



Uttar Pradesh Rajarshi Tondon
open University

SBSCS-02

Python Programming

Python Programming

Block - 1 BASICS OF PYTHON	3
Unit 1 Introduction to Python	6
Unit 2 Tokens and Statements	27
Unit 3 Data Types, Operators & Expressions	47
Block - 2 BASICS OF PYTHON	81
Unit 4 Data Structures	84
Unit 5 Functions	103
Unit 6 Modules and Packages	127
Block - 3 OOPS IN PYTHON	151
Unit 7 Object Oriented Programming in Python	154
Unit 8 Exception Handling	184
Unit 9 Python Libraries	203
Unit 10 GUI Programming and Testing	232
Block - 4 MACHINE LEARNING IN PYTHON	277
Unit 11 Machine Learning using Python	280
Unit 12 Regression and Classification in Machine Learning	306



Uttar Pradesh Rajarshi Tondon
open University

SBSCS-02

Python Programming

Block

1

BASICS OF PYTHON

Unit 1	6
Introduction to Python	
Unit 2	27
Tokens and Statements	
Unit 3	47
Data Types, Operators & Expressions	

Course Design Committee

Prof. Ashutosh Gupta Director (In-charge) School of Computer and Information Science, UPRTOU Prayagraj	Chairman
Prof. R. S. Yadav Department of Computer Science and Engineering MNNIT Prayagraj	Member
Dr. Marisha Assistant Professor (Computer Science), School of Science, UPRTOU Prayagraj	Member
Mr. Manoj Kumar Balwant Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Member

Course Preparation Committee

Dr. Brajesh Kumar Associate Prof. Dept. of CSIT, MJPRU, Bareilly, 243006	Author Block – 01 to 04 Unit – 01 to 12
Prof. Manu Pratap singh Prof. Department of Computer Science Dr. Bhimrao Ambedkar University, Agra - 282002	Editor Block – 01 to 04 Unit – 01 to 12
Dr. Chandrakant Kumar Singh Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Coordinator

© UPRTOU , Prayagraj

ISBN :

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar Pradesh Rajarshi Tandon Open University.

Reprinted and Published by Vinay Kumar, Registrar Uttar Pradesh Rajarshi Tandon Open University, 2024

Printed by - Chandrakala Universal Pvt. Ltd. 42/7 Jawahar Lal Nehru Road, Prayagraj.

Overview:

This block is about the basic fundamentals of Python programming language. It provides the insight on the concepts that a programmer must understand before learning the advanced concepts of any programming language. This block is divided into 3 units. The 'Unit 1' provides information on the development of Python, its need, and application areas. In 'Unit 2', preliminary concepts of Python programming language such as variables, constants, identifiers, input-output, keywords, and statements, etc. Some other important basic concepts such as data types, operators, expressions, loops, and flow-control, etc. are discussed in 'Unit 3'.

UNIT-1: Introduction to Python

Structure:

- 1.0 Introduction**
- 1.1. Objective**
- 1.2. History of Python**
- 1.3. Features of Python**
- 1.4. Need of Python**
- 1.5. Applications of Python**
- 1.6. Installation of Python**
- 1.7. REPL Shell**
- 1.8. Python IDLE**
- 1.9. Summary**
- 1.10. Review Questions**

1.0 Introduction

A program is a set of instructions that directs a computer to do specific tasks. Programs are written in a computer programming language to control the behaviour of a machine. A program written in the programming language is a collection of human-readable computer instructions. However, for a computer to understand and execute a source program it must be translated into machine language. This translation is done using either compiler or interpreter. A compiler is a program that transforms source code into a low level machine language. The process of converting program into machine language is known as compilation. The machine code generated by compiler can be later executed any number of times using different data. An interpreter is a program that reads source code statement by statement, translates the statement into machine code and executes the statement. After executing the statement, it continues with the next statement. Generally it is faster to execute a compiled code than to run a program by an interpreter. Python, Perl, and Ruby are some leading interpreter based programming languages.

There are various programming paradigms such as imperative, logical, functional, and object oriented. In imperative programming paradigm a program describes a sequence of steps that change the state of the computer by explicitly instructing the computer to accomplish a certain goal. Structured programming is a kind of imperative programming that makes use of looping structures. Procedural programming is another kind of imperative programming, where procedures are used to describe operations to perform. A sequence of instructions is combined into a procedure such that these instructions can be invoked any number of times without duplicating the same instructions. A procedure is similar to function with a major difference that unlike functions a procedure cannot return a value. Programming language C is imperative and structured in nature. The logical paradigm uses basic facts and rules. Rules and facts are written as logical clauses. A rule clause has a head and a body, while facts are expressed with a head only. PROLOG and Datalog are the examples of logical programming language.

In functional programming paradigm, the programs are composed as functions. Functional programming languages consider functions as first-class objects, which can be passed as an argument to another function as arguments and may be returned by another function just like data types. LISP and Haskell are examples of functional programming languages. In recent decades, the object oriented programming paradigm has become popular. It allows to organize the program as a collection of objects that consist of both data and functions. Object oriented programming considers data as a critical element in the program development. It does not allow free access to the data. It ties data and functions more closely to protect it from accidental modification. The data

of an object is supposed to be accessed only by the associated functions. However, functions of one object can be accessed by the functions of other objects. Abstraction, encapsulation, inheritance, and polymorphism, etc. are the important properties of object oriented programming. Python, C++, C#, and Java are the popular object oriented programming languages. Some languages may support features of multiple paradigms. On the other hand a language may not support all the features of a particular paradigm.

1.1 Objectives

The major objectives of this unit are outlined as follows.

1. To provide an insight into the development history of Python.
2. To discuss the basic characteristics of Python.
3. To discuss need and applications of Python.
4. To give an overview of the procedure to execution of Python scripts.

1.2 History of Python

Python is a programming language that follows multiple programming paradigms. It can be considered as a strong, procedural, object-oriented, functional programming language. Python was conceived in the late 1980s by Guido Van Rossum at CWI in the Netherlands as a successor to the ABC programming language. Its implementation started in December 1989 by its principal author Guido Van Rossum. The language was named after a popular comedy TV show “Monty Python’s Flying Circus” by Guido Van Rossum who was a big fan of the show. The design philosophy of Python emphasizes on code readability and simplicity. Python follows multiple programming paradigm with full support to object oriented programming and Structured programming along with some features of functional programming.

The first ever version of Python called Python 1.0 was introduced in year 1991. Later, the Python 2.0 was released in 2000. The second version had many new features including a cycle detecting garbage collector, list comprehension, argument assignment, string operations, support for unicode, IDLE improvements, DOM support, and SAX2 support, etc. With Python 2.0, the Python programming language became really popular with community backing. A major revision of the language was released in 2008 as Python 3.0 after a rigorous testing. The Python 3.0 is not completely backward compatible. After Python 3.0, several versions have been released with minor changes and additions. At present (in April 2021), the current version is Python 3.9x that provides excellent support to machine learning, data science, natural language processing, and many other advanced technologies with its efficient libraries.



Figure 1.1: Creator of Python- Guido Van Rossum

(Source: www.wikipedia.org)

According to *The Zen of Python*, there are 19 nineteen principles for writing computer programs.

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one— and preferably only one —obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea—let's do more of those.

The 20th entry left open as “Guido to fill in”, which still could not be filled up. There are many third party modules for accomplishing the above tasks. For example Django is an immensely popular framework for web development in Python. There are many third party libraries for software development. For examples Scions, which is used for “build controls”.

Check your progress

1. What are the major programming paradigms?
2. What are the major differences between compiler and interpreter?
3. Give examples of compiler based programming languages.
4. Give examples of interpreter based programming languages.
5. Python programming language follows which programming paradigms?
6. Differentiate procedure and function.
7. Write examples of the programming languages that follow object oriented programming paradigm.
8. Who is the creator of the Python?
9. How was the Python named?
10. List the major new features that were incorporated in Python version 2.0.
11. What is the current version of Python?
12. List principles to write a computer program as per *The Zen of Python*.

1.3 Features of Python

Python is a powerful programming language, which is simple but having a number of attractive features, built-in objects, and libraries. The important features of Python are discussed here.

1.3.1 Free and Open-Source

Python is developed under an OSI-approved open source license. It is available for everyone completely free for all kind of usage including commercial purposes. Anyone can freely download it from its official website (www.python.org) without any cost. Its code is available in public domain, therefore it can also be modified and redistributed free of cost.

1.3.2 Easy to Code

Python is known for its simple syntax similar to English language that makes it easy to learn and understand. Python emphasizes on the readability that reduces the cost of program maintenance compared to languages like C, C++, and Java. The removal of braces and parentheses makes the code short and sweet. As stated by many authors, three main features that make Python attractive are 'simple', 'small', and 'flexible'. Despite being simple, it is developer friendly. Python is highly expressive that allows to write code for complex tasks within a few lines. Therefore, Python is attractive both for beginners and developers. The Python code can also be extended to other programming languages such as C, C++, and Java.

1.3.3 Portable

Python is a portable programming language. The same code written on one machine can be used on different machines without making any changes to the code. For example, if a Python program is written on Mac machine, it can also run on Windows or Linux machine. Therefore, there is no need to write the same program separately for several platforms.

1.3.4 Dynamic Typing

Python is a dynamically-typed language. It has its own way of managing memory associated with objects. It means that when an object is created in Python, memory is dynamically allocated to it. At the end of the scope of the object, the memory is automatically deallocated. It allows efficient utilization of the system memory. Python does not require to declare the type of the variable in advance. The data type of the variables is decided at run time. Therefore, the programmers need not to specify the type of the variables in the program.

1.3.5 Interpreted

Python is an interpreter based programming language. It means that the source code is executed line by line, and there is no need to check the entire program all at once. Although it makes execution of Python programs little slower compared to compiler based languages like C and C++, but it can be run easily. On the other hand, the testing and debugging of the code becomes easy.

1.3.6 Object-Oriented, procedural, and functional high level language

Python is high level programming language that means that programmers do not need to have the knowledge of the underlying machine architecture to write the program. Python borrows the features from multiple programming paradigms. Primarily it is an object-oriented programming language that supports the concepts of class, object, abstraction, encapsulation, inheritance, etc. In addition to object-oriented features, Python also supports some features from procedural and functional programming.

1.3.7 GUI Programming Support

Graphical User Interface (GUI) provides a way to the users to easily interact with the software. One of the key aspects of any programming language is support for GUI. Python provides good support for writing GUI based programs. Python offers various toolkits, such as Tkinter, wxPython and JPython, which allows for GUI's easy and fast development. PyQt5 is the most popular option for creating graphical apps with Python.

1.3.8 Numerous Robust Libraries and Tools

Python makes programming really easy especially for the complicated tasks. There are numerous and robust libraries and tools in Python that provide a rich set of modules and functions so that programmers do not need to write their own code for every single thing. There are libraries for various applications and technologies such as image manipulation, pattern analysis, data visualization, scientific calculations, databases, remote sensing, machine learning, unit-testing, natural language programming, and a lot of other functionalities. In addition to the standard libraries, there are also a growing collection of thousands of components available as Python packages.

1.4 Need of Python

Python is becoming immensely popular among programmers and developers. Tech Giants like Cisco, IBM, Mozilla, Google, NASA, Quora, Hewlett-Packard, Dropbox, and Qualcomm are using this language owing to its simplicity and elegance. Most developers prefer Python over the plethora of programming languages out there because of its emphasis on readability and efficiency. The incredible growth of Python during last few years is shown in Figure 1.2. As shown in the graph, Python has become immensely popular during past 2-3 years due to its attractive features.

Python is used in Artificial Intelligence, Automation Scripting, Data Mining, Data Science, Embedded Systems, Gaming, Graphic Design, Machine Learning, Network Development, Product Development, Rapid Application Development, Web Development, Web Frameworks, Testing, and many more areas. Python also has a strong community around machine learning, data

modelling, data analysis and AI with extensive resources and libraries built for these purposes. In the past few years, many packages have been developed for data analysis and machine learning using Python. This includes numpy, pandas, tensorflow, and keras, etc. Nowadays Linux/Unix distributions include recent Python. Some Windows based computers also come with pre-installed Python.

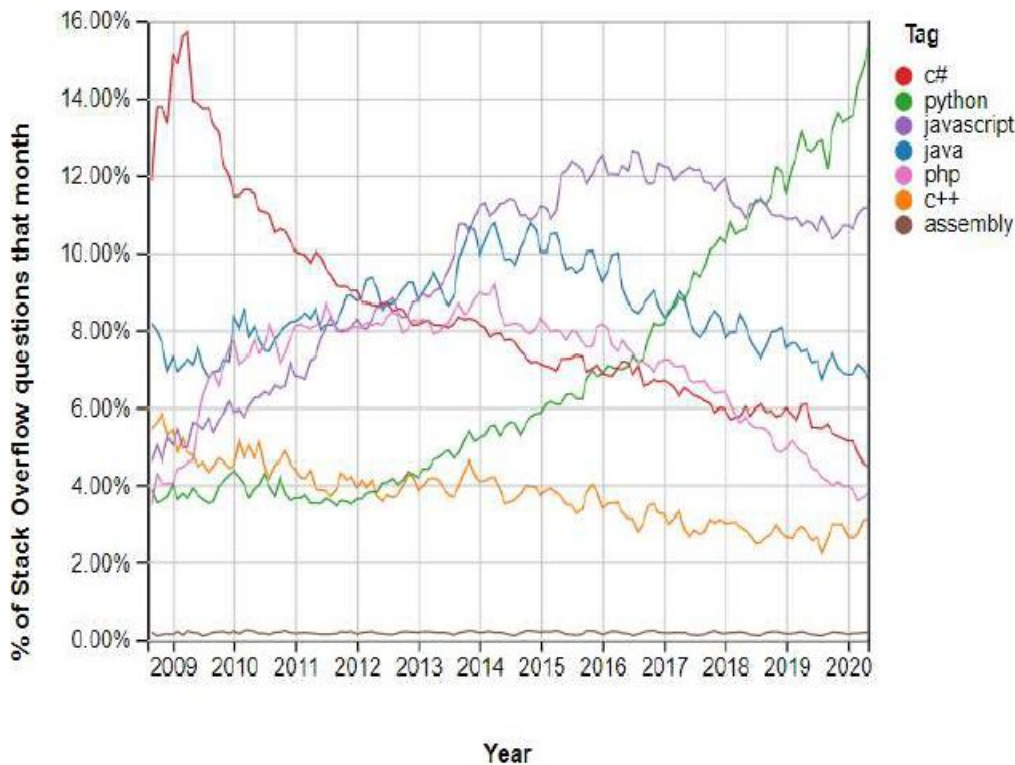


Figure 1.2: Incredible growth of Python.
(Source: StackOverflow)

Python is widely used in academia, industry, and scientific research around the world. Python is highly appreciated for facilitating the productivity, quality, and maintainability of software.

1.5 Applications of Python

Python is being used nowadays to accomplish many tasks in many different areas. The major thrust areas of Python are discussed here.

1.5.1 Artificial Intelligence and Machine Learning

Artificial Intelligence (AI) and Machine Learning (ML) projects are inherently different from traditional software projects. The tools, technologies, and skillset required for AI/ML are totally different from those required in the development of conventional software development. AI/ML applications require a stable, secure, and flexible language that is equipped with tools to

handle various unique requirements of such projects. Python has all these qualities, and hence, it has become one of the most preferred languages for such applications. The simplicity, consistency, portability, good collection of libraries and packages, and an active community of Python make it the perfect choice for developing AI/ML applications. Some of the best Python packages for AI/ML are: SciPy, Pandas, Seaborn, Keras, TensorFlow, Scikit-learn, and NumPy. Scikit-Learn, Keras, and TensorFlow are a well-known ML tools. Scikit-Learn is built on top of other scientific tools such as NumPy, SciPy and Matplotlib. It supports various models for classification, clustering, regression, feature selection, and dimensionality reduction, etc. Seaborn and Matplotlib provides support for data visualization.

1.5.2 Natural Language Processing

Natural language is a language such that English, Hindi, and Tamil etc. that is used for general communication by human beings. Natural Language Processing (NLP) is a branch of AI that deals with any kind of computer manipulation of natural language. It provides a way to human and machine interaction using natural languages. The ultimate objective of NLP is to read, understand, and make sense of the human languages in a manner that is valuable. NLP is about developing applications and services that are able to understand human languages. Some practical examples of NLP are as follows:

1. Text data processing
2. Audio to text conversion
3. Data to audio conversion
4. Human-machine interaction through voice e.g. Interactive Voice Response (IVR)
5. Language to language translation e.g. Google Translate
6. Sentiment analysis
7. Word processing to check grammatical accuracy e.g. Grammarly
8. Personal assistant applications e.g. Alexa, OK Google, etc.
9. Search engines e.g. Yahoo, Google, etc.
10. Spam filters
11. Social website feeds

Natural Language Toolkit (NLTK) is the popular library used for NLP in Python. NLTK consists of numerous trained algorithms for text processing. NLTK has huge collection of datasets and lexical resources such as chat logs, journals, and movie reviews, etc. NLTK provides an infrastructure to build NLP programs in Python. It defines basic classes for data representation relevant to NLP, standard interfaces, syntactic parsing, text classification, and standard implementations for solving complex problems.

1.5.3 Data Analysis

Data analysis is the process of mining large amounts of complex data with the purpose of discovering relevant information and supporting decision making. Data analysis makes decision making more scientific to help businesses operating effectively. Data analysis is used in different kind of organizations to make better decisions.

Python makes Pandas library available under BSD license for data analysis. Pandas provides two important data structures, namely Series and Data Frame, which can hold any type of data such as integer, float, string, and objects, etc. Each of the data stored in series is labelled after the index. Data Frame is a tabular data structure with labelled rows and columns similar to Excel spreadsheet.

1.5.4 Cloud Computing

Cloud computing technology is used to provide on-demand delivery of IT resources over the Internet especially the data storage and computing resources. The users do not need to purchase and maintain the devices or software. Instead, they can get access on need basis with pay-as-you-go pricing.

Python has the ability to deploy applications on any platform including the cloud. The Python programs can be executed on the cloud servers. OpenStack is the most widely deployed open source cloud software entirely written in Python. OpenStack is highly reliable, vendor independent and has decent load balancing and built-in security. Linux distributions like Ubuntu and Fedora include OpenStack as part of their packages. OpenStack is well supported by well-known cloud service providers including Microsoft Azure, Amazon Web Services (AWS), Google App Engine, and Heroku. PiCloud is a cloud computing platform that integrates into the Python and leverages the computing power of AWS without having to manage, maintain, or configure virtual servers.

1.5.5 Web Development

Python offers excellent support for web development. It provides many web development frameworks including bottle.py, CherryPy, Django, Flask, Pyramid, and web2py. Django and Flask are the two most popular Python web frameworks. Django is a full-fledged framework having built-in support for services such as caching, serialization, internationalization, ORM support, and automatic admin interface, etc. Django is fast, secure, and scalable that provides strong community support. Flask is a micro-framework that is used to create small web applications with minimal requirements. User can configure

web services with Flash according to their needs using external libraries. Both Django and Flash frameworks are available under BSD derived licenses.

1.5.6 Database Programming

Python supports various databases such as MySQL, Oracle, PostgreSQL, and Sybase, etc. for database programming. It supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements. For database programming, the Python DB API is a widely used module that provides a database application programming interface. It provides a set of standard interfaces to access specific databases through Python programs.

1.5.7 Game Development

Python is becoming an exceptional choice for the game developers for the prototyping of the video games. Python provides a number of libraries that allows the game development with less efforts. Battlefield 2 and Pirates of the Caribbean are two well-known games developed in Python. Pygame is an excellent tools for rapid game development in Python. It is built on top of the SDL library, which is a combination of C, Python, Native, and OpenGL. Pygame enables programmers to develop fully featured highly portable games as well as multimedia programs. PyKyr, Pyglet, PyOpenGL, Kivy, Panda3D, and Cocos2D, Python-Ogre, Ren'Py, etc. other successful frameworks for game development.

1.5.8 Scientific Applications

Python is a good choice for high performance computing in academia and scientific projects for its simplicity and efficiency. Python supports the use of external libraries typically written in other faster languages like C and Fortran for matrix operations. SciPy is a library that comes with modules for various commonly used tasks in scientific programming. It uses NumPy for more mathematical functions for linear algebra, calculus, differential equation solving, and signal processing. Numba, xarray, PsychoPy, Matplotlib, and Rpy2 are other scientific libraries in Python. It provides good support for parallel processing with processes and threads, interprocess communication, and GPU computing. Python has a large community of users to find help and documentation.

1.5.9 Image Processing

Image processing is the process of analysing and manipulating a digital image for improving its quality or for extracting useful information aiming at specific purpose. Due to growing popularity and availability of state of the art image

processing tools, Python is an excellent choice for image processing. A number of Python libraries are available that provide in-built functions for performing tasks such as classification, segmentation, feature extraction, image restoration, and image recognition, etc. Commonly used Python libraries for image manipulation are Scikit-image, Scipy, Numpy, PIL/Pillow, Mahotas, OpenCV-Python, SimpleITK, Pycairo, and pgmagic, etc.

1.5.10 Graphical User Interface

Python provides multiple options for creating Graphical User Interface (GUI). The most important options are Tkinter, wxPython, Jpython, and PySimpleGUI. Tkinter is the standard library for GUI that provides a powerful object-oriented interface to create GUI applications in Python in fast and easy way. Tkinter provides various controls, such as buttons, labels and text boxes commonly called as widgets. Python and the PySimpleGUI package can be used to create a simple GUI that works across multiple platforms. PySimpleGUI is gaining a lot of interest recently for creating nice-looking user interfaces. PySimpleGUI wraps Tkinter, PySide2, and some portion of wxPython. After the installation of PySimpleGUI, the user gets Tkinter variant by default. A large variety of different cross-platform GUIs can be created using PySimpleGUI. Anything from desktop widgets to full-blown user interfaces can be created using this tool.

1.5.11 Academia

Academic use of Python has made significant progress at various levels in recent years. Its simple syntax and procedural, functional, and object oriented approach along with the availability of high level data structures makes it an ideal choice for problem solving. It has been introduced as part of the curricular in academia to teach the programming. The researchers from different fields have adopted Python for developing their frameworks and performing experiments. According to StackOverFlow, use of Python in academic research is growing fast. Students are making use of Python for developing their projects more than any other language nowadays.

1.5.12 Business Applications

Success of Python in business applications is incredible. Python is one of the most preferred and fastest growing language for developing modern applications. The top applications of Python development in businesses include web development, data science solutions, machine learning frameworks, natural language processing, video gaming, entertainment applications, scientific computing, GUI, blockchain enabled applications, and finance, etc. Due to its numerous benefits and massive potential Python is likely to

dominate the software development world with its brilliant libraries, modules, and frameworks.

1.5.13 Audio-Video Processing

Python has some built-in multimedia modules for audio video processing. The `winsound` module provides an interface to implement audio-playing elements. `Audioop` module can be used for manipulating the raw audio data. Several operations can be performed on sound fragments using it. `PyMedia` is another popular media library that allows manipulation of audio/video supports a wide range of multimedia formats like `mp3`, `mpg`, `avi`, and `Ogg`, etc. `Pyglet` is a popular multimedia frameworks for the development of graphically intense applications including games. It is an `OpenGL`-based library supports animation and gaming applications. It provides an API for developing multimedia applications using Python. `MoviePy`, `vidgear`, and `pyAudioAnalysis`, etc. are some other libraries that support audio and/or video manipulation in Python.

1.5.14 3D CAD Applications

Python supports a number of functionalities for 3D CAD as well as CAM applications like `AnyCAD`, `Blender`, `CAMVOX`, `Fandango`, `FreeCAD`, `HeeksCNC`, `HeeksPython`, `PythonCAD`, `PythonOCC`, and `Vintech RCAM`, etc. `CAMVOX` is a 3D CAM application written in `C++` and Python to calculate tool paths. `FreeCAD` is based on `Open Cascade`, `Qt` and Python that supports full macro recording and editing, fully customizable GUI, and Workbenches, etc. `PythonCAD` is an easy to use CAD package for Linux/Unix. `Blender` is an open source 3D graphics Python application that can be used for animating, compositing, modelling, non-linear editing, rendering, rigging, skinning, texturing, UV unwrapping, water and other simulations, and creating interactive 3D applications. `PythonOCC` is a 3D CAD/PLM development library that is built upon the `OpenCASCADE` 3D modelling kernel. All these libraries help 3D CAD/CAM applications.

Check your progress

1. Write official website of Python.
2. Can Python code be extended to other programming languages?
3. Is Python portable?
4. How is the data-type of a variable declared in Python?
5. Write important Python packages for AI and ML applications.
6. What is NLTK?
7. Write two most popular Python web frameworks.
8. Give major successful frameworks for game development.
9. Write major Python libraries for scientific computing.
10. What are standard libraries for GUI development?
11. What is purpose of PyMedia library?
12. Which libraries are used for 3D CAD/CAM applications?

1.6 Installation of Python

Most Linux distributions include Python language by default. But Windows does not come with preinstalled Python. However, Python can be installed on Windows computers in a few easy steps. The step by step procedure for Python installation is explained here.

1. Download Python.exe installer from the official website of Python www.python.org according to the architecture of your computer. It could be Python 2 or Python 3 as per need as shown in Figure 1.3.

Python Releases for Windows

- [Latest Python 3 Release - Python 3.9.4](#)
- [Latest Python 2 Release - Python 2.7.18](#)

Stable Releases

- [Python 3.9.4 - April 4, 2021](#)
Note that Python 3.9.4 cannot be used on Windows 7 or earlier.
 - [Download Windows embeddable package \(32-bit\)](#)
 - [Download Windows embeddable package \(64-bit\)](#)
 - [Download Windows help file](#)
 - [Download Windows installer \(32-bit\)](#)
 - [Download Windows installer \(64-bit\)](#)
- [Python 3.9.3 - April 2, 2021](#)
Note that Python 3.9.3 cannot be used on Windows 7 or earlier.
 - No files for this release.
- [Python 3.8.9 - April 2, 2021](#)
Note that Python 3.8.9 cannot be used on Windows XP or earlier.
 - [Download Windows embeddable package \(32-bit\)](#)

Pre-releases

- [Python 3.10.0a7 - April 5, 2021](#)
 - [Download Windows embeddable package \(32-bit\)](#)
 - [Download Windows embeddable package \(64-bit\)](#)
 - [Download Windows help file](#)
 - [Download Windows installer \(32-bit\)](#)
 - [Download Windows installer \(64-bit\)](#)
- [Python 3.10.0a6 - March 1, 2021](#)
 - [Download Windows embeddable package \(32-bit\)](#)
 - [Download Windows embeddable package \(64-bit\)](#)
 - [Download Windows help file](#)
 - [Download Windows installer \(32-bit\)](#)
 - [Download Windows installer \(64-bit\)](#)
- [Python 3.9.2rc1 - Feb. 16, 2021](#)
 - [Download Windows embeddable package \(32-bit\)](#)
 - [Download Windows embeddable package \(64-bit\)](#)

Figure 1.3: Python releases for Windows

2. Run the Python Installer once downloaded. A dialog box appears as shown in Figure 1.4. It allows you to customize your installation or you can also choose the default option. There are two checkboxes **Install launcher for all users** and **Add Python 3.9 to PATH**. Do not forget to tick both the checkboxes. Click on the **Install Now** for the recommended installation option. For all recent versions of Python, the recommended installation options include **Pip** and **IDLE**. Older versions might not include such additional features. When you click on **Install Now**, the installation starts as shown in Figure 1.5.
3. The next dialog prompts you to select whether to **Disable path length limit**. As shown in Figure 1.6. If this option is chosen, it allows Python to bypass the 260-character `MAX_PATH` limit. Effectively, it enables Python to use long path names. The **Disable path length limit** option does not affect any other system settings. If you turn it on, it resolves potential name length issues on Linux. Therefore you can leave it as it is and close the dialog box without any potential issues.



Figure 1.4: Python installer

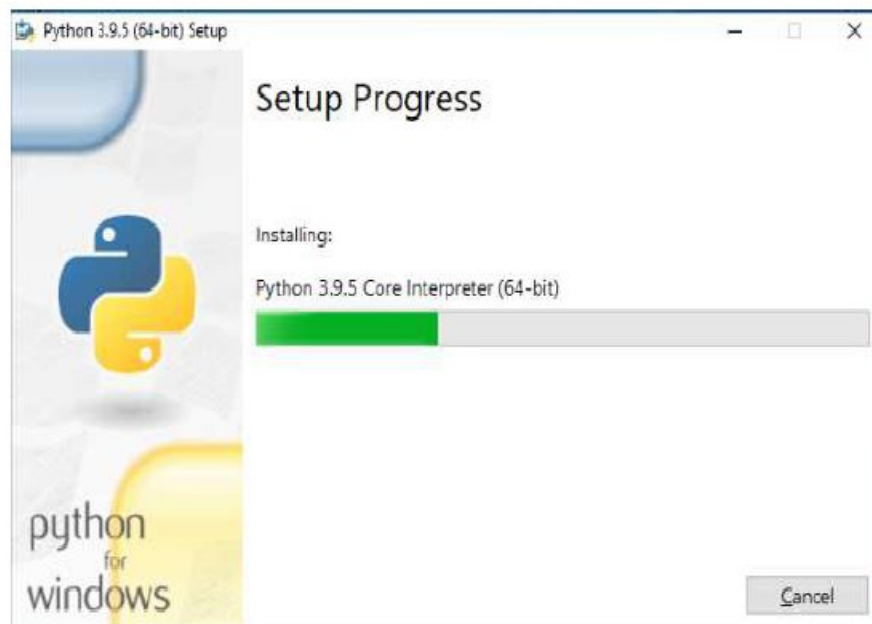


Figure 1.5: Progress of installation

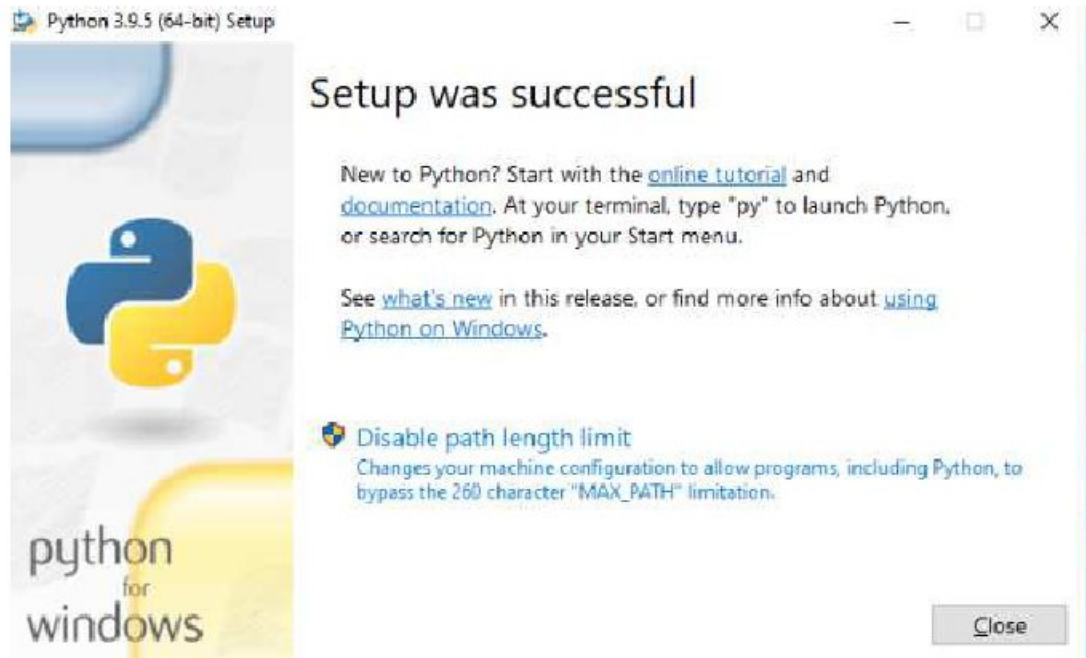


Figure 1.6: Successful installation of Python

4. After successful installation process, you need to verify the installation. Follow the given process to verify the Python installation.
 1. Navigate to the directory where Python was installed.
 2. Double-click **python.exe**. As a result **Command Prompt-Python** appears.
 3. You can also verify the version of the Python from the command prompt.
5. You can also verify the installation of Pip as follows:
 1. Open the **Start** menu and type “cmd.”
 2. Select the **Command Prompt** application.
 3. Enter **pip -V** in the console. If Pip was installed successfully, you should get version of Pip.

The above process can be used to install any version of Python. User should carefully choose appropriate version of Python as per need.

