out[: ['two', 'three', 'one']
```

The `in` operator works on dictionaries; it tells whether something appears as a key in the dictionary (appearing as a value is not good enough).

```
In[: 'one' in eng2sp
```

```
Out[: True
```

```
In[: 'uno' in eng2sp
```

```
Out[: False
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As a list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python

uses an algorithm called a hash table, so, the in operator takes about the same amount of time no matter how many items there are in a dictionary.

### 10.8.3.2 Dictionary Methods

Various methods associated with dictionary are listed below:

**TABLE 16: Various Dictionary Methods**

| Dictionary Methods | Syntax                                     | Description                                                                                                                                                                                                                                                                                    |
|--------------------|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| clear()            | dictionary_name.clear()                    | The clear() method removes all the key:value pairs from the dictionary.                                                                                                                                                                                                                        |
| fromkeys()         | dictionary_name.fromkeys(seq[, value])     | The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.                                                                                                                                                                          |
| get()              | dictionary_name.get(key[, default])        | The get() method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to None, so that this method never raises a KeyError.                                                |
| items()            | dictionary_name.items()                    | The items() method returns a new view of dictionary's key and value pairs as tuples.                                                                                                                                                                                                           |
| keys()             | dictionary_name.keys()                     | The keys() method returns a new view consisting of all the keys in the dictionary.                                                                                                                                                                                                             |
| pop()              | dictionary_name.pop(key[, default])        | The pop() method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If the default is not given and the key is not in the dictionary, then it results in KeyError.                                                       |
| popitem()          | dictionary_name.popitem()                  | The popitem() method removes and returns an arbitrary (key, value) tuple pair from the dictionary. If the dictionary is empty, then calling popitem() results in KeyError.                                                                                                                     |
| setdefault()       | dictionary_name.setdefault(key[, default]) | The setdefault() method returns a value for the key present in the dictionary. If the key is not present, then insert the key into the dictionary with a default value and return the default value. If key is present, default defaults to None, so that this method never raises a KeyError. |
| update()           | dictionary_name.update([o                  | The update() method updates the dictionary with the key:value pairs from other dictionary object and it                                                                                                                                                                                        |

|          |                          |                                                                                        |              |
|----------|--------------------------|----------------------------------------------------------------------------------------|--------------|
|          | ther])                   | returns None.                                                                          | Data<br>Cont |
| values() | dictionary_name.values() | The values() method returns a new view consisting of all the values in the dictionary. |              |

Note: Replace the word “dictionary\_name” mentioned in the syntax with the actual dictionary name.

```
In[]: million_dollar = {"sanju" : 2018, "tiger zindahai" : 2017,
 "baahubali 2" : 2017, "dangal" : 2016, "bajrangibhaijaan" : 2015}
```

```
In[]: bolwd_million_dollar =
million_dollar.fromkeys(million_dollar, "1,00,00,000")
```

```
In[]: bolwd_million_dollar
```

```
Out[]: {'sanju': '1,00,00,000',
 'tiger zindahai': '1,00,00,000',
 'baahubali 2': '1,00,00,000',
 'dangal': '1,00,00,000',
 'bajrangibhaijaan': '1,00,00,000'}
```

```
In[]: million_dollar.get("bahubali 1")
```

```
In[]: million_dollar.get("bahubali 1", 2015)
```

```
Out[]: 2015
```

```
In[]: million_dollar.keys()
```

```
Out[]: dict_keys(['sanju', 'tiger zindahai', 'baahubali 2', 'dangal',
 'bajrangibhaijaan'])
```

```
In[]: million_dollar.values()
```

```
Out[]: dict_values([2018, 2017, 2017, 2016, 2015])
```

```
In[]: million_dollar.items()
```

```
Out[]: dict_items([('sanju', 2018), ('tiger zindahai', 2017),
 ('baahubali 2', 2017), ('dangal', 2016), ('bajrangibhaijaan', 2015)])
```

```
In[]: million_dollar.update({"bahubali 1" : 2015})
```

```
In[]: million_dollar
```

```
Out[15]:
```

```
{'sanju': 2018,
 'tiger zindahai': 2017,
```

'baahubali 2': 2017,  
'dangal': 2016,  
'bajrangibhaijaan': 2015,  
'bahubali 1': 2015}

One of the common ways of populating dictionaries is to start with an empty dictionary {}, then use the **update()** method to assign a value to the key using assignment operator. If the key doesnot exist, then the key:value pairs will be created automatically and added to the dictionary.

```
In[]: states = {}
```

```
In[]: states.update({"Haryana":"Chandigarh"})
```

```
In[]: states.update({"Bihar":"Patna"})
```

```
In[]: states.update({"west bengal" : "kolkata"})
```

```
In[]: states
```

```
Out[23]: {'Haryana': 'Chandigarh', 'Bihar': 'Patna', 'west
bengal': 'kolkata'}
```

If a dictionary is used as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
In[] : for key in states :
 print(key, states[key])
```

```
Out[]:
```

```
Haryana Chandigarh
```

```
Bihar Patna
```

```
west bengalkolkata
```

**Program 10.4 : Write a Program that Accepts a Sentence and Calculate the Number of Digits, Uppercase and Lowercase Letters**

1. sentence = input("Enter a sentence : ")
2. construct\_dictionary = {"digits":0, "lowercase":0, "uppercase":0}
3. for each\_character in sentence :
4.     if each\_character.isdigit() :
5.         construct\_dictionary["digits"] += 1
6.     elif each\_character.isupper() :
7.         construct\_dictionary["uppercase"] += 1
8.     elif each\_character.islower() :
9.         construct\_dictionary["lowercase"] += 1
10. print("The number of digits, lowercase and uppercase letters are")

11. print(construct\_dictionary)

Data  
Cont

Fig. 4: Screen Shot of execution of Program 10.4

```
sentence = input("Enter a sentence : ")
construct_dictionary = {"digits":0, "lowercase":0, "uppercase":0}
for each_character in sentence :
 if each_character.isdigit() :
 construct_dictionary["digits"] += 1
 elif each_character.isupper() :
 construct_dictionary["uppercase"] += 1
 elif each_character.islower() :
 construct_dictionary["lowercase"] += 1
print("The number of digits, lowercase and uppercase letters are")
print(construct_dictionary)
```

```
Enter a sentence : Ask not what your country can do for you; ask what you can do for your country, John Kennedy, 1961
The number of digits, lowercase and uppercase letters are
{'digits': 4, 'lowercase': 69, 'uppercase': 3}
```

To delete the key:value pair, use the **del** statement followed by the name of the dictionary along with the key you want to delete.

```
In[]: del states["west bengal"]
```

```
In[]: states
```

```
Out[]: {'Haryana': 'Chandigarh', 'Bihar' : 'Patna'}
```

### 10.8.3.3 Relation between Tuples and Dictionaries

Tuples can be used as key:value pairs to build dictionaries.

```
In[]: pm_year =
(('jln',1947),('lbs',1964),('ig',1966),('rg',1984),('pvn',1991),('abv',1998),('nm',2014))
```

```
In[]: pm_year
```

```
Out[]:
```

```
('jln', 1947),
```

```
('lbs', 1964),
```

```
('ig', 1966),
```

```
('rg', 1984),
```

```
('pvn', 1991),
```

```
('abv', 1998),
```

```
('nm', 2014))
```

```
In[]: pm_year_dict = dict(pm_year)
```

```
In[]: pm_year_dict
```

```
Out[]:
```



```
{'jln': 1947,
'lbs': 1964,
'ig': 1966,
'rg': 1984,
'pvn': 1991,
'abv': 1998,
'nm': 2014}
```

The tuples can be converted to dictionaries by passing the tuple name to the **dict()** function. This is achieved by nesting tuples within tuples, wherein each nested tuple item should have two items in it. The first item becomes the key and the second item as its value when the tuple gets converted to a dictionary.

#### Check your progress - 8:

Q43. In Python, Dictionaries and its keys both are immutable.

- a) True, True
- b) True, False
- c) False, True
- d) False, False

Q44. Select correct ways to create an empty dictionary:

- a) sampleDict = {}
- b) sampleDict = dict()
- c) sampleDict = dict{}

Q45. Items are accessed by their position in a dictionary and all the keys in a dictionary must be of the same type.

- a) True
- b) False

Q46. To obtain the number of entries in dictionary, d which command do we use?

- a) d.size()
- b) len(d)
- c) size(d)
- d) d.len()

Q47. Which one of the following is correct?

- a) In python, a dictionary can have two same keys with different values.
- b) In python, a dictionary can have two same values with different keys.
- c) In python, a dictionary can have two same keys or same values but cannot have two same key-value pair.
- d) In python, a dictionary can neither have two same keys nor two same

values.

Q48. What will be the output of above Python code?

```
d1={"abc":5,"def":6,"ghi":7}
```

```
print(d1[0])
```

- a) abc
- b) 5
- c) {"abc":5}
- d) Error

Q49. What will the above Python code do?

```
dict={"Phy":94,"Che":70,"Bio":82,"Eng":95}
```

```
dict.update({"Che":72,"Bio":80})
```

- a) It will create new dictionary as dict={"Che":72,"Bio":80} and old dict will be deleted.
- b) It will throw an error as dictionary cannot be updated.
- c) It will simply update the dictionary as dict={"Phy":94,"Che":72,"Bio":80,"Eng":95}
- d) It will not throw any error but it will not do any changes in dict

Q50. Select all correct ways to remove the key 'marks' from a dictionary

```
student = {
 "name": "Emma",
 "class": 9,
 "marks": 75
}
```

- a) student.pop("marks")
- b) del student["marks"]
- c) student.popitem()
- d) dict1.remove("key2")

Q51. Select all correct ways to copy a dictionary in Python

- a) dict2 = dict1.copy()
- b) dict2 = dict(dict1.items())
- c) dict2 = dict(dict1)
- d) dict2 = dict1

### 10.8.4Sets

A **set** is an unordered collection with no duplicate items. Primary uses of sets include membership testing and eliminating duplicate entries. Sets also support mathematical operations, such as union, intersection, difference, and symmetric difference.

Curly braces{} or the set() function can be used to create sets with a comma-separated list of items inside curly brackets{}. Note: to create an empty set you have to use set() and not{} as the latter creates an empty dictionary.

Sets are mutable. Indexing is not possible in sets, since set items are unordered. One cannot access or change an item of the set using indexing or slicing.

```
In[]: basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

```
In[]: basket
```

```
Out[]: {'apple', 'banana', 'orange', 'pear'}
```

A set is a collection of unique items. Duplicate items are removed from the set basket. Here, the set will contain only one item of 'orange' and 'apple'.

```
In[]: 'orange' in basket
```

```
Out[]: True
```

```
In[]: 'grapes' in basket
```

```
Out[]: False
```

One can test for the presence of an item in a set using **in** and **not in** membership operators.

```
In[]: len(basket)
```

```
Out[]: 4
```

```
In[]: sorted(basket)
```

```
Out[]: ['apple', 'banana', 'orange', 'pear']
```

Total number of items in the set basket is found using the `len()` function. The `sorted()` function returns a new sorted list from items in the set.

```
In[]: a = set('abracadabra')
```

```
In[]: a
```

```
Out[]: {'a', 'b', 'd', 'k', 'r'}
```

```
In[]: b = set('alexander')
```

```
In[]: b
```

```
Out[]: {'a', 'd', 'e', 'l', 'n', 'r', 'x'}
```

```
In[]: a-b #letters present in set a, but not in set b
```

```
Out[]: {'b', 'k'}
```

```
In[]: a | b #Letters present in set a, set b, or both
```

```
Out[]: {'a', 'b', 'd', 'e', 'k', 'l', 'n', 'r', 'x'}
```

```
In[]: a & b #letters present in both set a and set b
```

```
Out[]: {'a', 'd', 'r'}
```

```
In[]: a ^ b #letters present in set a or set b, but not both
```

```
Out[]: {'b', 'e', 'k', 'l', 'n', 'x'}
```

#### 10.8.4.1 Set Methods

A list of all the methods associated with the set can be obtained by passing the set function to dir().

Various methods associated with set are listed in the TABLE.