Red Hat Enterprise Linux (RHEL) is available with pre-installed Python. The updated version of Python can be obtained through **Red Hat Software Collections**. Various popular Python modules are installed on RHEL by default. In order to verify the availability of Python at RHEL system, the following procedure can be used.

1. Open a Terminal window.
2. Use **sucommand** to become root user.

\$su -

3. Run following **yum** command to see the listing of installed Python packages.

```
# yum list installed python\*
```

All the installed Python packages are listed by the above procedure. If Python is not installed, it can be installed by following yum command:

```
# yum install python
```

To check other Python modules included with RHEL, run the following command:

```
# yum list available python\*
```

A Python program or script is saved as a file having extension **.py** e.g. **test.py**. The program can be executed from the command prompt by making the file executable as follows.

```
$chmod +x test.py
```

```
$./test.py
```



Figure 1.7: Open REPL shell

1.7 REPL Shell

The Python experiments for learning purpose can be easily performed using the **interactive interpreter** also known as **shell**. The shell is a basic Read-Eval-Print Loop (REPL) that reads and evaluates a statement and then prints the result on the screen. After printing the result, it loops back to read the next statement. Unlike running a file containing Python code, you can type single Python command in the REPL and display the output instantly. REPL can also be used to print out help for methods and objects. To run the Python REPL Shell on Windows, open the command prompt and write **python** and press **enter** as shown in Figure 1.7. A Python Prompt comprising of three greater-than symbols **>>>** appears, as shown in Figure 1.8. Now a single command can be entered and the result is displayed on the screen in the next line. For example in Figure 1.9, first a command **a=5** is entered then commands **b=6**,

`c=a+b`, and `print(c)` are used in subsequent lines. Instead of single statement, Python script can also be executed in REPL shell. Write the script in a text editor like notepad and save the file with `.py` extension. Suppose if file is saved `test.py`, then it is executed using the following command.

```
>>>python test.py
```

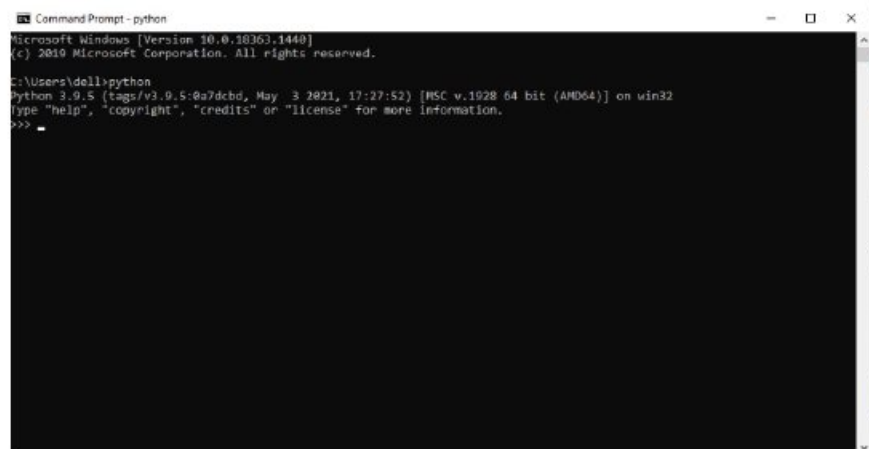


Figure 1.8: Python prompt

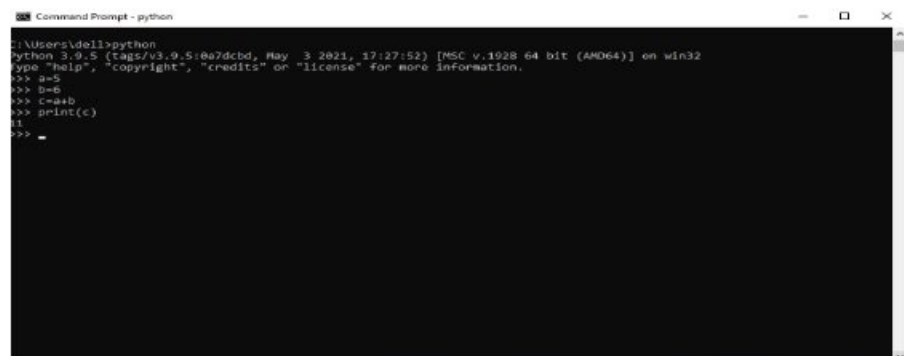


Figure 1.9: Python prompt

1.8 Python IDLE

Integrated Development and Learning Environment (IDLE) provides an integrated development environment for Python. It provides an interactive environment to write and execute the programs. Python IDLE comes included with installations for Windows and Mac platforms. Linux users have to download it from their package manager. Following command can also be used to install IDLE on Linux (Ubuntu distribution).

```
$sudo apt install idle
```

IDLE can be used to execute a single statement just like Python REPL Shell and execute Python scripts. IDLE also provides a fully-featured text editor to create and modify Python script that includes features like syntax highlighting, autocompletion, and smart indent. It also has a debugger with stepping and breakpoints features. To start an IDLE interactive shell, search for the IDLE icon in the start menu and double click on it to open IDLE as shown in Figure

1.10. You can write, modify, and execute the Python scripts in IDLE as shown below. You can execute Python statements or Python scripts same as in Python Shell as illustrated in Figure 1.11.

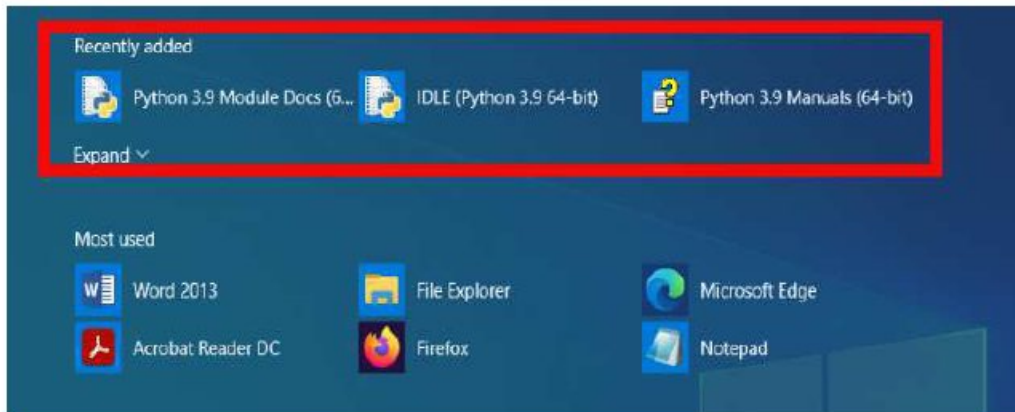


Figure 1.10: Python IDLE icon

1.9 Summary

Python is one of the fastest growing programming language that follows multiple programming paradigms. It can be considered as a strong, procedural, object-oriented, and functional programming language. Python was created by Guido Van Rossum at CWI in the Netherlands in late 1980s. The first ever version of Python called Python 1.0 was introduced in year 1991. Python 3.9x is its current version. Python is developed under an OSI-approved open source license. It is an open-source, which can be freely downloaded from its official website. Python emphasizes on the readability and is highly expressive. It is an interpreter based language that highly portable and supports a large number of libraries and tools for numerous applications. Python is widely used in academia, industry, and scientific research around the world. Python is highly appreciated for facilitating the productivity, quality, and maintainability of software. Python finds its application in a number of areas including AI, ML, NPL, data science, cloud computing, web development, game development, scientific applications, GUI, and audio-video processing, etc. Python comes pre-installed with Linux distributions. The Windows installer of Python can be downloaded from its official website. Python commands and scripts can be executed using REPL shell or IDLE.

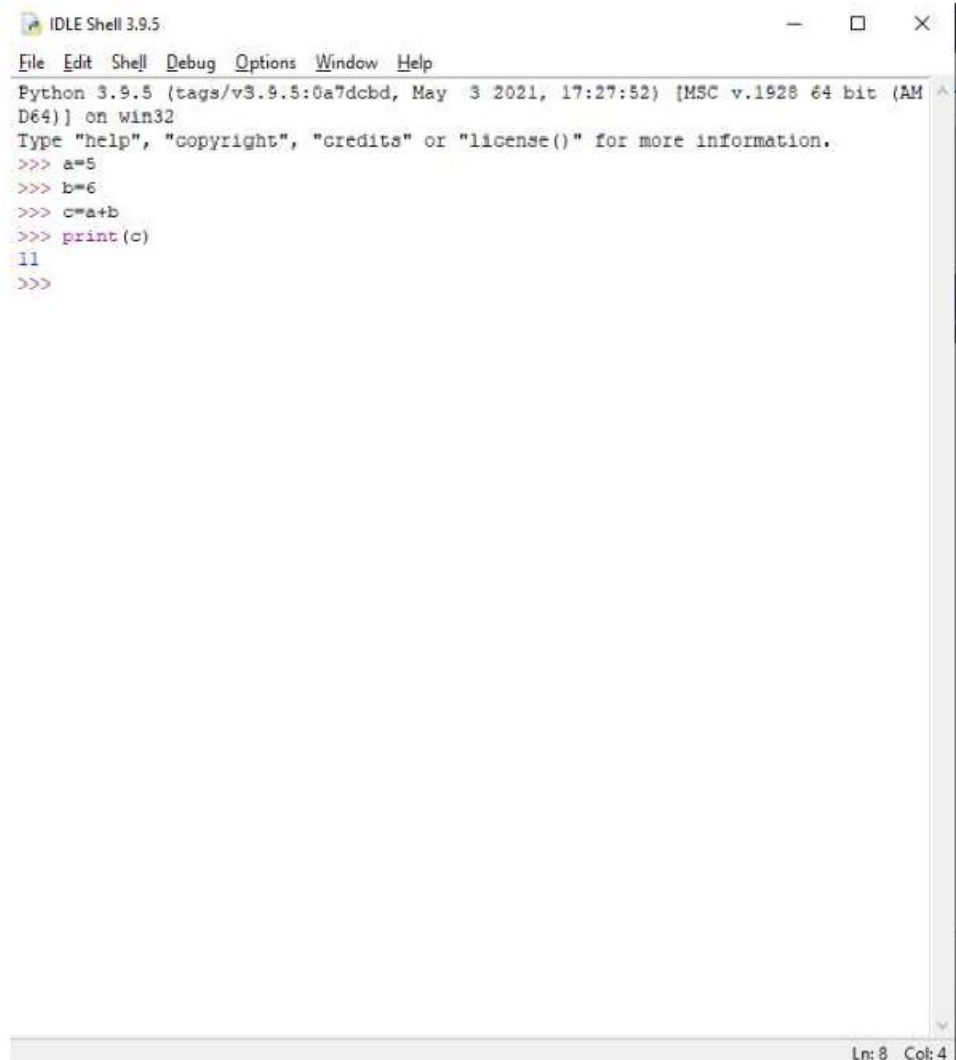


Figure 1.11: Python IDLE

1.10 Review Questions

- Q.1 What are major programming paradigms? Explain the major features of programming paradigms.
- Q.2 Write nineteen principles according to The Zen of Python.
- Q.3 Write and explain the major features of Python.
- Q.4 What are the major application areas of the Python.
- Q.5 Write important libraries for different applications of Python.
- Q.6 How is Python installed at different types operating systems? Explain.
- Q.7 What is REPL? What is the utility of REPL? Explain.
- Q.8 How is IDLE installed? Differentiate REPL and IDLE.

UNIT-2: Tokens and Statements

Structure:

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Identifiers
- 2.3 Variables
- 2.4 Constants
- 2.5 Keywords
- 2.6 Punctuation
- 2.7 Indentation
- 2.8 Statements
- 2.9 Comments
- 2.10 Input-Output
- 2.11 Summary
- 2.12 Review questions

Unit 2: Tokens and Statements

2.0 Introduction

In this unit, the fundamentals of Python programming language are discussed that are required to write simple Python programs. It is described how a Python program works at the most basic level and the fundamental concepts related to identifiers, variables, operators, keywords, and statements are discussed. One need to be familiar with these concepts before understanding many other complex functionalities. Tokens are the smallest individuals unit of a program. Tokens can be divided into following major categories: identifiers, literals, operators, separators, keywords, and comments, etc. An identifier is a name given to identify an entity or group of entities in programs. It is a string of characters and numbers. Each programming language provides some rule to write legitimate identifiers. It is necessary to store data or information somewhere that can be accessed or referenced by the programs. A variable provides the way out to store and label the data in memory. The labelling is done with a descriptive name. Identifiers provide a mechanism to name a variable. The variables should be named in such a way that variable is accurately descriptive for other readers. Some names are reserved for special purpose in the programming languages. These names or words are called keywords. Any user defined entity cannot be named with these keywords.

Programs are written in terms of instructions known as statements. Python statements are logical instructions that can be read and execute by the interpreter. A statement can be a single and multiline instruction. Python statements can be divided into two categories: assignment statement and expression statement. Assignment statements are usually used to create new variables, to assign values to variables, and to change the values of the variables. When a new variable is created and a value is assigned to it, Python allocates the appropriate memory location to the variable. The memory management of Python is different from some other well-known languages. Depending upon the assigned values, Python may assign same memory location to multiple variables to save the memory.

Multiline statements can be written in two different ways. Implicit method and explicit method. Implicit multiline statement is written within brackets `()`, `{}`, or `[]`. The explicit multiline statement makes use of a continuation character `\`. The end of a statement in both cases is marked by newline character. Unlike many other programming languages, Python does not use delimiter to terminate a statement.

Indentation is a highly important concept in Python. In some languages it is used to just for clarity and style of the program. It is especially used with control flow constructs such as loops and control statements. However, Python uses indentation not just readability of the program but also to convey the program structure to the interpreter. In Python, indentation cannot be ignored. It also helps to well organize and document the code. Comments are also useful to document the code, which highly useful when programmers revisit their code later on. Comments are used as reminders and they are not executable.

2.1 Objectives

The objectives of this unit are:

- 6 To discuss different types of tokens.
- 7 To define identifiers, keywords, and variables.
- 8 To use indentation and comments in writing Python programs.
- 9 To make use of input output functions.

2.2 Identifiers

A Python identifier is a user defined name given to a variable, function, class, module or other objects. It is a programmable entity. The following rules are used for identifiers in Python.

1. An identifier can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). It starts with a letter or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
2. An identifier cannot start with a digit. For example, 2num is not a valid identifier. However, num2, nu2m, n2um, _2num, num2_, and num_2 all are valid identifiers.
3. Python does not allow punctuation characters such as @, \$, %, and !, etc. and white spaces within or as identifiers.
4. Keywords cannot be used as identifiers.
5. Python is a case sensitive language. Therefore, num, Num, NUm, and NUM all are different identifiers.

In addition to above mentioned rules, there are several naming conventions used for the different identifiers in Python. Please note that conventions are not rules but good practices that are recommended to follow.

1. A class name should begin with an uppercase letter (A-Z) and other identifiers begin with lowercase letters (a-z).

2. The name of a private member should begin with an underscore (`_`). However, starting an identifier with an underscore does not automatically makes it a private identifier.
3. Use leading double underscore (`__`) only to avoid conflicts. It is also known as **mangling**.
4. A trailing double underscore indicates that an identifier is a language defined special name.
5. Use leading and trailing double underscore (`__`) with the names of magic methods.
6. Prefer using names longer than one character.
7. Use camel notation for naming. For example, to name a variable for “grand total”, use `grandTotal`.
8. Two words can be combined with an underscore (`_`). For example, “total marks” can be written as `total_marks`.

Python provides some methods to check the validity of an identifier. If you have followed all rules and conventions, still you can use these methods just to be sure that it is a valid identifier. There is a method `isidentifier()` that is used with strings to check if it makes a valid identifier. If given string is a valid identifier then it returns `True` otherwise `False` is returned. For example check strings `2num` and `_num` with `isidentifier()`.

Input:

```
>>>“2num”.isidentifier()
```

Output:

```
>>>False
```

Input:

```
>>>“_num”.isidentifier()
```

Output:

```
>>>True
```

According to `isidentifier()` method, `2num` is not a valid identifier. Therefore, it returns `False` for `2num`. Whereas, `_num` is a valid identifier and a `True` is returned for this. The `keyword` module provides a function `iskeyword(str)` to

check if the chosen name is a keyword or not. If 'str' is a Python keyword then it returns a **True** otherwise a **False**. To use the function **iskeyword(str)** one needs to import the module **keyword** as follows.

Input:

```
>>>import keyword  
  
>>>keyword.iskeyword('_num_')
```

Output:

```
>>>False
```

Input:

```
>>>keyword.iskeyword('for')
```

Output:

```
>>>True
```

Check your progress

1. What do you understand by token?
2. How many types of token are there in Python?
3. What is a statement in Python?
4. Write the categories of statements in Python.
5. What is expression statement?
6. How is the multiline statement written in Python?
7. What is identifier?
8. Check if following identifiers are valid or not:
 1. num\$\$b
 2. __num__

3. `Ssum`
4. `5_sum`
5. `sum_5`
6. `grand.total`
7. `total%`
8. `price@kg`
9. What is camel notation?
10. Which value is returned by `isidentifier()` for a valid identifier?

2.3 Variables

Variables in Python are named memory locations reserved to store values that a program can use or modify during the course of execution. When a variable is created, some space is reserved in memory. The amount of memory allocation depends on the data type of the variable.

2.3.1 Assigning values to variables

Python is not a statically typed language. It means that there is no need declare the type of the variable before using them. Therefore, Python has no command to declare a variable. A variable is created the moment a value is assigned to it first time. Depending on the assigned value, the interpreter determines its data type and allocates the memory accordingly. The assignment is done with a single equals sign (`=`). For example a variable of 'int' type can be created as follows.

```
>>>x = 5
```

Here, a variable of 'int' type named as 'x' is created by the interpreter as an integer value is assigned to it. Similarly a variable of type 'str' can be created by assigning a string to it as follows. Here, a variable of type 'str' named as 'y' is created by assigning it a string 'hello'.

```
>>>y = "hello"
```

The Python variables can even change their data type at some stage later during the execution. In the following example a variable named as 'i' is initially of type 'int' and later it becomes of type 'str' as it is assigned a string by the next statement.

```
>>>i = 10  
  
>>>i = "Jai Hind"
```

2.3.2 Multiple assignment

Several variables can be assigned a single value simultaneously in Python. This approach is known as multiple assignment, where a single statement is used to assign the same value to the multiple variables. But instead of allocating separate memory space to different variables, same memory location is shared by all such variables. For example,

```
>>>a = b = c = x = y = 5
```

In this example, an integer object is created with value 5 and all the five variables use the same memory location. A single statement can also be used to create multiple variables with different values. For example,

```
>>>x, y, z = 5, 2.5, 'hello'
```

Here, three variables x, y, and z are created in a single statement with different values of different data type. The integer value is assigned to variable x, the floating-point value is assigned to the variable y, and the string is assigned to the variable z. All three variables are allocated different amount of memory according to their data type.

2.3.3 Type casting

Type casting is used to explicitly specify the data type of a variable at the time of its creation. Python uses classes to define data types. Therefore, casting in

Python is done by using constructor functions. Some examples of casting are given as follows.

```
>>>x = int(3)

>>>y = str('5')

>>>z = float(7)
```

Here, `int()`, `str()`, and `float()` are the constructors or casting functions, `x` is a variable of type `int` with a value 3, `y` is a string '5', and `z` is a float having a value 7.0.

2.4 Constants

A constant is a value cannot be changed. Python does not support the easy creation of constants. In Python, constants can be defined in a module. A module is a file, which could be imported to access the defined constants and methods. Conventionally the constant should be named in upper case letters. For example,

```
PI = 3.17
GRAVITY = 9.8
RATE_OF_INTEREST = 7
```

The multi-word constant names should be separated by an underscore. But unlike some other languages like C++ and Java, there is no predefined data type in Python to create the constants in the program. There are some libraries to create a constant in the program itself instead of importing the constant from a module. One such library is `pconst`. Constant can be created using `pconst` as follows.

```
from pconst import const
const.RATE = 25
const.HEIGHT = 20
```

Here, `const.RATE` and `const.HEIGHT` are the constants.

2.5 Keywords

Keywords are the reserved words of the language that have predefined meaning and purpose. Keywords cannot be used as a variable name, constant name, function name or any other identifier. They are used to define the syntax and structure of the programming language. Keywords in Python are case sensitive. There are 36 keywords in Python 3.9. This number can vary over the course of time. All the Python keywords except `True`, `False`, and `None` are in lowercase. Keywords are always available and programmers do not need to import them in the program. A list of keywords for the Python version installed on the computer can be obtained using `help()` command. Type `help()` at the command prompt. After that type `keywords` to get the list of keywords as shown in Figure 2.1. To get help on specific keyword, type that keyword.

```
help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

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

Figure 2.1: Keywords in Python

Check your progress

1. What is a variable in Python?
2. What is the command to declare a variable in Python?
3. Can data type of a variable be changed during the course of execution of a program?
4. What is multiple assignment?
5. What is type casting?

6. What is difference between a variable and constant?
7. What is keyword?
8. How many keywords are there in Python?
9. Are Python keywords case sensitive?
10. Do you need to import the keywords?

2.6 Punctuation

A string contains letters, whitespace, numbers and it may also contain punctuation. The punctuation characters include commas, periods, and semicolon. All the common punctuation characters in Python are available with constant **string.punctuation**. The available punctuation characters can be printed as follows.

```
import string
for c in string.punctuation:
    print(c)
```

Consider a string "Hey, where are you?". The punctuation characters in the string can be find out using constant **string.punctuation** as follows.

```
import string
stmt = "Hey, where are you?"
for c in stmt:
    if c in string.punctuation:
        print(c)
```

The output of the piece of code would be:

```
,
?
```


2.7 Indentation

Indentation in Python refers to the spaces and tabs that are used at the beginning of a statement. The statements with the same indentation belong to the same code block called a suit. Python programs get structured through indentation. In other programming languages like C++/Java indentation is used just to improve the readability of the program. But in Python programs, indentation is mandatory. The indent level is increased to group the statements and it is decreased to close the group. If in a sequence of statements, there is a next statement with less indentation then it means the end of the previous code block. If indentation is not chosen carefully, it may lead to wrong output and even to errors. Incorrect indentation leads to *IndentationError*. Usually four whitespaces or a single tab is used create indentation but four whitespaces are preferred over a single tab. Let us an example of correct indentation.

```
if i==1:
    a=b*c
    print(a)
    print("This is a block of statements")
if i==2:
    a=b/c
    print(a)
    print("This is another block of statements")
print("This statement is the same suit with if statements")
```

In the above example, the two *if* statements and last *print* statement are in the same suit as there is no leading indentation with them. Three statements after the first *if* statement are belong to same group as they have same indentation. Similarly, three statements after the second *if* statement are in another same group. Consider next example with wrong indentation.

```
if i==1:
    a=b*c
    print(a)
    print("This is a block of statements")
```

In the above example, the second *print* statement inside *if* block has a different indentation level. But there is no matching statement, therefore it results in *IndentationError*. The indentation in Python makes the code clear and readable

and serves the purpose of grouping the statements. It is a mandatory requirement in Python. There should be same level of indentation in a block of code otherwise it gives an error.

2.7.1 Python indentation rules

For a proper indentation in Python, it is necessary to follow some rules. These rules are given as follows.

1. The first line of the program can not have any indentation. Otherwise it would throw *IndentationError*.
2. The indentation cannot be splitted into multiple lines using backslash.
3. Mixing of whitespaces and tabs for creating indentation should be avoided. Otherwise it could result in wrong indentation.
4. Use of four whitespaces is preferred over tabs.

2.7.2 Advantages of indentation

Indentation in Python is mandatory but it provides some default advantages. The major advantages of indentation in are as follows.

1. In other programming languages, indentation is used just for a better readability and making the code clear. In Python, indentation is used to structuring and grouping the code automatically making it readable and beautiful.
2. Indentation rules are very simple.
3. Most IDEs automatically do indentation of the code. Therefore, it is easy to write properly indented code.

2.7.3 Disadvantages of indentation

Besides several benefits, there are some disadvantages also of the automatic indentation. The major disadvantages are listed as follows.

4. In large programs, if indentation gets corrupted then it is really tedious to fix it. Use of whitespaces for indentation makes it difficult sometimes to identify the problem.
5. Copying a code from other source may also create problem related to indentation especially if it is copied from a Word or PDF document or online source.
6. Most of other languages use symbols like braces for indentation. Programmers migrating from other languages initially may find it hard to adjust.

2.8 Statements

Instructions that a Python interpreter can execute are called statements. The end of a statement is marked by a newline character. A simple statement is written within one single logical line of code. For example, following assignment statement and print statement are simple statements.

```
a=25  
print("This is a simple statement")
```

Several simple statements may occur on a single line separated by semicolons. The simple statements in above example can also be written in a single line as follows.

```
a=25; print("This is a simple statement")
```

2.8.1 Multi-line statements

The python statements terminate with newline character but a statement can be extended to multiple lines with the help of a line continuation character (\). For example,

```
a = 2 + 5 + 7 + \  
    3 + 9 + 4 + \  
    1 + 7 + 6
```

Here, one statement covers three lines. Such statements are known as multi-line statements. This approach explicitly continues the statement over multiple lines. Implicit line continuation is also possible in Python. It is done with the help of parenthesis (), braces { }, and brackets []. For example the above multi-line statement can be implemented with implicit line continuation as follows,

```
a = (2 + 5 + 7 +  
     3 + 9 + 4 +  
     1 + 7 + 6 )
```

The braces { } are used for Lists and brackets [] are used for Dictionary objects. For example,

```
countries = {"IN": "India", "FR": "France", "BR": "Brazil",  
            "UK": "United Kingdom"}
```

```
colors = ["red", "green", "cyan",  
         "blue", "pink"]
```

2.8.2 Compound statement

A compound statements comprise of group of other statements. The 'other' statements could be simple or compound statements. The compound statements are usually executed when a condition satisfies or through a function call or a code block is called directly. Compound Statements are generally spread into multiple logical lines but may also be contained in one line. The compound statement starts with a one-line *header* ending with a colon. The colon identifies the type of statement. A compound statement consists of one or more *clauses*. A header and *suits* of statements together form a clause. A set of indented statements is known as a suit. The general form of compound statement can be given as follows.

```
<header> :  
  
    <statement>  
  
    <statement>  
  
    <statement>  
  
    ...  
  
    <separating-header>  
  
    <statement>  
  
    <statement>  
  
    ...  
  
...
```

The *if*, *for*, *while*, *try*, *with*, *funcdef*, and *classdef* all are the compound statements. For example, in the following code, there is a *for* loop that starts with a header followed by a group of four simple statements.

```
for i in range(5):  
    j = i^2  
    print(j)  
    k = i^3  
    print(j+k)
```

In the following piece of code, there is compound statement that consists of suits of simple statements and compound statement.

```
if i > j:  
    print(i)  
    r = i%j  
    if r == 0:  
        print(j)  
        print(i + j)
```

2.8.3 Empty statement

It is the simplest statement in Python that is used to do 'nothing'. It is used to pass control flow to next statement under some conditions. It is usually used to avoid errors. The empty statement is written simply by using *pass* keyword. For example,

```
if(condition == False):  
    pass  
else:  
    print("Welcome")
```

Check your progress

1. What is punctuation?
2. How do you get the list of punctuation characters?
3. What is indentation in Python?
4. Is indentation is mandatory in Python?
5. What happens statements in a block do not follow same indentation level?
6. Is indentation allowed in first line of the program?
7. How is the end of statement marked in Python?
8. What is simple statement?
9. Write the continuation character in Python.
10. What is compound statement?
11. What is an empty statement?

2.9 Comments

Comments in a program are those lines that are completely ignored by the compiler or interpreter. Comments are used to make code more understandable. They make code more readable and provide explanation about what a part of a program is doing. The purpose of the comments is to explain intent and functionality of the program. Depending on the purpose of the program, comments can serve as notes and reminders to the programmer. It also helps other programmers to understand the code. It is a good practice to write comments in the program so that programmer can easily recall his/her thought process later on while updating the program. Generally comments are used for the following purposes:

- To improve code readability
- To explain the code or meta data of the project
- To prevent execution of code
- To insert reminders in the code

2.9.1 Single line comments

The single line comment is used to provide explanation for variables, expressions, and function declarations. In Python, a single line comment begins with a hash mark (#) and continue to the end of line. When a hash symbol is encountered by the interpreter

in a program, everything after # in that line is ignored by the interpreter. Here are some examples of single line comments.

```
# Code begins after this line
r = 65 # Rate of interest
y = 2  # no. of years
# y = 3
```

The first line in above code is a single line comment. The comment can also be placed at the end of the statement in the same line as the case is with second and third line. These type of comments are known as *inline* comments. The comment can also be used to prevent some code from execution as given in last line. If # is placed at the beginning of a statement then interpreter skips it.

2.9.2 Multiline comments

Multiline comments are used when single line is not sufficient to write a comment. A multiline comments spans several lines. In Python, a piece of text is enclosed within a delimiter(" " "). Delimiter (" " ") is placed at the beginning and at end of the text. For example,

```
"""
First line of multiline comment
Second line of multiline comment
Third line of multiline comment
Fourth line of multiline comment
Fifth line of multiline comment
"""
```

2.10 Input-Output

Input-Output functions are used to transfer data between user and the program. Input functions gather data from the user, while output functions allow a program to display data onto the console.

2.10.1 Reading input

Python provides a function *input()* to read the user data through keyboard. The general syntax of *input()* is as follows,

input(<prompt>)

where *<prompt>* is the string used to display the message at the time of taking input. For example,

```
name = input("Enter your name:")
```

Here, the statement is taking input in a variable 'name' from user with a message "Enter your name:". By default *input()* takes user input as a string. To take input as other data type, constructor for that data type is also required alongwith *input()* function. For example, *int()* is used alongwith *input()* to take integer type input as follows.

```
age = int(input("Enter your age:"))
```

2.10.2 Display output

There are several ways in Python to display the output to standard output device. The value of a variable can be displayed by simply typing the name of the variable or writing the expressions as follows.

```
>>>a = 25  
  
>>>a  
  
>>>25
```

Here, when 'a' is typed at command prompt followed by 'Enter' (second line), the value of 'a' is displayed (third line). Similarly, some expression can be written at command prompt and result is displayed in next line. There are some built-in functions in Python to display output to the console. The most popular output function is *print()*. The general syntax of *print()* is as follows.

Syntax: *print(value(s), sep=' ', end = '\n', file=file, flush=flush)*

Parameters:

value(s): A value or multiple values. Any value is converted to

string before printing.

sep='separator': (optional) Specify a separator to separate the multiple objects. By default it is single whitespace (' ').

end='end': (optional) Specify what to print at the end. By default it is: '\n'

file :(optional) An object with a write method. By default it is: sys.stdout

flush :(optional) Specifies if the output is flushed (True) or buffered (False). By default it is: False.

Returns: It returns output to the screen.

Some examples,

```
print('Hello')
Age = 25
print('Candidate age is: ', Age)
print(1,2,3, sep='-')
```

Output of code:

Hello

Candidate age is: 25

1-2-3

2.11 Summary

Objects like variable, function, class, or module, etc. in a program need to be named appropriately. The identifiers in Python follow some rules and conventions. Conventions are just good practices and are not mandatory but recommended to make the code more clear. Variables in Python are named memory locations reserved to store values that a program can use or modify during the course of execution. Data type of a variable is not strictly assigned and it can change to other type later on. Like other programming languages, there are a set of reserved words known as keywords in Python. They are used to define the syntax and structure of the programming language. Keywords cannot be used as identifiers. There are 36 keywords in Python 3.9.

Indentation is an important concept in Python. Whitespace and tabs are used for indentation in Python and it is mandatory. Usually four whitespaces or a single tab is used to create indentation but four whitespaces are preferred over a single tab. The indentation follows certain rules and correct indentation is required for the execution of the code. The instruction that can be executed by

interpreter are known as statement. The statement could be single line or multi-line statements. Further statements can also be classified as simple statement or compound statement. Compound statement is a group of statements.

Comments are those lines or strings of a program that are ignored by the interpreter. These are used just to improve the readability of the program. The programmers can use comments as reminders for themselves or to explain the code to others so that other programmer can extend the program. Python provides a set of input-output functions. The function `input()` is used to take user input through keyboard. There some output functions in Python out of which `print()` is one of the most popular output function.

2.12 Review questions

- Q.1 Write down the rules for identifiers in Python.
- Q.2 What are the conventions for identifiers in Python? What is the major difference between rules and conventions?
- Q.3 What is type casting? How are the variables type casted in Python?
- Q.4 How are the constant created in Python? Explain with examples.
- Q.5 Why is the indentation important in Python programs?
- Q.6 Write the rules for indentation. Explain the advantages and disadvantages of indentation.
- Q.7 What are the different methods to write multi-line statement in Python? Explain with the help of suitable examples.
- Q.8 What is compound statement? Write a piece of code to explain.
- Q.9 How are multi-line comments written in Python. Explain.
- Q.10 Write the syntax of `print()` function. With the help of suitable examples, explain different options.

UNIT-3: Data Types, Operators, and Expressions

Structure:

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Data Types
- 3.3 Operators
- 3.4 Expressions
- 3.5 Control Flow
- 3.6 Summary

Unit 3: Data Types, Operators, and Expressions

3.0 Introduction

Data type is an important concept in programming languages. The variables can hold values and each value is associated with a data type. Data types are a kind of classification or categorization of data items. It tells the compiler or interpreter what type of values a variable is supposed to take and which operations are allowed on those values. In other words, data type specifies the range of values and operations that can be safely performed for the variables in computation. Some programming languages attempt to define the type of all the variables during compilation. The behind this approach is reduce the errors at run time. The most of the data type related errors and exceptions happen at compile time. Such programming languages are known as *strongly typed* languages. Programming languages that do not have strict typing rules are called *weakly typed* languages. With loose typing rules, implicit type conversion can take place at runtime and the programs may generate unpredictable or erroneous results. *Dynamically typed* language perform type checking at runtime. A strongly typed language can also be dynamically typed. Most strongly typed languages use some form of dynamic typing as some properties are difficult to verify statistically. However, dynamic typing may sometimes cause program failure at runtime. Although, such errors can be anticipated and managed accordingly. Python is a strongly typed as well as dynamically typed language.

Operators in a programming language inform compilers or interpreters the kind of operations to perform on elements known as operands. The operators are represented with the help of some symbols. These operations include arithmetical, relational, and logical operations, etc. Functionality of programming languages is incomplete without operators.

Expression statements are those statements that return some value. These statement can also appear on the right hand side of an assignment statement. Usually they involve some mathematical expression or single variable. Expression statement can also take place as a parameter to some method call. The expression statements can be categorized into three different categories:

- Value-based expression on right hand side
- Current variable on right hand side
- Operation on right hand side

In “Value-based expression on right hand side” statement, there is a variable on left hand side of the assignment operator and a value is given on the right hand side. Python allocates new memory location to the variable on left hand side of the expression in this case. In case of “Current variable on right hand side” statement, instead of a value, there is another variable on the right hand side. No new memory location is allocated to the variable on left hand side but the same memory location is shared by both variables in the expression.

3.1 Objectives

The major objectives of this unit are as follows.

1. To have an insight into data types available in Python.
2. To learn the operators and their usages in Python programming.
3. To learn how to write loops for repeated execution of statements.
4. To learn how to control flow of execution in Python.

3.2 Data Types

Data types are used to specify the type of data used in a program to store and manipulate. Different data types represent different kind of values and specify what kind of operations can be performed on them. Since Python is an object oriented programming language and entities are represented as objects, all the data types in Python are actually classes and variables are objects of these classes. The standard built-in data types in Python can be divided into following categories.

1. Numeric
2. Boolean
3. Sequence type
4. Set
5. Dictionary

3.2.1 Numeric Data Types

Numeric data types deal with different kind of numbers. There are three distinct numeric data types in Python: int, float, and complex.

3.2.1.1 Integer

The data type *int* is used to represent integer type values. The integer type values are zero, positive, or negatives numbers without any fractional part. The integers have unlimited precision for example 0, 10, and -10, etc. The integers

in Python can take decimal, binary, octal, and hexadecimal values. All integer literals are objects of the class `int`. In Python there is no limit on the length of the integer number. The variables of data type `int` can be created using the constructor `int()` of class `int` as follows.

```
a = int(5)
```

Here `a` is a variable of integer type having a value 5. The binary numbers are represented with a leading '0b' for example 0b11010 is binary equivalent of 26.

```
a = int(0b11010)
```

Similarly, 0o or 0O is used for octal numbers and hexadecimal numbers are represented by 0x or 0X.

3.2.1.2 Floating-point numbers

Floating-point numbers are real numbers, which are written with a decimal point that divides the number into integer and fractional parts. For example, 7.85 is a floating-point number. Python floats are implemented as 64-bit double-precision values. The maximum value of any floating-point number in Python can be approximately 1.79×10^{308} . Any number greater than this is indicated by the string `inf`. Floating-point numbers are stored as base 2 fractions. Floating-point numbers are created with `float` data type. The `float()` method returns a floating-point number from the input as follows.

```
b = float(25.74)
```

3.2.1.3 Complex numbers

Complex numbers are written in the form `x+yj`, where `x` and `y` are the real and imaginary components of the complex number. There are two ways to create complex numbers. The real and imaginary parts can be specified as `(real)+(imaginary)j` as shown here.

```
c = 3+8j
```

In another way, the complex numbers can be created as instances of `complex` class using the constructor `complex()` as follows.

```
c = complex(3+8j)
```

3.2.2 Boolean Data Type

Boolean data type in python is used to represent truth values of the Python expressions. The truth values are defined by **True** and **False** keywords. The Boolean data type is indicated by `<class, bool>`. Generally Boolean values are generated as a result of the some kind of comparison in the expressions. For example, if `a=5` and `b=8` then the expression `(a==b)` generates the Boolean value **False**. Numbers can also be converted to Boolean values using the Python built-in method `bool()`. It could be any integer, floating-point, or complex number. A number having a value zero is converted to **False**, while a number having any negative or positive value is considered as **True**. Some examples,

```
a = bool(0)      #It converts the number to False
b = bool(5)      #It converts the number to True as its a non-zero number
c = bool(-2)     #It converts the number to True as its a non-zero number
```

3.2.3 Sequence Types

Sequences allow to store multiple values in an organized and efficient fashion. The data are stored in a kind of containers. There are several sequence data types in Python. Out of them, three widely used sequence data types are listed here.

11. Strings
12. Lists
13. Tuples

The elements of strings, lists, and tuples are ordered in a defined sequence and can be accessed via indices. Python uses the same syntax and functions to operate on sequential data types.

3.2.3.1 Strings

A String is a sequence of single or more characters. Python does not provide character data type, instead a character is considered a string of length one. A

string can be considered as an array of bytes representing Unicode characters. Strings are considered instances of `str` class in Python. There are multiple ways to create strings in Python using single quotes, double quotes, and triple quotes.

```
strg1 = 'Greetings of the day' #String created with single quotes
strg2 = "Welcome home"        #String created with double quotes
strg3 = '''How are you doing''' #String created with triple quotes
print(strg1)
print(strg2)
print(strg3)
```

Output:

```
Greeting of the day
Welcome home
How are you doing
```

Generally strings are created with single quotation marks. However, if single quotes are part of the string for example **Have you visited ‘New York’ city?** then double quotation marks can be used to create such strings as follows.

```
strg = "Have you visited 'New York' city?"
print(strg)
```

Output:

```
Have you visited 'New York' city?
```

With three quotation marks strings can go over multiple lines. For example,

```
strg = '''Hello
        how are you
        doing'''
print(strg)
```

Output:

```
Hello
how are you
doing
```

It is also possible to access individual characters of the string using indexing. The first character of the string has index 0 and subsequent characters can be accessed with 1,2,... indices respectively. The negative indices are also allowed, where -1 indicates last character. Subsequently -2, -3 refer second last and third last characters respectively. If index goes out of range then `IndexError` is generated. Some examples,

```
strg = 'Himalya'
print(strg[0]) #First character
print(strg[-1]) #Last Character
print(strg[2]) #Third character
```

Output:

H

a

m

3.2.3.2 Lists

Lists in Python are ordered collection of data like arrays in other programming languages but unlike array all the elements in Lists do not need to be of same type. A List can be created by placing the elements inside a pair of brackets [] as follows. The elements of a List can be accessed by indexing as it is used with Strings.

```
colors = ['red', 'green', 'blue', 'yellow']
print(colors[2])
```

Output:

Blue

The multi-dimensional Lists can be created by nesting Lists inside a List. For example,


```
ml = [[100, 200, 300, 400], ['red', 'green', 'blue', 'yellow']]
print(ml[0][0])
print(ml[1][0])
```

Output:

100

Red

The lists are mutable, which can be changed after creation. A number of operations can be performed on Lists directly or using some built-in methods. Some of the methods for operations on Lists are as follows.

1. `list.append(x)`: add an element 'x' to the end of the list.
2. `list.insert(i,x)`: add an element 'x' at given position 'i' in the list.
3. `list.extend(y)`: extend the list by appending all the elements from 'y'.
4. `list.remove(x)`: remove the first item with value 'x' from the list.
5. `list.pop([i])`: remove and return the element from given position 'i' in the list. If no index is specified, it removes the last element in the list.
6. `list.clear()`: Removes all the elements in the list.
7. `list.copy()`: Returns a copy of the list.
8. `list.reverse()`: The order of the elements is reversed.
9. `list.count(x)`: Returns the number of times the element 'x' appears in the list.
10. `list.sort()`: Sorts the elements in place.

3.2.3.3 Tuples

Tuple is a built-in data type in Python that is a collection of ordered objects. The indexing, repetition, and nesting in Tuples is similar to Lists. The major difference between Tuples and Lists is that unlike a List a Tuple is immutable. An immutable object cannot be modified once it is created. Tuples are unchangeable after their creation. Tuples can be created simply by comma separated values as follows.

```
t = 1, 5, 25, 'a', 'x'
```


Optionally the comma separated values can be put within the parenthesis. For example,

```
t = (1, 5, 25, 'a', 'x')
```

It is mandatory to separate elements with commas in a Tuple even though there is only one element in it. Otherwise, it would not be considered as a Tuple. For example,

```
t = (1,)
```

Tuples are immutable and therefore no elements can be removed or no new elements can be added to a Tuple. Indexing is done with the help of square brackets to access the Tuple elements. Only following operations are allowed on Tuples.

- | | |
|--------------------|---|
| 1. len(tuple) | Returns the number of elements in the Tuple. |
| 2. tuple1 + tuple2 | Concatenates the two Tuples and returns the result. |
| 3. tuple*i | Repeats the Tuple i times and returns the result. |
| 4. x in tuple | Returns True if x is in Tuple otherwise False. |

3.2.4 Set

Set is another built-in data type in Python, which is an unordered collection of elements. Set is an iterable and mutable data type that cannot have duplicate elements. There could be elements of different data types in a set. The order of elements in a set is undefined. A set can be created either by placing the comma separated elements within curly brackets or by using a built-in method `set()`. The input to the method `set()` could be any iterable sequence type such as string, list, tuple, or dictionary. Some examples,

```
set1 = {52, 65, 'Ram', 'Shyam'}           #Creating a set using curly brackets
set2 = set(['Sea', 'Sky', 32.5, 5])         #Creating a set using set() method
set3 = set('Welcome')                      #Creating a set from string
set4 = set((5,6,12, 25))                   #Creating a set from tuple
print(set1)
print(set2)
print(set3)
print(set4)
```

Output:

{65, 'Ram', 52, 'Shyam'}

{32.5, 'Sky', 5, 'Sea'}

{'l', 'o', 'W', 'm', 'e', 'c'}

{25, 12, 5, 6}

An empty set can be created by using method `set()` without passing any parameter. A set contains unique elements only. Although at the time of set creation multiple duplicate items can be passed but it keeps only one copy of an element. The order of elements in a set is undefined and it cannot be changed. The elements in a set cannot be accessed with the help of indexing as the sets are unordered data type. However, it is possible to check if some specified value is present in the set. Some more examples,

```
set1 = {'hi', 'hello', 'hi', 5, 5, 5, 8} #Creating a set with duplicate entries
print(set1)
print('hi' in set1) #Check if 'hi' is in set1
```

Output:

{'hi', 8, 'hello', 5}

True

Since sets are mutable, new elements can be added to the set or elements can be removed from it. A number of methods are available for set operations as given here.

- | | |
|---|--|
| 1. <code>set.add(x)</code> | Adds one element 'x' to the set |
| 2. <code>set.update(set1)</code>
in set1.
also be passed to it. | Adds multiple elements that are contained
Instead of set, a list can |
| 3. <code>set.remove(x)</code>
Error message
in the set. | Removes an element 'x' from the set.
is generated if 'x' is not available |

- | | |
|--|--|
| 4. <code>set.discard(x)</code> | Removes an element 'x' from the set. No error message generated if 'x' is not present in the set and it remains unchanged. |
| 5. <code>set.pop()</code> | Removes and returns the last element of the set. But it is unknown which element is the last element. |
| 6. <code>set.clear()</code> | Removes all elements from the set. |
| 7. <code>set.copy()</code> | Returns a copy of the set. |
| 8. <code>set.union(set1,...)</code> | Returns union of the sets. |
| 9. <code>set.difference(set1)</code> | Returns difference of two sets. |
| 10. <code>set.intersection(set1)</code> | Returns intersection of two sets. |
| 11. <code>set.difference_update(set1)</code> | Removes all elements of set1 from the set. |
| 12. <code>set.isdisjoint(set1)</code> | Returns True if two set are disjoint. |
| 13. <code>set.issubset(set1)</code> | Returns True if this set is a subset of set1. |
| 7. <code>set.issuperset(set1)</code> | Returns True if this set is a superset of set1. |
| 8. <code>set.symmetric_difference((set1)</code> | Returns symmetric difference of two sets. |
| 9. <code>set.intersection_update(set1)</code> | Updates the set with the intersection of two sets. |
| 10. <code>set.symmetric_difference_update(set1)</code> | Updates the set with the symmetric difference of two sets. |

3.2.5 Dictionary

Dictionary is another composite data type in Python similar to List. Like a list it is mutable, dynamic, and can be nested. The major difference between a dictionary and a list the way they access their elements. Instead of an index, elements in a dictionary are accessed with the help of an associated key. The elements of dictionaries in Python are key-value pairs. Any Python data type can be used as keys in a dictionary but usually numbers and strings are preferred as keys. The mixed keys of different types are also allowed in a dictionary. The values in dictionary can also be any Python objects. A Python dictionary is like a hash table.

A dictionary in Python can be created by writing key-value pairs inside the curly braces. A key and its associated value are separated by a colon (:), while different elements (key:value) are separated by a comma (,). The dictionaries can also be created with built-in method `dict()`. The elements within curly braces are passed to `dict()` as parameter. For example,

```

#Creating dictionaries with { }
dic1 = {1:'one', 2:'two', 3:'three'}
#Dictionary with mixed keys
dic2 = {1:'one', 2:'two', 'Name':'Ram', 'Course':'BCA'}
#Creating dictionary dict() method
dic3 = dict({1:'one', 2:'two', 'Name':'Ram', 'Course':'BCA'})
#Creating nested dictionary
dic4 = {1:'Ten', 2:'Hundred', 'colors':{1:'red', 2:'green', 3:'blue'}}
print(dic1)
print(dic2)
print(dic3)
print(dic4)

```

Output:

```

{1:'one', 2:'two', 3:'three'}
{1:'one', 2:'two', 'Name':'Ram', 'Course':'BCA'}
{1:'one', 2:'two', 'Name':'Ram', 'Course':'BCA'}
{1:'Ten', 2:'Hundred', 'colors':{1:'red', 2:'green', 3:'blue'}}

```

The elements of a dictionary are accessed using the keys. The key is used inside brackets [] with dictionary name. The built-in method `get()` can also be used to access the dictionary elements. Some examples,

```

print(dic1[1])
print(dic2['Name'])
print(dic3.get('Course'))
print(dic4['colors'][2])

```

Output:

```

'one'
'Ram'
'BCA'
'green'

```

Various operations can be performed on dictionaries using the built-in methods. Some important built-in methods are listed and explained here.

1. `dictionary.pop(key)` Removes and returns the value of the 'key'.
2. `dictionary.popitem()` Removes and returns an element randomly.
3. `dictionary.clear()` Removes all the elements at once.
4. `dictionary.copy()` Returns the copy of the dictionary.
5. `dictionary.update(dictionary1)` Adds all the elements of `dictionary1` to the `dictionary`.
6. `dictionary.str()` Produces a printable string representation of the `dictionary`.
7. `dictionary.keys()` Returns the list of keys in the `dictionary`.
8. `dictionary.items()` Returns a list of (key, value) pairs in the `dictionary`.
9. `dictionary.cmp(dictionary1)` Compares the two dictionaries.
10. `dictionary.has_key(key)` Returns True if 'key' is in the `dictionary`.

Check your progress

1. Differentiate strongly typed and weakly typed languages?
2. What is dynamically typed language?
3. What are major categories of built-in data type?
4. What is the range of *int* data type?
5. How are the complex numbers created?
6. Write the output of `book(-25)`.
7. How do you create multi-line strings?
8. What do you understand by mutable data type?
9. List the mutable data types in Python.

10. List the immutable data types in Python.
11. How do you create multi-dimensional lists?
12. What is the major difference between Tuple and List?
13. Create a tuple with single element.
14. How do you remove an item from a Tuple?
15. Can you add duplicate items to a set?
16. How do you create an empty set?
17. How is indexing done in List, Tuple, and Set?
18. What happens when duplicate items are passed at the time of set creation?
19. What is major difference between Dictionary and List?
20. How are the key and value separated in a dictionary?
21. How do you access values in a dictionary?

3.3 Operators

Operators are special symbols designated for the logical and arithmetic operations on data values. An operator manipulates data values called operands. There are different types of operators in Python including:

1. Arithmetic operators
2. Logical operators
3. Relational operators
4. Bitwise operators
5. Assignment operators
6. Special operators

3.3.1 Arithmetic operators

Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, division, multiplication, etc. These operators are used with numeric values. There are seven arithmetic operators as listed in Table 3.1. Operator precedence to ensure which operator goes before other operators. Python uses PEMDAS (parentheses, exponentiation, multiplication, division, addition, subtraction) ordering for arithmetic operators.

Table 3.1: Description of arithmetic operators in Python

Operator	Function	Description	Example	Output
+	Addition	Performs addition of two operands to produce their summation.	5 + 2	7
-	Subtraction	Performs subtraction of one operand from other one to produce the difference.	5 - 2	3
*	Multiplication	Produces product of two operands.	5 * 2	10
**	Exponentiation	Performs exponentiation of operands ($a^{**}b$ means a^b).	5 ** 2	25
/	Division	Performs division operation between two operands (a/b). The left operand (a) is dividend and right operand (b) is divisor.	5 / 2	2.5
//	Floor division	Performs division operation between two operands and returns integral part of the quotient.	5 // 2	2
%	Modulus	Performs division operation between two operands and returns a remainder.	5 % 2	1

3.3.2 Logical operators

Logical operators are used to perform logical operations on the values of variables. The resultant value is either TRUE or FALSE. The result of the logical operations is always Boolean but the operands of the logical operators may be Boolean or other numbers. There are three logical operators in Python AND, OR, and NOT. The description of logical operators in Python is given in Table 3.2.

Table 3.2: Description of logical operators in Python

Operator	Function	Description	Example	Output
and	Logical AND	Performs logical AND operations on operands.	True and False	False
			True and True	True
			1 and 1	1
			1 and 0	0
or	Logical OR	Performs logical OR operations on operands.	True and False	True
			False and False	False
			1 and 1	1
			1 and 0	1
not	Logical NOT	It performs operation on a single operator only that reverses the state of the operand.	not True	False
			not False	True

3.3.3 Relational operators

The relational operators are used to compare the values of two operators. They always return either TRUE or FALSE according to the condition. The relational operators are also known as comparison operators. The operands of relational operators could be numbers or strings or boolean values. For strings comparison is performed character by character using ASCII values. The relational operators are given in Table 3.3.

Table 3.3: Description of relational operators in Python

Operator	Function	Description	Example	Output
<code>==</code>	equal to	If values of two operands are equal then results is True otherwise False.	<code>a=2, b=5 (a==b)</code>	False
<code>!=</code>	not equal	If values of two operands are not equal then results is True otherwise False.	<code>a=2, b=5 (a!=b)</code>	True
<code>></code>	greater than	If left operand is greater than right operand then True otherwise False.	<code>a=2, b=5 (a>b)</code>	False
<code><</code>	less than	If left operand is smaller than right operand then True otherwise False.	<code>a=2, b=5 (a<b)</code>	True
<code>>=</code>	greater than or equal to	If left operand is greater than or equal to the right operand then True otherwise False.	<code>a=2, b=5 (a>=b)</code>	False
<code><=</code>	less than or equal to	If left operand is less than or equal to the right operand then True otherwise False.	<code>a=2, b=5 (a<=b)</code>	True

3.3.4 Bitwise operator

Bitwise operators perform operations on numbers at binary level. It means that they operate bit-by-bit on the binary representation of a number. Each number is considered as a string of zeros and ones. For example a number 57 is converted to its binary representation 111001 then a particular binary operation is carried out at each bit. Some bitwise operators are unary operators while some other are binary operators as listed on Table 3.4.

3.3.5 Assignment operators

After evaluating an expression, the result is usually required to assign some variable. The assignment operators are used to assign the result of right hand

operand or expression to the left hand operand. There are two types of assignment operators- simple assignment operator and compound assignment operators. Simple assignment operator is denoted by = and it simply assigns the right hand value to the left hand operand. Compound assignment operators are a kind of shorthand notation that combines simple assignment operator with some other operator. All kind of assignment operators work from right to left. The assignment operators in Python are listed in Table 3.5.

Table 3.4: Description of bitwise operators in Python

Operator	Function	Description	Example	Output
&	Binary AND	If two bits at a bit position in operands are both 1 then resultant bit is 1 otherwise it is 0.	6 & 3	0110 0011 0010 = 2
	Binary OR	If two bits at a bit position in operands are both 0 then resultant bit is 0 otherwise it is 1.	6 3	0110 0011 0111 = 7
^	Binary XOR	If two bits at a bit position in operands are both same then resultant bit is 0 otherwise it is 1.	6 ^ 3	0110 0011 0101 = 5
~	Binary ones complement	It is unary operator that inverts each bit of the operand.	~6	~(0110) =(1001) = 9
<<	Binary left shift	The bits of left operand are moved towards left by the positions mentioned by right operand.	6<<2	0110<<2 =(1000) = 8
>>	Binary right shift	The bits of left operand are moved towards right by the positions mentioned by right operand.	6>>2	0110>>2 =(0001) = 1

3.3.6 Special operators

In addition to the operator discussed so far, there are some special operators in Python such as *identity operator* and *membership operators* as given and explained in Table 3.6

Table 3.5: Description of assignment operators in Python (continue)

Operator	Function	Description	Example	Output
=	Assignment	Simple assignment operator. It assigns value of right hand operand to right hand operand.	b=5, c=3 a=b+c	a=8
+=	Addition assignment	It adds the value of right hand operand to the value of left hand operand and assigns it to right hand operand.	b=5, c=3 b+=c (equivalent to b=b+c)	b=8
-=	Subtraction assignment	Performs subtraction and assigns the result to right hand operand.	b=5, c=3 b-=c (equivalent to b=b-c)	b=2
=	Multiplication assignment	Performs multiplication and assigns the result to right hand operand.	b=5, c=3 b=c (equivalent to b=b*c)	b=15
/=	Division assignment	It divides the value of right hand operand by the value of left hand operand and assigns it to right hand operand.	b=15.9, c=3 b/=c (equivalent to b=b/c)	b=5.3
=	Exponentiation assignment	It raises the value of right hand operand to the power of left hand operand and assigns it to right hand operand.	b=5, c=3 b=c (equivalent to b=b**c)	b=125
//=	Floor division assignment	It divides the right hand operand by left hand operand and assigns the integral part of the quotient to right hand operand.	b=15.9, c=3 b//=c (equivalent to b=b//c)	b=5.0
%=	Remainder assignment	Performs division and assigns the remainder to right hand operand.	b=5, c=3 b%=c (equivalent to b=b%c)	B=2
&=	Bitwise AND assignment	Performs bitwise AND operation and assigns the result to right hand operand.	b=5, c=3 b&=c (equivalent to b=b&c)	b=1
=	Bitwise OR assignment	Performs bitwise OR operation and assigns the result to right hand operand.	b=5, c=3 b =c (equivalent to b=b c)	B=7
^=	Bitwise XOR assignment	Performs bitwise XOR operation and assigns the result to right hand operand.	b=5, c=3 b^=c (equivalent to b=b^c)	b=6
>>=	Bitwise right shift assignment	Performs bitwise right shift operation and assigns the result to right hand operand.	b=5, c=3 b>>=c (equivalent to b=b>>c)	B=0
<<=	Bitwise left shift assignment	Performs bitwise left shift operation and assigns the result to right hand operand.	b=5, c=3 b<<=c (equivalent to b=b<<c)	b=40

Table 3.6: Special operators in Python

Operator		Description	Example	Output
Identity operators	is	True if both operands are identical.	<code>a=5, b=5, c=7</code> <code>a is b</code>	True
			<code>a is c</code>	False
	is not	True if both operands are not identical.	<code>a is not b</code>	False
			<code>a is not c</code>	True
Membership operators	in	True if value is found in the given sequence.	<code>x = {3,4,5,6}</code> <code>5 in x</code>	True
			<code>7 in x</code>	False
	not in	True if value is not found in the given sequence.	<code>5 not in x</code>	False
			<code>7 not in x</code>	True

Check your progress

1. Why is operator precedence used?
2. Write the precedence for arithmetic operators.
3. Write down logical operators.
4. Which operators are also known as comparison operators?
5. How do bitwise operators perform their operations?
6. How do compound assignment operators differ from simple assignment operator?
7. Write down special operators in Python.
8. Let `a=15`, `b=25`, `s='hello'`, `t='namaste'`. Write the output of following Python statements.
 1. `a>b`
 2. `a==b`
 3. `a>=b`
 4. `a & b`
 5. `a ^ b`
 6. `a >> 2`
 7. `a+=b`
 8. `b/=a`

9. `b%=a`
 10. `a*=b`
 11. `a is b`
 12. `b is not a`
 13. `s is t`
 14. `t is not s`
9. Let `a=True`, `b=False`. Write the output of following Python statements.
1. `a or b`
 2. `a and b`
10. Let `x = {4, 'a', 6, 'b'}`. Write the output of following Python statements.
1. `'a' in x`
 2. `8 in x`
 3. `'c' not in x`
 4. `6 not in x`

3.4 Expressions

An expression is a combination of values, variables, operators, and calls to functions. The expressions are evaluated to produce a value, which can be displayed, assigned to a variable, or passed to other expression or method. Usually expressions appear on the right hand side of an assignment statement. A variable or value itself can be considered a simple expression. Expressions are different from statement as statements do something while expressions are representation of a value.

Python has some advanced constructs for representing values and hence these constructs are also known as expressions. Python expressions contain identifiers, literals, and operators only. The operators combine many data objects into expressions. The result of evaluating an expression is a value or object. There can be two types of expression in Python: operator expression and function call expression. The operator expression has the following general form,

`<sub expression> operator <sub expression>`

There is a sub expression before and after the expression, which itself could be a simple or complex expression. The general form of function call expression is,

`<sub expression>(<sub expression>, <sub expression>, ..., <sub expression>)`

Complicated expressions can be built with a combination of operators and function calls. Some examples,

```
a = 2
b = 4
print(a+b)
print(a+b*len('Welcome'))
c = a*b+a
d = pow(a,3)+pow(b,pow(a,2))
```

In Python, the left operand is always evaluated before the right operand. That also applies to function arguments. While evaluating expressions involving and/or operators, the second operand is not evaluated unless it is necessary to obtain the result.

3.5 Control Flow

The controlflow of a program is the order in which the program code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls. There are three different types of control structures in Python: sequential, selection, and repetition. The sequential flow is the default structure of execution of the Python programs. The selection statements allow the condition based execution of the code. While, repetition allows multiple times execution of the code.

3.5.1 Sequential mode

The default mode of execution of the program code is sequential. In this mode the statements in the program code are executed sequentially from top to bottom in the same order they appear in the program. The next statement in the sequence can only be executed after the execution of the current statement. The major problem with sequential statements is that if there is any break in the logic at any statement in the program, the execution of the remaining statements is also terminated. Although, the execution flow of the program is inherently sequential at different levels, there may arise situation when a change in the sequence is required according to some conditions or due to break in the logic.

3.5.2 Selection or Conditional statements

The selection statements or conditional statements are also known as decision control statements or branching statements. The selection statement allows a

program to execute instructions based on several conditions. Depending on the condition the flow of execution may skip a block of statements or it may execute a block of statements instead of other. Python provides various conditional statements including *simple if*, *nested if*, *if-else*, and *if-elif-else*.

3.5.2.1 Simple if statement

The simple *if* conditional statements help us to execute a particular code only when a certain condition is satisfied. It allows only one condition to check. The *if* statement evaluates the expression for given condition, if expression evaluates to True, then the intended code in the “*if* block” is executed otherwise those statements are skipped and flow is transferred to the next line of code which is at the same level of indentation as the *if* statement. The general syntax of the *if* is as follows.

```
if(<expression>):  
    statement or block of statements
```

For example,

```
a=5  
b=8  
if (a<b):  
    print("'a' is smaller than 'b'")
```

Output:

'a' is smaller than 'b'

The expression after keyword *if* can also be written without brackets (). Another example,

```
colors = ['red', 'green', 'blue', 'pink']  
print(colors)  
if ('blue' in colors):  
    print('Blue is in the list of colors')  
    colors.remove('blue')  
    print('Blue is removed from the list of colors')  
print(colors)
```

Output:

['red', 'green', 'blue', 'pink']

Blue is in the list of colors

Blue is removed from the list of colors

['red', 'green', 'pink']

3.5.2.2 If-Else statement

Sometimes we need to execute one set of statements if a condition is True and another set of statements if the condition is False. In this situation, *if-else* control statement is more useful than the simple *if* control statement. The *else* statement is a kind of default statement. It can be demonstrated with the help of the following flow chart.

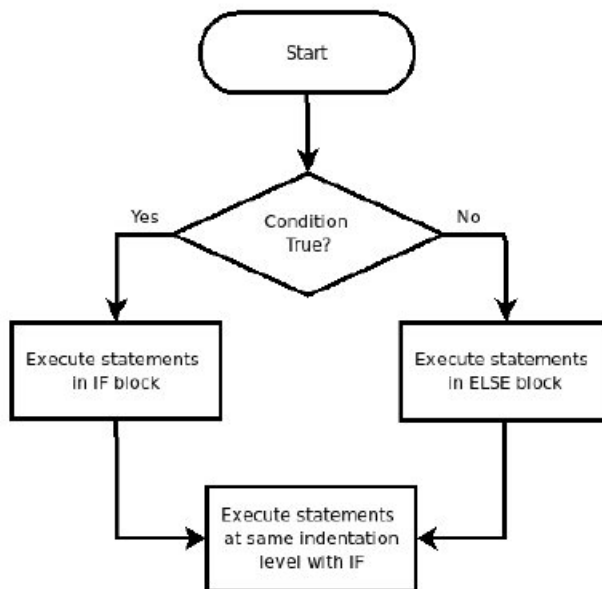


Fig. 3.1: If-Else statement control flow

The general syntax of *if-else* control statement is as follows.

```
if (<expression>):  
    <statement/statements>  
  
else:
```

<statement/statements>

For example,

```
a=25
if (a%2==0):
    print("'a' is an even number")
else:
    print("'a' is an odd number")
```

Output:

'a' is an odd number

3.5.2.3 If-Elif-Else statement

Sometimes we need to test multiple conditions. In this situation *elif* is useful, which means if previous condition is not met than try this. We can use *elif* any number of times after *if* block and *elif* block can be followed by optional *else* block. The *else* statement (if used) must come last. It is also known as multi-way decision control statement. The general syntax for *elif* control statement is as follows.

```
if(<expression>):
    <statement/statements>
elif(<expression>):
    <statement/statements>
elif(<expression>):
    <statement/statements>
.
.
.
.
else:
    <statement/statements>
```



```
percentage = 46
if (percentage >= 60):
    print('Its a First division')
elif (percentage >= 45):
    print('Its a second division')
elif (percentage >= 33):
    print('Its a third division')
else:
    print('Fail')
```

Output:

Its a second division

3.5.2.4 Nested if statement

In some situations, a condition is needed to test only after another condition tests True. In this situation an *if* statement is placed inside another *if* or *else* block. An *if* statement that contains another *if* statement either in its *if* block or *else* block is called a nested *if* statement. The general syntax for nested *if* is as follows.

```
if (<expression>):
    <statement/statements>
    if (<expression>):
        <statement/statements>
    else:
        <statement/statements>
else:
    <statement/statements>
```

Here, expression for inner *if* statement is tested only if expression of outer *if* statement results in Boolean value True. An example,

```
a, b = 5, 5
if (a>b):
    print("'a' is greater than 'b'")
else:
    if (a<b):
        print("'a' is smaller than 'b'")
    else:
        print('Both numbers are equal')
```

Output:

Both numbers are equal

3.5.3 Loop control flow statements

The loop control flow statements are also known as repetition statements or iteration statements. A repetition or iteration or loop statement allows the execution of a block of statements multiple times until a loop termination condition is met. Python provides two major types of loops: *while* and *for*. The basic functionality of both loops is same but they differ in syntax. In addition, both *while* and *for* loops can also be nested with other loops.

3.5.3.1 The while loop

The while loop in Python starts with the *while* keyword and ends with a colon (:). Between keyword *while* and colon a Boolean expression is written. The syntax for while loop is given as follows.