**TABLE 17 : Various Set Methods**

| Set Methods    | Syntax                         | Description                                                                                                                                                           |
|----------------|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| add()          | set_name.add(item)             | The add() method adds an item to the set set_name.                                                                                                                    |
| clear()        | set_name.clear()               | The clear() method removes all the items from the set set_name.                                                                                                       |
| difference()   | set_name.difference(*others)   | The difference() method returns a new set with items in the set set_name that are not in the others sets.                                                             |
| discard()      | set_name.discard(item)         | The discard() method removes an item from the set set_name if it is present.                                                                                          |
| intersection() | set_name.intersection(*others) | The intersection() method returns a new set with items common to the set set_name and all other sets.                                                                 |
| isdisjoint()   | set_name.isdisjoint(other)     | The isdisjoint() method returns True if the set set_name has no items in common with other set. Sets are disjoint if and only if their intersection is the empty set. |
| issubset()     | set_name.issubset(other)       | The issubset() method returns True if every item in the set set_name is in the other set.                                                                             |
| issuperset()   | set_name.issuperset(other)     | The issuperset() method returns True if every element in other set is in the set set_name.                                                                            |
| pop()          | set_name.pop()                 | The method pop() removes and returns an arbitrary item from the set set_name. It raises KeyError if the set is empty.                                                 |
| remove()       | set_name.remove(item)          | The method remove() removes an item from the set set_name. It raises KeyError if                                                                                      |

|                                     |                                                   |                                                                                                                                    |
|-------------------------------------|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
|                                     |                                                   | the item is not contained in the set.                                                                                              |
| <code>symmetric_difference()</code> | <code>set_name.symmetric_difference(other)</code> | The method <code>symmetric_difference()</code> returns a new set with items from the set <code>set_name</code> and all other sets. |
| <code>union()</code>                | <code>set_name.union(*others)</code>              | The method <code>union()</code> returns a new set with items from the set <code>set_name</code> and all others sets.               |
| <code>update()</code>               | <code>set_name.update(*others)</code>             | Update the set <code>set_name</code> by adding items from all others sets                                                          |

Note: Replace the words “set\_name”, “other” and “others” mentioned in the syntax with your actual set names in your code.

```
In[: flowers1 = {'sunflower', 'roses', 'lavender', 'tulips',
'goldcrest'}
```

```
In[: flowers2 = {'roses', 'tulips', 'lilies', 'daisies'}
```

```
In[: flowers2.add('orchids')
```

```
In[: flowers2.difference(flowers1)
```

```
Out[: {'daisies', 'lilies', 'orchids'}
```

```
In[: flowers2.intersection(flowers1)
```

```
Out[: {'roses', 'tulips'}
```

```
In[: flowers2.isdisjoint(flowers1)
```

```
Out[: False
```

```
In[: flowers2.issuperset(flowers1)
```

```
Out[: False
```

```
In[: flowers2.issubset(flowers1)
```

```
Out[: False
```

```
In[: flowers2.symmetric_difference(flowers1)
```

```
Out[: {'daisies', 'goldcrest', 'lavender', 'lilies', 'orchids',
'sunflower'}
```

```
In[: flowers2.union(flowers1)
```

```
Out[:
```

```
{'daisies',
```

```
'goldcrest',
```

```
'lavender',
```

```
'lilies',
'orchids',
'roses',
'sunflower',
'tulips'}
```

```
In[]: flowers2.update(flowers1)
```

```
Out[]: flowers2
```

```
Out[]:
{ 'daisies',
 'goldcrest',
 'lavender',
 'lilies',
 'orchids',
 'roses',
 'sunflower',
 'tulips' }
```

```
In[]: flowers2.discard("roses")
```

```
In[]: flowers2
```

```
Out[]:
{ 'daisies',
 'goldcrest',
 'lavender',
 'lilies',
 'orchids',
 'sunflower',
 'tulips' }
```

```
In[]: flowers1.pop()
```

```
Out[]: 'roses'
```

```
In[]: flowers2.clear()
```

```
In[]: flowers2
```

```
Out[]: set()
```

#### 10.8.4.2 Traversing of Sets

One can iterate through each item in a set using a for loop.

```
In[]: for flower in flowers1 :
```

```
print(f"{flower} is a flower")
```

```
Out[]: lavender is a flower
```

```
tulips is a flower
```

```
goldcrest is a flower
```

```
sunflower is a flower
```

#### 10.8.4.3Frozenset

A **frozenset** is basically the same as a set, except that it is immutable. Once a frozenset is created, its items cannot be changed. Since they are immutable, they can be used as members in other sets and as dictionary keys. The frozensets have the same functions as normal sets, except none of the functions that change the contents (update, remove, pop, etc.) are available.

```
In[]: fs = frozenset(["g","o","o","d"])
#creating/declaring frozenset

In[]: fs
of frozenset #displaying the contents

Out[]: frozenset({'d', 'g', 'o'})

In[]: pets = set([fs, "horse", "cow"])
is used within a set #Frozenset type

In[]: pets

Out[]: {'cow', frozenset({'d', 'g', 'o'}), 'horse'}

In[]: lang_used = {"english":59, "french":29, "spanish":21}

In[]: frozenset(lang_used)
#keys in a dictionary are
returned when a dictionary is
passed as an argument to
frozenset() function.

Out[]: frozenset({'english', 'french', 'spanish'})

In[]: frs = frozenset(["german"])

#Frozenset is used as a key in dictionary

In[]: lang_used = {"english":59, "french":29, "spanish":21, frs:6}

In[]: lang_used

Out[]: {'english': 59, 'french': 29, 'spanish': 21, frozenset({'german'}): 6}
```

|                          |
|--------------------------|
| Check your progress - 9: |
|--------------------------|

Q52. Select which is true for Python set:

- a) Sets are unordered.
- b) Set doesn't allow duplicate
- c) sets are written with curly brackets {}
- d) set Allows duplicate
- e) set is immutable
- f) a set object does support indexing

Q53. What is the output of the following union operation?:

```
set1 = {10, 20, 30, 40}
```

```
set2 = {50, 20, "10", 60}
```

```
set3 = set1.union(set2)
```

```
print(set3)
```

- a) {40, 10, 50, 20, 60, 30}
- b) {40, '10', 50, 20, 60, 30}
- c) {40, 10, '10', 50, 20, 60, 30}
- d) SyntaxError: Different types cannot be used with sets

Q54. What is the output of the following set operation?:

```
set1 = {"Yellow", "Orange", "Black"}
```

```
set2 = {"Orange", "Blue", "Pink"}
```

```
set3 = set2.difference(set1)
```

```
print(set3)
```

- a) {'Yellow', 'Black', 'Pink', 'Blue'}
- b) {'Pink', 'Blue'}
- c) {'Yellow', 'Black'}

Q55. What is the output of the following set operation?:

```
set1 = {"Yellow", "Orange", "Black"}
```

```
set1.discard("Blue")
```

```
print(set1)
```

- a) {'Yellow', 'Orange', 'Black'}
- b) KeyError: 'Blue'

Q56. The isdisjoint() method returns True if none of the items are present in both sets, otherwise, it returns False.

- a) True
- b) False

Q57. Select all the correct ways to copy two sets

- a) set2 = set1.copy()
- b) set2 = set1
- c) set2 = set(set1)
- d) set2.update(set1)



Q58. The `symmetric_difference()` method returns a set that contains all items from both sets, but not the items that are present in both sets.

- a) False
- b) True

## 10.9 CONTROL FLOW STATEMENTS

Python supports a set of control flow statements that you can integrate into your program. The statements inside your Python program are generally executed sequentially from top to bottom in the order that they appear. Apart from sequential control flow statements, you can employ decision making and looping control flow statements to break up the flow of execution thus enabling your program to conditionally execute particular block of code. The term control flow details the direction the program takes.

The control flow statements in Python Programming Language are:

**10.9.1 Sequential Control Flow Statements :** This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.

**10.9.2 Decision Control Flow Statements :** Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if...else and if...elif...else).

**10.9.3 Loop Control Flow Statements :** This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (for loop and while loop). Loop Control Flow statements are also called Repetition statements or Iteration Statements.

### 10.9.2.1 The if Decision Control Flow Statement

In order to write useful programs, we almost need the ability to check conditions and change the behavior of the program accordingly. The Decision Control Flow Statements give us this ability. The simplest form is the if statement.

The syntax for if statement is:

```
if Boolean_Expression :
 statement (s)
```

The Boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented. The if statement decides whether to run statement (s) or not depending upon the value of the Boolean expression. If the Boolean expression evaluates to True then indented statements in the if block will be executed, otherwise if the result is False then none of the statements are executed. E.g.,

If  $x > 0$  :

is  $x > 0$ ?

Yes

print ( 'x is positive' )

print('x is positive')

**Fig. 5: If Logic**

Here, depending on the value of  $x$ , print statement will be executed i.e. only if  $x > 0$ .

There is no limit on the number of statements that can appear in the body, but there must be at least one. In Python, the if block statements are determined through indentation and the first unindented statement marks the end. You don't need to use the `==` operator explicitly to check if the variable's value evaluates to True as the variable name can itself be used as a condition.

**Program 10.5: Program reads your age and prints a message whether you are eligible to vote or not???**

1. `age = int(input("Enter your age : "))`
2. `if age >= 18 :`
3.     `print ("Congratulations!!! you can vote")`
4.     `print ("!!! Thanks !!!")`
5.     `print ("Voting is your birth right. Use judiciously")`

Fig. 6: Screen Shot of execution of Program 10.5

```
age = int(input("Enter your age : "))
if age >= 18 :
 print ("Congratulations!!! you can vote")
 print ("!!! Thanks !!!")
 print ("Voting is your birth right. Use judiciously")

Enter your age : 22
Congratulations!!! you can vote
!!! Thanks !!!
Voting is your birth right. Use judiciously
```

Here, condition or Boolean expression is true, indented block would be executed

```
Enter your age : 17
Voting is your birth right. Use judiciously
```

Here, condition is False, so indented block would not be executed and only the unindented statement after the if block would be executed.

### 10.9.2.2 The if...else Decision Control Flow Statement

An if statement can also be followed by an else statement which is optional. An else statement does not have any condition. Statements in the if block are

executed if the Boolean\_Expression is True. Use the optional else block to execute statements if the Boolean\_Expression is False. The if...else statements allow for a two – way decision.

The syntax for if...else statement is:

```
if Boolean_Expression :
 statement_blk_1
else
 statement_blk_2
```

If the Boolean\_Expression evaluates to True, then statement\_blk\_1 (single or multiple statements) is executed, otherwise it is evaluated to False then statement\_blk\_2 (single or multiple statements) is executed. Indentation is used to separate the blocks. After the execution of either statement\_blk\_1 or statement\_blk\_2, the control is transferred to the next statement after the if statement. Also, if and else keywords should be aligned at the same column position.

**Program 10.6 : Program to find if a given number is odd or even**

1. num = int(input("Enter a number : "))
2. if num % 2 == 0 :
3. print (num," is even number")
4. Else:
5. print (num," is odd number")

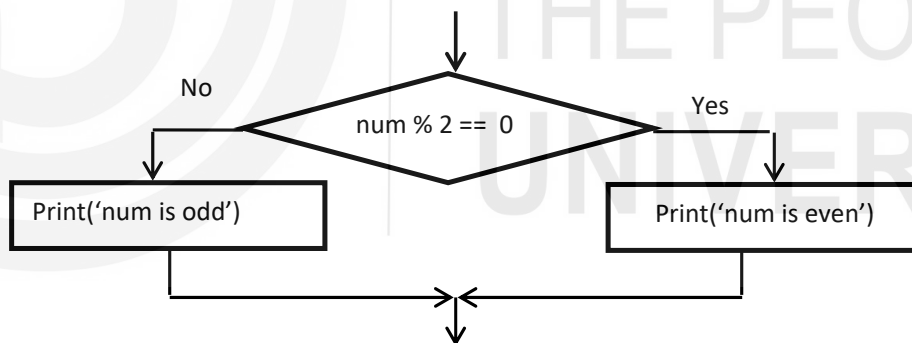


Fig. 6: IF – ELSE LOGIC

A number is read and stored in the variable named num. The num is checked using modulus operator to determine whether the num is perfectly divisible by 2 or not. Num entered is 15 which makes the evaluated expression False, so the else statement is executed and num is odd.

Fig. 7: Screen Shot of execution of Program 10.6

```

num = int(input("Enter a number : "))
if num % 2 == 0:
 print (num," is even number")
else:
 print (num," is odd number")

```

```

Enter a number : 15
15 is odd number

```

### 10.9.2.3 The if...elif...else Decision Control Statement

The if...elif...else is also called as multi-way decision control statement. When you need to choose from several possible alternatives, an elif statement is used along with an if statement. The keyword 'elif' is short for 'else if' and is useful to avoid excessive indentation. The else statement must always come last, and will again act as the default action.

The syntax for if...elif...else statement is,

```

if Boolean_Expression_1 :
 statement_blk_1
elif Boolean_Expression_2 :
 statement_blk_2
elif Boolean_Expression_3 :
 statement_blk_3
:
:
:
else :
 statement_blk_last

```

This if...elif...else decision control statement is executed as follows:

- In the case of multiple Boolean expression, only the first logical Boolean expression which evaluates to True will be executed.
- If Boolean\_Expression\_1 is True, then statement\_blk\_1 is executed.
- If Boolean\_Expression\_1 is False and Boolean\_Expression\_2 is True, then statement\_blk\_2 is executed.
- If Boolean\_Expression\_1 and Boolean\_Expression\_2 is False and Boolean\_Expression\_3 is True, then statement\_blk\_3 is executed and so on.
- If none of the Boolean\_Expression is True, then statement\_blk\_last is executed.

**Program 10.7 : Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error. If the score is between 0.0 and 1.0, print a grade using the following table**

| Score      | Grade |
|------------|-------|
| $\geq 0.9$ | A     |
| $\geq 0.8$ | B     |
| $\geq 0.7$ | C     |
| $\geq 0.6$ | D     |
| $< 0.6$    | F     |

```

1. score = float(input("Enter your score : "))
2. if score < 0 or score > 1 :
3. print('Wrong Input')
4. elif score >= 0.9 :
5. print('Your Grade is "A"')
6. elif score >= 0.8 :
7. print('Your Grade is "B"')
8. elif score >= 0.7 :
9. print('Your Grade is "C"')
10. elif score >= 0.6 :
11. print('Your Grade is "D"')
12. else :
13. print('Your Grade is "F"')

```

if score < 0  
or score > 1      Yes      Wrong Input'

score >= 0.9      Yes      Your Grade is "A"

score >= 0.8      Yes      Your Grade is "B"

Fig. 9: Screen Shot of execution of Program 10.7

```

score = float(input("Enter your score : "))
if score < 0 or score > 1 :
 print('Wrong Input')
elif score >= 0.9 :
 print('Your Grade is "A"')
elif score >= 0.8 :
 print('Your Grade is "B"')
elif score >= 0.7 :
 print('Your Grade is "C"')
elif score >= 0.6 :
 print('Your Grade is "D"')
else :
 print('Your Grade is "F"')

```

>= 0.7      Yes      Your Grade is "C"

>= 0.6      Yes      Your Grade is "D"

le is "F"

Enter your score : 0.82  
Your Grade is "B"

Fig.8 : IF – ELIF - ELSE LOGIC

#### 10.9.2.4 Nested if Statement

In some situations, you have to place an if statement inside another statement. An if statement that contains another if statement either in its if block or else block is called a Nested if statement.

The syntax of the nested if statement is,

```

if Boolean_Expression_1 :
 if Boolean_Expression_2 :
 statement_blk_1
 else :
 statement_blk_2
else :
 statement_blk_3

```

If the Boolean\_Expression\_1 is evaluated to True, then the control shifts to Boolean\_Expression\_2 and if the expression is evaluated to True, then statement\_blk\_1 is executed. If the Boolean\_Expression\_2 is evaluated to False then the statement\_blk\_2 is executed. If the Boolean\_Expression\_1 is evaluated to False, then the statement\_blk\_3 is executed.

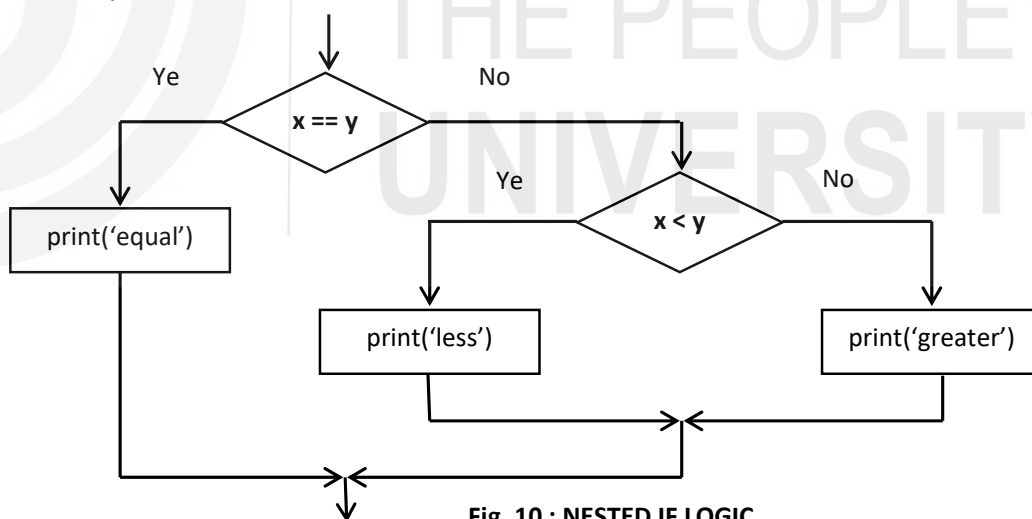
A three – branch example could be written as :

```

if x == y :
 print('x and y are equal')
else :
 if x < y :
 print('x is less than y')
 else :
 print('x is greater than y')

```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.



**Fig. 10 : NESTED IF LOGIC**

#### **PROGRAM 10.8 : Program to check if a given year is a leap year**

```

1. year = int(input('Enter a year : '))
2. if year % 4 == 0 :
3. if year % 100 == 0 :
4. if year % 400 == 0 :
5. print(f'{year} is a Leap Year')
6. else :
7. print(f'{year} is not a Leap Year')

```

```

8. else :
9. print(f' {year} is a Leap Year')
10. else :
11. print(f' {year} is not a Leap Year')

```

Fig. 11: Screen Shot of execution of Program 10.8

```

year = int(input('Enter a year : '))
if year % 4 == 0:
 if year % 100 == 0 :
 if year % 400 == 0 :
 print(f'{year} is a Leap Year')
 else :
 print(f'{year} is not a Leap Year')
 else :
 print(f'{year} is a Leap Year')
else :
 print(f'{year} is not a Leap Year')

```

```

Enter a year : 2014
2014 is not a Leap Year

```

All years which are perfectly divisible by 4 are leap years except for century years (years ending with 00) which is a leap year only if it is perfectly divisible by 400. For example, years like 2012, 2004, 1968 are leap years but 1971, 2006 are not leap years. Similarly, 1200, 1600, 2000, 2400 are leap years but 1700, 1800, 1900 are not.

Although the indentation of the statements makes the structure apparent, nested conditional becomes difficult to read very quickly. In general, it is a good idea to avoid them when you can.

### 10.9.3.1 The while loop

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making error is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the while statement. The syntax for while loop is :

```

while Boolean_Expression :
 statement(s)

```

**Program 10.9 :** Program that counts down from five and then says “Blastoff!”

```

1. n = 5
2. while n > 0 :

```

3.        print(n)
4.        n = n - 1
5.    print('Blastoff!')

Fig. 12: Screen Shot of execution of Program 10.9

```
n = 5
while n > 0 :
 print(n)
 n = n - 1
print('Blastoff!')
```

```
5
4
3
2
1
Blastoff!
```

You can almost read the while statement as if it were English. It means, “While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!”

More formally, the flow of execution for a while statement is :

1. Evaluate the condition or the Boolean\_Expression, yielding True or False.
2. If the Boolean\_Expression is false, exit the while statement and then continue execution at the next statement.
3. If the Boolean\_Expression is true, execute the body and then go back.

This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. for the above loop, we would say, “It had five iterations”, which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition or the Boolean\_Expression becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

**PROGRAM 10.10 : Program to display the Fibonacci Sequences up to nth term where n is provided by the user**

1. nterms = int(input('How many terms?'))
2. current = 0
3. previous = 1



```

4. next_term = 0
5. count = 0
6. if nterms <= 0 :
7. print('Please enter a positive number')
8. elif nterms == 1 :
9. print('Fibonacci Sequence')
10. print('0')
11. else :
12. print("Fibonacci Sequence")
13. while count < nterms :
14. print(next_term)
15. current = next_term
16. next_term = previous + current
17. previous = current
18. count += 1

```

Fig. 13: Screen Shot of execution of Program 10.10

```

nterms = int(input('How many terms? '))
current = 0
previous = 1
next_term = 0
count = 0
if nterms <= 0 :
 print('Please enter a positive number')
elif nterms == 1 :
 print('Fibonacci Sequence')
 print('0')
else :
 print("Fibonacci Sequence")
 while count < nterms :
 print(next_term)
 current = next_term
 next_term = previous + current
 previous = current
 count += 1

```

```

How many terms? 5
Fibonacci Sequence
0
1
1
2
3

```

In a Fibonacci sequence, the next number is obtained by adding the previous two numbers. The first two numbers of the Fibonacci sequence are 0 and 1. The next number is obtained by adding 0 and 1 which is 1. Again, the next number is obtained by adding 1 and 1 which is 2 and so on. Get a number from user up to which you want to generate Fibonacci sequence. Assign

values to variables `current`, `previous`, `next_term` and `count`. The variable `count` keeps track of number of times the `while` block is executed. User is required to enter a positive number to generate a single number in the sequence, then print zero. The `next_term` is obtained by adding the `previous` and `current` variables and the statements in the `while` block are repeated until `while` block conditional expression becomes `False`.

### 10.9.3.2 The for loop

Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a definite loop using a `for` statement. We call the `while` statement an indefinite loop because it simply loops until some condition becomes `False`, whereas the `for` loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax for the `for` loop is :

**for iteration\_variable in sequence :**

**statement(s)**

The `for` loop starts with `for` keyword and ends with a colon. The first item in the sequence gets assigned to the iteration variable `iteration_variable`. Here, `iteration_variable` can be any valid variable name. Then the statement block is executed. This process of assigning items from the sequence to the `iteration_variable` and then executing the statement continues until all the items in the sequence are completed.

The `for` loop is incomplete without the use of **`range()`** function which is a built-in function. It is very useful in demonstrating `for` loop. The `range()` function generates a sequence of numbers which can be iterated through using `for` loop. the syntax for `range()` function is,

**`range([start ,] stop [, step])`**

Both `start` and `step` arguments are optional and is represented with the help of square brackets and the `range` argument value should always be an integer.

`start` → value indicates the beginning of the sequence. If the `start` argument is not specified, then the sequence of numbers start from zero by default.

`stop` → generates numbers up to this value but not including the number itself.

`step` → indicates the difference between every two consecutive numbers in the sequence. The `step` value can be both negative and positive but not zero.

**PROGRAM 10.11 : Program to find the sum of all odd and even numbers up to a number specified by the user.**

1. `number = int(input("Enter a number"))`
2. `even = 0`

3. odd = 0
4. for i in range(number):
5.     if i % 2 == 0:
6.         even = even + i
7.     else:
8.         odd = odd + i
9. print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")

Fig. 14: Screen Shot of execution of Program 10.11

```
number = int(input("Enter a number"))
even = 0
odd = 0
for i in range(number):
 if i % 2 == 0:
 even = even + i
 else:
 odd = odd + i
print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")
```

Enter a number10  
Sum of Even numbers are 20 and Odd numbers are 25

A range of numbers are generated using range() function. As only stop value is indicated in the range() function, so numbers from 0 to 9 are generated. Then the generated numbers are segregated as odd or even by using the modulus operator in the for loop. All the even numbers are added up and assigned to even variable and odd numbers are added up and assigned to odd variable and print the result. The for loop will be executed / iterated number of times entered by user which is 10 in this case.

**PROGRAM 10.12 : Program to find the factorial of a number.**

1. number = int(input('Enter a number'))
2. factorial = 1
3. if number < 0:
4.     print("Factorial doesn't exist for negative numbers")
5. elif number == 0:
6.     print("The factorial of 0 is 1")
7. else:
8.     for i in range(1, number + 1):
9.         factorial = factorial \* i
10. print(f"The factorial of number {number} is {factorial}")

Fig. 15: Screen Shot of execution of Program 10.12

```

number = int(input('Enter a number'))
factorial = 1
if number < 0 :
 print("Factorial doesn't exist for negative numbers")
elif number == 0 :
 print("The factorial of 0 is 1")
else :
 for i in range(1, number + 1) :
 factorial = factorial * i
 print(f"The factorial of number {number} is {factorial}")

```

```

Enter a number5
The factorial of number 5 is 120

```

Read the number from user. A value of 1 is assigned to variable factorial. To find the factorial of a number it has to be checked for a non – negative integer. If the user entered number is zero then the factorial is 1. To generate numbers from 1 to the user entered number range() function is used. Every number is multiplied with the factorial variable and is assigned to the factorial variable inside the for loop. The for loop block is repeated for all the numbers starting from 1 up to the user entered number. Finally, the factorial value is printed.

### 10.9.3.3 The continue and break Statements

The break and continue statements provide greater control over the execution of code in a loop. Whenever the break statement is encountered, the execution control immediately jumps to the first instruction following the loop. To pass control to the next iteration without exiting the loop, use the continue statement. Both continue and break statements can be used in while and for loops.

#### PROGRAM 10.13 : Program that prints integers from zero to 5.

1. count = 0
2. while True :
3.     count += 1
4.     if count > 5 :
5.         break
6.     print (count)

Fig. 16: Screen Shot of execution of Program 10.13

```
count = 0
while True :
 count += 1
 if count > 5 :
 break
 print (count)
```

```
1
2
3
4
5
```

In the while loop, as soon as count value reaches 6, because of break statement print statement will not be executed and control will come out of the while loop.

**PROGRAM 10.14 : Program that processes only odd integers from 0 to 10.**

1. count = 0
2. while count < 10 :
3. count += 1
4. if count % 2 == 0 :
5. continue
6. print (count)

Fig. 17: Screen Shot of execution of Program 10.14

```
count = 0
while count < 10 :
 count += 1
 if count % 2 == 0 :
 continue
 print (count)
```

```
1
3
5
7
9
```

The continue statement in the while loop will not let the print statement to execute if the count value is even and control will go for next iteration. Here, loop will go through all the iterations and the continue statement will affect only the statements in the loop to be executed or not occurring after the continue statement. Whereas, the break statement will not let the loop to complete its iterations depending on the condition specified and control is transferred to the statement outside the loop.

**PROGRAM 10.15 : Program to check whether a number is prime or not**

```

1. number = int(input('Enter a number: '))
2. prime = True
3. for i in range(2, number) :
4. if number % i == 0 :
5. prime = False
6. break
7. if prime :
8. print(f'{number} is a prime number')
9. else :
10. print(f'{number} is not a prime number')

```

Fig. 18: Screen Shot of Program 10.15

```

number = int(input('Enter a number: '))
prime = True
for i in range(2, number) :
 if number % i == 0 :
 prime = False
 break
if prime :
 print(f"{number} is a prime number")
else :
 print(f"{number} is not a prime number")

Enter a number: 9
9 is a prime number

```

#### Check your progress-10 :

Q59. In a python program, a control structure:

- Defines program-specific data structures
- Directs the order of execution of the statements in the program
- Dictates what happens before the program starts and after it terminates
- None of the above

Q60. Which of the following is False regarding loops in python?

- Loops are used to perform certain tasks repeatedly
- While loop is used when multiple statements are to be executed repeatedly until the given condition becomes False
- While loop is used when multiple statements are to be executed repeatedly until the given condition becomes True.
- for loop can be used to iterate through the elements of lists.

Q61. We can write if/else into one line in python.