```
while (<boolean expression>):  
    statement/statements
```

In while loop, the Boolean expression is evaluated before the statements in the loop block is executed. If the Boolean expression evaluates to True, then the statements in the loop block are executed. If the Boolean expression evaluates to False, then the loop block is never executed. After each iteration, the control flow returns to evaluate the Boolean expression again. If it evaluates to True, then the loop is iterated again. This process continues until the Boolean expression evaluates to False, then control goes to the first statement after the while loop.

Example,

```
i, j = 5, 1  
while (j<=10):  
    print(i*j)  
    j = j + 1
```

Output:

5
10

15
20
25
30
35
40
45
50

3.5.3.2 The while-else loop

The *while* loop executes the block as long as condition is satisfied. When the condition for the loop becomes false, the statement immediately after the loop is executed. The *while-else* loop is a minor variation of *while* loop. An *else* clause is added at the same indentation level with *while*. The *else* clause is executed only when *while* condition becomes false. If it breaks out of the loop or if an exception is raised, the *else* block is not executed. For example,

```
i = 1
while (i<=3):
    print(i)
    i = i + 1
else:
    print('In else block')
```

Output:

1
2
3
In else block

```
i = 1
while (i<=3):
    print(i)
    i = i + 1
    break;
else:
    print('In else block')
```

Output:

1

In the first example of *while-else* loop, the *while* loop terminates when condition becomes false and after that *else* block is executed. On the other hand, in the second example, the *while* is broken before condition becomes false. Therefore *else* block is not executed here. (Note *break* statement will be discussed later in details.)

3.5.3.3 The for loop

The *for* loop can be used to repeat the execution of statements just like the *while* loop do. However, the *for* loop is more suitable to iterate sequence such as list, tuple, or string and other iterable objects in Python. The general syntax of the *for* loop is given as follows.

```
for iterating_var in sequence:
    statement/statements
```

Here *iterating_var* is a variable that is used for indexing in the sequence. The loop continues until the entire sequence is exhausted. Some examples,

```
colors = ['red', 'green', 'blue', 'pink']
for clr in colors:
    print(clr)
```

Output:

red

green

blue

pink
<pre>for st in 'Hello': print(st)</pre>
Output: H e l l o
<pre>for i in range(0, 4): print(i)</pre>
0 1 2 3

Here in the third example, the *range()* function is used to generated a sequence of numbers between user given limits. An *else* clause can be used with *for* loop just like it is used with *while* loop. The *else* block is executed when sequence is exhausted. Otherwise, it is not executed.

3.5.4 Loop control statements

Loop control statements are used to provide more control over the loops, which can alter the flow of a loop. Python provide following loop control statements,

- break
- continue
- pass

These statements change normal course of execution of a loop. Sometimes it may be needed to exit a loop before its condition becomes False or it may be required to skip a part of the loop. The loop control statements can handle such situation.

3.5.4.1 The break statement

The *break* statement is used to bring the control out of the current loop and resume the execution at next statement. In case of nested loops, the *break* statement exits the inner loop and proceeds to outer loops. Usually *break* statement is associated with some condition. When some external condition is triggered and if there is an associated *break* statement then it abruptly stops the loop. Consider following examples,

```
seq = [2, 5, 0, 3, 4]
num, i = 15, 0
while(i<5):
    d = seq[i]
    if(d==0):
        print('Denominator is zero, operation not allowed')
        break
    print(num/d)
    i = i + 1
```

Output:

7

3

Denominator is zero, operation not allowed

```
seq = [2, 5, 0, 3, 4]
num = 15
for d in seq:
    if(d==0):
        print('Denominator is zero, operation not allowed')
        break
    print(num/d)
```

Output:

7

3

Denominator is zero, operation not allowed

The *break* statement can be used both with *while* and *for* loops. In both above examples, break statement was used to handle the situation of division by zero.

3.5.4.2 The continue statement

The *continue* statement returns the control to the beginning of the loop discarding the remaining statements. Like *break* statement, the continue statement is also associated with a condition. If condition is satisfied, all the statements in the loop after the *continue* statement are rejected and control flow is transferred to the beginning of the loop. The *continue* statement can be used in both *while* and *for* loops. For example,

```
for st in 'welcome':  
    if(st=='e'):  
        print('-')  
        continue  
    print(st)
```

Output:

w
-
l
c
o
m
-

Here in this example, whenever it reads the character 'e', the *continue* statement transfers the control to the top of the loop.

3.5.4.3 The pass statement

The *pass* statement is a null statement, which is generally used as a placeholder for future code. It is not only used with loops but it can also be used with constructs of the program such as function, and class, etc. The *pass* is useful in loops or functions where desired code is not written yet but will eventually be

available. In such constructs *pass* statement is written instead of having a blank body otherwise interpreter may generate the error. The *pass* statement does nothing but it is different from a comment. The comment is ignored completely by the interpreter but *pass* is not ignored. Although nothing happens when *pass* statement is executed. For example,

```
for st in 'welcome':  
    if(st=='e'):  
        pass  
    print(st)
```

Here, nothing happens when condition in *if* statement is True.

3.6 Summary

Data types are important to store and manipulate data for a program. In Python a data type is implemented as a class and objects of that class are considered as variables of that type. The built-in data types in Python can be divided into following categories: numeric, Boolean, sequence, set, and dictionary. There are three main numeric data type including int, float, and complex. True and False are two Boolean data types, which are mainly used for making comparison in expressions. Sequence data types are used to efficiently organize multiple values. The values in a sequence data type can be accessed with the help of indices. String, List, and Tuple are sequence data type in python. Strings are sequence of characters, which are instances of str class. A list is a collection of data elements like arrays but unlike an array a list can contain elements of different types. Tuples are similar to lists but they cannot be manipulated once they are created. Set is another ordered and mutable data type in Python but it cannot have duplicate elements. The order of elements in a set is undefined. Therefore, the elements in set cannot be accessed with an index but an element can be searched in the set. Dictionary is an ordered, mutable, and dynamic data type in which elements are stored as (key, value) pairs. The values in a dictionary cannot be accessed with the help of an index. Instead values are accessed by using it associated key.

Operators are designed to perform different type of operations on data elements. The operators are represented by special symbols. The important operators in Python are as follows: arithmetic, logical, relational, bitwise, assignment, and special operators. Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, division, multiplication, and exponentiation, etc. Python uses PEMDAS ordering for arithmetic operators. Logical operators perform logical operations such as AND, OR, and

NOT. The relational or conditional operators are used to perform comparison between values. While, bitwise operators do operations at binary level. Assignment operators can assign the result of an expression to a variable.

An expression is a combination of values, variables, operators, and calls to functions. Python expressions contain identifiers, literals, and operators only. The operators combine many data objects into expressions. The result of evaluating an expression is a value or object. There can be two types of expression in Python: operator expression and function call expression. Complicated expressions can be built with a combination of operators and function calls.

The control flow of a program is the order in which the program code executes. The default mode of execution of the program code is sequential. The selection or conditional statement allows a program to execute instructions based on several conditions. Depending on the condition the flow of execution may skip a block of statements or it may execute a block of statements instead of other. Python provides various conditional statements including simple if, nested if, if-else, and if-elif-else. The 'if' statement evaluates the expression for given condition, if expression evaluates to True, then the intended code is executed. In if-else control statement the else statement is a kind of default statement. The if-elif-else control statement is useful to test multiple conditions.

Loop control flow statements allow the execution of a block of statements multiple times until a loop termination condition is met. Python provides two major types of loops: while and for. The basic functionality of both loops is same but they differ in syntax. To better control the loops some special statements including break, continue, and pass are used. The break statement is used to bring the control out of the current loop and resume the execution at next statement. The continue statement returns the control to the beginning of the loop discarding the remaining statements. The pass is a kind of dummy statement that executed by interpreter but nothing happens as a result.

3.7 Review questions

Q.1 What do you understand by data types? What are the major categories of data types in Python? Give some examples of data types in each category.

Q.2 Create a list of number and perform major operations with built-in functions.

Q.3 Create two mixed sets of numbers and strings. Perform following operations on created set: add, pop, remove, union, intersection, and clear.

Q.4 Explain different types of operators in Python with suitable code.

Q.5 Briefly explain binary shift operators with examples.

- Q.6 Write a program to read the marks of five subjects and find the average of them.
- Q.7 What do you understand by loop? What are the major loop statements in Python? Explain.
- Q.8 What is use of else statement with while or for loops? Explain with code.
- Q.9 What is the utility of selection or conditional statements in a program? What are different conditional statements in Python?
- Q.10 Explain the use of if-elif-else statement with the help of a program.
- Q.11 Explain the syntax of for loop with example.
- Q.12 Explain the syntax of while loop with example.
- Q.13 Explain the purpose of continue, break, and pass statements.
- Q.14 Write a program to find the sum of digits of a number.
- Q.15 Write a program to find the factorial of a number.
- Q.16 Write a program to display the Fibonacci series.
- Q.17 Write a program to iterate through each character in a string.
- Q.18 Write a program to demonstrate an infinite loop and break.
- Q.19 Write a program to check whether a number is prime or not.
- Q.20 Demonstrate the use of continue statement in a loop.



**Uttar Pradesh Rajarshi Mondon
Open University**

SBSCS-02

Python Programming

Block

2

BASICS OF PYTHON

Unit 4	
Data Structures	84
Unit 5	
Functions	103
Unit 6	
Modules and Packages	127

Course Design Committee

Prof. Ashutosh Gupta Director (In-charge) School of Computer and Information Science, UPRTOU Prayagraj	Chairman
Prof. R. S. Yadav Department of Computer Science and Engineering MNNIT Prayagraj	Member
Dr. Marisha Assistant Professor (Computer Science), School of Science, UPRTOU Prayagraj	Member
Mr. Manoj Kumar Balwant Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Member

Course Preparation Committee

Dr. Brajesh Kumar Associate Prof. Dept. of CSIT, MJPRU, Bareilly, 243006	Author Block – 01 to 04 Unit – 01 to 12
Prof. Manu Pratap singh Prof. Department of Computer Science Dr. Bhimrao Ambedkar University, Agra - 282002	Editor Block – 01 to 04 Unit – 01 to 12
Dr. Chandrakant Kumar Singh Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Coordinator

© UPRTOU , Prayagraj

ISBN :

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar pradesh Rajarshi Tandon Open University.

Overview:

In this block, some important features of Python are discussed including data structures, functions, modules, and packages. Data structures are the important building blocks of any programming languages. Python provides both in-built and user defined options to implement various data structures. A discussion on the implementation of data structures is given in Unit 3. The need of functions is explored in Unit 5. Python functions allow us to group and generalize code and reduces code redundancy by reusing the segment again once needed, further reduces the maintenance efforts. Python modules are the highest organization unit that packages the program code and other information/data to reuse that. Unit 6 starts with a discussion on fundamental ideas about Modules and Packages in Python language. Later, it moves on exploiting some advanced modules.

UNIT 4: Data Structures

Structure:

4.0 Introduction

4.1 Objectives

4.2 Lists

4.3 Tuple

4.4 Sets

4.5 Dictionaries

4.6 Stack

4.7 Queue

4.8 Sequences

4.9 Comprehensions

4.10 Summary

UNIT 4: Data Structures

4.0 Introduction

In this unit, the concepts of data structures are discussed that are required to store, organize and manage data in such a way that it enables easier access of related data and efficient modifications. Python support implicit data structures such as List, Tuple, Sets and Dictionaries. On the other hand, like other programming languages, it also supports user defined data structures such as Stack, Queue, Linked list, Tree and Graphs enabling users to create their own data structures having full control over their functionalities. One needs to be familiar with the basics of data structure before proceeding further to understand and create complex programs. The fundamental in-built data structure in Python is the List. The Lists are sequenced collection of different types of data elements. Here, unique addresses are assigned to every elements of the lists. The addresses assigned to the list elements are known as 'indices'. The Python supports positive as well as negative indices. The positive index starts from 0 and goes up to the last element of the list, whereas the negative index starts from -1 in reverse direction i.e. from the last to the first element of the list. The various operations that can be performed on the list include create, adding elements, deleting elements and accessing elements, etc. The tuples are similar to the list but unlike the list, the data once entered into a tuple cannot be changed. The various operations that can be performed on tuples include creating tuples, accessing elements, concatenation, and many more. The dictionaries contain key value pair, where keys must be unique within a dictionary but values may or may not be unique. The various operations including accessing values in dictionary, updating dictionary and deleting elements from dictionary, etc. can be performed on dictionaries. The sets are another type of data structure which resembles much like sets used in mathematics in the sense that both of them do not allow duplicate values. It means that the sets in python are unordered collection of unique elements. Various operations including Union, Intersection, difference, and symmetric difference, etc. can be performed on the sets using in-built functions like union, intersection, difference, and symmetric_difference, etc.

Apart from above discussed in-built data structures, there are some user defined data structures including stacks, queues, trees, graphs and linked list. Stack is a linear data structure that follows a particular order of operations on its elements. The element which is inserted at last in a stack is removed first from it. Therefore, it is also referred to as Last-In-First-Out (LIFO) data structure. Another user defined data structure is Queue. Queue is also a linear data structure in which elements are inserted from one end of the queue called as rear and deleted from the other end called as front. The PUSH and POP operations in a stack are performed from only one end called as top of stack.

While in a queue the insertion and deletion are performed from different ends known as rear and front respectively. There are many more user defined data structures like Tree, Graphs and Linked list, etc.

4.1 Objectives

The main objectives of this unit are given as follows.

1. To provide the knowledge of basic data structures and their implementations.
2. To explore different types of data structures.
3. To define List, Tuple, Sets, Dictionary, Stack and Queue.
4. To understand significance of various data structures in writing efficient python programs.

4.2 Lists

The fundamental in-built data structure in Python is the *list*. The lists are sequenced collection of different types of data elements. An element in the list can be accessed with help of an index. The Python supports positive as well as negative indices. The positive index starts from 0 and goes up to the last element of the list, whereas the negative index starts from -1 in reverse direction i.e. from last to first element of the list. The various operations that can be performed on list include create, add elements, delete elements and access elements, etc.

4.2.1 Creating a List

A list in Python is created using square brackets. The list can be empty or it may have some elements. If you do not pass anything within square brackets, an empty list is created. If you pass some values within square bracket then a list with those values is created. You can better understand creation of lists using the following example.

```
new_list=[]#creating an empty list
print(new_list)
new_list=[0,1,2,'python',1000.50]#creating a List having some data
print(new_list)
```

Output:

```
[]
[0, 1, 2, 'python', 1000.5]
```

In the above example, an empty list is created with square brackets. When it is printed, square brackets without any elements are printed. The third statement creates a list with some elements. On printing, all those elements are printed.

4.2.2 Adding Elements

You can add more elements in an existing list using in-built functions like `append()`, `insert()` and `extend()`. These functions are briefly explained here.

- `append(x)`: This function is used to add elements at the end of the list. The multiple elements are appended as a single unit.
- `extend(y)`: This function is used to append all the elements from `y` to the list.
- `insert(i,x)`: This function is used to add the element '`x`' at given index '`i`'.

The use of above three functions is illustrated with the help of the following example. The explanation of the statements is given as comments.

```
new_list=[10,20,30]#creating a List data as 10 20 and 30
print(new_list)
new_list.append([100,200])#append data [100,200] to the existing list
print(new_list)
new_list.extend([111,'python'])#extends the list by adding data as different elements
print(new_list)
new_list.insert(10,'programming')#Insert element at index i in existing list
print(new_list)
```

Output:

```
[10, 20, 30]
[10, 20, 30, [100, 200]]
[10, 20, 30, [100, 200], 111, 'python']
[10, 'programming', 20, 30, [100, 200], 111, 'python']
```

4.2.3 Deleting Elements

To delete elements from a list, the Python provides a built-in keyword `del` and some functions like `pop()`, `remove()` and `clear()` as explained here.

- `del`: This is a built-in keyword used to delete an element, but it does not return anything after deleting the element.
- `pop()`: This function deletes an element from the specific index and returns the deleted element.
- `remove()`: This function removes a particular element, passed into the function as parameter.
- `clear()`: This function clears the list, and the list becomes empty.

The significance of above functions can be better understood with the help of an example given as follows.

```
new_list=[10,20,30,'python',50,10,40,50,60]#creating a List
```

```

print(new_list)
del new_list[6]#delete the element present at index 6 (50)
print(new_list)
a=new_list.pop(2)#To pop element from the given index (30)
print('Element Popped:',a,'Remaining List:',new_list)
print(new_list)
new_list.remove('python')#It will remove element with given value
print(new_list)
new_list.clear()#Delete all elements of the list to make list EMPTY
print(new_list)

```

Output:

```

[10, 20, 30, 'python', 50.1, 40, 50, 60]
[10, 20, 30, 'python', 50.1, 40, 60]
Element Popped: 30 Remaining List: [10, 20, 'python', 50.1, 40, 60]
[10, 20, 'python', 50.1, 40, 60]
[10, 20, 50.1, 40, 60]
[]

```

4.2.4 Accessing Elements

The elements in a list are accessed using indices. To access any element, the corresponding index value is passed. The following piece of code illustrates the way a user can access elements from the list.

```

new_list=[10,20,30,'python',50.1,40,50,60]#creating a List data
print(new_list)
for element in new_list:# one by one accessing elements of the list
print(element)
print(new_list)#To access whole elements of the list
print(new_list[2])#To access element at index 2
print(new_list[0:3])#To access list elements from index 0 to 2 and exclude 3
print(new_list[::-1])#To access list of elements in reverse order

```

Output:

```

[10, 20, 30, 'python', 50.1, 40, 50, 60]
10
20
30
python

```



```

50.1
40
50
60
[10, 20, 30, 'python', 50.1, 40, 50, 60]
30
[10, 20, 30]
[60, 50, 40, 50.1, 'python', 30, 20, 10]

```

4.2.5 Other Functions

In addition to above functions, there are several other functions which are used while working with the lists. Some of the functions are briefly explained here.

- `len()`: This function returns the total length of the list
- `index()`: This function gives the index corresponding to the value passed, when encountered for very first time.
- `count()`: This function gives the count of the values being passed to it.
- `sort()`: This function is used to sort the elements of the list. It simply modifies the list, but does not return anything.
- `sorted()`: This function serves the same function as `sort()`, except it has return type.

The usage of some of the functions is illustrated by the following code.

```

new_list=[100,20,10,90,50,10,40,70,60,85,75]#creating a List data
print(new_list)
print(len(new_list))#To find the length
print(new_list.index(20))#To find index of the corresponding element passed as parameter and
occurs very first
print(new_list.count(100))#To find the count of a particular element in the list
print(sorted(new_list))#To print sorted list
new_list.sort(reverse=True)#To sort the original list
print(new_list)

```

Output:

```

[100, 20, 10, 90, 50, 10, 40, 70, 60, 85, 75]
11
1
1
[10, 10, 20, 40, 50, 60, 70, 75, 85, 90, 100]
[100, 90, 85, 75, 70, 60, 50, 40, 20, 10, 10]

```

4.3 Tuple

The tuples are similar to the list but they are mutable. Unlike a list, the data once entered into a tuple cannot be changed. The various operations that can be performed on tuples include creating tuples, accessing elements, and concatenation, etc. Let us understand the creation of tuple and various other operations with example.

4.3.1 Creating a Tuple

In Python, a tuple can be created in the following two ways: using parenthesis and using constructor of the tuple class.

- i) Using parenthesis

```
new_tuple=(1,2,3,4,5,6,7,8,9)
print(new_tuple)
```

Output:

(1, 2, 3, 4, 5, 6, 7, 8, 9)

- ii) Using tuple() function

```
list1=[1,2,3,4,5,6,7,8,9]
tuple1=tuple(list1)
print(tuple1)
```

Output:

(1, 2, 3, 4, 5, 6, 7, 8, 9)

4.3.2 Accessing Elements

The elements of a tuple are accessed in the same way as the elements are accessed in a list. In order to access an element in a tuple, one needs to pass the corresponding index value. The following piece of code better illustrates the way a user can access elements in a tuple.

```
list2=[10,20,30,'python',50,10,40,50,60]#creating a tuple data
tuple2=tuple(list2)
print(tuple2)
for element in tuple2:# one by one accessing elements of the list
print(element)
print(tuple2)#To access whole elements of the list
print(tuple2[2])#To access element at index 2
print(tuple2[0:3])#To access list elements from index 0 to 2 and exclude 3
```



```
print(tuple2[::-1])#To access list of elements in reverse order
```

Output:

```
(10, 20, 30, 'python', 50.1, 40, 50, 60)
10
20
30
python
50.1
40
50
60
(10, 20, 30, 'python', 50.1, 40, 50, 60)
30
(10, 20, 30)
(60, 50, 40, 50.1, 'python', 30, 20, 10)
```

4.3.3 Concatenation of Tuples

It is not possible to add, append, or delete elements in Tuples once they are created because of their mutable property. However, a new tuple can be generated by concatenating two or more tuples. The concatenation operation is performed using '+' operator. Let us consider an example,

```
tuple1=(1,2,3,4,5,5.5,'python',8,9)
# Concatenate some elements with tuple1 to generate a new tuple 'tuple2'
tuple2= tuple1+(4.9,'programming',6)
print(tuple2)
```

Output

```
(1, 2, 3, 4, 5, 5.5, 'python', 8, 9, 4.9, 'programming', 6)
```

Check your progress

1. What is the range of positive index in a List?
2. What is the range of negative index in a List?
3. Write the names of methods for adding elements to the List.

4. Write the names of methods for deleting elements to the List.
5. How an element is accessed in a List?
6. Write the function to determine the length of a List.
7. What is the difference between List and Tuple?
8. How do you append elements of another Tuple to a Tuple?

4.4 Sets

Sets are another type of data structure which resemble to the sets used in the mathematics in the sense that both of them do not allow duplicate elements. It means that the sets in Python are unordered collection of unique elements. Various operations including Union, Intersection, Difference and Symmetric difference can be performed on sets using the in-built functions such as `union()`, `intersection()`, `difference()` and `symmetric_difference()`, etc. Let us understand the sets with the help of some examples.

4.4.1 Creating a Set

To create a set in Python curly/flower braces are used. Here, elements are passed instead of key-value pairs as follows.

```
set1={1,2,3,4,5,5}#creating set
print(set1)
```

Output

```
{1, 2, 3, 4, 5}
```

4.4.2 Adding Elements

Addition of elements in an existing set is performed using `add()` function. Here, the elements to be added are passed as arguments to the function `add()`. For Example,

```
set2={5,10,15,20,25}#Creating set
print(set2)
set2.add(18)#to add 4 as new element in existing set
print(set2)
```

Output

```
{5, 10, 15, 20, 25}
```

```
{5, 10, 15, 18, 20, 25}
```

4.4.3 Other operations on Sets

Several other functions like `union()`, `intersection()`, `difference()` and `symmetric difference()` can also be performed on sets. The purpose of all these functions is given as follows.

- `union()`: This function is used to combine unique elements of two sets
- `intersection()`: This function gives the common elements present in two sets.
- `difference()`: The difference of `set1` and `set2` returns the elements that are in `set1` but not in `set2`.
- `symmetric_difference()`: It is also performed on two sets. It returns the elements that are not in the intersection of the sets.

Let us consider the following example illustrating the use of the above functions for performing different operations on sets.

```
set1={10,20,30,40}
set2={5,10,25,30,45,55,65}
print(set1.union(set2),'-----', set1 | set2)
print(set1.intersection(set2),'-----', set1 & set2)
print(set1.difference(set2),'-----', set1 - set2)
print(set1.symmetric_difference(set2),'-----', set1 ^ set2)
set1.clear()
print(set1)
```

In the above code, each operation except 'clear' is performed in two different ways: using function and using operator. Both methods are separated by '-----' inside the print function. It can be observed from the output that both results are same.

Output:

```
{65, 5, 40, 10, 45, 20, 55, 25, 30} ----- {65, 5, 40, 10, 45, 20, 55,
25, 30}
{10, 30} ----- {10, 30}
{40, 20} ----- {40, 20}
{65, 5, 20, 25, 40, 45, 55} ----- {65, 5, 20, 25, 40, 45, 55}
set()
```

4.5 Dictionary

Dictionary is a data structure that contains key value pairs, where keys must be unique within a dictionary but values may or may not be unique. The various

operations including accessing values in dictionary, updating dictionary, and deleting elements from dictionary can be performed.

4.5.1 Creating a Dictionary

Dictionaries can be created in two ways, one using flower/curly braces and another using dict() function. Here new elements can be added as key-value pairs. Let us understand it with the help of an example.

```
dict1={}#To create empty dictionary
print(dict1)
dict2={1:'Python',2:'Java'}# To Create dictionary having elements
print(dict2)
```

Output:

```
{ }
{1: 'Python', 2: 'Java'}
```

4.5.2 Making changes to Dictionary

In dictionary a value is changed using the corresponding key. It means that first a user needs to access the element with the help of a key then the user can write the new value replacing the older value. Whereas to add new key-value pair user needs to simply write the element value at the corresponding key. Let us consider the following example.

```
dict1={'One':'C language','Two':'XML'}
print(dict1)
dict1['One']=Python#changing the value at key one
print(dict1)
dict1['Three']='Java'#To add new key-value pair after Two
print(dict1)
```

Output:

```
{'One': 'C language', 'Two': 'XML'}
{'One': 'Python', 'Two': 'XML'}
{'One': 'Python', 'Two': 'XML', 'Three': 'Java'}
```

4.5.3 Delete Key-Value pair

In dictionary key-value pair can be deleted in a variety of ways such as:

- ❑ `pop()`: This function returns the deleted value at the given key.
- ❑ `popitem()`: This function randomly deletes and returns the value.
- ❑ `clear()`: This function clears the dictionary to make it empty.

The significance of above functions can be better understood with the example given as follows.

```
dict1={'One':'C Language','Two':'XML','Three':'Python'}
del_value=dict1.pop('Three')# To delete value at Key: three
print('Deleted Value:',del_value)
print('Dictionary after deleting last value:', dict1)
del_key_pair=dict1.popitem()#To randomly delete the key-value pair
print('Deleted Key-value pair:',del_key_pair)
print('Dictionary after deleting key-value pair', dict1)
dict1.clear()#To clear dictionary, make it empty
print(dict1)
```

Output:

```
Deleted Value: Python
Dictionary after deleting last value: {'One': 'C Language', 'Two':
'XML'}
Deleted Key-value pair: ('Two', 'XML')
Dictionary after deleting key-value pair {'One': 'C Language'}
{}

```

4.5.4 Accessing Dictionary Elements

Dictionary elements are accessible only using keys. If you want to access a value then you need to pass the corresponding key. There are two ways to access values. Let us understand it with the help of an example.

```
dict1={'One':'C Language','Two':'XML'}
print(dict1['One'])#To access dictionary elements using key
print(dict1.get('Two'))# To access dictionary element using get() function

```

Output:

```
C Language

```


4.5.5 Other Functions

There are several other functions used to return key, value, or key-value pair. These functions are given as follows.

- ☐ `keys()`: This function returns all the keys of dictionary
- ☐ `values()`: This function returns all the values of dictionary
- ☐ `items()`: This function returns all the key-value pair of dictionary

Let us understand all these functions with an example.

```
dict1={'One':'C Language','Two':'XML','Three':'Python'}
print(dict1.keys())#To access all keys
print(dict1.values())#To access all values
print(dict1.items())#To access key-value pairs
```

Output:

```
dict_keys(['One', 'Two', 'Three'])
dict_values(['C Language', 'XML', 'Python'])
dict_items([('One', 'C Language'), ('Two', 'XML'), ('Three', 'Python')])
```

Check your progress

1. Differentiate 'difference' and 'symmetric difference' operations on sets.
2. Differentiate 'pop()' and 'popitem()' methods in Dictionary.
3. How do you remove all the elements in a dictionary?
4. How do you create a Dictionary?
5. How do you add new elements to a set?
6. How do you take union of two sets?
7. How do you make changes to Key-Value pair in a dictionary?

4.6 Stack

Stack is a linear data structure, in which insertion and deletion of items occurs only from one end of the data structure known as top of stack. In general a variable named as 'Top' is used to represent the top of stack. In stack PUSH() function is used to add an element, while POP() function is used to remove an

element. In this data structure elements are removed in reversed order in which they were added. So it is also called as Last-In-First-Out (LIFO) data structure.

4.6.1 Operations on Stack

In general, there are three main operations that are performed on stack including PUSH, POP, and Traversal. The purpose of these operations is given as follows.

- **PUSH:** This operation is used to insert an element at the top of the stack. As a result of PUSH operation, the size of the stack is increased by the number of items added. The addition of element always takes place at the top of the stack. Initial value of top is -1 i.e. $Top = -1$. It shows that the stack is empty. One should always check the value of variable Top before performing PUSH operation. If $Top = MAX - 1$, then condition of OVERFLOW occurs. Where MAX is the size of the stack. Overflow means that there is no space available in the stack to accommodate the new items, hence PUSH operation is not possible.
- **POP:** This operation removes the elements from the top of the stack. It reduces the size of stack by the number items being popped. One should always check the value of variable Top before performing POP operation. If $Top = -1$ (stack is empty) and still user wants to POP an item then UNDERFLOW occurs. Underflow means that there are no more elements in the stack, and hence POP can not be performed.
- **Traversal:** This operation displays the items left over in the stack. If $Top = -1$, then it shows that the stack is empty.

An example is given here to illustrate some operations on the stack.

```
# Implementing stack in Python using List
stack=[]#Initially stack is empty
print('stack is empty',stack)
# Add 2 items in Stack
stack.append("Python")
stack.append("programming")
print('stack after pushing two items is:', stack)
# Remove one item from stack (top most item)
print('popped item is:',stack.pop())
print('Stack after removing top item is:',stack)
# Add 2 items in Stack
stack.append("is")
stack.append("Not")
print('stack after pushing two items:', stack)
```

```

# Removing item from stack
print('popped item is:',stack.pop())
print('Stack after removing top item is:',stack)
# Add 1 item in Stack
stack.append("Easy")
print('stack after pushing one item is:', stack)

```

Output:

```

stack is empty []
stack after pushing two items: ['Python', 'programming']
popped item is programming
Stack after removing top item is: ['Python']
stack after pushing two items: ['Python', 'is', 'Not']
popped item is Not
stack after removing top item is: ['Python', 'is']
stack after pushing one item is: ['Python', 'is', 'Easy']

```

4.7 Queue

Queue is another linear data structure where elements are stored and processed sequentially. It is based on the principle of First-In-First-Out (FIFO) or First-Come-First-Serve (FCFS), in which element first entered is processed first. Queue differs from the stack as it has two operational ends namely Front and Rear. Insertion of elements is performed from Rear end, whereas the deletion is performed from Front end of the queue.

4.7.1 Operations Associated with Queue

Following operations are associated with the queue data structure.

- Enqueue- This operation adds elements/items in the queue, thereby increasing the total number of elements in the queue. If the queue is full and user wants to perform Enqueue operation, then OVERFLOW condition occurs. Overflow simply means no more elements/items can be added to the queue.
- Dequeue- This operation removes elements/items from the queue, thereby decreasing the total number of elements in the queue by the number of elements/items being removed. If queue is empty then dequeue operations leads to UNDERFLOW condition.

4.7.2 Implementation of Queue

In Python the queue data structure can be implemented in several ways as follows.

- ☐ Using list
- ☐ Using collections.deque
- ☐ Using queue.Queue

The List is a built-in data structure in Python which can be used like a queue. However insertion and deletion of elements at the front of the queue is quite slow because it requires shifting rest of the element by one position. We can better understand implementation of queue using list with the following example.

```
# implementation of queue using list

# queue Initialization
new_queue=[] # Empty queue
print('Queue is Empty:',new_queue)
# Insertion of 3 elements in queue
new_queue.append('r')
new_queue.append('g')
new_queue.append('b')

print("Initially elements in queue are:")
print(new_queue)

# Deletion of 2 elements from queue
print("\nElements deleted from queue are :")
print(new_queue.pop(0))
print(new_queue.pop(0))

print("\nRemaining Queue after performing insertion and deletion of elements: ")
print(new_queue)
```

Output:

```
Queue is Empty: []
Initially elements in queue are:
['r', 'g', 'b']
Elements deleted from queue are :
r
g
Remaining Queue after performing insertion and deletion of elements:
['b']
```

Queue can also be implemented using deque class of Python's collections module. Whenever quicker append and pop operations are needed, deque is preferred over the list because it has time complexity of $O(1)$ against list time complexity of $O(n)$.

```
# Implementation of Queue using collections.deque
from collections import deque
# Queue Initialization
queue=deque()

# Inserting 3 elements in the queue
queue.append('r')
queue.append('g')
queue.append('b')

print("Elements in the Queue are: ")
print(queue)

# deleting 2 element from queue
print("\n Deleted elements from the queue are: ")
print(queue.popleft())
print(queue.popleft())

print("\nRemaining element in the queue are: ")
print(queue)
```

Output:

```
Elements in the Queue are:
deque(['r', 'g', 'b'])
Deleted elements from the queue are:
r
g
Remaining element in the queue are:
deque(['b'])
```

In Python the queue can also be implemented using a built-in module known as Queue. In this module several functions are available, which are given as follows.

- ☐ `maxsize()`: It signifies the total number allowed items in queue. Zero value of maxsize means infinite queue.
- ☐ `empty()`: It returns True for the empty queue and false otherwise.
- ☐ `get()`: Returns the deleted item from the queue.
- ☐ `get_nowait()`: It returns item if available immediately.
- ☐ `put_nowait(item)`: It inserts an item in the queue without blocking.
- ☐ `put(item)`: This method is used to insert one item in the queue.
- ☐ `qsize()`: This method provides the size of queue.
- ☐ `full()`: This method returns True if queue is full otherwise it returns False.


```

# Implementing queue using module Queue
from queue import Queue
# queueInitializ by maxsize
queue= Queue(maxsize=3)
# qsize() function provides maxsize of Queue
print("Size of queue is:",queue.qsize())
# inserting 3 elements in queue using put() function
queue.put('r')
queue.put('g')
queue.put('b')
# Boolean value is returned when queue is Full
print("queue is Full: ",queue.full())
# Deleting 2 elements using get() function from queue
print("Deleted elements are: ")
print(queue.get())
print(queue.get())
# Boolean value returned for Empty queue
print("queue is Empty: ",queue.empty())
queue.put(1)
print("queue is Empty: ",queue.empty())
print("queue is Full: ",queue.full())

```

Output:

```

Size of queue is: 0
queue is Full: True
Deleted elements are:
r
g
queue is Empty: False
queue is Empty: False
queue is Full: False

```

4.8 Summary

Data structures are highly important for implementing software in any programming language. Python supports various in-built data structures such as List, Dictionary, Sets, and Tuple, etc. In addition, the programmers can define their own data structures including well known data structures such as stack, queue, link list, and tree, etc. The elements in many data structures can be accessed using index. Both negative and positive indices are allowed in Python. Various operations can be performed on data structures using in-built methods such as add, remove, and change elements of the data structure. Some specific operations are also defined for data structures in Python. For example, union and intersection operations are allowed on set data structure.

Lists are sequenced collection of different types of data elements. Here unique addresses are assigned to every element of the lists. The addresses assigned to list elements are known as Indexes. Tuples are similar to list with

one difference that data once entered into list can be changed while the same is not possible in case of tuples. Sets are another type of data structure which resembles much like sets used in mathematics in the sense that both does not allow duplicate values, which means sets in python are unordered collection of unique elements. Dictionary is a data structure that contains key value pair, where keys must be unique within a dictionary but values may or may not be unique. Stack is a linear data structure, in which insertion and deletion of items occurs only from one end of the data structure known as top of stack. Queue is another linear data structure where elements are stored and processed sequentially. It is based on the principle of FIFO. Different types of data structure can be implemented in Python either by using in-built constructs or as user defined entities.

4.9 Review Questions

- Q.1 What do you understand by data structure? How do data structure implemented in Python?
- Q.2 What are the major in-built data structures in Python? Explain with their features and associated operations and methods.
- Q.3 Create a list in Python and illustrate use of various in-built methods with help of a suitable code.
- Q.4 Demonstrate the creation of a dictionary in Python. Perform the allowed the operations on the created dictionary.
- Q.5 Perform various operations on sets in Python using a suitable code.
- Q.6 What is stack? How do create stack in Python? Write a code to perform PUSH and POP operations on stack.
- Q.7 What is the difference between stack and queue? Implement a queue using two different approaches in Python and perform the related operations.

UNIT 5: Functions

Structure:

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Defining a Function
- 5.3 Calling Functions
 - 5.3.1 Call by reference
 - 5.3.2 Call by value
- 5.4 Types of functions
- 5.5 Function Arguments
 - 5.5.1 Required Arguments
 - 5.5.2 Keyword Arguments
 - 5.5.3 Default Arguments
 - 5.5.4 Variable-length Arguments
- 5.6 Scope of the Variable
- 5.7 Anonymous Functions
- 5.8 Fruitful Function
- 5.9 Summary
- 5.10 Review Questions

Unit 5: Functions

5.0 Introduction

In this unit, the basics of functions in Python programming are discussed and the requirements for a functions are covered. The fundamental concepts related to functions such as defining a function, scope of the function, calling a function, passing arguments and other basic aspects of functions are described. Before understanding complex Python programming using functions, one should familiar with the need of function and one should understand how it works. A set of additional statement are explored related to creating a function for efficient programs in Python language.

In simple terms, a function is group or set of statements that can perform single, specific and well defined task more than once in a program. The parameters or arguments serve as input to functions. Functions can also compute a result with the specified parameters which may differ each time a code runs. Python enables the programmers to code a program through the functions with a specific task. The functions are a useful tool, which can be used repetitively in a variety of contexts.

More fundamentally, functions provide alternative to cutting and pasting a program segment by the redundant use of any specific operation's code. We can address it as a single function. Thus, the future work is reduced and according to the change in usability of the function we need to update only a single copy of that function instead of many. Moreover, functions are the basic building blocks of the programs, which minimizes the code redundancy maximizing the code reuse. Also, functions act as designing tool which breaks the complex programs into manageable short program segments/operations.

To create a function, the programmers write its definition including function header and function body. In order to use the function, the program control is passed to the first statement of the function. After executing all the statements one-by-one inside the function body, the control goes back to the calling function. A function, which uses another function is known as "Calling function" and function which is being called is termed as "Called function".

Before getting into the details, let's establish a clear picture of what functions are and why we need these in programming. Functions are the universal structure of the code to accomplish a specific task. It might be used repeatedly. You may be aware or may have come across with them before in other programming languages. Briefly, function serves following primary roles:

1. Maximizing code reuse and minimizing code redundancy:

Similar to other languages, the Python programming allows programmers to pack a logic that he wish to use more than once. Python functions allow us to group and generalize code and

reduces code redundancy. Further reduces the maintenance efforts. In simple words, we need to write the code of the operation only once and it can be reused each time you need to perform that task.

2. **Procedural decomposition:**

Functions provide a way to split the program in code segments, each of which has a well-defined set of task. A program coded using functions appears simpler and easier to understand. And, it is easier to read a program that consists of several small functions instead of one long sequence of statements.

3. **Better Testing:**

Testing and debugging would be simple if the program is coded using functions. Each function can be tested individually and bugs can be removed easily.

4. **Faster Development:**

The entire program is divided into several other small tasks, which can be developed or coded by individual programmer or team. Thus, it allows the faster development of programs.

5.1 Objectives

The objectives of this unit are:

1. To discuss the functions and their need in the programming.
2. To define function with its body and function header.
3. To discuss types of function.
4. To use arguments and the parameter passing to a function.
5. To understand scope of a function and function calls.

5.2 Defining a Function

A function is a block of code, which only runs when it is called. You can pass the data, known as parameters, into a function. A function can return data as a result. We need to define a function to achieve the required functionality. A function can return data as a result.

Creating a Function: The general way to define a function is given as follows.

```
def function_name(): #function header
    statement1
    statement2
    .....
    Statement N
```



Function Body

The simple rules related to the defining a function in Python programming are given as follows.

- ❖ In Python, a function is defined using the *def* keyword. It creates a function object and assign a name to the function. The block of a function starts with the *def* keyword followed by function name and parentheses '()' followed by a colon ':'. This statement is known as **function header**.
- ❖ The input parameter or argument should be written inside the parentheses. Also, inside the parentheses the parameters can be defined. The function can have zero or more arguments.
- ❖ The code block or the **function body** within every function begins with the colon and indentation. The function body may have one or more statements to be executed in sequence to perform the specific task for which it has been defined. The first statement of the function body is documentation string or *docstring*, which is optional statement.
- ❖ The *return* statement exits the function passing back the control to the caller function optionally. A return statement without any argument is like return none.

Syntax:

```
def functionname( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

The parameters have a positional behavior and used in the same order as they were expressed in the function definition.

Example 1:

The following function takes a string as the input parameter and prints it on standard screen.

```
def printme( str ):
    "This prints a passed string into this function"
    print str
    return
```

The above function is defined as “printme” and takes the string as an argument. The string “str” is the parameter and the first line shows the docstring which is optional. The statement inside the function body simply prints the string received as argument.

Example 2: Function definition to find out if the given number is an even number or an odd number.

```
#function definition
def eventest(x):
    if x%2==0:
        print("even")
    else:
        print("odd")
```

In the above function definition, the function having name “eventest” takes “x” as input argument and body of the function declares whether the given number is an even number or an odd number.

5.3 Calling Functions

The function definition only specifies its name, parameter, and tells about the task to be performed. To execute the function we must call it the program segment. On finalizing the structure of the function, it can be directly executed at the Python prompt. It can also be called by another function with specified arguments.

At the time of function call, the interpreter jumps to the function definition and all the statements of the function body are executed sequentially. And, it jumps back to the calling function once it reaches at the end of the function body. It means that the function returns to the calling position in the code.

Example 3: In this example, the function printme() is called:

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str
    return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Output: When the code is executed, we get the following output.


```
$python main.py
I'm first call to user defined function!
Again second call to the same function
```

Example 4: Function `eventest()` is called in the following way:

```
#function definition
def eventest(x): #function header
    if x%2==0:
        print("even")
    else:
        print("odd")

n=int(input("Enter any number:"))
#function calling
eventest(n)
```

Output: On executing the code, the following output will be generated.

```
Python 3.8.3 Shell
File Edit Shell Debug Options Window Help
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:37:02) [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Enter any number:12
even
>>>
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Enter any number:5
odd
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Enter any number:2
even
>>>
```

The functions can be called in two ways: 1) call by reference and 2) call by Value.

5.3.1 Call by Reference

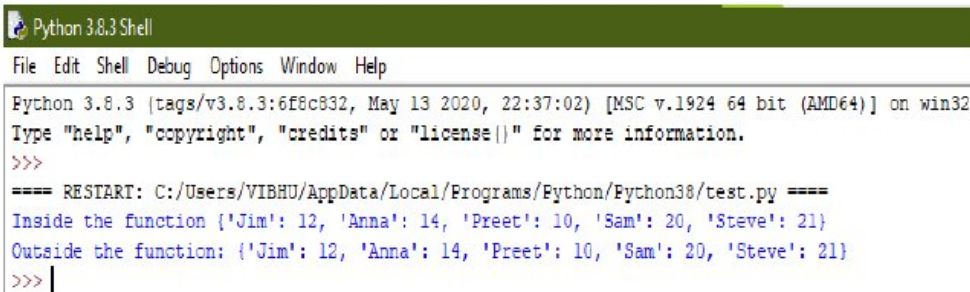
When a function is called using the addresses of the variables instead of copying the values of variables, it is termed as calling by reference in programming languages. In this strategy, a variable itself is passed as an argument in the function. When arguments are updated inside the function body, the changes are also reflected outside the function. Thus, it allows us to modify the actual value by function calls. All parameters in Python programming are passed using the reference. If we change the parameter inside a function it reflects changes in the calling function also.

Example 5: Demonstrating the call by reference in Python programming.


```
# Python code to demonstrate call by reference

student = {'Jim': 12, 'Anna': 14, 'Preet': 10}
def test(student):
    new = {'Sam':20, 'Steve':21}
    student.update(new)
    print("Inside the function", student)
    return
test(student)
print("Outside the function:", student)
```

Output: When the dictionary variable 'student' gets modified using update function, it reflects the changes outside the function also.



```
Python 3.8.3 Shell
File Edit Shell Debug Options Window Help
Python 3.8.3 [tags/v3.8.3:6f8c832, May 13 2020, 22:37:02] [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Inside the function {'Jim': 12, 'Anna': 14, 'Preet': 10, 'Sam': 20, 'Steve': 21}
Outside the function: {'Jim': 12, 'Anna': 14, 'Preet': 10, 'Sam': 20, 'Steve': 21}
>>> |
```

Example 6: Demonstrating Call by Reference in Python program using a list.

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist.append([1,2,3,4]);
    print "Values inside the function: ", mylist
    return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Output:

```
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

5.3.2 Call by Value

It is the condition when we called a function by copying the values of variable as the arguments. It means a copy of variable is passed instead the variable itself. Thus, changes in the copy of variable are not reflected outside the function. More clearly, changes made inside the function are not reported to the calling function. It does not allow us to modify the original values of variables.

Example 7: Demonstrating the call by value in Python programming.

```
# Python code to demonstrate call by value

student = {'Jim': 12, 'Anna': 14, 'Preet': 10}
def test(student):
    student = {'Sam': 20, 'Steve': 21}
    print("Inside the function", student)
    return
test(student)
print("Outside the function:", student)
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Inside the function {'Sam': 20, 'Steve': 21}
Outside the function: {'Jim': 12, 'Anna': 14, 'Preet': 10}
>>> |
```

Example 8: Demonstrating Call by Value in Python program using a list.

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist = [1,2,3,4]; # This would assign new reference in mylist
    print "Values inside the function: ", mylist
    return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Output: Here, the parameter “mylist” is local to the function changeme(). Updating mylist inside the function does not affect it outside. And, the following result is generated:

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```

Check your progress

1. What is a function?
2. How are the functions created in Python?
3. What is function header?
4. What do you understand by function arguments?
5. What is *docstring* in a Python function?
6. How do you call a function?
7. List some advantages of a function.
8. Differentiate the *call by value* and *call by reference* approaches of making a call to Python functions.

5.4 Types of Functions

The functions in Python programming language are of two types: 1) built-in functions and 2) user-defined functions.

5.4.1 Built-in Functions

Built-in functions or in-built functions or pre-defined functions are the functions that already exist in Python or in Python toolboxes. Once we install the Python, these functions automatically get installed along with their full specification/definitions. Therefore, we do not need to define these functions but before using them one must be aware of their definition. In simple words, these are readily available for use.

Example 9: The built-in function `len(s)` returns the length of any object. Arguments may be a list, bytes, tuple, range or string. And, `type()` function returns the type of the object.

```
#In built functions

x = [1,2,3,4,5]
print(len(x)) #it return length of list
print(type(x)) #it return object type
```

Output:

```
# 5
# <class 'list'>
```

Here, some built-in functions along with their specifications are given as follows:

Function name	Description
len()	It returns the length of an object/value.
list()	It returns a list.
max()	It is used to return maximum value from a sequence (list,sets) etc.
min()	It is used to return minimum value from a sequence (list,sets) etc.
open()	It is used to open a file.
print()	It is used to print statement.
str()	It is used to return string object/value.
sum()	It is used to sum the values inside sequence.
type()	It is used to return the type of object.
tuple()	It is used to return a tuple.

To get more built-in functions with detailed description follow the source: [Built-in Functions — Python 3.9.6 documentation](#).

5.4.2 User-defined functions

User defined functions are the functions that are defined, customized, and written by the user to accomplish a specific task. Also, functions written by others and used after installing any library are also user defined functions. Such functions are also known as library functions. These function can be from third party libraries developed by others. The programmers define these function for specific purpose and reduce the complexity of the big problem.

Example10: Example of user defined function.

```
#Example of user defined function

x = 3
y = 4
def add():
    print(x+y)

add()
```

Output: # 7

Example 11: Another example of user defined function.

```
def product(a, b):  
    return a * b  
  
print (product (2, 7))
```

Output: After executing above code, we get the following output.

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====  
14  
>>> |
```

5.5 Function Arguments

Information or any piece of input data can be passed into functions as arguments. Arguments/ parameters are specified after the function name, within the parentheses. A function can have one or more arguments separated by comma. The arguments mentioned in function call are known as Actual Arguments and arguments used in definition of function are called as Formal Arguments. Consider the following code to better understand the concept of arguments.

```
#function definition  
def eventest(x): #function header, here x is called Formal Argument  
    if x%2==0:  
        print("even")  
    else:  
        print("odd")  
  
n=int(input("Enter any number:"))  
#function calling  
eventest(n) # here n is called Actual argument
```

Example 12: The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):  
    print(fname + " Refsnes")  
|  
my_function("Emil")  
my_function("Tobias")  
my_function("Linus")
```

Output:

[113]


```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Emil Refsnes
Tobias Refsnes
Linus Refsnes
>>>
```

One can call a function through the following mentioned formal arguments, which are described in later subsections.

- ❖ Required arguments
- ❖ Keyword arguments
- ❖ Default arguments
- ❖ Variable –length arguments.

5.5.1 Required Arguments

The required arguments are the arguments, which are passed in a function in correct way and right positional order. That means, the number of arguments and the order of arrival must exactly match the function definition. If it fails to follow, the syntax errors are raised during the compile time.

Example 13: Let consider the following function definition for *printme()*. Here, we need to pass exactly one argument, otherwise a syntax error results as follows.

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;

# Now you can call printme function
printme()
```

On executing above code, the following error message is produced: **Syntax Error**

```
>>>
===== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Traceback (most recent call last):
  File "C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py", line 10, in <module>
    printme()
TypeError: printme() missing 1 required positional argument: 'str'
>>>
```

A syntax error is raised due to incorrect arguments position and numbers. The correct use of the function and the output is as follows.

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;

# Now you can call printme function
printme("Python Programming")
```

Correct Output

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Python Programming
>>>
```

5.5.2 Keyword Arguments

The keyword arguments are related to the function calls. When the keyword arguments are used in a function call, the arguments are identified by the caller through the argument names. With the help of keyword arguments, the Python allows a function call in which the order or position of the arguments can be changed. The argument values are not identified by their position but by their names. The actual arguments in the function call can be written as follow.

Function_name (Argument_name1=value1, argument_name2=value2)

An argument that is written according to this syntax is known as “**Keyword Argument**”. Also, it allows the programmer to use the argument out of order or skip them.

Example14: Use of keyword arguments in the function call printme().

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;

# Now you can call printme function
printme(str= "Python Programming")
```

Output: On executing above code, the following output is produced:

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Python Programming Keyword argument demo
>>> |
```

Example 15: Calculating Simple interest using keyword arguments.

```
#keyword arguments
def simpleinter(principal,rate,time): #function Header
    i=(principal*rate*time)/100
    print("The interest is:",i)
    print("Total amount is:",principal+i)
#function call
simpleinter(rate=7.25,time=3,principal=5000)
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
The interest is: 1087.5
Total amount is: 6087.5
>>> |
```

5.5.3 Default Arguments

Python language allows to skip some or all arguments in a function call. A default value is assigned to all the arguments, if in case any argument is not present in function call than default value is used. The default value is assigned to an argument using the assignment operator(=). More clearly, if all the arguments have passed during the function call then the default values remains unconsidered. The values of actual arguments are copied to formal arguments in this case otherwise default arguments are considered. Therefore, the formal arguments are overwritten by the actual arguments if available.

Syntax: A general format to use default arguments in Python language is given as follows.

```
#function definition
def function_name(arg1=val1,arg2=val2,arg3=val3)
    Statement 1
    Statement 2
    Statement 3
#function call
function_name() #without arguments
function_name(val1,val2) #with two arguments, third argument is taken from default argument
```

Example 16: The following example gives an idea on default arguments, it prints default age if it is not provided.

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)|
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Name: miki
Age 50
Name: miki
Age 35
>>>
```

Example 17: Adding three numbers

```
#default arguments

def add(x=12,y=13,z=14):
    t=x+y+z
    print("The sum is:",t)

#function call without arguments
add()
#function call with one argument
add(1)
#function call with two arguments
add(10,20)
#function call with three arguments
add(10,20,30)
```

Output:


```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
The sum is: 39
The sum is: 28
The sum is: 44
The sum is: 60
>>>
```

5.5.4 Variable-length Arguments

In some situations, it is not known in advance that how many number of arguments have to be passed to the function. In such cases, Python allows programmers to make function calls with arbitrary (or any) number of arguments. When we use arbitrary arguments or variable-length arguments, then the function definition uses an asterisk (*) before the formal parameter name. Syntax for a function with non-keyword variable arguments is as follows.

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example.

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print ("Output is: ")
    print (arg1)
    for var in vartuple:
        print (var)
    return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output:


```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Output is:
10
Output is:
70
60
50
>>> |
```

Example 18:

```
#variable-length argument
def var_len_arg(name,*args):
    print ("\n",name,"Hobbies are:")
    for x in args:
        print(x) #function call

#subash is assigned to name, and rest of arguments are assigned to *args
var_len_arg("Subash","Cricket","Movies","Traveling")

#rajani is assigned to name, and rest of arguments are assigned to *args
var_len_arg("Rajani","Reading Books","Singing","Tennis")
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py

Subash Hobbies are:
Cricket
Movies
Traveling

Rajani Hobbies are:
Reading Books
Singing
Tennis
>>>
```

Check your progress

1. How many types of functions are there in Python?
2. What do you understand by built-in or in-built functions?
3. The functions available in a Python library are built-in functions or user defined functions?
4. How do you create your own functions in Python?
5. How do you separate different arguments in a function?
6. What do you understand by formal arguments?
7. What is required argument in a function?

8. Can you change the order of arguments in a function call?
9. What do you understand by default arguments?
10. Is it mandatory to fix the number of arguments in a Python function?

5.6 Scope of the Variable

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable. The part of the program in which a variable can be accessed is called its scope. The duration for which a variable exists is called its "Lifetime". If a variable is declared and defined inside a function its scope is limited to that function only. It cannot be accessed outside that function. If an attempt is made to access it outside that function an error is raised. The scope of a variable determines the portion of the program where you can access it. There are two basic scopes of variables in Python:

Global Scope: The variables that are the part of main program have global access. These are not defined inside any function. They can be accessed anywhere in the program.

Local Scope: The variables that are defined inside a function have local scope. They can be accessed inside that function only.

Global vs. Local variables: The variables defined within the body of the function have a local scope and known as local variables. Whereas the variables defined outside of the function body have a global scope and are known to be a global variable. Therefore, we can access the local variable inside the scope of the function where it was declared. But, the global variables can be accessed anywhere in the program body including the main function and all the other functions.

Example 19:

```
#!/usr/bin/python

total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2; # Here total is local variable.
    print ("Inside the function local total : ", total)
    return total;

# Now you can call sum function
sum( 10, 20 );
print ("Outside the function global total : ", total)
```

Output:

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Inside the function local total : 30
Outside the function global total : 0
>>>
```

Example20:

```
tot=10
def sum(a,b):
    global tot #refering the global variable
    tot=a+b
    print("The sum is:",tot)

sum(12,3)
print("The global value of tot is:",tot)
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
The sum is: 15
The global value of tot is: 15
>>> tot
15
>>> sum(14,3)
The sum is: 17
>>> tot
17
>>>
```

5.7 Anonymous Functions

The lambda or anonymous functions are not defined using the *def* keyword like other functions in Python programming. These function are defined using the *lambda* keyword. These are known as throw-away function as they can just be used where they have been created. The important characteristics of lambda functions are mentioned as follows.

- ❖ A lambda function can take any number of arguments. However it returns only one value in the form of an expression.
- ❖ The lambda function cannot be called directly to print as they require expression.
- ❖ These functions have their own local namespace and can access variable that are in their parameter list.
- ❖ These function are online version of the functions in Python language.
- ❖ The lambda functions have no name.

Syntax: The syntax of the anonymous function defined using the lambda statement consists of a single line given as follows:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

The arguments composed of a comma separated list of variables. The expression is an arithmetic expression that employs these arguments. These functions can be assigned to a variable to give them a name.

Example 21: Following is the example to show how the lambda form of functions works.

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

Output: On executing the above code, it produces the following result:

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Value of total : 30
Value of total : 40
>>> |
```

Example 22: Python program to find the power of a number using lambda function.

```
#lambda or anonymouse function

n=lambda x,y: x**y

x=int(input("Enter value of x :"))
y=int(input("Enter value of y :"))

#function call in the print() function
print(x,"power",y,"is",n(x,y))
```

Output:


```

==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Enter value of x :5
Enter value of y :5
5 power 5 is 3125
>>>
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Enter value of x :2
Enter value of y :3
2 power 3 is 8
>>> |

```

Calling lambda function from other functions:

The lambda functions can be called by other functions. The function that calls a lambda function needs to pass its arguments to lambda function itself. Thus the lambda function performs its task and return the value to the caller.

Example 23: A Python program to increase the value of a number by one using the lambda function.

```

#lambda or anonymous function
def inc(y):
    return(lambda x: x+1)(y)

x=int(input("Enter value of x :"))

#function call in the print()function

print("the increased value of ",x," by one is",inc(x))

```

Output:

```

>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ====
Enter value of x :4
the increased value of  4  by one is 5
>>> |

```

5.8 Fruitful Function

The fruitful functions are the functions that return a value. Every function after completing its operation returns the control to the calling function. This is done using the keyword 'return'. If the keyword 'return' is used without any variable/argument then it is known as implicit return statement. The implicit return statement returns nothing to its caller except the program control. If this return statement is used with a variable/argument then it is called explicit return statement. The explicit return statement returns the given value as well as program control to its caller. More clearly, the statement 'return [expression]'

exits a function with optionally passing back an expression to the caller. A return statement with no arguments is same as a return statement that returns 'None'.

Syntax:

`return (expression)`

The expression is written inside the parenthesis and it computes a single value. The return statement provides the two benefits to the programmers: 1) it returns a value to the calling function and 2) it passes the program control back to the caller function by exiting the current function.

Example 24: The return statement that returns a value.

```
#!/usr/bin/python

# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2
    print ("Inside the function : ", total)
    return total;

# Now you can call sum function
total = sum( 10, 20 );
print ("Outside the function : ", total)
```

Output: On executing the above program, the following output is produced:

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Inside the function : 30
Outside the function : 30
>>>
```

Example25: Python program using the fruitful function to find the area of a circle.

```

#finding the area of a circle
def area(r):
    a=3.14*r*r
    return(a) #function returning a value to the caller

#begining of main function

x=int(input("Enter the radius :"))
#calling the function

r=area(x)
print("The area of the circle is:",r)

```

Output:

```

>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Enter the radius :2
The area of the circle is: 12.56
>>>

```

5.9 Summary

The functions in Python language are the basic building blocks for complex and long programs. The functions make the program easier to read and simpler to understand. With the help of functions, a larger program can be divided smaller units that implemented independently by different programmers. The function definition and arguments must be given according to the syntax otherwise it raise error during compile time. Arguments are the input data on which a function operates and perform the specific task. There are two types of functions: built-in functions and user-defined functions. The users can define their own set of functions for ease of programming. The built-in functions are already available with the languages that can be used by the programmer directly. However, a programmer must be well-aware of the structure of the functions available in Python.

Arguments or the parameters of the function must be in the same number and in the same order while calling a function. However, Python provides relaxation with some additional key features related with arguments such as keyword arguments, default argument, variable length arguments. These all add some special functionalities to the way of using arguments while calling a function. Default arguments allow the use of default values of arguments defined in the function definition if any argument is missing in caller. Moreover, anonymous functions defined using the lambda statements are the only functions which are based on the list of arguments and arithmetical expressions. Thus, lambda function cannot be called directly because it is based on the expression.

Accessing the variable declared in the function body or outside the body depends on the scope of variable. A local variable defined inside the function has a local scope and cannot be accessed outside of the function. Whereas global variables can be accessed anywhere in the program. The return statement helps the programmer to return the program control as well as some value to caller. It returns a single value as output and the expression shows what to return. A return statement with no arguments just returns nothing. Conclusively, we can say that a function is a block of statements that is independent of the remaining code. In this way, we can call the function at any time. The functions are a kind of small building blocks that together can create large programs.

5.10 Review Questions

- Q.1 Define a function. Write the difference between user-defined and built-in functions in Python.
- Q.2 Write down the benefits of functions in Python programming.
- Q.3 What are the two ways of calling a function? Illustrate with the help of Python programs.
- Q.4 How do you define the scope of function variable in Python language?
- Q.5 What are function arguments? Explain different kinds of arguments in a function. Explain with the help of suitable Python programs.
- Q.6 What is the output of the following program?

```
result = lambda x: x * x
print(result(5))
```

- Q.7 Explain the role of lambda function in Python. Illustrate with the help of a Python program.
- Q.8 Explain the role of the return statement. Explain fruitful functions with the help of a Python program.
- Q.9 Write a function that converts Kelvin temperature (KK) to degrees Celsius (CC). The formula to convert between the two temperature scales is $C = K - 273.15$.
- Q.10 Run the following code and explain the error if any. Rewrite the code and make it error free.

```
def first_a(a):
    return a[0]

first_a(1)
```

UNIT-6: Modules and Packages

Structure:

- 6.0 Introduction
- 6.1 Objectives
- 6.2 Module
- 6.3 Creating Modules
- 6.4 Module usage
 - 6.4.1 *import* statement
 - 6.4.2 *from.import* statement
- 6.5 Namespacing
- 6.6 Python Packages
 - 6.6.1 Creating Python packages
 - 6.6.2 Importing the packages
- 6.7 Introduction to PIP
- 6.8 Installing packages via PIP
- 6.9 Usage of Python Packages
- 6.10 Summary
- 6.11 Review Questions

Unit 6: Modules and Packages

6.0 Introduction

This unit starts with the discussion on fundamental ideas about Modules and Packages in Python language. Further, we can see that Python modules are the highest organization unit that group the program code and other information/data to reuse that. Modules usually correspond to Python program files. The concept of the core module is explained in this unit. Later some advanced concepts related to the modules are discussed. It provides a look on the role of modules in overall program structure. The program code or data available in a module can be used by a programmer with the help of 'import' and 'from' keywords. The 'import' statement gives a name to the whole module object and helps to fetch its attribute to reuse them. Whereas, the 'from' statement copies the names from one file to another within the space.

Each module determines its own namespace, which permits that the same name can be used in distinct function or modules. Python packages group a large set of classes and modules together. If we have a higher number of classes and modules in a program, it would be preferred to wrap them in a package. PIP is the name of a tool that is used for installing packages in Python. The Python package index is a repository of all Python packages. Currently, there are more than eleven thousand packages stored in this repository. PIP tool can be used to install these packages as per need.

6.1 Objectives

The objectives of this unit are as follows:

1. To understand Python Modules and their roles in Python programming.
2. To learn how to create a module in Python.
3. To understand the ways of accessing a modules.
4. To understand the Python packages and their creation.
5. To learn about Namespacing and use of Python Packages.

6.2 Module

A module is a file that contains Python code. This code contains functions and variables that perform related tasks. This approach is called “Modularization”. This makes the program easier to understand, test and maintain. Modules also make it much easier to reuse same code in more than one program. If we have written a set of functions that are needed in several different programs, we can place them in modules. Then we can import these modules in each program to call one of the functions.

It allows us to well organize the Python code. The related code segments are grouped together to make a module. It is a kind of Python object having arbitrarily named attributes that we can bind and reference. In simple words, it can define classes, functions, and variables. Also, it can have the runnable code. The major roles of modules in Python specifically have the following roles.

1. Code reuse

Module is a file of the Python code that is saved permanently. Reusing the code through modules is easy as the files can be reloaded and executed as many times we need. Moreover, the modules help to define names, known as attributes, which may be further referenced by multiple other modules.

2. Implementing Shared services or data

With the operational perspective, modules are used for implementing the information shared across a system. Thus, only a single copy of the code is required that can be shared by many people. We may define a global object that can be used by other functions.

3. Partitioning the system namespace

Modules are the high level organizational unit for the Python programs. These are the best suitable tool for grouping the system components. The modules group the names into self-contained packages, which helps avoid name classing. Everything grouped in to a module is always implicitly enclosed. The information cannot be seen in any other file or function unless we import that module. Thus, it helps to partition the namespaces.

6.3 Creating the Modules

In order to create a module, we simply write Python code into a file, and save it with a “.py” extension. These kind of files are accepted as a Python module automatically. A set of names that are assigned at the top level of the module are considered as its attributes and then exported to use. For an instance, we write following function definition into a file named *module1.py*. In this way, we have created a module object with one attribute—the name *printer*, which happens to be a reference to a function object.

```
def printer(x):                # Module attribute
    print(x)
```

The following rules are followed in naming the modules.

- ❖ A module file name should end with .py. If it is not ended with .py we cannot import it into other programs.
- ❖ A module name cannot be a keyword.

- ❖ The module must be saved within the same folder (directory) where you are accessing it.