- True
- False

Q62. Given the nested if-else below, what will be the value x when the code is executed successfully?

x = 0

```
a = 5
b = 5
if a > 0:
 if b < 0:
 x = x + 5
elif a > 5:
 x = x + 4
else:
 x = x + 3
else:
 x = x + 2
```

print(x)

- a) 0
- b) 4
- c) 2
- d) 3

Q63. if -3 will evaluate to true

- a) True
- b) False

Q64. What is the output of the following nested loop

```
numbers = [10, 20]
```

```
items = ["Chair", "Table"]
```

```
for x in numbers:
```

```
 for y in items:
```

```
 print(x, y)
```

- a) 10 Chair
- 10 Table
- 20 Chair
- 20 Table
- b) 10 Chair
- 20 Chair
- 10 Table
- 20 Table

Q65. What is the value of x?:

```
X = 0
```

```
While (x < 100):
```

```
 x+=2
```

```
print(x)
```

- a) 101
- b) 99
- c) None of the above, this is an infinite loop
- d) 100

Q66. What is the value of the var after the for loop completes its execution?

```
var = 10
for i in range(10):
 for j in range(2, 10, 1):
 if var % 2 == 0:
 continue
 var += 1
 var+=1
else:
 var+=1
print(var)
```

a) 20  
b) 21  
c) 10  
d) 30

Q67. Find the output of the following program:

```
a = {i : i * i for i in range(6)}
print (a)
```

Dictionary comprehension doesn't exist

- a) {0:0, 1:1, 2:4, 3:9, 4:16, 5:25, 6:36}  
b) {0:0, 1:1, 4:4, 9:9, 16:16, 25:25}  
c) {0:0, 1:1, 2:4, 3:9, 4:16, 5:25}

## 10.10

## SUMMARY

After the completion of this chapter, I am sure, you would be able to learn and understand the basics of Python language. Using the contents explained in this chapter, you would be able to turn your logic into codes where identifiers and variables would help you to form statements and expressions using operators, lists, tuples, sets and dictionaries which are the back bone of Python language which makes it unique and easy to program with. All the basic structures of Python can be bound together with the control flow statements which ultimately form a Python program. However, this is not all; there is still a long way to go. Python has many more unique features which make it a programmer's language. So, Happy Programming!



---

## SOLUTIONS TO CHECK YOUR PROGRESS

---

Data Structures and  
Control Statements  
in Python

### Answers to Check your Progress 1 to 10 :

|       |           |           |           |       |       |
|-------|-----------|-----------|-----------|-------|-------|
| 1. a  | 2. d      | 3. d      | 4. a      | 5. c  | 6. d  |
| 7. b  | 8. e      | 9. b      | 10. a     | 11. b | 12. a |
| 13. d | 14. c     | 15. b     | 16. e     | 17. d | 18. a |
| 19. a | 20. d     | 21. b     | 22. c     | 23. c | 24. c |
| 25. b | 26. b     | 27. b     | 28. b     | 29. c | 30. b |
| 31. c | 32. b     | 33. b     | 34. c     | 35. a | 36. b |
| 37. c | 38. d     | 39. a,c   | 40. b     | 41. b | 42. c |
| 43. c | 44. a,b   | 45. b     | 46. b     | 47. b | 48. d |
| 49. c | 50. a,b,c | 51. a,b,c | 52. a,b,c | 53. c | 54. b |
| 55. a | 56. a,c,d | 57. a,c,d | 58. b     | 59. b | 60. b |
| 61. a | 62. d     | 63. a     | 64. a     | 65. d | 66. b |
| 67. d |           |           |           |       |       |



ignou  
THE PEOPLE'S  
UNIVERSITY

---

# UNIT 11      FUNCTIONS AND FILE HANDLING IN PYTHON

---

Functions and Files  
Handling in Python

---

## Structure

- 11.0 Introduction
- 11.1 Objectives
- 11.2 Function definition and calling
- 11.3 Function Scope
- 11.4 Function arguments
- 11.5 Returning from a function
- 11.6 Function objects
- 11.7 Lambda / Anonymous Functions
- 11.8 File Operations
- 11.9 Summary

---

## 11.0      INTRODUCTION

---

Python provides a way of organizing the tasks into more manageable units called functions. Functions make the code modular which is one of the characteristics of an object oriented programming language (OOPs). Modularity is the process of decomposing a problem into set of sub-problems so as to reduce the complexity of a problem. It also makes the code reusable. File handling is another important aspect discussed in this unit. File handling includes- creation, deletion and manipulations of files using python programming.

---

## 11.1      OBJECTIVES

---

After completing this unit, you will be able to

- Perform functions definition and calling
- Understand scope of functions
- Define function objects
- Create lambda functions
- Understand basic file operations

---

## 11.2      FUNCTION      DEFINITION      AND CALLING

---

A function is block of code that performs a specific task. When the size of a program increases, its complexity also increases. Hence, it becomes important to organize the program into more manageable units. Further, it makes the code reusable.

There are three types of functions in python-

1. Built-in functions – these are the functions which are already defined in the language. Example `print()`, `max()`, `input()`, etc.
2. User Defined functions- these are the functions that can be created or defined by the users according to their needs.
3. Anonymous functions

### 11.2.1 Creating user defined functions

A function can be defined using *def* statement and giving suitable name to a function by following the rules of identifiers. The process of defining a function is called *function definition*.

Syntax of function definition

```
Def function_name(list_of_parameters) :
Statement (s)
```

Where,

`function_name()` - is the name of the function. A function name must be followed by a set of parenthesis. Any name can be given to a function following the rules of identifiers.

`List_of_parameters` – is an optional field which is used to pass values or inputs to a function. It can be none or a comma separated list of variables.

`Statements(s)` – is a set of statements or commands within the function. Each time the function is called, all the statements will be executed. These statements together form the body of a function.

Even if the function is defined, it can never be executed till the time it is called. Hence, for using a function, it must be called using *function call statement*. A function can be called by its name with set of parenthesis and optional list of arguments.

Syntax of Function Calling

```
function_name(list_of_arguments)
```

Example1. A function to display HELLO WORLD

```
def first_function(): # function definition
 print("HELLO WORLD...!!!")

first_function() # function calling

===== RESTART: C:/Users/test/Desktop/python_programs/abc.py =====
HELLO WORLD...!!!
>>>
```

Function needs to be defined only once, but it can be called any number of times by using calling statements. They can not only be called in the same program, but they can also be called in different programs. This will further be discussed in the next chapter.

Another example shown below is the program having a function to calculate factorial of a number.

Example2. A function to display factorial of a number.

```
def factorial(): # function definition
 fact=1
 n=int(input("Enter a number :"))
 for i in range(1,n+1):
 fact=fact*i
 print("Factorial of ",n," is ",fact)

factorial() # function calling
```

===== RESTART: C:/Users/test/Desktop/python\_programs/factorial.py =====

```
Enter a number :4
Factorial of 4 is 24
>>>
```

## 11.3 FUNCTION SCOPE

Part of a code in which a variable can be accessed is called *Scope of a variable*. Scope of a variable can be global or local.

**Local variables** are the variable created within a function's body or function's scope. They cannot be accessed outside the function. Their scope is only limited to the function in which they are created.

**Global variables** are the variable created outside any function. Their scope is global, hence can be used anywhere. Global variables can also be created within function using keyword *global*.

Example 3. Use of local and global variables.

```
x=1 # global variable
def test():
 y=2 # local variable
 print("Global inside function x=",x) # both local and global can be used here
 print("Local inside function y=",y)

test()
print("Global outside function x=",x)
print("Local outside function y=",y) # shows error:local variable cannot be used here
```

===== RESTART: C:/Users/test/Desktop/python\_programs/test.py =====

```
Global inside function x= 1
Local inside function y= 2
Global outside function x= 1
Traceback (most recent call last):
 File "C:/Users/test/Desktop/python_programs/test.py", line 9, in <module>
 print("Local outside function y=",y) # shows error: local variable cannot be used here
NameError: name 'y' is not defined
>>>
```

In Example 3, variable created x is global, hence, can be used both inside and outside any function. Variable y is created inside function test(), therefore, its scope is only limited to this function and hence cannot be used outside. Global variables can be created in functions using *global* keyword as shown in example 4.

Example 4: Program to create global variable inside function.

```
x=1 # global variable
def test():
 global x # to access global variable for modification
 print("inside function before modification x=",x)
 x=x+2
 print("inside function after modification x=",x)

test()
print("outside function x=",x)
```

===== RESTART: C:/Users/test/Desktop/python\_programs/test.py =====

```
inside function before modification x= 1
inside function after modification x= 3
outside function x= 3
```

If we create a variable inside function having same name as that of global variable, then a separate variable of same name gets created. Function in this case can access local variable and hence, cannot refer to global variable as shown in example 5.

Example 5: Program showing use of Local and global variables of same name.

```
x=1 # global variable
def test():
 x=2 # local variable
 print("inside function x=",x)

test()
print("outside function x=",x)
```

===== RESTART: C:/Users/test/Desktop/python\_programs/test.py =====

```
inside function x= 2
outside function x= 1
>>>
```

In python, global variable can be accessed in function directly (as shown in Example 3) but they cannot be directly modified inside the function (shown in example 6). Then how can we modify global variables inside function? Answer to this question is through *global* keyword. It can not only be used for creating global variables inside functions, but it can also be used for modifying global variables inside functions. See example 7.

Example 6: Program to show modification of global variable is not allowed inside function

```
x=1 # global variable
def test():
 x=x+2 # shows error: modification of global variable is not allowed
 print("inside function x=",x)

test()
print("outside function x=",x)
```

===== RESTART: C:/Users/test/Desktop/python\_programs/test.py =====

```
Traceback (most recent call last):
 File "C:/Users/test/Desktop/python_programs/test.py", line 6, in <module>
 test()
 File "C:/Users/test/Desktop/python_programs/test.py", line 3, in test
 x=x+2 # shows error: modification of global variable is not allowed
UnboundLocalError: local variable 'x' referenced before assignment
>>>
```

Example 7: Program to show modification of global variable using global keyword

```
x=1 # global variable
def test():
 global x # to access global variable for modification
 print("inside function before modification x=",x)
 x=x+2
 print("inside function after modification x=",x)

test()
print("outside function x=",x)
```

===== RESTART: C:/Users/test/Desktop/python\_programs/test.py =====

```
inside function before modification x= 1
inside function after modification x= 3
outside function x= 3
```

### Check your Progress 1

Ex1. What are the benefits of using functions? Also differentiate between function definition and function calling.

Ex2. What is Scope of a variable? Explain local and global variables with example.

Ex3. Write a function named *pattern* to display the pattern given below-

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

---

## 11.4 FUNCTION ARGUMENTS

---

Arguments are a way of passing values or input to a function. Arguments are passed to a function during function calls. Their values replace the functions parameters in the function definition.

Python provides various mechanisms of passing arguments to a function.

1. Required arguments
2. Default arguments
3. Keyword arguments
4. Arbitrary arguments

### 1. Required Arguments or Positional Arguments

In required arguments, number of arguments should match number of function parameters and should be given in same order. If number of arguments is not the same, then error will be shown and if correct order is not followed then output can be incorrect.

Example 8: Passing arguments to a function.

```
def calculator(a,b):
 print("Addition :",a+b)
 print("Subtraction :",a-b)
 print("Multiplication:",a*b)
 print("Division :",a/b)

calculator(5,2)
```

===== RESTART: C:/Users/test/Desktop/python\_programs/arguments.py =====

```
Addition : 7
Subtraction : 3
Multiplication: 10
Division : 2.5
>>>
```

## 2. Default arguments

Default values of variables can be given in the parameters itself. So if any argument is not given at the calling time, it will be replaced by the default value. Default values can be given for any number of arguments. It must be taken care that default argument must follow non-default arguments.

### Example 9: Use of Default Arguments

```
def simple_interest(p,r=2,t=5):
 interest=(p*r*t)/100
 print(" Simple Interest is:",interest)

simple_interest(1000) # p=1000, r=2 (default) ,t=5(default)
simple_interest(1000,5) # p=1000, r=5 ,t=5(default)
simple_interest(1000,5,10) # p=1000, r=2 ,t=5
```

===== RESTART: C:\Users\test\Desktop\python\_programs\arguments.py =====

```
Simple Interest is: 100.0
Simple Interest is: 250.0
Simple Interest is: 500.0
>>>
```

## 3. Keyword Arguments

We have seen above that arguments must be given in the order in which parameters are defined otherwise the result will be effected. Keyword arguments are one way by which we can give arguments in any order. They are given by specifying the parameter name with each argument during function call. In this case position does not matter but name matters hence they are also called named arguments.

#### Example 10: Use of Keyword Arguments

```
def compound_interest(p,r,t):
 amount = p*pow(1+ (r/100),t)
 interest = amount-p
 print("Compound Interest is:",interest)

compound_interest(r=13,p=100000,t=5) # passing keyword arguments
compound_interest(100000,13,5) # passing positional arguments
```

```
===== RESTART: C:/Users/test/Desktop/python_programs/compound_interest.py =====
Compound Interest is: 84243.51792999991
Compound Interest is: 84243.51792999991
>>>
```

#### 4. Arbitrary Arguments

Sometimes it is required to pass different number of arguments to a same function or it is not known at function definition time that how many arguments could be used at calling time. Python provides the feature of arbitrary arguments to deal with this issue. Arbitrary arguments can be declared using \* symbol.