Example 1: For example, we define the functions such as area of rectangle, circumferences of rectangle, area of circle and circumference of circle in many programs. Instead of doing so, we need to create two modules such as: rectangle and circle. And, write all the related functions in these two modules. Now, we can import these modules into any program as their applicability.

6.4 Module Usage

Python language offers many statements to use the modules created in any program. These statement helps to access the names from any modules and provide the facility to reuse them accordingly. The statements *import* and *from* find, compile, and run a code of module. The major difference between *import* and *from* statements is that *import* fetches the whole module however *from* statement copies specific code or data from that module. Both statements are described here in the form of programs.

6.4.1 The import statement

We can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax.

```
import module1[, module2[,... moduleN]
```

When the interpreter reads import statement, it attempts to fetch that module if available in the search path. The search path is a list of directories that interpreter traces before importing the module.

Example 2: Let us create a module *support.py* and import it in another program.

```
#Creating a Module support.py
def print_func( par ):
    print ("Hello : ", par)
    return
```

Save the above function definition and give a name to this file as *support.py*. Now, the module *support.py* can be imported as follows.

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Python Programming")
```

Output:

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test1.py
Hello : Python Programming
>>>
```

It is important to note that a module is loaded only once regardless of number of times it is imported. If there are multiple statements importing the same module, Python prevents multiple execution of the module. A module is imported only once in a program despite the multiple import statements involving it.

Example 3: Creating a module 'circle.py' and importing it in another Python program.

```
# creating the module circle

#definition of area function
def area(r):
    a=3.14*r*r
    return(a)

#definition of circumference
def circum(r):
    c=2*3.14*r
    return(c)
```

Now, import 'circle.py' in another program.

```
#importing circle in another program
import circle

x=float(input("Enter the Radius:"))

#function call to area
res=circle.area(x)
print ("The area of the Circle is:",res)

#function call to circum
res=circle.circum(x)
print ("The Circumference of the Circle is:",res)
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test1.py
Enter the Radius:4
The area of the Circle is: 50.24
The Circumference of the Circle is: 25.12
>>> |
```

6.4.2 The from....import statement

The *from...import* statement in Python language helps to import specific attributes from a module into the current namespace. The *from...import* has the following syntax –

```
from modname import name1[, name2[, ... nameN]]
```

There are four ways of using import statement to access a module and subsequently use what was imported. We use the math module to illustrate it.

```
# Way 1
import math
print (math.cos(math.pi/2.0))
```

Output:

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test1.py
6.123233995736766e-17
>>> |
```

Here, the statement 'import math' imports everything from the math module. You can access any members of the module using member operator (.).

Output:

```
# Way 2
from math import cos
print (cos(3.14159265/2.0))
```

Here, the statement 'from math import cos' imports only the definition of 'cos' function from the math library. Nothing else is imported.

Output:

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test1.py
1.7948965149208059e-09
>>>
```


Now, See the third way of using the 'from...import' statement. Here, we import only the definitions of 'cos' and 'pi' from the math library. Nothing else is imported.

```
#way 3
from math import cos, pi
print (cos(pi/2.0))
```

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test1.py :
6.123233995736766e-17
>>> |
```

Let us consider another form of 'from...import' statement.

```
#way 4
from math import*
print (cos(pi/2.0))
```

Output: This method also imports everything from the math module. The difference between 'import' and 'from...import' is that member operator (.) is not needed to access module members.

```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test1.py :
6.123233995736766e-17
>>> |
```

Example 4: A program demonstrating the fourth way, the 'circle' module is being imported.

```
from circle import*
x=float(input("Enter the Radius:"))
#function call to area
res=area(x)
print ("The area of the Circle is:",res)
#function call to circum
res=circum(x)
print ("The Circumference of the Circle is:",res)
```

Output:

[133]


```
>>>
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test1.py
Enter the Radius:4
The area of the Circle is: 50.24
The Circumference of the Circle is: 25.12
>>> |
```

6.4.3 Locating the Modules

When a module is imported in a program, the Python interpreter searches it in the current working directory. If it is not available in the current directory, the interpreter then searches it in every directory available in a shell variable known as `PYTHONPATH`. If the interpreter is still not able to find the module in the shell, then it checks the default path. The path `/usr/local/lib/python/` is considered as the default path in UNIX systems. The system module 'sys' represented as `sys.path` variable is used to store the search path for the modules. This variable contains the current working directory i.e. `PYTHONPATH`, and the installation-dependent default variables.

The PYTHONPATH Variable:

The variable `PYTHONPATH` is an environment variable. It consists of a list of directories. The syntax of this variable is similar to the shell variable `PATH`. In windows system the `PYTHONPATH` variable is:

```
set PYTHONPATH = c:\python20\lib;
```

And, in the UNIX system it is as follows:

```
set PYTHONPATH = /usr/local/lib/python
```

6.4.4 Executing the Modules

A given module is located using the option '-m', which is a command line option. Then a module known as `__main__` module of the program is executed. The module `runpy`, a standard module of Python language, is used for this mechanism internally. It allows a script to be located by the namespace of the module instead of the filesystem. This module gives two functions: `run_module()` and `run_path()`.

The function `run_module()` executes the code containing a specific module and returns the result of the module. The argument 'module_name' must be the actual module name. The variables such as `__name__`, `__file__`, `__loader__`, `__name__`, `__cached__`, and `__package__` are the special global variables and sets in the global dictionary prior to module execution.

The `run_path()` function is utilized to execute the programs given in a file stored at a given path. It returns the global dictionary of the module as a

result. The given path may be for a Python source file, valid `sys.path` entry containing the `__main__` module, or a compiled bytecode file.

Example 5: The following program demonstrates the use of *runpy* module. The program file is saved as *runpy_demo.py*. The user can run the following program using the following commands at Python shell and the output would be as follows.

```
#runpy demonstration
def add(p, q, r, s, t):
    return p + q + r + s + t

def main():
    p = 4
    q = 6
    r = 2
    s = 8
    t = 7
    print ("sum of p, q, r, s, t = ")
    print (add(p,q,r,s,t))
    return

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

```
>>> import runpy_demo as runp
>>> runp.main()
sum of p, q, r, s, t =
27
```

Although, the given source code can be executed without importing the file *runpy_demo.py*, the following output is generated in this way.

```
>>> import runpy
>>> runpy.run_module('runpy_demo', run_name='__main__')
sum of p, q, r, s, t =
27
```

Moreover, the `run_path()` function can be used to find the output of the above given script named *runpy_demo.py*.

```
>>> runpy.run_path('runpy_demo.py', run_name='__main__')
sum of p, q, r, s, t =
27
```

In addition to this, the runpy also supports `-m` option at python command line:

```
C:\Users\VIBHU>python -m runpy_demo
sum of p, q, r, s, t =
27
C:\Users\VIBHU>
```

Check your progress

1. What is a module in Python?
2. List the major advantages of a module?
3. How do you create a module?
4. What are the naming rules for modules?
5. What is the utility of *import* statement?
6. Differentiate the usage of *from...import* and *import* statements.
7. What kind of information the PYTHONPATH variable provides?
8. What is the utility of `run_path()` function?

6.5 Namespacing

The namespace is a syntactic container that allows to use the same name in different functions or modules. In more compact form, namespace is considered as a dictionary of the variable names as keys and the corresponding objects as values. Every module has its own namespace. The namespace could be local or global. The Python statements can read variables in both namespaces i.e. local namespace and global namespace. A local variable is considered as a copy of global variable if they both have same names. Each function has its own local namespace and Class methods follow the same rules for scopes as the ordinary function do.

Python interpreter checks whether the variable is local or global. If a variable is assigned inside a function it is local to that function only. In other case, if we want to assign a value to a global variable inside a function then we need to use global statement. The statement `global VarName` indicates that `VarName` is a global variable and it stops the interpreter to search local namespace for this variable. For example, a variable *Money* is needed to be

declared as a global variable. It is done using the global statement otherwise it results in an error named as UnboundLocalError.

```
#Namespacing Demo
#!/usr/bin/python

Money = 2000
def AddMoney():
    # Comment the following line to see the error:
    global Money
    Money = Money + 1

print (Money)
AddMoney()
print (Money)
```

Output:

```
>>>
===== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/money.py
2000
2001
>>>
```

The built-in function `globals()` and `locals()` are used to find the global and local namespace dictionaries. The function `globals()` is used to access a reference to the global namespace dictionary. It provides access to objects in the global namespace. Here is the global dictionary related to above script of namespace demo.

```
>>> type(globals())
<class 'dict'>
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class 'frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotation
s__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'C:/Us
ers/VIBHU/AppData/Local/Programs/Python/Python38/money.py', 'Money': 2001, '
AddMoney': <function AddMoney at 0x00000217E606F5E0>}
>>> |
```

The built-in function `locals()` provides the dictionary related to local namespace. On calling `locals()`, it returns a dictionary that refers to the local namespace of the function. If we call it outside the function in a program then it behaves similar to `globals()`. In simple words, for a variable declared global inside a function using the global statement, it returns the same output. For global variable `Money`, see the output:

```
>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class 'frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotation
s__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'C:/Us
ers/VIBHU/AppData/Local/Programs/Python/Python38/money.py', 'Money': 2001, '
AddMoney': <function AddMoney at 0x00000217E606F5E0>}
>>> |
```

However, it returns the dictionary related to local namespace of a function. For instance:

```
#locals() demo

def locals_demo(x,y):
    s='foo'
    z=x+y
    print('The sum is:',z)
    print('The dictionary for local namespace is:')
    print(locals())

locals_demo(5,3)
```

Output:

```
>>>
== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/money.py
The sum is: 8
The dictionary for local namespace is:
{'x': 5, 'y': 3, 's': 'foo', 'z': 8}
>>> |
```

The *dir()* function can be used to check the sorted list of strings that contains the names related to a module. This function returns a list that contains the names of all the modules, functions, and variables defined in a module. For instance, execute the following script `dir_demo.py` and see the results:

```
# function dir() demo
#!/usr/bin/python

# Import built-in module math
import math

content = dir(math)
print (content)
```


Output:

```
>>>
= RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/dir_demo.py
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
```

6.6 Python Packages

In Python language, a package refers to a hierarchical structure that represent a single Python application environment consisting of modules and a series of subpackages. It means a package is a bundle of multiple modules. Basically, it is a directory that contains the Python files and a special file named `__init__.py`. It indicates that a directory that contains `__init__.py` file is considered to be a Python package.

6.6.1 Creating Python Packages

When we have a large set of modules then it is better to organize them into Python packages. It requires placing the similar files/modules (related with a project) into a single directory. The directory is named after the desired package name. Following is the set of rules to define a package.

- ❖ First, a directory is created and named after the desired package name.
- ❖ Put all the related modules in the created directory.
- ❖ Create an empty `__init__.py` file in this directory.

With the file `__init__.py` the Python interpreter recognize a directory as a Python package.

Example 6: For demonstrating the creation of a package, let us create two python files *a.py* and *b.py*. We these files into a directory that is named as `simple_package`. i.e. the package name would be `simple_package`. Now, create empty Python file and save it with the name `__init__.py` and put it also inside the created directory.

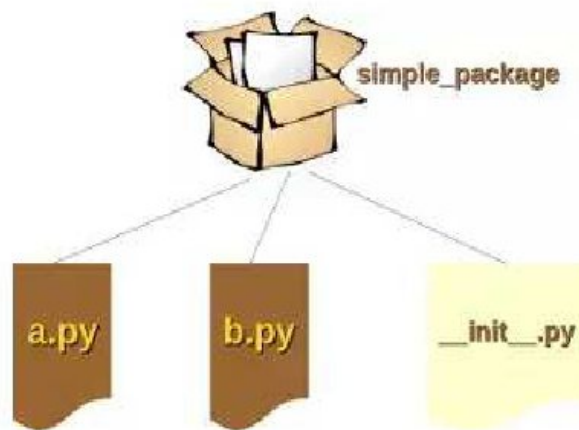
The code of file *a.py*:

```
def file_a():
    print("Hello, function 'file_a' from module 'a' calling")
```

The code of file *b.py*:

```
def file_b():
    print("Hello, function 'file_b' from module 'b' calling")
```

And further create an empty Python file `__init__.py`.



The directory `simple_package` is represented in the following way.

This PC > Local Disk (C:) > Users > VIBHU > AppData > Local > Programs > Python > Python38 > simple_package				
	Name	Date modified	Type	Size
	<code>__init__</code>	7/14/2021 2:20 PM	Python File	0 KB
	<code>a</code>	7/14/2021 2:25 PM	Python File	1 KB
	<code>b</code>	7/14/2021 2:25 PM	Python File	1 KB

6.6.2 Importing the packages

Now, execute following code to import the created package and the generated output will be as follows:

```
>>> import simple_package
>>> from simple_package import a,b
>>> a.bar()
Hello, function 'bar' from module 'a' calling
>>> b.foo()
Hello, function 'foo' from module 'b' calling
>>> |
```

Example 7: Create a file `arith.py` and put it inside the `simple_package` directory (instead of the files `a.py` and `b.py`) along with the `__init__.py` file. Then the directory structure would be as follows.

This PC > Local Disk (C:) > Users > VIBHU > AppData > Local > Programs > Python > Python38 > simple_package

Name	Date modified	Type	Size
 _init_	7/14/2021 2:20 PM	Python File	0 KB
 arith	7/14/2021 2:42 PM	Python File	1 KB

Code of file arith.py:

```
#this module contains all function to perform arithmetic operations

def add(x,y):
    return(x+y)
def sub(x,y):
    return(x-y)
def mul(x,y):
    return(x*y)
def div(x,y):
    return(x/y)
def rem(x,y):
    return(x%y)
```

Now import module arith.py in another file test.py. The code inside the file test.py is given as follows.

```
#Demonstrating package import

from simple_package.arith import *

#reading the data from keyboard
a=int(input("Enter a value:"))
b=int(input("Enter a value:"))
#function calls
print("The sum is :",add(a,b))
print("The subtraction is :",sub(a,b))
print("The product is:",mul(a,b))
print("The quotient is:",div(a,b))
print("The remainder is:",rem(a,b))
```

Output:

```
== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
Enter a value:5
Enter a value:6
The sum is : 11
The subtraction is : -1
The product is: 30
The quotient is: 0.8333333333333334
The remainder is: 5
>>>
```


6.7 Introduction to PIP

A Python package is considered as a group of Python files that offers particular features for programming. Python PIP manages all the packages and acts as a package manager in Python language. It is preinstalled in Python language toolbox and the pip command is used at command prompt. Sometimes we want to install the packages that are created by third parties. The Python package index offers a repository of software related to Python programming. We can install a package with the help of the pip command. The pip command assists the user to install and manage packages. The following command can be used to check an installed package and its details.

```
pip show numpy
```

Result:

```
C:\Users\VIBHU>pip show numpy
Name: numpy
Version: 1.18.5
Summary: NumPy is the fundamental package for array computing with Python.
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email: None
License: BSD
Location: c:\users\vibhu\appdata\local\programs\python\python38\lib\site-packages
Requires:
Required-by: tifffile, tensorflow, tensorboard, scipy, scikit-learn, scikit-image, PyWavelets, pandas, opt-einsum, openv-python, matplotlib, Keras, Keras-Preprocessing, imageio, h5py
```

Check pip version: The following command is used to check pip version.

```
pip --version
```

```
C:\Users\VIBHU>pip --version
pip 20.2.4 from c:\users\vibhu\appdata\local\programs\python\python38\lib\site-packages\pip (python 3.8)
```

Searching a package:

Using the following command, we can search for the existing package.

```
pip search numpy
```

```
Command Prompt
C:\>pip search numpy
numpy (1.18.1)
  INSTALLED: 1.18.1 (latest)
numpy-alignments (0.0.2)
  - NumPy Alignments
  - NumPy utilities.
  - NumPy in the cloud
numpy-utis (0.1.6)
  - Turtle graphics with NumPy
numpy-cloud (0.0.5)
  - Missing NumPy functionalities
numpy-turtle (0.2)
  - The interface between ROOT and NumPy
numpy-sugar (1.5.0)
  - Numpy data serialization using msgpack
root-numpy (4.8.0)
  - Fast SQLite to numpy array loader
msgpack-numpy (0.4.4.3)
  - Add built-in support for quaternions to numpy
sqlite-numpy (0.2.1)
  - SQL PARTITION BY and window functions for NumPy
numpy-quaternion (2019.12.11.22.25.52)
  - Mapchete NumPy read/write extension
numpy-partition (1.18.9)
  - NumPy is the fundamental package for array computing with Python.
mapchete-numpy (0.1)
  - posit (unum type III) integrated NumPy.
numpy-mips64 (1.17.4)
  - NumPy is the fundamental package for array computing with Python.
numpy-aarch64 (1.16.4)
  - NumPy optimized with Intel(R) MKL library
intel-numpy (1.15.1)
  - A package providing an unit system for numpy multidimensional arrays.
numpy-unit (0.1.1)
  - NumPy: array processing for numbers, strings, records, and objects.
numpy-mkl (1.10.2)
  - Manage a Numpy data structure using Peewee-Sync
numpy-syncer (0.0.1)
  - Wrapper module for ccv using numpy arrays interface
ccv-numpy (0.0.2)
  - A package which contains a simple implementation of neural network with numpy
nn-numpy (0.0.1)
```

List of installed Packages: The following command is used to display the list of packages that are installed in the system.

```
pip list
```

```
C:\Users\VIBHU>pip list
Package            Version
-----
absl-py            0.11.0
astunparse         1.6.3
cachetools         4.1.1
certifi            2020.11.8
chardet            3.0.4
cyclер            0.10.0
decorator          4.4.2
gast               0.3.3
google-auth        1.23.0
google-auth-oauthlib 0.4.2
google-pasta       0.2.0
grpcio            1.33.2
h5py              2.10.0
idna               2.10
imageio            2.9.0
joblib             0.17.0
Keras              2.4.3
Keras-Preprocessing 1.1.2
kiwisolver         1.3.1
Markdown           3.3.3
matplotlib         3.3.3
networkx           2.5
numpy              1.18.5
oauthlib           3.1.0
opencv-python     4.4.0.46
opt-einsum         3.3.0
pandas             1.1.4
Pillow            8.0.1
pip               20.2.4
protobuf          3.14.0
pyasn1            0.4.8
pyasn1-modules    0.2.8
pyparsing          2.4.7
python-dateutil    2.8.1
python-igraph      0.8.2
pytz              2020.4
PyWavelets        1.1.1
PyYAML            5.3.1
requests          2.25.0
requests-oauthlib  1.3.0
rsa               4.6
```

6.8 Installing Packages using PIP

We can install the required package using the following pip command:

```
<pip install numpy
```


If the required package is not installed, the pip command initiated to install that after downloading that from the repository. If the package is already installed, it is again installed and overwriting the previous copy of the package.

```
Command Prompt
C:\>pip install numpy
Collecting numpy
  Downloading https://files.pythonhosted.org/packages/ab/38/f6d6d86350496db4ed5d8ca4b9f193d8edc39999c3a63779cbc38a659f/numpy-1.18.1-cp37-cp37m-win_amd64.whl (11.8MB)
    12.8MB 32kB/s
Installing collected packages: numpy
Successfully installed numpy-1.18.1
```

A package can be uninstalled using the following command if needed.

```
pip uninstall numpy
```

```
Command Prompt
C:\>pip uninstall numpy
Uninstalling numpy-1.16.5:
  Would remove:
    d:\software\anaconda\lib\site-packages\numpy
    d:\software\anaconda\lib\site-packages\numpy-1.16.5-py3.7.egg-info
    d:\software\anaconda\scripts\f2py-script.py
    d:\software\anaconda\scripts\f2py.exe
Proceed (y/n)? y
Successfully uninstalled numpy-1.16.5
```

Check your progress

1. Explain the concept of the namespace.
2. How is the namespace related to module?
3. Differentiate local and global namespace.
4. What are the functions `locals()` and `globals()` used for?
5. What are the usage of `dir()` function?
6. What do you understand by a package in Python?
7. Differentiate module and package?
8. How do you create a package in Python?
9. How do you install a Python package?
10. How do you uninstall a Python package?

6.9 Usage of Python packages

NumPy, SciPy, Pandas, Matplotlib, Pillow or PIL, OpenCV, Keras, TensorFlow, Theano, NLTK, PyTorch, and ScikitLearn, etc. are some popular Python packages, which are used for different purposes. Some of these packages are described here.

NumPy package:

NumPy is a Python package that is used to deal with arrays. It stands for Numerical Python. We can install it using the command “pip install numpy”. An example of the simple usage of the NumPy package is given here.

```
>>> import numpy as np
>>> arr=np.array([1,2,3,4,5])
>>> print(arr)
[1 2 3 4 5]
>>> |
```

In the above example, an array of integer elements is created and displayed. Let us consider another example that deals with the arrays.

```
# use of NumPy package
import numpy #numpy package

l=[12.34,45.56,68.78,98.78]

a=numpy.array(l) #numpy array() function
print("Temp in 'C' are:",a)
f=(a*9/5+32)
print("The temp in 'F' are:",f)
```

Output:

```
>>>
== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py ==
Temp in 'C' are: [12.34 45.56 68.78 98.78]
The temp in 'F' are: [ 54.212 114.008 155.804 209.804]
>>> |
```

In the following example, the random module from NumPy package is used to create an array with random numbers.

```
>> from numpy import random
>> x=random.randint(100, size=(3,5)) # generate a 2-d array
>> print(x)
[80 39 12 70 74]
[65 46 10 86 5]
[16 13 49 51 31]]
>> |
```

SciPy package:

SciPy stands for Scientific Python and it is used for scientific computation. SciPy package uses the services of NumPy package underneath. It can be installed using the “pip install scipy” command. Let us consider the following example that uses a constant ‘pi’ from the scipy package.

```
>>> from scipy import constants
>>> print(constants.pi)
3.141592653589793
>>>
```

Pandas Package:

Pandas is a Python package that offers facilities to work on datasets. It provides various functionalities for cleaning, manipulating, and exploring the data used for analysis. It refers to both “Panel Data”, and “Python Data Analysis”. This package allows us analyze the big data based on the statistical theories. Data cleaning is the most relevant facility offered by the Pandas that makes the datasets more relevant and useful. Python command “pip install pandas” can be used to install Pandas package. A Pandas Series is like a column in a table. It is a one-dimensional array. Whereas, Pandas Dataframe is a 2-D structure like a 2-D array. Let us consider the following examples that demonstrate the usage of Series and Dataframe.

```
# Pandas Series demonstration

import pandas as pd
a=[1,7,2] #labels by default starts with 0.

myvar=pd.Series(a)
print(myvar)

#create own label using index arguments.
myvar=pd.Series(a,index=["x","y","z"])
print(myvar)
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
0    1
1    7
2    2
dtype: int64
x    1
y    7
z    2
dtype: int64
>>>
```



```
# Pandas Dataframe demonstration
# refer to the row index

import pandas as pd
data={"calories": [420, 380, 390],
      "duration": [50, 40, 45]
      }
#load data into Dataframe object.
df=pd.DataFrame(data)
print(df)
print(df.loc[0])
# use alist of indexes
print(df.loc[[0,1]])
#add a list of names to give each row a name:
df=pd.DataFrame(data, index=["Day1", "Day2", "Day3"])
print(df)
```

Output:

```
==== RESTART: C:/Users/VIBHU/AppData/Local/Programs/Python/Python38/test.py
  calories  duration
0       420        50
1       380        40
2       390        45
calories    420
duration     50
Name: 0, dtype: int64
  calories  duration
0       420        50
1       380        40
Day1    420        50
Day2    380        40
Day3    390        45
>>>
```

Matplotlib Package:

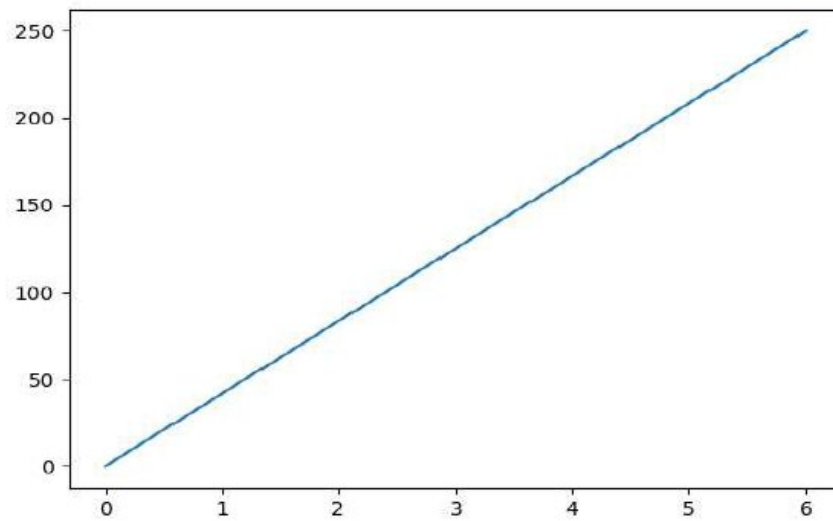
The Matplotlib package is a low level package that is used to plot graphs and serves as a visualization utility. It is open source and can be used without any cost. Python command “pip install matplotlib” can be used to install this package. Let us consider the following examples, where some graphs are plotted with the help of the utilities of the matplotlib packages.

```
#Matplotlib Package demonstration

import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
#plot a line from position (0,0) to position(6,250)
plt.plot(xpoints, ypoints)
plt.show()
```

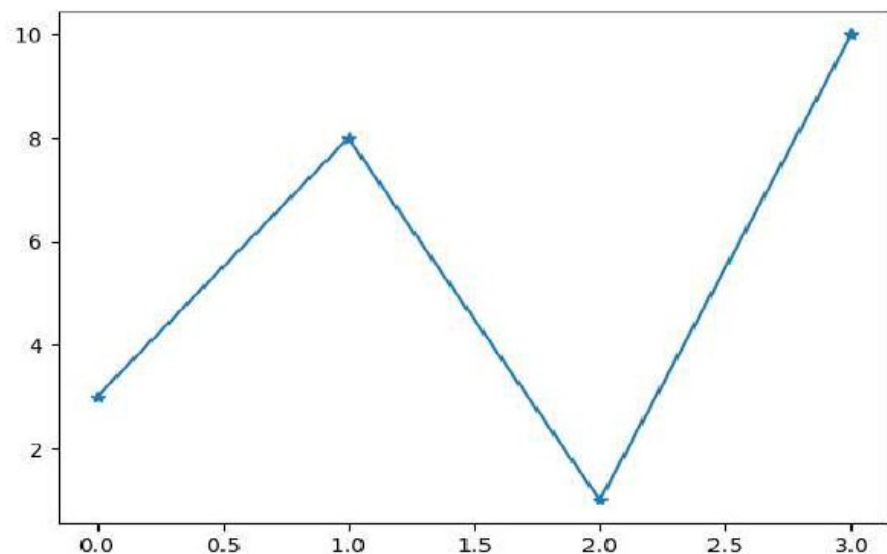
Output:



Plot with Marker:

```
#plot with marker
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, marker = 'x')
plt.show()
```

Output:



6.10 Summary

Modules are important to organize the Python programs. The related programs or program segments are grouped together to form a modules. The Python code is placed in a file that is saved with extension '.py' to make a module. The modules facilitate the code reusability. The code available with a module can be used by other programmers also. The 'import' and 'from...import' statements allow the programmers to copy the exiting code in a module into their own programs. The 'import' statement loads whole of the module whereas the statement 'from' is used to access only the required objects/code from any module.

Every module has its own namespace. A namespace is a syntactic container that allows to use the same name in different functions or modules. The namespace is considered as a dictionary of the variable names as keys and the corresponding objects as values. It could be a local or global namespace. The functions locals() and globals() are used to access the dictionaries related to local and global namespaces respectively.

A package refers to a hierarchical structure that represent a single Python application environment consisting of modules and a series of subpackages. Python packages are the directories where Python files are stored. Each package contains a special file named `__init__.py`. All the related modules and files are placed in a directory, which is named after the desired package. The name of the directory represents the name of the Python package. PIP is the manager tool for Python packages. The third party packages can be installed with the help of PIP. There are many Python packages such as Pandas, NumPy, SciPy, Keras, TensorFlow, and Matplotlib, etc. These Python packages were developed with special functionalities that help the programmer to easily write the complicated programs.

6.11 Review Questions

- Q.1 How do you create a module in Python? Explain with the help of suitable programs.
- Q.2 Explain the usage of *import* and *from* statements.
- Q.3 What is the purpose of `__init__.py` file in module package directory?
- Q.4 How can you avoid repeating the full package path every time you reference a package's content?
- Q.5 Explain the role of `dir()` function related with Python modules.
- Q.6 Create a module for Fibonacci series, and import it in a python program to print the series.
- Q.7 Write a Python program to calculate simple interest using the concept of module and packages.
- Q.8 Demonstrate the usage of Pandas and Matplotlib packages in a single program.
- Q.9 Write a Python program to demonstrate NumPy, Pandas, and SciPy packages in a single program.



Uttar Pradesh Rajarshi Tondon
open University

SBSCS-02

Python Programming

Block

3

OOPS IN PYTHON

Unit 7	
Object Oriented Programming in Python	154
Unit 8	
Exception Handling	184
Unit 9	
Python Libraries	203
Unit 10	
GUI Programming and Testing	232

Course Design Committee

Prof. Ashutosh Gupta Director (In-charge) School of Computer and Information Science, UPRTOU Prayagraj	Chairman
Prof. R. S. Yadav Department of Computer Science and Engineering MNNIT Prayagraj	Member
Dr. Marisha Assistant Professor (Computer Science), School of Science, UPRTOU Prayagraj	Member
Mr. Manoj Kumar Balwant Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Member

Course Preparation Committee

Dr. Brajesh Kumar Associate Prof. Dept. of CSIT, MJPRU, Bareilly, 243006	Author Block – 01 to 04 Unit – 01 to 12
Prof. Manu Pratap singh Prof. Department of Computer Science Dr. Bhimrao Ambedkar University, Agra - 282002	Editor Block – 01 to 04 Unit – 01 to 12
Dr. Chandrakant Kumar Singh Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Coordinator

© UPRTOU , Prayagraj

ISBN :

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar pradesh Rajarshi Tandon Open University.

Overview:

This block is about the object oriented programming (OOP) constructs in Python programming language. The major OOP concepts supported by Python are discussed in different units of this block. This block is divided into 4 units from Unit-7 to Unit-10. The Unit-7 deals with the basic concepts of OOP supported by Python. Later on some advanced concepts are covered in the subsequent units. The Unit-8 deals with the exceptions and exception handling. Different methods of exception handling available in Python are explained with the help of suitable programs. The strength of Python lies in its vast libraries. There are thousands of libraries that support many different technologies and application development in Python. These libraries provide a lot ready-made programs that make the job of a programmer easy. Therefore, it is important to have knowledge of the important libraries. The Unit-9 provides an insight into the Python standard library. Graphical user interface (GUI) is another important concept that is highly desirable to make the software user friendly. Unit-10 deals with the GUI development in Python.

UNIT 7: Object-Oriented Programming in Python

Structure:

- 7.0 Introduction
- 7.1 Objectives
- 7.2 Classes
- 7.3 Operations for Classes
- 7.4 Method and Self Argument
- 7.5 The `__init__()` Method (Constructor Method)
- 7.6 Class variables and object variables
- 7.7 The `__del__()` Method
- 7.8 Data Hiding and Access specifiers
- 7.9 Private Methods
- 7.10 Class Methods
- 7.11 Static Method
- 7.12 Inheritance
- 7.13 Types of Inheritance
- 7.14 Abstract Classes
- 7.15 Operator Overriding
- 7.16 Operator Overloading Operator
- 7.17 Summary
- Review Questions

Unit 7: Object-Oriented Programming in Python

7.0 Introduction

In this chapter, major object-oriented programming features supported by Python are discussed. Object-oriented programming is an effective approach of writing computer programs. It is a programming paradigm that provides a tools for structuring the computer programs so that properties and behaviours real world objects are combined into individual objects. For example, an object can represent a citizen with properties such as name, age, weight, and address with his behaviours such as walking, talking, smiling, and breathing. Or it could represent an email with properties such as a recipient, subject, and message body with its behaviours such as attachments and sending.

Object-oriented programming provides a way for modelling real-world things such as student, as well as its relations between other things like teachers and college, etc. Object-oriented programming allows to model real-world entities as software objects. These objects are associated with some data and they can perform certain functions.

In object-oriented programming, the *classes* represents the real-world things and situations. The instances of these classes are known as objects. When a class is written, it defines the general behaviour of the whole category of objects. Python is not only a popular scripting language but it is a great object-oriented programming language also. It has good support for the object-oriented programming paradigm. In Python, almost everything is represented as an object with its attributes and methods.