#### Example 11: Use of Arbitrary Arguments

```
def sum(*a): # declaration of arbitrary argument
 sum=0
 for i in a:
 sum=sum+i
 print("Sum of Series",a," is:",sum)

sum(1,2,5,7,8,10,2) # passing arbitrary arguments
sum(10,20,25)
```

```
===== RESTART: C:/Users/test/Desktop/python_programs/Arbitrary.py =====
Sum of Series (1, 2, 5, 7, 8, 10, 2) is: 35
Sum of Series (10, 20, 25) is: 55
>>>
```

## 11.5 RETURNING FROM A FUNCTION

As we can pass values as input to function parameters, similarly we can also return or extract values out of a function. This can be done using *return* statement. By default when we do not include return statement, value None is returned from the function.

Syntax of returning from function

```
def function_name(arg1,arg2....) :

 return value
```



#### Example12: Function with return statement

```
def example(a,b,c): # function with return statement
 avg=(a+b+c)/3
 return avg

result=example(2,3,4) # returned value passed to result variable

print("AVG OF NUMBERS:",result) # display return value by print statement

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
AVG OF NUMBERS: 3.0
>>>
```

There can be only one return statement in a function. It doesn't mean that we can return only one value. It means that the return statement can return only one object and object in python can contain single (variable) or multiple values (List, tuple). Hence, we can return multiple values with a single return statement. See example13.

#### Example 13: Function returning more than one value

```
def example(a,b,c):
 sum=a+b+c
 avg=sum/3
 return sum,avg # function returning two values with single statement

result1,result2=example(2,3,4)
print("SUM OF NUMBERS:",result1)
print("AVG OF NUMBERS:",result2)

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
SUM OF NUMBERS: 9
AVG OF NUMBERS: 3.0
>>>
```

Various benefits of using return statement -

1. Values returned by function can be used again in the rest of the program.
2. They can also be passed as argument to other functions providing better flow of control. See example 14.
3. They can also be used to break out of function in the middle.

#### Example 14: Function returning more than one value

```
def first(a,b):
 sum=a+b
 return sum

def second(a,b):
 avg=a/b
 return avg

value1=float(input("Enter first input:"))
value2=float(input("Enter second input:"))
result1=first(value1,value2) # result1 contains value return from first() function
result2=second(result1,2) # result1 is passed as argument to second() function
print("AVG OF NUMBERS:",result2) # display return value by print statement

print("10 / 2 =:",second(10,2)) # function with return statement can also be
 # directly called with print statement

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
Enter first input:3
Enter second input:6
AVG OF NUMBERS: 4.5
10 / 2 =: 5.0
```

It is also possible to break out of a function in the middle of statements in function using return. The statements following the return statement will never be executed. In example 15, inside a function, there is loop ranging from 1 to 9, but it will only be executed 4 times, since the function will return when value of i reaches 5, hence remaining iterations will be skipped.

Example 15: Function returning in between the loop

```
def example():
 for i in range(1,10):
 if(i==5):
 return # function returns when i=5, no further execution
 else:
 print(i)

example()
```

===== RESTART: C:/Users/test/Desktop/python\_programs/return\_example.py =====

```
1
2
3
4
>>>
```

### Check your Progress 2

Ex 1. What are the various ways of passing arguments to a function?

Ex 2. Write a function which takes list of numbers as argument and returns a list of unique elements from it.

Ex 3. Write a function to display all prime numbers between a range which is passed as arguments.

---

## 11.6 FUNCTION OBJECTS

---

In python, data is represented as objects. Like Lists, Strings etc, functions are also treated as objects. Functions in python are first class objects. Since functions are objects, it provides various features with additional to the existing ones. These are-

1. functions can be assigned to a variable
2. functions can be used as elements of data structures
3. functions can be sent as arguments to another function
4. functions can be nested

### 1. Functions assigned to variable

As functions are object, they can be assigned to a variable (object). Hence, function can then be called using variable and set of parenthesis. Assigning function to a variable creates a reference to the same function or a function with two names. If one function or variable is deleted, function can still be referenced by another name. Following is the example of function assigned to a variable. (example 16)

Example 16: Function assigned to variable

```
def example():
 print("Testing function")

a = example # function assigned to variable
a() # function call using variable

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
Testing function
>>>
```

In the following console window it can be clearly seen that after deleting a function, it cannot be accessed with the same name. But it can still be called using another name as shown in example 17.

Example 17: Deleting reference to a function

```
>>> del example
>>> example()
Traceback (most recent call last):
 File "<pyshell#1>", line 1, in <module>
 example()
NameError: name 'example' is not defined
>>> a()
Testing function
>>> |
```

## 2. Functions as element of data structure

Functions can be passed as an element to any data structure. This is very useful at times when we want to apply different functions to same input. Below is the example 18 to show that. We are using various built-in functions to demonstrate that.

Example 18: Function assigned as elements to List

```
abc=[str.upper,str.lower,len] # list of functions

for i in abc:
 print(i,i("function as element"))# applying different function to same input

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
<method 'upper' of 'str' objects> FUNCTION AS ELEMENT
<method 'lower' of 'str' objects> function as element
<built-in function len> 19
>>>
```

## 3. Function as argument to another function

Objects can be passed as arguments to a function. Since functions are objects in python, they can also be passed as arguments to functions. This technique allows application of multiple operations on a particular set of inputs. Given below is an example of function calling another function as arguments.

Example 19: Function calling another function as argument

```

def add(n):
 sum=0
 for i in n:
 sum=sum+i
 return sum

def display(l,func):
 print("Addition of elements:",func(l))

display([1,2,3,4,5,6,7],add) # function display() calling function add()

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
Addition of elements: 28
>>>

```

#### 4. Nested function

Functions created within another function are called nested function. This is one of the unique properties in python since functions are objects. The inner function which is created within some outer function has access to all the variables of enclosing scope. Inner functions can never be directly called outside outer function. Hence, it provides security feature called Encapsulation. It can only be called by the use of enclosing function.

Example 20: Nested function

```

def outer(text): # outer function
 print(str.upper(text))

 def inner(): # nested or inner function
 print(str.lower(text))

 inner() # nested function can only be called inside outer function

outer("Nested Function") # call to outer function makes indirect call to inner
inner("Nested Function") # shows error: nested function cannot be called outside

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
NESTED FUNCTION
nested function
Traceback (most recent call last):
 File "C:/Users/test/Desktop/python_programs/return_example.py", line 12, in <module>
 inner("Nested Function") # shows error: nested function cannot be called outside
NameError: name 'inner' is not defined
>>>

```

## 11.7 LAMBDA / ANONYMOUS FUNCTIONS

Python allows to create functions without names called Anonymous functions. They are not declared with `def` keyword, instead *lambda* keyword is used to create these functions. Hence, these functions are also called **lambda functions**. Like other

functions, these functions can also contain arguments but there can be only one statement in the body of these functions. Statement is evaluated and the result is returned. There are situations when a function needs to be created for a short operation (similar to macros in C language) or usable for a short period of time, in that case lambda functions are best suited.

#### Syntax of Anonymous function

Variable = lambda arg : statement

Here, arg is one or more arguments passed to a function and statement consists of operation performed by a function.

Given below is example 21 in which two lambda functions are created. First function calculates cube of a given argument and second function gives addition to two given arguments. Lambda function is then assigned to a variable (function assigned to variable, discussed in previous section), which can then be used to call function with arguments and set of parenthesis.

#### Example 21: Use of Lambda function

```
cube = lambda x : x*x*x # lambda function to calculate cube
add = lambda x,y: x+y # lambda function to calculate sum

print("cube of 3:",cube(3)) # first lambda function called
print("addition of 10 & 20:",add(10,20) # second lambda function called

===== RESTART: C:/Users/test/Desktop/python_programs/lambda.py =====
cube of 3: 27
addition of 10 & 20: 30
>>>
```

Lambda functions are mostly used as argument to other high-order function like map () and filter () function.

**map() function** : map () is a built-in function which takes two arguments, first argument given is function (can be lambda function), second argument is given as a list. map () function returns a list of elements obtained by applying given function to each element of a list.

**filter() function** : filter() is another built-in function which takes two arguments. First argument is a function and second is list of elements, just as map() function. Here, given function is applied to each of the elements of list and returns a list of items for which the function evaluates to True.

#### Syntax of map and filter function

map ( function , [ list ] )

filter ( function , [ list ] )

Example 22: Application of map() function with lambda function

```
def factorial(n):
 fact = 1
 for i in range(1,n+1):
 fact = fact*i
 return fact

a=[1,2,3,4,5]

b = list(map(lambda x : x*x*x,a)) # map() applied on lambda function
c = list(map(factorial,a)) # map() applied on user defined function

print("CUBE OF SERIES",b)
print("FACTORIAL OF SERIES",c)

===== RESTART: C:/Users/test/Desktop/python_programs/lambda.py =====
CUBE OF SERIES [1, 8, 27, 64, 125]
FACTORIAL OF SERIES [1, 2, 6, 24, 120]
>>>
```

Example 22: Application of filter() function with lambda function

```
a=[1,2,3,4,5,6,7,8,9,10]

b = list(filter(lambda x : x%2==0,a)) # filter() applied on lambda function

print("EVEN NUMBER SERIES",b)

===== RESTART: C:/Users/test/Desktop/python_programs/lambda.py =====
EVEN NUMBER SERIES [2, 4, 6, 8, 10]
>>>
```

### Check your Progress 3

Ex 1. Write a program to find cube of numbers in a list using lambda function.

Ex 2. Write a program to add two given lists using map function.

Ex 3. Write a program to display vowels from a given list of characters using filter function.

---

## 11.8 FILE OPERATIONS

---

Main memory is volatile in nature, hence, nothing can be stored permanently. File handling is a very important functionality which must be provided by any language to deal with this problem. Inputs can be taken from files instead of users, output can be displayed and saved permanently in files which can further be accessed later and appended or updated. All these functionalities come under file handling. Like other languages, python also provides various built-in functions for file handling operations.

There are two types of files supported by python- Binary and Text.

**Binary files** – These are the files that can be represented as 0's and 1's. These files can be processed by applications knowing about the file's structure. Image files are example of binary files.

**Text files** – These files are organized as sequence of characters. Here, each line is separated by a special end of line character.

Any file handling operation can be performed in three steps-

1. Opening a file
2. Operating on file – read , write , append etc
3. Closing a file

### 1. Opening a file

A file can be opened in a Python using built-in function *open()*. By default the files are opened in read text ('r') file mode.

Syntax of open() function

```
file = open ("file_name.ext ","mode_of_operation")
```

Where,

File\_name.ext - is the name of the file to be opened and ext is the extension of file.

Mode\_of\_operation - is the mode in which file is needed to be opened.

file – is the object returned by the function. This object can be used for further operations on file

The possible mode of operations in python –

| Mode of opening file | Functionality                                                                                      |
|----------------------|----------------------------------------------------------------------------------------------------|
| 'r'                  | Opens a file for reading. It is default mode                                                       |
| 'w'                  | Opens a file for writing. Overwrites a file if already exists. Creates new file if does not exist. |
| 'a'                  | Opens a file for appending.                                                                        |
| 'r+'                 | Opens a file for both reading and writing.                                                         |
| 'w+'                 | Same as 'r+' but creates new file if does not exist and overwrites if exists                       |
| 'a+'                 | Opens a file for appending and reading                                                             |
| 'x'                  | Creates new file. Fails if already exists. Added in python 3.                                      |
| 'rb'                 | Opens a file for reading in binary mode.                                                           |
| 'wb'                 | Same as 'w' except data is in binary                                                               |
| 'ab'                 | Same as 'a' except data is in binary                                                               |

## 2. Operating on file

There are various operations -

### 2.1 Reading from a file

There are many ways to read from a file. Given below are the functions available for reading from a file.

In the below table assume that *file* is the object returned from open() function.

| Function    | Syntax           | Description                                                    |
|-------------|------------------|----------------------------------------------------------------|
| read()      | file.read ()     | Returns entire file contents as a string                       |
|             | file.read (n)    | Returns n characters from beginning of file as string          |
| readline()  | file.readline()  | Returnssingle line of file at a time. First line in this case. |
| readlines() | file.readlines() | Returns a list of all lines                                    |

### 2.2 Writing to a file

Similar to read operations, functions are there in python to write data to file.

| Function | Syntax             | Description                     |
|----------|--------------------|---------------------------------|
| write()  | file.write("text") | Writes text or string to a file |

### 2.3 Operations on file

Python allows many other operations to be done in with files. Listed below are some functions used for file handling.

| Function  | Syntax            | Description                                                              |
|-----------|-------------------|--------------------------------------------------------------------------|
| tell()    | file.tell()       | Returns the current cursor position in file                              |
| seek()    | file.seek(pos)    | Places the file cursor to the position pos                               |
| os.stat() | os.stat(filename) | This method is used to get display status of the file given as argument. |



|                   |                           |                                                           |
|-------------------|---------------------------|-----------------------------------------------------------|
| os.path.exists()  | os.path.exists('arg')     | Returns true if the file or directory of name 'arg' exist |
| os.path.isfile()  | os.path.isfile('arg')     | Returns true if the file of name 'arg' exists             |
| os.path.isdir()   | os.path.isdir('arg')      | Returns true if the directory of name 'arg' exists        |
| shutil.copy()     | shutil.copy(src,dst)      | Copies a file from src to des file.                       |
| os.path.getsize() | os.path.getsize(filename) | Returns the size of file                                  |

### 3. Closing a file

After completing all the operations on file, it must be closed properly. This step frees up the resources and allows graceful termination of file operations.

Syntax of closing a file

```
file.close()
```

#### 11.8.1 Reading data from a file

There are various ways to read a file by following the steps explained in the above section. For reading data from the file, it must be existing. Below is given the program (Example 23) to read a file named "first.txt" in the directory files within the present directory.

Example 23: Program to display file contents

```
file = open ("files\\first.txt","r") # step 1: opening file for reading
s = file.read() # step 2: reading entire file
print(s) # displaying file contents
file.close() # step 3: closing file
```

```
===== RESTART: C:/Users/test/Desktop/python_programs/file.py =====
Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.[28]

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural,) object-oriented, and functional programming. Python is often described as a "batteries included"
```

It may be noted that the file that we want to read or write, may be present in some other directory or folder. In that case, we can give absolute or relative path of that file along with its name, shown in the example below. The path separated by \ must

be proceeded by one more \ (backslash) to turn-off special feature of this character. If the file to be read or written is in the same folder, in which python program is present, we need not give its path.

## 11.8.2 Creating a file

For creation of new file, the file can be opened in 'w' mode. If the file already exists, the contents can be overwritten. Also, if we want to be sure that no existing file gets overwritten, then 'x' mode can be used to create new file. If the file already exists, it will show error message. Given below is the example to open a file and write contents to it.

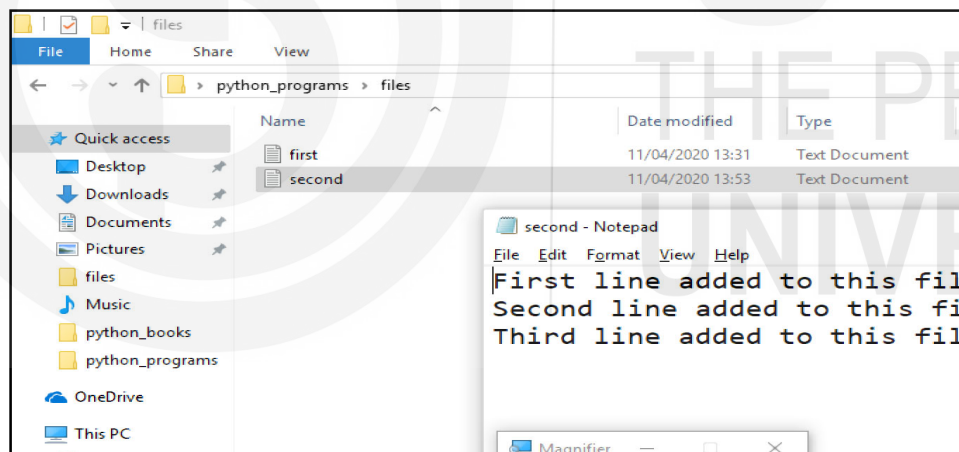
Example24 : Program to create file and add contents to it.

```
file = open ("files\\second.txt","w") # step 1: opening file for reading

file.write("First line added to this file\n")# step 2: writing contents to file
file.write("Second line added to this file\n")
file.write("Third line added to this file\n")
file.close() # step 3: closing file
print("file created successfully")

===== RESTART: C:/Users/test/Desktop/python_programs/file.py =====
file created successfully
>>>
```

After executing this program a file name second.txt gets created in the mentioned drive with the added contents.



### Reading, writing, appending with *with* statement

The methods of dealing with files used in the above section may not be safe sometimes. It may happen that an exception occurs as a result of operations on file and the code exits without closing the file. To make it more convenient python introduced another statement called *with* statement which ensures that file is closed when the code within *with* is exited. Call to close() function is not required explicitly. The syntax of how to use *with* statement for reading, writing and appending file is given below.

Syntax of read, write and append with *with* statement:

```
with open("file_name","r") as file: # to read file
 file.read()

with open("file_name","w") as file: # to write file
 file.write("contents to add")

with open("file_name","a") as file: # to append file
 file.write("\n contents to append")
```

### 11.8.3 Copying a file

To copy a file from one to another various methods can be used. One such way of copying file is by importing a module named 'shutil' which contains a built-in function to create copy of a file.

Example 25 : Program to copy a file from 'second.txt' to 'third.txt'.

```
import shutil
shutil.copy("files\\second.txt", "files\\third.txt")
```

### 11.8.4 Deleting a file or folder

A file can be deleted using remove() function from os module. Before using this function you should move to the directory in which the file to be removed exists. Similarly, to delete a folder or directory, os.rmdir() function can be used. The directory to be removed must be empty or otherwise error will be shown. In the example given below, we want to delete a file name "delete\_123.py". os.listdir() function is used to display the list of files present in a directory.

Example 26: Deleting a file.

```
>>> import os
>>> os.chdir("C:\\Users\\test\\Desktop\\python_programs")
>>> os.listdir()
['abc.py', 'Arbitrary.py', 'arguments.py', 'arg_test.py', 'compound_interest.py',
'delete_123.py', 'demo.py', 'demo2.py', 'factorial.py', 'file.py', 'file2.py',
'files', 'lambda.py', 'modules_test.py', 'module_2.py', 'pack', 'path.py', 'rand_test.py', 'return_example.py', 'series.py', 'simple_interest.py', 'test.py', 'test_2.py', '__pycache__']
>>> os.remove("delete_123.py")
>>> os.listdir()
['abc.py', 'Arbitrary.py', 'arguments.py', 'arg_test.py', 'compound_interest.py',
'demo.py', 'demo2.py', 'factorial.py', 'file.py', 'file2.py', 'files', 'lambda.py', 'modules_test.py', 'module_2.py', 'pack', 'path.py', 'rand_test.py', 'return_example.py', 'series.py', 'simple_interest.py', 'test.py', 'test_2.py', '__pycache__']
>>>
```

### Check your Progress 4

Ex 1. Write a program to display frequency of each word in a file.

Ex 2. Write a program to display first n lines from a file, where n is given by user.

Ex 3. Write a program to display size of a file in bytes.

---

## 11.9 SUMMARY

---

A **Function** is a block of code that performs a specific task. In this chapter, we have discussed various aspects of functions such as how to create functions, their scope, passing arguments to function, and Lambda functions. In addition to these topics, file handling operations are also discussed in detail such as how to interact with file, copying, deleting, etc.

---

## SOLUTIONS TO CHECK YOUR PROGRESS

---

### Check your Progress 1

Ex 1. The various benefits of using functions are –

- i) It decomposes a larger problem into more manageable units
- ii) It allows reusability of code
- iii) It reduces duplication
- iv) It makes code easy to understand, use and debug

The process of defining a function is called *function definition*. It actually describes the working of a function. It includes – function name, list of arguments and function body.

Syntax `def function_name():`  
`.....`                      `# body of function`  
`.....`

Even if the function is defined, it can never be executed till the time it is called. Hence, for using a function, it must be called using *function call statement*. Calling of a function includes function name and list of arguments (no function body).

Syntax `function_name()`

Ex 2. Part of a code in which a variable can be accessed is called *Scope of a variable*.

Local variables are the variable created within a function's body or function's scope. They cannot be accessed outside the function. Their scope is only limited to the function in which they are created.

Global variables are the variable created outside any function. Their scope is global, hence can be used anywhere. Global variables can also be created within function using keyword *global*.

```
def f():
 s = 1 # s is local variable
 print(s)
 r = "IGNOU" # r is global variable
f()
```

```
print(r)
```

Ex 3.

Fu  
Ha

```
def pattern():
 for i in range(1,6):
 for j in range(1,i+1):
 print(j,end=' ')
 print()

pattern()
```

## Check your Progress 2

Ex 1. Python provides various mechanisms of passing arguments to a function.

1. Required arguments
2. Default arguments
3. Keyword arguments
4. Arbitrary arguments

Ex 2.

```
def unique(numbers):
 out = []
 for i in numbers:
 if i not in out:
 out.append(i)
 return out

ans=unique([1,2,1,1,2,2,3,4,6,3])
print(ans)
```

Ex 3.

```
def prime(a,b):
 for i in range(a,b+1):
 if i==0 or i==1:
 continue
 test=0
 for j in range(2,i):
 if i%j==0:
 test=1
 break
 if test==0:
 print(i,end=' ')

prime(1,50)
```

## Check your Progress 3

Ex 1.

```
nums = [1, 2, 3, 4, 5]
print("\nCube of numbers:")
cube = list(map(lambda x: x ** 3, nums))
print(cube)
```

Ex 2.

```
a = [1, 2, 3]
b = [10, 20, 6]

result = map(lambda x, y: x + y, a, b)
print("\nAddition of two list:")
print(list(result))
```

Ex 3.

```
def vowel(a):
 v=['a','e','i','o','u']
 if a in v:
 return a

l=['a','b','f','o','x','z','y']
print("LIST OF VOWELS:")
print(list(filter(vowel,l)))
```

### Check your Progress 4

Ex 1.

```
from collections import Counter
def wordcount(fname):
 with open(fname) as f:
 return Counter(f.read().split())

print("Frequency of words:",wordcount("abc.txt"))
```

Ex 2.

```
n=int(input("enter number of lines:"))
c=1
file=open("test.txt","r")
for i in file:
 if c<=n:
 print(i)
 c+=1
file.close()
```

Ex 3.

```
def fsize(fname):
 import os
 status = os.stat(fname)
 return status.st_size

print("File size in bytes: ",fsize("test.txt"))
```

---

## UNIT 12 MODULES AND PACKAGES

---

### Structure

|      |                                |
|------|--------------------------------|
| 12.0 | Introduction                   |
| 12.1 | Objectives                     |
| 12.2 | Module Creation and Usage      |
| 12.3 | Module Search Path             |
| 12.4 | Module Vs Script               |
| 12.5 | Package Creation and Importing |
| 12.6 | Standard Library Modules       |
| 12.7 | Summary                        |

---

## 12.0 INTRODUCTION

---

Modules are files that contain various functions, variables or classes which are logically related in some manner. Modules like functions are used to implement modularity feature of OOPs concept. Related operations can be grouped together in a file and can be imported in other files. Package is a collection of modules and other sub-modules. Modules can be well organized and easily accessible if collectively stored in a package.

## 12.1 OBJECTIVES

---

After going through this unit, you will be able to :

- Understand usage of Modules
- Create your own Modules
- Compare Modules and scripts
- Import packages and Create your own packages
- Understand the standard library modules

## 12.2 MODULE CREATION AND USAGE

---

Module is a logical group of functions, classes, variables in a single python file saved with .py extension. In other words, we can also say that a python file is a module. We have seen few examples of built-in modules in previous chapters like os, shutil which are used in the program by import statement. Python provides many built-in modules. We can also create our own module.

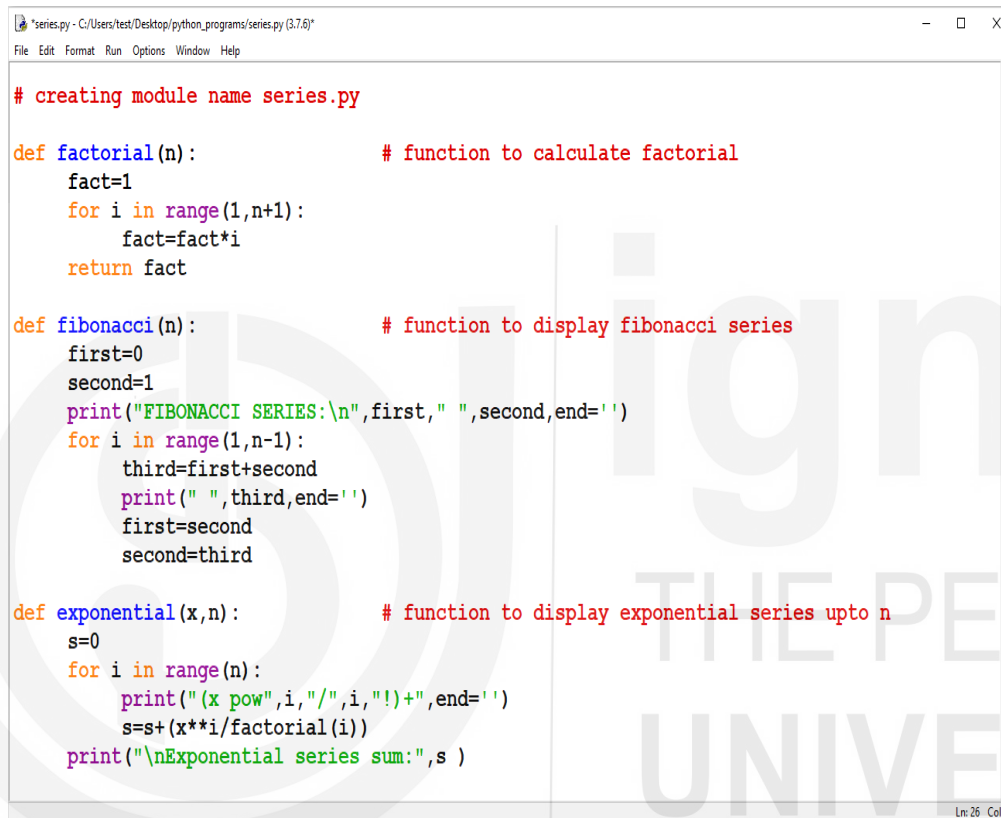
A major benefit of a module is that functions, variable or objects defined in one module can be easily used by other modules or files, which make the code re-usable. A module can be created like any other python file. Name of the module is the same as the name of a file. Let us create our first module- series.py.

In this module, we have created three functions- to find factorial of a number, fibonacci series upto given number of terms and a function to display exponential series and its sum. After creating a file, save it with name series.py.