Object-Oriented programming focuses designing and developing applications using objects and classes. The major building blocks of object-oriented programming paradigm are abstraction, encapsulation, inheritance, and polymorphism as shown in Figure 7.1.

Abstraction allows to hide the necessary details of a class not important for the outside world and only the essential features are exposed. A car is a good example of abstraction, where details many parts like engine, gear box, battery, and many other necessary mechanical/electrical components are hidden to the user. The driver knows that by pushing a button, the engine gets start but he need to aware the underlying technology that is actually responsible for starting the engine with a push of button. In the same way necessary data and methods in a program hide inside the class or object.

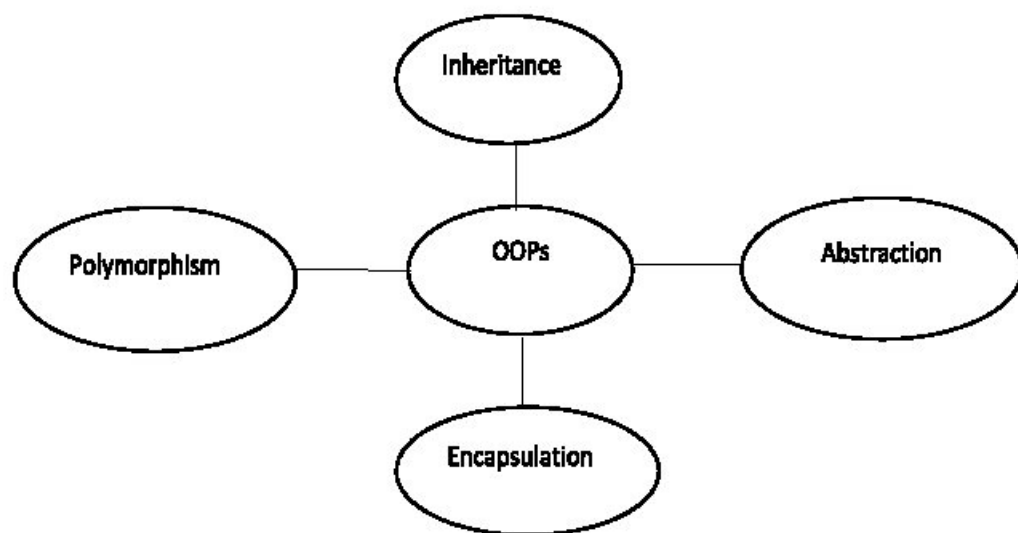


Figure 7.1. Four pillars of OOPs

Encapsulation refers to wrapping of data and related methods in a single unit. Both data and methods are wrapped simultaneously in a class. It helps to hide the values or state of data elements to prevent the unauthorized access. It can be achieved by making the data members as private.

Inheritance is a technique that allows to establish a hierarchal relationship between classes and objects. It is also known as generalization that allows to reuse the existing program that could be enhanced further. For example, 'Citizen' is a class that is generalization of another class 'Employee'. An employee has all the characteristics of a Citizen with some additional properties. Therefore, instead of writing a class for 'Employee' from scratch, it can be built upon the existing class 'Citizen'. The main advantage of inheritance is the ability to reuse the code.

Polymorphism allows the methods and operators in a program to take many forms. A function or method in a program can have different forms and meanings with the same name. This types of methods and operators have different meanings in different context. Overloading and overriding are the examples of different forms of polymorphism.

7.1 Objectives

The objectives of this unit are:

1. To discuss the features of Object-oriented programming.
2. To define classes, objects, class methods and self-argument.
3. To understand constructor method.
4. To discuss the concept of data hiding.
5. To discuss inheritance, its types, overriding methods.
6. To understand the concept of abstraction and operator overloading

7.2 Classes

A class acts as a template that defines the basic characteristics of a particular object. Classes are the way to implement abstract data types in python. A class in python is a collection of data and functions. These functions operate on the data of the class. More precisely, data are the properties of the real-world object. Functions are the actions that operate on these attributes.

In python, class itself is an object of type *type*. As functions are equivalent to data in python and are first class object, classes are also first class object and the class itself is an object. In languages like java, the class definition does not create an object rather, it needs to be instantiated. But in python, the class definition itself creates an object. Even class methods are objects that have *type* called *method_descriptor*.

7.2.1 Creating Classes

Python has a very simple way of creating class. To create class keyword *class* is used. General syntax of creating class is given as follows.

```
class class_name:
    <statement-1>
    <statement-2>
    .
    .
    .
    <statement-n>
```

For example, a class named Citizen with a property, age=35 can be created as follows:

```
class Citizen:
    name = "kartik"
    age = 35
```

Variables defined in a class are called *class variables* and functions defined inside a class are called *class methods* or simply *methods*. Class variables and class methods are together known as *class members*. The class members can be accessed through class objects. Class methods have access to all the data contained in the instance of the object.

7.3 Operations for classes

There are two types of major operations that can be performed on classes:

- Instantiation
- Referencing

7.3.1 Class Instantiation

This operation creates instances of class. Creating an object or instance of a class is known as class instantiation. Both instance and object are synonym of each other, they are used interchangeably. The syntax to create an object is given as follows.

```
Object_name = class_name( )
```

Example:

```
ob1 = Citizen( )
```

The above statement creates the object *ob1* of the class *Citizen*.

7.3.2 Referencing

The second possible operation for classes is *referencing*. Referencing means accessing data and methods belonging to a class. Referencing is done using dot (.) operator. Once the object is created, it can then access class variables and class methods using dot (.) operator.

Example:

```
print(ob1.name)  
print(ob1.age)
```

It prints the value of attributes name and age belonging to object ob1 i.e. kartik and 35. Complete program that access class variable using class object is shown as follows.

```
class Citizen:
```

```
    name = "kartik"
```

```
    age = 35
```

```
ob1 = Citizen()
```



```
print(ob1.name)
print(ob1.age)
```

Output:

```
Kartik
35
```

Check your progress

1. What is Object Oriented Programming?
2. Write the basic properties of OOPS?
3. What is a class?
4. How can you define class?
5. What is an Object?
6. What is the main difference between a class and an object?
7. Define class variables?
8. Explain class instantiation? How it is done?
9. Define Referencing operation on class.
10. What are methods?

7.4 Method and Self Argument

A member function of a class in Python is known as **instance method** or **simply method**. An instance method does not require a decorator to call it. An instance of the class is used in order to call it. A special parameter **self** is needed in the definition of the instance methods. The **self** must be placed before other parameters in the method. The parameter **self** is automatically passed when an instance method is called. If an instance method does not consist any parameters or arguments, still a **self** parameter is required to give the individual instances access to the data and methods in the class. It is also possible to give any other name to **self** parameter, but conventionally it is recommended to use the word 'self'. A program that illustrates how to access class members using the class object is given as follows.

```
class Citizen:
    city="mumbai"
    def set_attributes(self,name,age):
        self.name=name
        self.age=age
```

```

def display(self):
    print(self.name)
    print(self.age)
ob1 = Citizen()
ob1.set_attributes("ravi",32)
ob1.display()
print(ob1.city)
ob1.city="delhi"      # this statement modifies the value of attribute city.
print(ob1.city)
print(ob1.name)
print(ob1.age)

```

Output:

```

ravi
32
mumbai
delhi
ravi
32

```

In above program, a class `Citizen` is created with three object variables named `city`, `name` and `age`. Further, two methods `set_attributes` and `display` are defined, which are used to set and display the values of object variables `name` and `age`. Variable `city` is initialized to value `mumbai`. All three variables can be accessed through the object of class `Citizen`. The attributes `name` and `age` can be accessed directly through the object as well as through the methods `set_attributes` and `display` as shown in the program. The above example illustrates how class members can be accessed using class object.

7.5 The `__init__()` Method (Constructor Method)

The `__init__()` method is a special method that python runs automatically whenever a new instance of a class is created. It is a constructor method however unlike other languages like `c++`, `java` which allows multiple constructors python allows only one constructor per class. This method has two leading underscores and two trailing underscores which is a convention that help prevent python's default method names from conflicting user defined our method names. The syntax to declare the `__init__()` method is given as follows.

```

def __init__(self,[args...])

```

The first parameter is always a *self* followed by zero or more parameters.
The following program shows the use of `__init__()` method.

```
class Citizen:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def update_attributes(self,name,age):
        self.name=name
        self.age=age
    def display(self):
        print(self.name)
        print(self.age)
obl = Citizen("Rohit", 25)
obl.display()
obl.update_attributes("Ramanujan", 28)
obl.display()
```

Output:

```
Rohit
25
Ramanujan
28
```

In above program, the `__init__()` accepts two arguments *name* and *age*. Like any other class first argument has to be *self*. Inside the definition of `__init__()` method, there are two arguments *self.name* and *self.age*, which have exactly the same names as specified in the parameter list of `__init__()` method. Although the two variables have same name but they are different variables. The *self.name* and *self.age* belong to the instance variable, while the parameters *name* and *age* in the `__init__()` method are used only to receive the values and assign those values to the instance variables.

7.6 Class variables and object variables

In object-oriented programming the variables can be used both at class level and instance level. The variables that are defined inside the Class definition but not inside any instance method are called *Class variables*. The class variables are not tied to a particular object but shared across all the objects of that class. If a modification is made to a class variable by any object, it is reflected to all the objects instance at the same time. On the hand the *Instance variables* tied to a particular object only. The changes made to instance variables by a particular

object are not reflected to the other objects of the class. The instance variables are declared inside the `__init__` method, which is the constructor of the class. If a class variable and an instance variable in a class have same name then instance variable overrides the class variable. It may lead to bugs or undesirable behaviour of the program. Therefore, it is should be avoided to have class variables and instance variables with the same names or they must be carefully handled. The following program shows the use of class variables and object variables.

```
class Citizen:
    count=0          # count is a class variable which is shared among all
                    # object variables
    def __init__(self, name, age):
        Citizen.count+=1          # class variable count is accessed
        using classname
        self.name=name           # self.name is an object variable
        self.age=age             # self.age is an object variable
    def update_attributes(self, name, age):
        self.name=name
        self.age=age
    def display(self):
        print(self.name)
        print(self.age)
ob1 = Citizen("Rohit",25)
print(f"Number of objects created:{ob1.count}")
ob2= Citizen ("Ravi",28)
print(f"Number of objects created:{ob1.count}")
print(f"Number of objects created:{ob2.count}")
ob3= Citizen("Ramesh",32)
print(f"Number of objects created:{ob1.count}")
print(f"Number of objects created:{ob2.count}")
print(f"Number of objects created:{ob3.count}")
```

Output:

Number of objects created:1

Number of objects created:2


```
Number of objects created:2
Number of objects created:3
Number of objects created:3
Number of objects created:3
```

In above program, there is a class variable count, which is shared by all three objects of the class Citizen. It is initialized to zero and incremented by 1 each time an object variable is created. Any changes made to class variable count by one object is reflected to all other object also.

7.7The __del__() Method

The __del__() method is a special method of a class. It is also called the destructor method and it is called when the instance (object) of the class is about to get destroyed. The __del__() method is called automatically when an object goes out of scope. The same can be done using the keyword 'del'. The following program illustrates the use of __del__() method

```
class Citizen:

    def __init__(self,name,age):
        self.name=name
        self.age=age
        print(f'object with name {self.name} created')

    def display(self):
        print(self.name)
        print(self.age)

    def __del__(self):
        print(f'object with name {self.name} destroyed')

ob1 = Citizen ("Rohit",25)
ob2 = Citizen("Ravi",32)

del ob1
```

Output:

```
object with name Rohit created
object with name Ravi created
object with name Rohit destroyed
object with name Ravi destroyed
```


In above program obj1 is deleted explicitly by using the 'del' keyword and obj2 gets destroyed automatically. Hence it shows that any object get deleted or destroyed automatically when it goes out of scope or it can be deleted explicitly by using the 'del' keyword.

7.8 Data Hiding and Access Specifiers

Data hiding is the process that ensures exclusive data access to class members and provides object integrity by preventing unintended or intended changes. Data hiding insulates the data from the straight access by the program.

Data hiding, also called encapsulation, organizes the data and methods into a structure that prevents data access by any function defined outside the class. Encapsulation or data hiding defines different levels for data variable and member function (or method) of the class. These access levels specify the access rights. There are three types of access specifiers:

- Public Access specifiers
- Protected Access specifiers
- Private Access specifiers

The members of a class that are declared *public* are easily accessible from any part of the program. All data members and member functions of a class are public by default. The members of a class that are declared *protected* are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class. The members of a class that are declared *private* are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class. The following program demonstrates the use of public and private variables

```
class Employee:

    def __init__(self, name, age):
        self.name=name           #variable self.name is made public
        self.__age=age           #variable self.__age is made private

    def display(self):
        print(self.name)
        print(self.__age)

obj = Employee("Rohit",25)

obj.display()

print(obj.name)
```

```
print(obj.__age)
```

Output:

Rohit

25

Rohit

Traceback (most recent call last):

File "C:\Python39\access_specifier.py", line 24, in <module> print(obj.__age)

AttributeError: 'Employee' object has no attribute '__age'

In above program the class Employee has two variables self.name and self.__age. The variable self.name is made public while self.__age is made private. Hence name attribute can be accessed by object directly or through the class method. But the private variable can be accessed through the class method only. Hence it produces error when it is tried to print private variable __age directly using object obj. Private members of class can be accessed from outside the class using following syntax:

```
objectname._classname__privatevariable
```

So, to remove the error from above code the last print statement should be replaced by following statement:

```
print(obj._Employee__age)
```

Protected access specifiers are used in inheritance.

7.9 Private Methods

Python class can also have private methods, which can be identified by a double underscore '__' symbol before them. Like private attributes, private methods should not be used from outside the class. A private method can be accessed using the object name as well as the class name from outside the class. However, if it is very necessary to access them from outside the class, then they can be access using following syntax:

```
objectname._classname__privatemethodname
```

Program to illustrate the use of private method.

```
class Employee:
```

```

def __init__(self, name, age):
    self.name=name          #variable self.name is made public
    self.__age=age          #variable self.__age is made private

def __display(self):
    print(self.name)
    print(self.__age)

def display(self):
    print("display from inside the class")
    self.__display()

obj = Employee("Rohit",25)

obj.display()

print("display from outside the class")
obj._Employee__display()    # private method __display is
                             # called from outside
                             # the class

```

Output:

```

display from inside the class
Rohit
25
display from outside the class
Rohit
25

```

The protected methods in a class is declared by adding a symbol '_' before the method name. Similarly protected data members are declared by adding a symbol '_' before the data variable name.

7.10 Class Methods

The class methods are special methods that are bound to the class itself and not to the class objects. These methods are called using the class name but can also be called using class objects. The class methods have access to the class state and can therefore modify the class state. These changes are reflected across all the objects of the class. However, it cannot make modifications to the object state as it does not have access to `self` parameter. The first argument of a class method is `cls` instead of `self`. The class methods can be used to create objects using the same or different parameters used with the class constructor.

A classmethod decorator `@classmethod` is used to declare the class method. The decorator `@classmethod` is a built-in function of the class. The class method can be called using `Classname.MethodName()`. The decorator `@classmethod` is an alternative to `classmethod()`. The following program illustrates the use of class methods.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    # a class method to create an Employee object
    @classmethod
    def salary(cls, name, salary):
        return cls(name, salary)

obj1 = Employee('Rohit', 85000)
obj2 = Employee.salary('Ram', 97000)

print(obj1.salary)
print(obj2.salary)
```

Output:

```
85000
97000
```

In above program, method `salary(cls, name, salary)` is a class method, which receives two arguments 'name' and 'salary' explicitly and one argument 'cls' implicitly. `Obj2` is created by using the class method.

7.11 Static Method

A static method is another method that is not bound to class objects similar to a class method. But unlike a class method, a static method does not receive implicit first argument. It is associated neither with a `cls` nor with a `self` argument. It is free to accept any number of arguments. Similar to a class method, a static method is also bound to class name instead of class objects. However, it can not access or modify the object state or class state. A static method is marked by `@staticmethod` decorator. These are primarily used for namespace in the programs. A program to demonstrate the use of static method is given as follows.

```
class Citizen:
    def __init__(self, name, age):
        self.name = name
```

```

        self.age=age
    @staticmethod
    def isVoter(age):
        return age > 18
ob1 = Citizen("Rohit",25)
ob2 = Citizen("Ravi",12)
print (Citizen.isVoter(ob1.age))
print (Citizen.isVoter(ob2.age))

```

Output:

```

True
False

```

In above program, isVoter(age) is a static method which is defined to check whether the Citizen is adult or not using value of parameter age.

Check your progress

1. What is self parameter in python?
2. Explain the use of __init__() method?
3. Differentiate between class variable and instance variable.
4. What is the use of __del__() method?
5. What do you mean by data hiding?
6. What are access specifiers?
7. How variables are made private in python class?
8. How private members are access from outside the class in which they declared?
9. What are class methods?
10. How class methods are different than static methods?

7.12 Inheritance

Inheritance is the concept of reusability that allows to reuse the existing code for creating another code with some additional features. A programmer can first create a general class also known as a *base class* and later on, it can be extended to a more specialized class when needed. The specialized class that extends the existing base class is known as *derived class*. Class being inherited

by another class is called base class or parent class. The derived class inherits the properties of the base class. Therefore, a base class is also called parent class and the derived class is called child class. It allows programmer to write extended or better program code. The idea of inheritance in Python is similar to the concept of inheritance in other languages like C++ and Java. Syntax to create child class is given as follows.

```
Class childclass(baseclass):  
    body of child class
```

Syntax to call base class method from child class is given as follows.

```
Class childclass(baseclass):  
    Parentclass.method_of_parentclass(self,[arg1,arg2,...])
```

From above statement, it can be seen that self is passed as a first argument. It is necessary to pass self as a first argument when method of parent class is called from child class. Program to illustrate the use of Inheritance

```
class Citizen:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def getname(self):  
        return self.name  
    def getage(self):  
        return self.age  
  
class Student(Citizen):  
    def __init__(self, name, age, roll_no):  
        Citizen.__init__(self, name, age)  
        self.roll_no=roll_no  
  
    def getrollno(self):  
        return self.roll_no  
  
p1 = Citizen ("Rohit", 25)  
p2 = Student("Ramesh",15,2)  
  
print (f"Citizen name: {p1.getname()}")  
print (f"Citizen age: {p1.getage()}")  
print (f"Student name: {p2.getname()}")  
print (f"Student age: {p2.getage()}")  
print (f"Student roll no: {p2.getrollno()}")
```

Output:

Citizen name: Rohit
Citizen age: 25
Student name: Ramesh
Student age: 15
Student roll no: 2

In above program, class Student is inherited from class Citizen. Therefore, the child class Student have all the properties (attributes and methods) of the base class Citizen. The base class Citizen have two attribute (self.name and self.age) and two methods (getname() and getage()). These attributes and methods defined in parent class now belongs to Student class also as Student class is inherited from Citizen class. Further Student class have one unique attribute roll_no. When object of Student class is created it automatically calls the `__init__()` method that belongs to it. The `__init__()` method of Student class calls the `__init__()` method of base class, Citizen, to set the values of derived attributes self.name and self.age. From the code it can be seen that object of class Student (p2) can access both the methods getname() and getage() defined in class Citizen.

The `super()` function is used to give access to methods and properties of a parent or sibling class. In an inherited subclass, a parent class can be referred to with the use of the `super()` function. The `super` function returns a temporary object of the superclass that allows access to all of its methods to its child class. When a method of a parent class is called by child class using `super()` function than self argument is not required in calling statement. Syntax to call base class method from child class is given as follows.

Class childclass(baseclass):

Super().method_of_parentclass([arg1,arg2,...])

From above statement, it can be observed that self is not passed. It is not required to pass self argument when method of parent class is called from child class. Program to illustrate the use of `super()` function.

```
class Citizen:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def getname(self):
        return self.name
    def getage(self):
```

```

        return self.age
class Student(Citizen):
    def __init__(self, name, age, roll_no):
        super().__init__(self, name, age)
        self.roll_no=roll_no
    def getrollno(self):
        return self.roll_no
p1 = Citizen("Rohit", 25)
p2 = Student("Ramesh", 15, 2)
print (f"Citizen name: {p1.getname()}")
print (f"Citizen age: {p1.getage()}")
print (f"Citizen name: {p2.getname()}")
print (f"Student age: {p2.getage()}")
print (f"Student roll no: {p2.getrollno()}")

```

Output:

Citizen name: Rohit

Citizen age: 25

Student name: Ramesh

Student age: 15

Student roll no: 2

In above program, the `__init__()` method of parent class (class Citizen) is called from child class (class Student) using `super()` function.

7.13 Types of Inheritance

In Python, there are three types of Inheritance:

- Multiple inheritance
- Multilevel Inheritance
- Multipath Inheritance

7.13.1 Multiple Inheritance

In multiple inheritance, derived class inherits the features from more than one class. The derived class have features of all base classes. Figure 7.2 shows that class C inherits the properties of class A and class B.

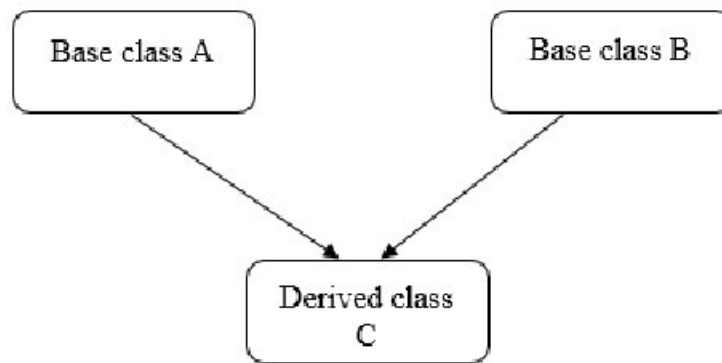


Figure 7.2: Multiple Inheritance

The following demonstrates the use of multiple inheritance.

```

class A:
    def method1(self):
        print("A class method is called here")

class B:
    def method2(self):
        print("B class method is called here")

class C(A,B):
    def method3(self):
        print("C class method is called here")

c=C()
c.method3()
c.method2()
c.method1()
  
```

Output:
 C class method is called here
 B class method is called here
 A class method is called here

In above program, class C is derived from class A and class B. Object of class C (c) is created which can access the method of all the base classes derived by class C. Let us consider another program to demonstrate the use of multiple inheritance.

```

class Citizen:
    def __init__(self,name,age):
        self.name=name
  
```



```

        self.age=age

class Specialist:
    def __init__(self,specialization):
        self.specialization=specialization

class Doctor(Citizen, Doctor):
    def __init__(self, name, age, specialization, collegeName):
        Citizen.__init__(self, name, age)
        Specialist.__init__(self,specialization)
        self.college_name=collegeName

ram=Doctor ("ram",26,"cardiologist","AIIMS Delhi")
print(ram.name)
print(ram.age)
print(ram.specialization)
print(ram.college_name)

```

Output:

```

ram
26
cardiologist
AIIMS Delhi

```

In above program, Class Doctor inherits the attributes of class Citizen and Specialist. The `__init__()` method of Doctor calls the `__init__()` method of class Citizen and Specialist to initialize the attributes name, age and specialization of Doctor. Attribute `college_name` belongs to the class Doctor, hence it is initialized by the `__init__()` method of class Doctor. Now it can be seen from the output that all the attributes of Doctor (inherited attributes as well as attributes defined in Doctor class) can be accessed by the object ram of derived class Doctor.

7.13.2 Multilevel Inheritance

Multi-level inheritance is achieved when a derived class inherits another derived class. Number of levels go up to any number based on the requirements. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python. In Figure 7.3, Class A is a base class, class B is derived from class A, and class C is derived from class B. Class B have features of class A and class C have features of class B and class A.

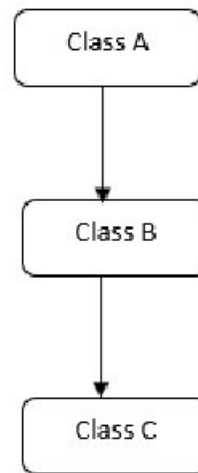


Figure 7.3: Multilevel Inheritance

Program to demonstrate the use of multiple inheritance.

```

class Citizen:
    def assign_basic(self,name,age):
        self.name=name
        self.age=age
class Specialist(Citizen):
    def assign_specialization(self,specialization):
        self.specialization=specialization
class Doctor(Specialist):
    def assign_college(self, collegeName):
        self.collegeName=collegeName
ravi=Doctor()
ravi.assign_basic("Ravi",27)
ravi.assign_specialization("cardiologist")
ravi.assign_college("AIIMS Delhi")
print(ravi.collegeName)
print(ravi.age)
print(ravi.specialization)
print(ravi.collegeName)
  
```

Output:

```

Ravi
27
cardiologist
AIIMS Delhi
  
```

In above program, class Citizen is a base class. Specialist is a derived by class that inherits the attributes of class Citizen. Class Doctor is another derived class that inherits the class Specialist. The method `assign_basic()` is defined in base class Citizen and the method `assign_specialization()` is defined in derived

class Specialist. The method `assign_college()` is defined in class Doctor. An object 'ravi' of class Doctor is created that can access all the attributes and methods of class Specialist and class Citizen as shown in the program.

7.13.3 Multipath Inheritance

In multipath inheritance a child class is derived from other derived classes that in turn derived from the same base class. Figure 7.4 shows multipath inheritance, Class B and class C are derived from class A. Class D is derived from class B and class C.

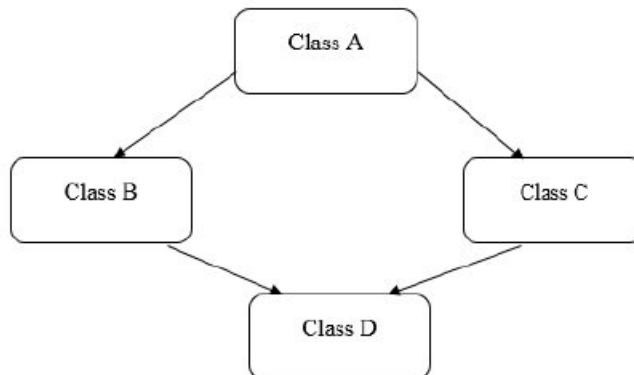


Figure 7.4: Multipath Inheritance

A program to illustrate the use of multipath inheritance is given as follows.

```
class A:
    def displayA(self):
        print("Method display() of class A invoked here")

class B(A):
    def displayB(self):
        print("Method display() of class B invoked here ")

class C(A):
    def displayC(self):
        print("Method display() of class C invoked here ")

class D(B,C):
    def displayD(self):
        print("Method display() of class D invoked here ")

d=D()
d.displayD()
d.displayC()
d.displayB()
```

```
d.displayA()
```

Output:

Method display() of class D invoked here

Method display() of class C invoked here

Method display() of class B invoked here

Method display() of class A invoked here

In above program, class A is a base class. Class B and class C are derived from class A. Class D is derived from class B and class C. Object of class D is created which access the method display() of class B, class C and class A.

Check your progress

1. What is Inheritance?
2. What does the super() do in python?
3. How does the super() method work?
4. What is method overriding?
5. How can we call the base class method overridden by the child class using child class object?
6. What do you mean by base class?
7. What do you mean derived class?
8. What are different types of Inheritance?
9. What is multilevel Inheritance?
10. What is multiple Inheritance?
11. Explain multipath Inheritance?

7.14 Abstract Classes

An Abstract Class is a template or blueprint for other classes to be defined. The other classes can be inherited from the abstract class and those must define the methods specified by the abstract class. An abstract class is not meant to create its instances rather it serves as template for other classes. Abstract class can have abstract methods and concrete methods. If there is any abstract method in a class, that class must be abstract.

A concrete method is a method whose action is defined in the abstract class itself but it can be overridden in the inherited class. An abstract method is

a method that has a declaration but does not have an implementation. Abstract methods must be implemented in child class derived from abstract class. A program to demonstrate the use of Abstract class is given here.

```
class Fruit:

    def cost(self):
        raise NotImplementedError("cost method not implemented!")
    def colour(self):
        raise NotImplementedError("colour method not implemented!")

class Apple(Fruit):

    def cost(self):
        return "200 Rs/KG"
    def colour(self):
        return "red"

class Banana(Fruit):

    def cost(self):
        return "60 Rs/Dozen"
    def colour(self):
        return "yellow"

ob1=Apple()
print(f"Apple cost: {ob1.cost()}")
print(f"Apple colour: {ob1.colour()}")
ob2=Banana()
print(f"Banana cost: {ob2.cost()}")
print(f"Banana colour: {ob2.colour()}")
```

Output:

```
Apple cost:200 Rs/KG
Apple colour: red
Banana cost:60 Rs/Dozen
Banana colour: yellow
```

In above program, Fruit is an abstract having two abstract methods, cost() and colour(). Class Apple and Banana are derived from Fruit class and implements the abstract methods of Fruit class. If child class does not implement the abstract methods than it raises an error defined in the abstract methods as shown in above program.

7.15 Method Overriding

Method overriding is a form of polymorphism in object-oriented programming languages that allows a derived class to redefine a method that already exists in its one of the parent classes. It allows the derived class to have a method with the same name, same parameters, and same return type as exists in its parent class. Method overriding is not allowed within the same class. However, it is still possible to call the base class function using the child class object as follows.

```
Base_classname.method_name(child_class_object)
```

Program to illustrate the use of method overriding is given as follows.