Note: it is important to check where we have saved this file or module. Currently, it is saved in my present working directory. You can check current working directory by using built-in function `getcwd()` under `os` module by using following syntax in console window or python prompt.

```
>>> import os
>>> os.getcwd()
'C:\\Users\\test\\Desktop\\python_programs'
>>> |
```

Example 1: Creating module named series.py

A screenshot of a Python IDE window titled "series.py - C:/Users/test/Desktop/python\_programs/series.py (3.7.6)". The window contains the following Python code:

```
creating module name series.py

def factorial(n):
 # function to calculate factorial
 fact=1
 for i in range(1,n+1):
 fact=fact*i
 return fact

def fibonacci(n):
 # function to display fibonacci series
 first=0
 second=1
 print("FIBONACCI SERIES:\n",first," ",second,end='')
 for i in range(1,n-1):
 third=first+second
 print(" ",third,end='')
 first=second
 second=third

def exponential(x,n):
 # function to display exponential series upto n
 s=0
 for i in range(n):
 print("(x pow",i,"/",i,"!)+",end='')
 s=s+(x**i/factorial(i))
 print("\nExponential series sum:",s)
```

Our module is successfully created. Now let us test our module by importing in some other file and check whether it is working or not. For verifying that, in a new file, two steps are needed to be done-

1. Import the module we have created to make it accessible
2. Call the functions of that module with module name and a dot (.) symbol.

Example 2: Accessing function in module created in Example 1.



```

import series # importing module series.py

print ("Here we are using module series.py")

n=int(input("enter number of terms in fibonacci series "))

series.fibonacci(n) # calling a function present in other module

===== RESTART: C:/Users/test/Desktop/python_programs/demo.py =====
Here we are using module series.py
enter number of terms in fibonacci series 10
FIBONACCI SERIES:
0 1 1 2 3 5 8 13 21 34
>>>

```

Similar to the above example, we can call another function created in the module fibonacci() by following the same process.

```
import series
```

```
Series.fibonacci (2, 10)
```

When a module is imported in a file, a folder named `__pycache__` folder gets created by interpreter which contains .pyc file of a module imported. This file contains the compiled bytecode of module so that conversion from source code to bytecode can be skipped for subsequent imports and making execution faster.

## Importing a module

Importing is the process of loading a module in other modules or files. It is necessary to import a module before using its functions, classes or other objects. It allows users to reference its objects. There are various ways of importing a module.

1. using *import* statement
2. using *from import* statement
3. using *from import \** statement

### 1. Importing Complete module

In this method, we can import the whole module all together with a single import statement. In this process, after importing the module, each function (or variable, objects etc.) must be called by the name of the module followed by dot (.) symbol and name of the function.

Syntax of function calling within module

```
import module
module.function_name()
```

For example, let us import the built-in module random, and call its function randint(), which generates a random integer between a range given by user.

This can be done by running the code below in console window directly or in a python file.

```
>>> import random
>>> random.randint(10,100)
74
>>> random.randint(10,100)
95
>>>
```

In this method, other functions or objects present in module random can be called similarly.

```
>>> import random
>>> random.random()
0.62009496454466
>>> |
```

Here, random () is function present within module random.

## 2. Importing using *from import* statement

In this method of importing, instead of importing the entire module function or objects, only a particular object needed can be imported. In this method, objects can be directly accessed with its name.

Let us take an example of another module called math. This module contains various functions and variables. One such variable is pi, which contains value of  $\pi$ .

```
>>> from math import pi
>>> pi
3.141592653589793
>>>
```

In this example, only a variable called pi is imported from math module, hence it can be directly accessed with its name. In this case, module name cannot be used for calling its objects, doing this will show NameError. Also other functions within the module math cannot be accessed, since only one variable pi is imported. You will be able to access them only after importing them individually with *from import* statement.

```
>>> from math import pi
>>> pi
3.141592653589793
>>> math.pi
Traceback (most recent call last):
 File "<pyshell#4>", line 1, in <module>
 math.pi
NameError: name 'math' is not defined
>>>
```

### 3. Importing entire module using *from import \**

This method can be used to import the entire module using *from import \** statement. Here, *\** represents all the functions of a module. Like previous method, an object can be accessed directly with its name.

```
>>> from math import *
>>> pi
3.141592653589793
>>> log(2)
0.6931471805599453
>>> log10(100)
2.0
>>> |
```

### Check your Progress 1

Ex 1. What are modules in Python and how can we create modules ?

Ex 2. What are the various ways of importing modules ?

Ex 3. Name any 3 built-in modules in python.

---

## 12.3 MODULE SEARCH PATH

---

When we use import statements to import a module, it is searched in a list of directories or search paths stored by the environment variable PYTHONPATH. This list of directories can be checked using sys.path variable.

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\Lib\\idlelib',
 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\DLLs', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\lib', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\lib\\site-packages']
>>>
```

These directories include:

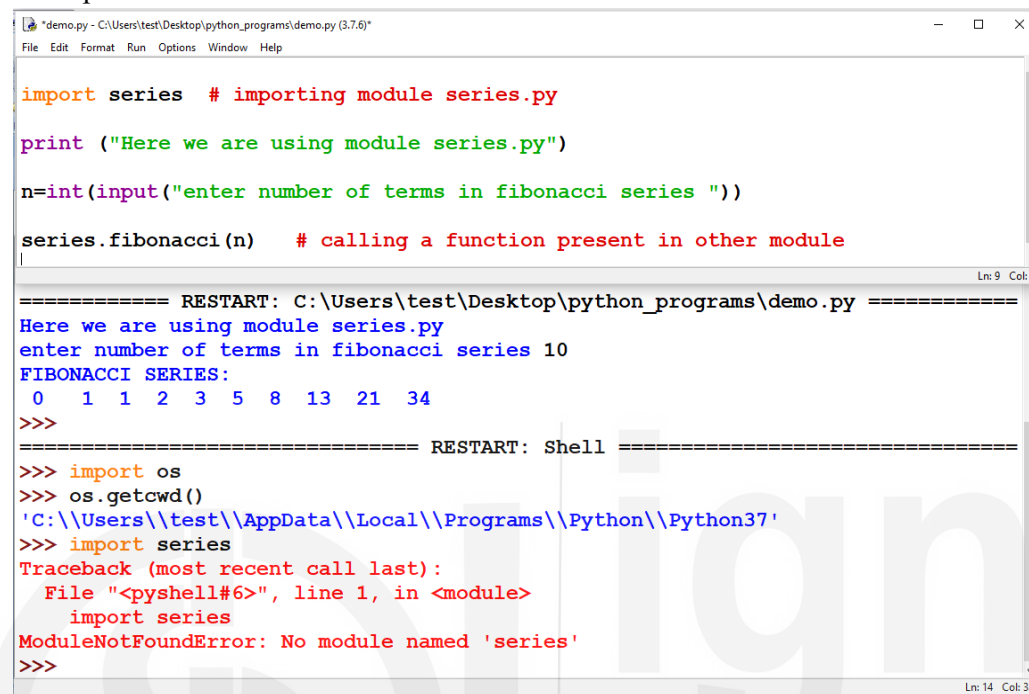
1. The current directory in which user is working [ ' ' ]
2. Installation dependent paths
3. Directories stored in variable PYTHONPATH

Upto now, we were able to import our modules without doing anything special because they were all created in the same current directory. But if we move to some other directory, and try to import modules located in previous directories, we will not be able to use it.

In example 1 of this unit, we have created a module named series.py and used this module in a file named demo.py in example2. We were able to import

module since both of them were in the same directory. But when we re-start shell, we move to python's default location. In this location, we will not be able to import of series.py module. Shown in example 3 below.

### Example 3:



```
demo.py - C:\Users\test\Desktop\python_programs\demo.py (3.7.6)
File Edit Format Run Options Window Help

import series # importing module series.py
print ("Here we are using module series.py")
n=int(input("enter number of terms in fibonacci series "))
series.fibonacci(n) # calling a function present in other module

===== RESTART: C:\Users\test\Desktop\python_programs\demo.py =====
Here we are using module series.py
enter number of terms in fibonacci series 10
FIBONACCI SERIES:
0 1 1 2 3 5 8 13 21 34
>>>

===== RESTART: Shell =====
>>> import os
>>> os.getcwd()
'C:\Users\test\AppData\Local\Programs\Python\Python37'
>>> import series
Traceback (most recent call last):
 File "<pyshell#6>", line 1, in <module>
 import series
ModuleNotFoundError: No module named 'series'
>>>
```

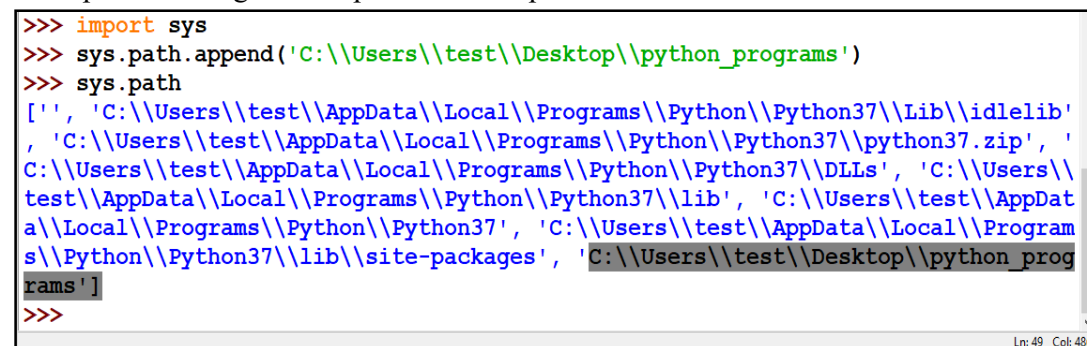
Therefore, any modules created must be located in python's search path for its global identification. This can be done in either of the ways-

1. Creating module in one of the locations already present in search path
2. Adding your module path in the search path using sys.path.
3. Updating PYTHONPATH environment variable.
- 4.

### Adding module location to search path

A module path can be added to python's module search path by appending the sys.path variable. This can be done by using the append( ) function of sys.path. Directory in which your module is located should be appended as shown below example 4.

### Example 4: Adding module path to search path



```
>>> import sys
>>> sys.path.append('C:\\Users\\test\\Desktop\\python_programs')
>>> sys.path
['', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\Lib\\idlelib',
 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip',
 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\DLLs', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\lib',
 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37', 'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\lib\\site-packages',
 'C:\\Users\\test\\Desktop\\python_programs']
>>>
```

As we can see in above example, our directory is now present in the list of search directories. Hence, now we can import series module from any location. This method is not robust since it adds modules only for current session. For each new session, path needs to be added again.

```
>>> import series
Traceback (most recent call last):
 File "<pyshell#0>", line 1, in <module>
 import series
ModuleNotFoundError: No module named 'series'
>>> import sys
>>> sys.path.append('C:\\Users\\test\\Desktop\\python_programs')
>>> import series
>>> series.fibonacci(10)
FIBONACCI SERIES:
0 1 1 2 3 5 8 13 21 34
>>> |
```

---

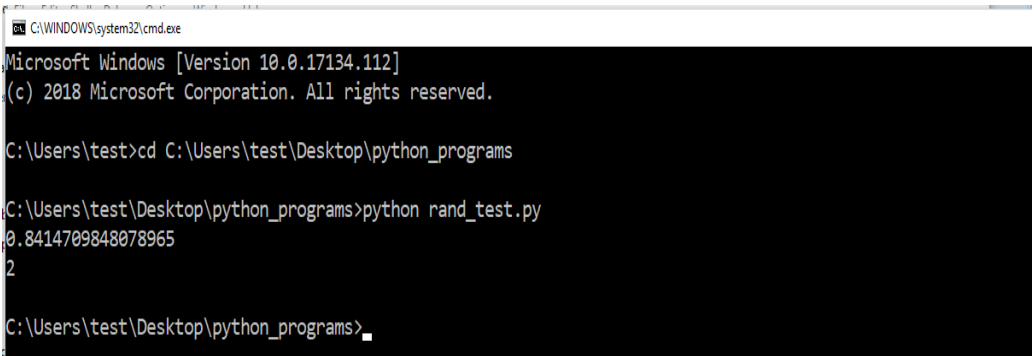
## 12.4 MODULE VS SCRIPT

---

For large number of instructions to be used together instead of directly running in shell or console, we used to write the code text files. These files can be modules or scripts. Extension of both the files is .py. Though there are several similarities, there are few differences as well.

### SCRIPTS

Scripts are the files with sequence of instructions, which are executed each time the script is executed. There are various ways to execute a script, provided by different IDEs. It can also be executed in the console ( shell in Unix/Linux and cmd in windows) using the command given below.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.17134.112]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\test>cd C:\Users\test\Desktop\python_programs

C:\Users\test\Desktop\python_programs>python rand_test.py
0.8414709848078965
2

C:\Users\test\Desktop\python_programs>_
```

It should be noted that this command should be run in the directory where your python script exists otherwise *no file or directory exists* error will be shown.

## MODULES

Functions which can be called from multiple scripts should be created within a module or we can say that a module is a file which is created for the purpose of importing. They are used to organize code in hierarchy. Module after creation should be added to search path.

When a module is imported, it runs the file from top to bottom. But when a module is executed, it runs the entire file and set the `__name__` attribute to the value `"__main__"`. This allows us to put a special code in a particular section which we want and we execute only when the module is executed directly. This section will not be executed during import.

Here, a module named `module_2.py` is created. If this module is executed, output received is given in the below screenshot. The whole file will be executed.

```
print("this section will execute when module is imported")

if __name__ == "__main__":
 print("this part is executed only if module is executed")
 print("we can add more useful stuff here which we dont want to export")

===== RESTART: C:/Users/test/Desktop/python_programs/module_2.py =====
this section will execute when module is imported
this part is executed only if module is executed
we can add more useful stuff here which we dont want to export
>>>
```

But when the above module is imported, the section under `if __name__ == "__main__":` will not be executed as show below.

```
import module_2

===== RESTART: C:/Users/test/Desktop/python_programs/demo2.py =====
this section will execute when module is imported
>>>
```

---

## 12.5 PACKAGE CREATION AND IMPORTING

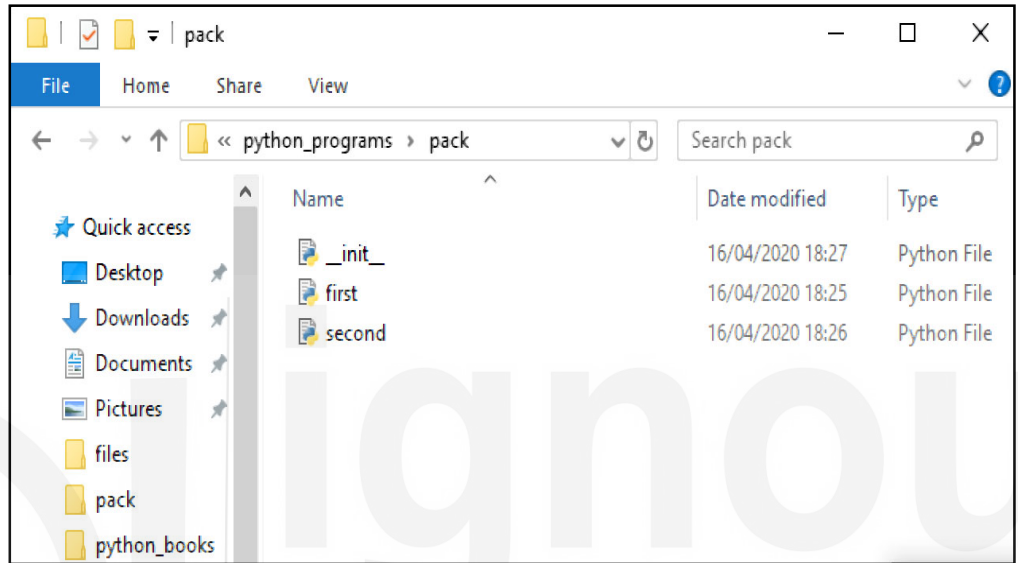
---

Packages like modules are also used to organize the code in a better way. A package is a directory which contains multiple python modules. It is used to group multiple related python modules together. A python package in addition to modules must contain a file called `__init__.py`. This file may be empty or contains data like other modules of package.

### File `__init__.py`

It is a file that makes the package importable. When a package is imported in a script, this file is automatically executed. It initializes variables, objects and makes the functions in the package accessible.

Let us create a package named *pack* and within this package create two modules *first.py* and *second.py*.



The `__init__.py` file created is empty. Module one contains function `abc()` and module two contains function `xyz()`.

Packages can be imported in the same way as we import modules.

The various ways in which we can import from package are-

```
import pack.first
pack.first.abc()
```

```
from pack import first
first.abc()
```

```
from pack.first import abc
abc()
```

There are more methods to import. We have used `*` to import all the functions from a module in the previous section. This method can also be used here. But by default importing package modules using `*` will show error.



```
from pack import *
first.abc()

===== RESTART: C:/Users/test/Desktop/python_programs/test_2.py =====
Traceback (most recent call last):
 File "C:/Users/test/Desktop/python_programs/test_2.py", line 3, in <module>
 pack.first.abc()
AttributeError: module 'pack' has no attribute 'first'
>>>
```

This can be made possible using `__all__` variable. This variable when added to `__init__.py` file, can make modules within package accessible outside using `from import *` statement.

Hence, we need to add `__all__` statement in `__init__.py`

```
__all__ = ['first']
```

The above statement makes module `first.py` accessible using `from import *` statement.

Adding statement `__all__` to `__init__.py` file.

```
__init__.py - C:\Users\test\Desktop\python_programs\pack__init__.py (3.7.6)
File Edit Format Run Options Window Help

__all__ = ['first']

===== RESTART: C:\Users\test\Desktop\python_programs\pack__init__.py =====
>>>
```

Now another way to import the module is given below:

Syntax to import using `from import *`

```
from pack import *
first.abc()
```

```
from pack import *
first.abc()
second.xyz()

===== RESTART: C:/Users/test/Desktop/python_programs/test_2.py =====
under module first
Traceback (most recent call last):
 File "C:/Users/test/Desktop/python_programs/test_2.py", line 4, in <module>
 second.xyz()
NameError: name 'second' is not defined
>>>
```

Here, we can clearly see that `first.py` module is now accessible, since we have added it to `__all__` variable. But `second.py` module is not accessible simultaneously, since it was not added to `__all__` attribute in `__init__.py`.



## Check your Progress 2

Ex. 1 What are packages ?How are they different from modules ?

Ex. 2 What is module search path ?How can we check it ?State the ways of adding a user defined module to search path.

Ex. 3 Create a package named Area and create 3 module in it named – square, circle and rectangle each having a function to calculate area of square, circle and rectangle respectively. Import the module in separate location and use the functions.

---

## 12.6 STANDARD LIBRARY MODULES

---

Python standard library provides number of built-in modules. They are automatically loaded when an interpreter starts. We have already used few of them in previous chapters. It should be noted that before using any module, it should be imported first. Some of the commonly used library modules are-

- sys
- os
- math
- random
- statistics

### Module Attributes

There are some attributes or functions that work for every module whether it is built-in library module or custom module. These attributes help in smooth operations of these modules. Some of them are explained below:

1. help () – it is a function used to display modules available for use in python or to get help on specific module.

```
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

__future__ atexit html search
__main__ audioop http searchbase
__abc__ autocomple hyperparser searchengine
__ast__ autocomplete_w idle secrets
__asyncio autoexpand idle_test select
__bisect base64 idlelib selectors
__blake2 bdb imaplib setuptools
__bootlocale binascii imghdr shelve
__bz2 binhex imp shlex
__codecs bisect importlib shutil
__codecs_cn browser inspect sidebar
__codecs_hk builtins io signal
__codecs_iso2022 bz2 iomenu site
__codecs_jp cProfile ipaddress smtpd
__codecs_kr calendar itertools smtplib
__codecs_tw calltip json sndhdr
__collections calltip_w keyword socket
__collections_abc cgi lib2to3 socketserver
__compat_pickle cgitb linecache sqlite3
__compression chunk locale squeezer
__contextvars cmath logging sre_compile
```

2. `dir()` - it is a function which is used to display objects or functions present in a specific module. Before using `dir()` function, module should be first imported.

```
>>> import math
>>> dir(math)
['_doc_', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan',
'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
>>>
```

Ln: 114 Col: 4

### 3. `__name__` attribute

This attribute returns name of the module. By default its value is the same as the name of the module.

When a module or script is executed, its value becomes `'__main__'`. Also, when called without module name, it returns `'__main__'`.

```
>>> import math
>>> math.__name__
'math'
>>> __name__
'__main__'
>>>
```

Ln: 870 Col: 4

### 4. `__file__` attribute

This attribute returns the location or path of the module.

```
>>> import os
>>> os.__file__
'C:\\Users\\test\\AppData\\Local\\Programs\\Python\\Python37\\lib\\os.py'
>>>
```

Ln: 892 Col: 4

### 5. `__doc__` attribute

This attribute displays documentation given at the beginning of the module file.

```
>>> math.__doc__
'This module provides access to the mathematical functions\ndefined by the C standard.'
>>>
```

Ln: 904 Col: 4

## OS MODULE

This is a module responsible for performing many operating system tasks. It provides functionality like- creating directory, removing directory, changing directory, etc. Some of the functions in `os` modules are given in the table

below. It should be noted that before using these functions, the module should be imported.

#### **import os**

| function               | Description                                                                                                                                                |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| os.mkdir("location")   | Creates new directory in a given location.                                                                                                                 |
| os.rmdir("location")   | Removes directory given by user. It should be taken care that current working directory cannot be removed and the directory to be removed should be empty. |
| os.getcwd()            | Displays the current working directory.                                                                                                                    |
| os.chdir("location")   | Changes current working directory to a given location.                                                                                                     |
| os.listdir("location") | Displays list of files and directories in a given location. If location is not given, files of current directory will be displayed.                        |

#### **SYS MODULE**

This module contains various variables and functions that can manipulate python runtime environment. Some of them are listed in table given below:

| Function              | Description                                                                |
|-----------------------|----------------------------------------------------------------------------|
| sys.path              | Shows list of directories used to search python modules                    |
| sys.argv              | Displays list of values passed as command line arguments to python program |
| sys.maxsize           | Returns the largest integer value a variable can store                     |
| sys.version           | Returns string representing python version                                 |
| sys.getsizeof(object) | Returns size of an object in bytes                                         |
| sys.exit              | Used to exit from a program in case of exception                           |

#### **MATH MODULE**

This module provides various mathematical functions and constant variables. It includes logarithmic, trigonometric functions etc. Some of the functions are listed in table below:

| Function                     | Description                                      |
|------------------------------|--------------------------------------------------|
| <code>math.pow(x,y)</code>   | Returns x to the power y i.e. $x^{**y}$          |
| <code>math.sqrt(x)</code>    | Returns square root of x i.e. $\sqrt{x}$         |
| <code>math.pi</code>         | Returns value of $\pi$                           |
| <code>math.e</code>          | Returns value of e, Euler's number               |
| <code>math.radians(x)</code> | Converts angle x from degree to radians          |
| <code>maths.degree(x)</code> | Converts angle x from radians to degree          |
| <code>math.sin(x)</code>     | Returns <code>sin()</code> of angle x in radians |
| <code>math.log(x)</code>     | Returns natural log of x                         |
| <code>math.log10(x)</code>   | Returns log base 10 of x                         |
| <code>math.floor(x)</code>   | Returns largest integer $\leq x$                 |
| <code>math.ceil(x)</code>    | Returns smallest integer $\geq x$                |

## STATISTICS MODULE

This module contains various functions used in statistics. These functions are widely used for data analysis or data science.

| Function                               | Description                                           |
|----------------------------------------|-------------------------------------------------------|
| <code>Statistics.mean(list)</code>     | Returns arithmetic mean of list or data given by user |
| <code>Statistics.median(list)</code>   | Returns median value of list given by user            |
| <code>Statistics.mode(list)</code>     | Returns mode (highest frequency) value given by user  |
| <code>Statistics.stdev(list)</code>    | Returns standard deviation of list given by user      |
| <code>Statistics.variance(list)</code> | Returns variance of list given by user                |

## 12.7 SUMMARY

In this unit, we have discussed modules and package creations in details. Modules are python files which can be imported in other files. A package is a folder which can store multiple modules and sub-packages within. Moreover,

built-in modules are also discussed in details that add real power to python programming.

---

## SOLUTION TO CHECK YOUR PROGRESS

---

### check your Progress 1

Ex.1 A **Module** is a logical group of functions , classes, variables in a single python file. A major benefit of a module is that functions, variable or objects defined in one module can be easily used by other modules or files, which make the code re-usable.

A module can be created like any other python file i.e. with .py extension. Name of the module is the same as the name of a file.

Ex. 2 The various methods of importing a module are

1. using *import* statement
2. using *from import* statement
3. using *from import \** statement

1. *import module*  
module.function()

This method is used to import the entire module. Individual functions can be used with module name and .symbol.

2. *from module import function*  
function()

This method is used to import individual function from a module. In this method, function can be directly called with its name.

3. *From module import \**  
function()

This method can be used to import the entire module using from import \* statement. Here, \* represents all the functions of a module. Like previous method, an object can be accessed directly with its name.

Ex. 3 The 3 built-in modules in python are –os, math, random.

### check your Progress 2

Ex. 1 A package is a directory which contains multiple python modules. It is used to group multiple related python modules together. A python package in addition to modules must contain a file called `__init__.py`. This file may be empty or may contain data like other modules of package.

Ex. 2 When we use import statements to import a module, it is searched in a list of directories or search paths stored by the environment variable PYTHONPATH. This is called **module search path**. This list of directories can be checked using **sys.path** variable. Any modules created must be located in python's search path for its global identification.

Modules can be added to search path by either of the ways-

1. Creating module in one of the locations already present in search path
2. Adding module path in the search path using sys.path.
3. Updating PYTHONPATH environment variable.

Ex. 3 First of all, create a folder named area and place 4 file named – circle.py, square.py, rectangle.py and \_\_init\_\_.py in folder.

circle.py

```
circle.py - C:\Users\test\Desktop\python_programs\area\circle.py (3.7.6)
File Edit Format Run Options Window Help

def circle(r):
 import math
 return math.pi*r**2
```

square.py

```
square.py - C:\Users\test\Desktop\python_programs\area\square.py (3.7.6)
File Edit Format Run Options Window Help

def square(side):
 return side*side
```

rectangle.py

```
rectangle.py - C:\Users\test\Desktop\python_programs\area\rectangle.py (3.7.6)
File Edit Format Run Options Window Help

def rectangle(l,b):
 return l*b
```

Now, we can import the package along with all the modules in any file.

```
from area.circle import *
from area.square import *
from area.rectangle import *

print("Area of Circle:", circle(4))
print("Area of Square:", square(4))
print("Area of Rectangle:", rectangle(3,4))
```