```
class Citizen:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print("\nIt is display function of Citizen class")
        print(f" Citizen name: {self.name}")
        print(f" Citizen age: {self.age}")

class Student(Citizen):
    def __init__(self, name, age, roll_no):
        Citizen.__init__(self, name, age)
        self.roll_no = roll_no

    def display(self):
        print("\nIt is display function of Child class")
        print(f"Student name: {self.name}")
        print(f"Student age: {self.age}")
        print(f"Student roll_no: {self.roll_no}")

p1 = Citizen('Rohit', 25)
p2 = Student("Ramesh", 15, 2)

p1.display()
p2.display()
Citizen.display(p2)
```

Output:

```
It is display function of Citizen class
Citizen name: Rohit
Citizen age: 25
```

```
It is display function of Child class
Student name: Ramesh
```



```
Student age: 15
Student roll_no: 2
```

```
It is display function of Citizen class
Citizen name: Ramesh
Citizen age: 15
```

In above program, method `display()` is defined in parent class, `Citizen`, which is overridden by child class, `Student`. You can see when `display()` function is called using object of class `Student` than `display()` function defined in the child class is executed. Hence, method overriding is implemented.

7.16 Operator Overloading

Operator overloading is another form of polymorphism. It gives additional meaning to an operator other than its predefined operational meaning. For example, the operator `+` is usually used to add two numbers. It can be given additional meaning to join two strings or to merge two lists. It is achieved with help of operator overloading. With operator overloading, a programmer can assign a different definition for a built-in operator in a class. This enables the programmer to perform some specific computation when the operator is applied on class objects and to apply a standard definition when the same operator is applied on a built-in data type. In Python, everything is an object. Each object has some special internal methods which it uses to interact with other objects. Generally, these methods follow the `__action__` naming convention. The Table 7.1 provides the operators that can be overloaded in classes, along with the method definitions that are required.

Table 7.1: Operators and their corresponding function names

Operator	Operation	Method	Expression
+	Addition	<code>__add__(self, other)</code>	<code>a1 + a2</code>
-	Subtraction	<code>__sub__(self, other)</code>	<code>a1 - a2</code>
*	Multiplication	<code>__mul__(self, other)</code>	<code>a1 * a2</code>
@	Matrix Multiplication	<code>__matmul__(self, other)</code>	<code>a1 @ a2</code>
/	Division	<code>__truediv__(self, other)</code>	<code>a1 / a2</code>
//	Floor Division	<code>__floordiv__(self, other)</code>	<code>a1 // a2</code>
%	Modulo/Remainder	<code>__mod__(self, other)</code>	<code>a1 % a2</code>

**	Power	<code>__pow__(self, other[, modulo])</code>	<code>a1 ** a2</code>
<<	Bitwise Left Shift	<code>__lshift__(self, other)</code>	<code>a1 << a2</code>
>>	Bitwise Right Shift	<code>__rshift__(self, other)</code>	<code>a1 >> a2</code>
&	Bitwise AND	<code>__and__(self, other)</code>	<code>a1 & a2</code>
^	Bitwise XOR	<code>__xor__(self, other)</code>	<code>a1 ^ a2</code>
 	Bitwise OR	<code>__or__(self, other)</code>	<code>a1 a2</code>
-	Negation (Arithmetic)	<code>__neg__(self)</code>	<code>-a1</code>
+	Positive	<code>__pos__(self)</code>	<code>+a1</code>
~	Bitwise NOT	<code>__invert__(self)</code>	<code>~a1</code>
<	Less than	<code>__lt__(self, other)</code>	<code>a1 < a2</code>
<=	Less than or Equal to	<code>__le__(self, other)</code>	<code>a1 <= a2</code>
=	Equal to	<code>__eq__(self, other)</code>	<code>a1 == a2</code>
!=	Not Equal to	<code>__ne__(self, other)</code>	<code>a1 != a2</code>
>	Greater than	<code>__gt__(self, other)</code>	<code>a1 > a2</code>
>=	Greater than or Equal to	<code>__ge__(self, other)</code>	<code>a1 >= a2</code>
[index]	Index operator	<code>__getitem__(self, index)</code>	<code>a1[index]</code>
In	In operator	<code>__contains__(self, other)</code>	<code>a2 in a1</code>

Program to add two complex numbers by overloading '+' operator

```
class ComplexNum(object):

    def __init__(self,r,i):
        self.real=r
        self.imag=i

    def __str__(self):
        imag=self.imag
        join='+'
        if(imag<0):
            imag=-imag
            join='-'
        self.s="str"
        return '(' + str(self.real) + join + 'i' + str(imag) + ')'

    def __add__(self,oth):
        return ComplexNum(self.real+oth.real, self.imag+oth.imag)
```

```

z1 = ComplexNum(2,3)
print('z1 =',z1)
z2=ComplexNum(4,2)
print('z2 =',z2)
z3=z1+z2
print(f'z1+z2={z3}')
Output:

```

```

z1 = (2 + i3)
z2 = (4 + i2)
z1+z2 = (6 + i5)

```

In above program, '+' operator is overloaded by redefining the `__add__(self, other)` method in order to use '+' operator to add two complex numbers. The function, `__str__(self)`, is also redefined to print the complex numbers in nicely readable form, `x+iy` or `x-iy`. The string function `__str__(self)` is called internally whenever `print()` function is used to print an object.

7.17 Summary

Object-oriented programming is an effective approach of writing computer programs. It is a programming paradigm that provides a tools for structuring the computer programs so that properties and behaviours real world objects are combined into individual objects. Object-oriented programming provides a way for modelling real-world things such as student, as well as its relations between other things like teachers and college, etc. Object-oriented programming allows to model real-world entities as software objects. In object-oriented programming, the classes represents the real-world things and situations. The instances of these classes are known as objects. When a class is written, it defines the general behaviour of the whole category of objects. Python is not only a popular scripting language but it is a great object-oriented programming language also. It has good support for the object-oriented programming paradigm. In Python, almost everything is represented as an object with its attributes and methods.

In object-oriented programming the variables can be used both at class level and instance level. The variables that are defined inside the Class definition but not inside any instance method are called Class variables. The class variables are not tied to a particular object but shared across all the objects of that class. If a modification is made to a class variable by any object, it is reflected to all the objects instance at the same time. On the hand the Instance variables tied to a particular object only. The changes made to instance variables by a particular object are not reflected to the other objects of the class.

Data hiding is the process that ensures exclusive data access to class members and provides object integrity by preventing unintended or intended changes. Data hiding insulates the data from the straight access by the program. Data hiding, also called encapsulation, organizes the data and methods into a structure that prevents data access by any function defined outside the class. Encapsulation or data hiding defines different levels for data variable and member function (or method) of the class.

The class methods are special methods that are bound to the class itself and not to the class objects. These methods are called using the class name but can also be called using class objects. The class methods have access to the class state and can therefore modify the class state. These changes are reflected across all the objects of the class. However, it cannot make modifications to the object state as it does not have access to `self` parameter. The first argument of a class method is `cls` instead of `self`. The class methods can be used to create objects using the same or different parameters used with the class constructor. A static method is another method that is not bound to class objects similar to a class method. But unlike a class method, a static method does not receive implicit first argument. It is associated neither with a `cls` nor with a `self` argument. It is free to accept any number of arguments. Similar to a class method, a static method is also bound to class name instead of class objects. However, it can not access or modify the object state or class state. A static method is marked by `@staticmethod` decorator.

Inheritance is the concept of reusability that allows to reuse the existing code for creating another code with some additional features. A programmer can first create a general class also known as a base class and later on, it can be extended to a more specialized class when needed. The specialized class that extends the existing base class is known as derived class. Class being inherited by another class is called base class or parent class. The derived class inherits the properties of the base class.

Method overriding is a form of polymorphism in object-oriented programming languages that allows a derived class to redefine a method that already exists in its one of the parent classes. It allows the derived class to have a method with the same name, same parameters, and same return type as exists in its parent class. Operator overloading is another form of polymorphism. It gives additional meaning to an operator other than its predefined operational meaning. For example, the operator `+` is usually used to add two numbers. It can be given additional meaning to join two strings or to merge two lists. It is achieved with help of operator overloading. With operator overloading, a programmer can assign a different definition for a built-in operator in a class.

An Abstract Class is a template or blueprint for other classes to be defined. The other classes can be inherited from the abstract class and those must define the methods specified by the abstract class. An abstract class is not meant to create its instances rather it serves as template for other classes. Abstract class

can have abstract methods and concrete methods. If there is any abstract method in a class, that class must be abstract.

Review Questions

Q.1 Write a Python class named Student with two attributes `student_name`, `marks`. Modify the attribute values of the said class and print the original and modified values of the said attributes.

Q.2 Write a Python class to convert an integer to a roman numeral.

Q.3 Write a Python class to implement `pow(x,n)`.

Q.4 Write a Python class which accept two methods `get_String` and `print_String`. `Get_String` accept a string from the user and `print_String` print the string in upper case.

Q.5 Write a program that has a class Circle. Use a class variable to define the value of constant PI. Use this class variable to calculate area and circumference of a circle with specified radius.

Q.6 Write a program with class Bill. The users have the option to pay the bill either by cheque or by cash. Use the inheritance to model this situation.

Q.7 Differentiate between the multiple and multi-level Inheritance.

Q.8 With the help of example explain the significance of `super()` function.

Q.9 Write a program that overloads the '+' operator to add two objects of class Matrix.

Q.10 What are abstract classes? Explain its significance using appropriate example.

UNIT-8: Exception Handling

Structure:

- 8.0 Introduction
- 8.1 Objectives
- 8.2 Error and Exceptions
- 8.3 Handling Exceptions
- 8.4 Handling Multiple Exceptions
- 8.5 Handling Multiple Exceptions Using a Single Except Block
- 8.6 Except Block without Exception
- 8.7 The *else* Block
- 8.8 Raising Exception
- 8.9 The Finally Block
- 8.10 Built-In Exceptions
- 8.11 User-Defined Exceptions
- 8.12 Assertions in Python
- 8.13 Summary
- 8.14 Review Questions

Unit 8: Exception Handling

8.0 Introduction

In this unit, the exception handling support in Python is discussed. An exception occurs due to unexpected events such as abnormal user input, attempting to open a non-existing file, attempting to write to read-only file, and undefined mathematical operations such as division by zero, etc. Exceptions are also raised due to some internal events, which change the normal flow of the program. It is important to handle such exceptions to prevent abrupt termination of the program. Exception handling is a process that aims to gracefully handle the exceptions to prevent the program to crash. Exception handling interrupts the normal flow of execution in case an exception occurs and executes a pre-defined exception handler. There could be multiple exception handler in a program. The execution of a particular exception handler depends on the type of exception that occurs during the program execution. Exception handling in Python is supported by specialized constructs. Some exceptions can be handled gracefully so that program resumes its execution where it was interrupted. It is usually happens with hardware specific exceptions.

Every error in Python programs leads to an exception, which is identified by its error type. When something unusual happens in a program, a mechanism is required to handle it. The exception mechanism allows a program to raise an exception. In Python, the exception handling mechanism involves the keywords *try* and *except* to catch the exceptions. The code expected to generate some exception during the execution is placed inside a *try* block. Whenever an exception occurs within this *try* block, the exception handling mechanism looks for a matching *except* block to handle it. The code given in the *except* block is executed that takes actions necessary to handle that exception. Python allows to have multiple *except* blocks with a single *try* block to handle different types of situations.

8.1 Objectives

The major objectives of this unit are as follows.

1. To understand the concepts of errors and exceptions.
2. To learn the process of handling exception.
3. To understand the use of *try*, *except*, and *finally* blocks.
4. To manually raise the exceptions.

8.2 Errors and Exceptions

Despite a lot of efforts and removing all syntactic errors, still there could be some errors in the program that get generated at runtime. Such errors are called *exceptions*. For syntactical error, the interpreter immediately gives error message. However, this is not the case with exceptions. This is because the exception might occur due to several other reasons which may or may not encounter in the source code. For example, if two integers are taken as input from the user for dividing one number by other number. Suppose user provides 0 as divisor, which is a valid number. But it is not a valid divisor. It leads to the problems at runtime. Let us consider the following program to illustrate the exception.

```
a=int(input('Enter first number: '))
b=int(input('Enter second number: '))
c=a/b
print(c)
```

Output:

Enter first number: 19

Enter second number: 0

Traceback (most recent call last):

File "C: exception.py", line 11, in <module>

c=a/b

ZeroDivisionError: division by zero

It can be observed from the above program that when it is attempted to perform division by zero, an exception is raised at runtime. These exceptions are named according to the reason behind them. Here in this case, the exception is named as *ZeroDivisionError*. The major differences between error and exception are outlined as follows.

- Error may occur at compile time or runtime.
- Exceptions are raised when the program is syntactically correct but the execution results in an error due to some other reasons.
- Errors can be intercepted at compile time but exceptions cannot.
- All exception occurs only at runtime.
- An exception is an error that can be handled by a programmer.
- An exception which is not handled by programmer becomes an error.

8.3 Exceptions Handling

It is important to have mechanisms in the program to handle the exceptions to prevent the abrupt termination of the program. If such undeniable situation arises in the program during execution then there should be some alternative options to direct the control of execution. Python provides following mechanisms to handle the exceptions.

- **try/except**
- **try/except/finally**
- **raise**
- **assert**

The exception handling mechanisms allow a programmer to notify message or take some alternative action if something undesirable happens. The **try/except** blocks are the most popular and basic mechanism in Python to handle the exceptions in Python.

If a Python program contains suspicious code that may throw the exception, it must be placed in a try block. The try block must be followed with an 'except' statement, which contains a block of code that is executed if there is some exception during execution of the code in try block. The syntax of try/except blocks is given as follows.

```
try:
    #block of code
except ExceptionName:
    #block of code
#remaining code of program
```

The primary source code is placed in the try block followed by 'except' block that contains some alternative code. On execution, first, the try block is executed. If no exception occurs, the 'except' block is skipped. But if an exception occurs, then rest of the statements inside the try block are skipped and if the exception type matches the exception name (that is written just after the 'except' keyword), then 'except' block is executed and the execution continues after the try statement. If exception does not match the exception name, then it is an unhandled exception and the program is terminated with an error message. Let us consider the following program to illustrate the use of try/except block.

```
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
try:
```

```

c=a/b
print(c)
except ZeroDivisionError:
    print("Division by zero not allowed. Please enter the appropriate numbers")
print("Program ended after executing remaining statements")

```

Output:

Enter first number:12

Enter second number:0

Division by zero not allowed. Please enter the appropriate numbers.

Program ended after executing remaining statements

In above program, statement that can cause exception is written inside try block, and exception name, i.e. ZeroDivisionError in this case, was mentioned with 'except' clause, followed by the statements depicting the action to be taken is written inside 'except' block. As it can be observed in the above program, when it is attempted to divide a number with zero, the exception ZeroDivisionError is thrown by Python interpreter. This exception was matched with the exception mentioned in except block. Hence, statement written inside except block is executed and execution continued after except block. In case there is no exception, the code in except block is not executed and execution control goes to first statement after the 'except' block. In this way exceptions can be handled in the Python programs. Even if there is no provision in the program to handle the exceptions, the default error handling mechanism of the Python works automatically.

Check your progress

11. Define exception in a program.
12. Differentiate error and exception.
13. What are the mechanisms to handle the exception in Python?
14. What happens when exception is not handled by the program?
15. What is written after 'except' keyword?
16. What happens if there is no match to exception name?

8.4 Handling Multiple Exceptions

A particular block of code contains multiple statements that perform different operations. Different statements could generate different types of exceptions depending on the instruction type. Therefore, sometimes provisions may be

needed to handle multiple exceptions of different types for a particular block. In Python, it is possible to have multiple 'except' blocks for a single 'try' block to handle multiple exceptions. When the python interpreter encounters an exception, it checks all the 'except' blocks associated with that try block. If interpreter finds a matching exception, it executes corresponding 'except' block. The syntax for associating multiple 'except' blocks with a single 'try' block is given as follows.

```
try:
    statements which can throw an Exceptions
except Exception1:
    # block of code if exception1 occurs
except Exception2:
    # block of code if exception2 occurs
...
#remaining code of program
```

Let us illustrate the use of multiple 'except' blocks with a 'try' block with help of the following program.

```
try:
    a=int(input("Enter first number: "))
    b=int(input("Enter second number: "))
    c=a/b
    print(c)
except ZeroDivisionError:
    print("Division by zero not allowed")
except ValueError:
    print("Only integer values are allowed")
print("Program ended after executing remaining statements")
```

Output:

```
Enter first number: 12
Enter second number: x
Only integer values are allowed
Program ended after executing remaining statements
```

In the above program, two 'except' blocks are associated with a 'try' block. One 'except' block deals with a 'ZeroDivisionError' and second 'except' handles 'ValueError'. When the above program is executed, the user enters a character instead of a number. But division operation needs two numbers. Therefore, a 'ValueError' is thrown by Python interpreter. This exception has a match with the second 'except' block, and statement inside that except block is executed.

8.5 Handling Multiple Exceptions Using a Single Except Block

Python allows to handle multiple exceptions using a single 'except' block also. This approach can avoid unnecessary duplication of the code and can save the programmer's time. This approach is helpful if the same action can be taken for multiple exceptions. The syntax for handling multiple exceptions in single 'except' block is given as follows.

```
try:
    statements which can throw Exceptions
except (Exception1, Exception2, ... ExceptionN):
    # block of code if any of the Exception mentioned in except clause occurs
#remaining code of program
```

Let us consider the following example to illustrate this approach.

```
try:
    a=int(input("Enter first number:"))
    b=int(input("Enter second number:"))
    c=a/b
    print(c)
except (ZeroDivisionError, KeyboardInterrupt, ValueError, TypeError):
    print("Please check values before entering. Program Terminating")
print("Program ended after executing remaining statements")
```

Output 1:

```
Enter first number: 12
Enter second number: a
Please check before enter... Program Terminating
Program ended after executing remaining statements
```

Output 2:

Enter first number: 12

Enter second number: 0

Please check values before entering. Program Terminating

Program ended after executing remaining statements

In above program, four exceptions `ZeroDivisionError`, `KeyboardInterrupt`, `ValueError`, and `TypeError` are written in the same 'except' clause. If any of these exceptions is raised by the statements in 'try' block, the same 'except' block gets executed. The program is executed two times, one gives rise to 'ValueError' exception and second time 'ZeroDivisionError' exception is generated. In both cases, the same 'except' block is executed.

8.6 Except Block without Specific Exceptions

Python allows to specify except block without mentioning any exception. If a programmer finds it difficult to predict all types of possible exceptions, it is better to write a handler that would catch all types of exceptions. The syntax for 'except' block without mentioning specific exceptions is as follows.

```
try:
    statements which can throw Exceptions
except:
    # block of code to execute if any exception occurs
```

The 'except' block can be used along with other handlers which handle some specific types of exceptions. If an exception has no match to specific handlers then it can be handled by the 'except' block with no specific exception. It is a kind of default handler that must be placed after all other 'except' blocks. Let us consider the following program to illustrate this approach.

```
try:
    a=int(input("Enter first number: "))
    b=int(input("Enter second number: "))
    c=a/b
    print(c)
except ZeroDivisionError:
```

```

    print("Division by zero not allowed")
except KeyboardInterrupt:
    print("Interrupt occurred")
except:
    print("Unexpected error occurred")
print("Program ended after executing remaining statements")

```

Output:

```

Enter first number: 12
Enter second number: x
Unexpected error occurred
Program ended after executing remaining statements

```

In above program, specific exceptions are mentioned in the first two 'except' blocks and the last 'except' block has no specific exception. During the execution, the user has entered a wrong input causing an exception having no match to any of the 'except' blocks. Hence, the last 'except' block gets executed.

8.7 The else Block

In Python, keyword *else* can also be used along with the 'try' and 'except' clauses. The 'except' block is executed if the exception occurs inside the try block while the 'else' block gets processed if no exception is caused inside the 'try' block. The syntax of 'else' clause with try/except blocks is given as follows.

```

try:
    #statements in try block
Except ExceptionName:
    #executed when error in try block
else:
    #executed if try block is error-free

```

The following program demonstrates the use of 'else' block with try/except blocks.

```

try:

```



```

a=int(input("Enter first number: "))
b=int(input("Enter second number: "))
c=a/b
except ZeroDivisionError:
    print("Eivision by zero not allowed")
except KeyboardInterrupt:
    print("Interruption occurred")
except:
    print("Unexpected error occurred")
else:
    print("Inside 'else' block")
    print("c==>c)

```

Output:

```

Enter first number: 12
Enter second number: 2
Inside else block
c= 6.0

```

During the execution of the above program, user provided the proper input values. Therefore, no error occurred while executing 'try' block and hence 'else' block gets executed.

8.9 Raising Exceptions

In Python programming, exceptions are raised when errors occur at runtime. However, Python also allows us to raise exceptions manually using the 'raise' keyword. The syntax for the 'raise' statement is given as follows.

```

raise [exception [, args [, traceback ] ] ]

```

Here, the parameter 'exception' is the type of exception for example, ZeroDivisionError, and the parameter 'args' is used to provide values for the exception arguments. The parameter 'args' is optional; if it is not supplied, the exception argument is 'None'. The last parameter, 'traceback' is also optional and rarely used in practice. If it is present, the 'traceback' object is used for the exception. Let us consider the following program.


```

try:
    a=int(input("Enter first number: "))
    b=int(input("Enter second number: "))
    c=a/b
    print(c)
    raise
except:
    print("Exception occurred")
print("Program ended after executing remaining statements")

```

Output:

```

Enter first number: 12
Enter second number: 2
6.0
Exception occurred
Program ended after executing remaining statements

```

In above program, it can be observed that an exception is raised deliberately even if there was no error. The keyword 'raise' can be used to throw an exception if certain condition occurs. For example,

```

age=int(input("Enter your age: "))
try:
    if age < 18:
        raise
    if age > 40:
        raise
    print("You are eligible to apply")
except:
    print("You are not eligible to apply")
print("Program ended after executing remaining statements")

```

Output:

```
Enter your age: 15
You are not eligible to apply
Program ended after executing remaining statements
```

In above program, an exception is deliberately raised if the age of the applicant is less than 18 or greater than 40. Python allows to re-raise the exception in 'except' block. It is required to re-raise the exception to determine whether an exception was raised but don't intend to handle it. Let us consider the following example.

```
age=int(input("Enter your age: "))
try:
    if age < 18:
        raise ValueError
    if age > 40:
        raise ValueError
    print("You are eligible to apply")
except:
    print("You are not eligible to apply")
    raise
print("Program ended after executing remaining statements")
```

Output:

```
Enter your age: 15
You are not eligible to apply
Traceback (most recent call last):
  File "C:\Users\Manoj\Documents\Python39\raise2.py", line 11, in <module>
    raise ValueError
ValueError
```

In above program, error is re-raised in except block but not handled.

8.9 The finally block

The 'finally' code block is also a part of exception handling mechanism. When exception handling is done using the 'try' and 'except' block, a 'finally' block can be added after 'except' blocks. The 'finally' block is always executed even if

there is an exception in previous code blocks. Therefore, it is generally used for doing the concluding tasks such as closing file resources or closing database connection or gracefully terminating the program execution with a delightful message. If an 'except' block is unable to catch the exception, which interrupts code execution, still the finally block gets executed. A small program to illustrate the use of finally block is given here.

```
try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number: "))
    print("Result of Division: " + str(a/b))
except(ZeroDivisionError):
    print("Division by zero not allowed.")
finally:
    print("Finally block executed")
print("Program ended after executing remaining statements")
```

Output:

```
Enter numerator number: 12
Enter denominator number: 0
Division by zero not allowed.
Finally block executed
Program ended after executing remaining statements
```

In above program, finally block executed even after the occurrence of ZeroDivisionError. Let us consider another example to illustrate the use of 'try', 'except', and 'finally' blocks together.

```
try:
    print("Raising Exception")
    raise TypeError
except:
    print("Exception Caught")
finally:
```

```
print("Performing closing operations")
```

Output:

Raising Exception

Exception Caught

Performing closing operations

From the output of above program, it can be seen that finally block executed when error occurs and also when error does not occur.

Check your progress

1. How the multiple exceptions are handled in Python programs?
2. Explain the use of except block without exception?
3. What is the purpose of using else block?
4. How can we raise exceptions manually?
5. How can we re-raise an exception?
6. What is the use of re-raising the exception?
7. Explain the use of finally block?
8. What is the difference between 'except' block without exception and 'finally' block?

8.10 Built-In Exceptions

There are plenty of built-in exceptions in Python that can be raised when corresponding errors occur. The built-in exceptions can be displayed using the built-in `local()` function as follows.

```
print(dir(locals()['__builtins__']))
```

`locals()['__builtins__']` returns a module of built-in exceptions, functions, and attributes. *dir* allows us to list these attributes as strings.

Some of the common built-in exceptions in Python along with their description are listed in Table 8.1.

Table 8.1 Built-in Exceptions

Exception	Description
AssertionError	Raised when an <code>assert</code> statement fails.
AttributeError	Raised when attribute assignment or reference fails.

EOFError	Raised when the <code>input()</code> function hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's <code>close()</code> method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or Delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by <code>next()</code> function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by <code>sys.exit()</code> function
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.

UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.

8.11 User-Defined Exceptions

In Python, users can define customized exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class. Although not mandatory, most of the exceptions are named as names that end in “Error” similar to naming of the standard exceptions in Python. It is explained with the help of the following program.

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return(repr(self.value))
try:
    raise(MyError(10*2))
except MyError as err:
    print("User Defined Exception occurred: ",err.value)
```

Output:

User defined exception occurred: 20

In above program, the constructor (`__init__()`) of the Exception class is overridden to accept user defined custom argument *value*. The newly defined exception class can be used in any program. However, these classes are kept simple having limited attributes to provide the information about the error to be extracted by handlers for the exception.

8.12 Assertions in Python

Assertion is a kind of debugging tool and its primary task is to check the condition. If it finds a condition True, it moves to the next line of code, otherwise stops all its operations and raises the `AssertionError` exception with the specified error message. It points out the error in the code. The syntax for `assert` is given as follows.

```
assert Expression[, Arguments]
```

A sample program to illustrate the use of assertions is given as follows.

```
x = int(input("Enter value of x: "))
assert x > 0
print ("x' is a positive number.")
```

Output 1:

```
Enter value of x: 2
'x' is a positive number
```

Output 2:

```
Enter value of x:0
Traceback (most recent call last):
  File "C:\Python39\assert.py", line 9, in <module>
    assert x > 0
AssertionError
```

In above program, it can be observed that when `assert` condition is True as shown in output 1, it moves to next line of code. But when `assert` condition goes False, as shown in output 2, it stops all its operations and raises the `AssertionError` exception. The `assert` statement can optionally include an error message string, which gets displayed along with the `AssertionError`. Let

us consider another program to display error message along with `AssertionError`.

```
x = int(input("Enter value of x: "))
assert x > 0, "Only positive numbers are allowed"
print("'x' is a positive number.")
```

Output:

Enter value of x: 0

Traceback (most recent call last):

File "C:\Python39\assert.py", line 9, in <module>

assert x > 0, "Only positive numbers are allowed"

`AssertionError: Only positive numbers are allowed`

In above program, as condition specified by `assert` statement goes false, error message gets printed along with `AssertionError`.

8.13 Summary

Every error in Python programs leads to an exception, which is identified by its error type. When something unusual happens in a program, a mechanism is required to handle it. The exception mechanism allows a program to raise an exception. In Python, the exception handling mechanism involves the keywords *try* and *except* to catch the exceptions. The code expected to generate some exception during the execution is placed inside a *try* block. Whenever an exception occurs within this *try* block, the exception handling mechanism looks for a matching *except* block to handle it. The code given in the *except* block is executed that takes actions necessary to handle that exception.

Python allows to have multiple *except* blocks with a single *try* block to handle different types of situations. Python also allows to handle multiple exceptions using a single *except* block. This approach can avoid unnecessary duplication of the code and can save the programmer's time. This approach is helpful if the same action can be taken for multiple exceptions. In addition, it is possible to have an *except* block without mentioning any exception. If a programmer finds it difficult to predict all types of possible exceptions, it is better to write a handler that would catch all types of exceptions. In Python, keyword *else* can also be used along with the 'try' and 'except' clauses. The 'except' block is executed if the exception occurs inside the try block while the 'else' block gets processed if no exception is caused inside the 'try' block.

In Python programming, exceptions are raised when errors occur at runtime. However, Python also allows us to raise exceptions manually using the 'raise' keyword. The 'finally' code block is also a part of exception handling mechanism. When exception handling is done using the 'try' and 'except' block, a 'finally' block can be added after 'except' blocks. The 'finally' block is always executed even if there is an exception in previous code blocks. Therefore, it is generally used for doing the concluding tasks such as closing file resources or closing database connection or gracefully terminating the program execution with a delightful message. There are plenty of built-in exceptions in Python that can be raised when corresponding errors occur. The built-in exceptions can be displayed using the built-in local () function. The users can also define customized exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class. Assertion is a kind of debugging tool and its primary task is to check the condition. If it finds a condition True, it moves to the next line of code, otherwise stops all its operations and raises the AssertionError exception with the specified error message.

8.14 Review Questions

- Q.1 Explain the procedure to create user defined exception.
- Q.2 Explain the use of Assert statement.
- Q.3 Explain any two built-in exceptions with relevant example.
- Q.4 Write a program that re-raise an exception.
- Q.5. Write a program that finds smaller of two given numbers. If the first number is smaller than the second number.
- Q.6 Write a program that accepts date of birth along with other personal details of a person. Raise an exception if an invalid date is entered.
- Q.7 Write a program to print the square root of a number. Raise an exception if the number is negative.
- Q.8 Write a program that opens a file and writes data to it. Handle exceptions that can be generated during the I/O operations.
- Q.9 Write a program which infinitely prints natural numbers. Raise the StopIteration exception after displaying first 10 numbers to exit the program.
- Q.10 Write a program that validates name and age as entered by the user to determine whether the person can cast vote or not.

Unit 9: Python Standard Library

Structure

- 9.0 Introduction
- 9.1 Objectives
- 9.2 The Python standard library
- 9.3 OS module
- 9.4 MATH module
- 9.5 STRING module
- 9.6 Internet access
- 9.7 Date and time
- 9.8 Calendar
- 9.9 Data compression
- 9.10 Summary
- 9.11 Review questions

Unit 9: Python Standard Library

9.0 Introduction

The power of a programming language lies in its libraries. A library is a collection of modules containing many useful functions that can be used in other programs. In addition to functions or methods the libraries may also contain some data values or numerical constants and other things. The libraries eliminate the need of writing the entire code from scratch. The frequently used codes are included in the libraries that can be used iteratively. People can share their useful codes with other programmers through libraries. A number of standard methods for performing common tasks or operations are written and provided in ready to use form. A group of such programs is collectively known as a standard library. In most of the programming languages, many related functions/methods are organized into several libraries. These programs are written and tested by programmers and reviewed by several people to ensure they work as per expectations without errors and bugs. Once a library is published by the authorized forum, other programmers can use it confidently.

The Python standard library is an extensive collection of standard methods, modules, and data elements that is provided with Python itself. In addition to the standard library, there are many other Python libraries available for a variety of applications. Usually a library contains a number of modules but some libraries may also contain only a single module. A library or module is loaded into the program memory in order to use it. In Python, `import` keyword is used to load a library. It can be done in many different ways. For example, a Python module `math` can be loaded as follows,

```
import math
```

The above statement loads the `math` module. The module can be considered as a class and all its variables and functions can be accessed considering them class members. Instead of loading entire module, only a particular data element or method from the module can also be loaded as follows.

```
from math import factorial
```

In the above example, only *factorial* function from the `math` library is loaded. It uses `__import__()` to search the module for the function `factorial`. In case a function or data element is not found in the module, an error generated. All the data elements and functions in the library or module can be loaded at once as follows.

```
from math import *
```

It is also possible to create alias for a library while importing it. This approach helps to shorten the names as follows.

```
import string as st
```

Here, the `string` module is named as `st` while importing it. Now `st` can be used in place of `string` for related operations. It is more helpful when library names are long or a library is used frequently in a program. However, aliases may make program harder to understand for other as aliases are not standard names.

There are several thousand different libraries in Python at present. These libraries are related to many different applications such as data science, machine learning, natural language processing, data visualization, artificial intelligence, deep learning, pattern analysis, image processing, and data compression, etc. NumPy, Pandas, Matplotlib, TensorFlow, Scikit-Learn, Keras, SciPy, PyTorch, and Theano, etc. are some top Python libraries.

9.1 Objectives

The major objectives of this unit are outlined here as follows.

1. To introduce the concept of library in Python.
2. To provide an insight into Python standard library.
3. To discuss the usage of important modules in Python standard library.

9.2 The Python Standard Library

Python standard library is a large and extensive library that contains built-in modules to provide many basic facilities and access to system functionalities. The functionality such as file I/O would be inaccessible to programmers without standard library. The built-in modules in standard library are written in C programming language, which provide solution to many standard problems. In addition to provide basic functionality, some modules are designed to enhance the portability of Python programs. The Python standard library is included with the Python distributions. The Python installers for the Windows include the entire standard library and many other components. For Unix/Linux operating systems Python is normally provided as a collection of packages. There are more than 200 modules in the standard library. The list of standard library modules are available at <https://docs.python.org> given as follows.

A

<code>aifc</code>	Read and write audio files in AIFF or AIFC format.
<code>argparse</code>	Command-line option and argument parsing library.
<code>array</code>	Space efficient arrays of uniformly typed numeric values.
<code>ast</code>	Abstract Syntax Tree classes and manipulation.
<code>asynchat</code>	Support for asynchronous command/response protocols.

asyncio	Asynchronous I/O.
asyncore	A base class for developing asynchronous socket handling services.
atexit	Register and execute cleanup functions.
audioop	Manipulate raw audio data.

B	
base64	RFC 3548: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85
bdb	Debugger framework.
binascii	Tools for converting between binary and various ASCII-encoded binary representations.
binhex	Encode and decode files in binhex4 format.
bisect	Array bisection algorithms for binary searching.
builtins	The module that provides the built-in namespace.
bz2	Interfaces for bz2 compression and decompression.

C	
calendar	Functions for working with calendars, including some emulation of the Unix cal program.
cgi	Helpers for running Python scripts via the Common Gateway Interface.
cgitb	Configurable traceback handler for CGI scripts.
chunk	Module to read IFF chunks.
cmath	Mathematical functions for complex numbers.
cmd	Build line-oriented command interpreters.
code	Facilities to implement read-eval-print loops.
codecs	Encode and decode data and streams.
codeop	Compile (possibly incomplete) Python code.
collections	Container datatypes
colorsys	Conversion functions between RGB and other color systems.
compileall	Tools for byte-compiling all Python source files in a directory tree.
configparser	Configuration file parser.
contextlib	Utilities for with-statement contexts.
contextvars	Context Variables

copy	Shallow and deep copy operations.
copyreg	Register pickle support functions.
crypt (Unix)	The crypt() function used to check Unix passwords.
csv	Write and read tabular data to and from delimited files.
ctypes	A foreign function library for Python.
curses (Unix)	An interface to the curses library, providing portable terminal handling.

D

dataclasses	Generate special methods on user-defined classes.
datetime	Basic date and time types.
dbm	Interfaces to various Unix "database" formats.
decimal	Implementation of the General Decimal Arithmetic Specification.
difflib	Helpers for computing differences between objects.
dis	Disassembler for Python bytecode.
distutils	Support for building and installing Python modules into an existing Python installation.
doctest	Test pieces of code within docstrings.

E

email	Package supporting the parsing, manipulating, and generating email messages.
ensurepip	Bootstrapping the "pip" installer into an existing Python installation or virtual environment.
enum	Implementation of an enumeration class.
errno	Standard errno system symbols.

F

faulthandler	Dump the Python traceback.
fcntl (Unix)	The fcntl() and ioctl() system calls.
filecmp	Compare files efficiently.
fileinput	Loop over standard input or a list of files.
fnmatch	Unix shell style filename pattern matching.
formatter	Deprecated: Generic output formatter and device interface.
fractions	Rational numbers.
ftplib	FTP protocol client (requires sockets).
functools	Higher-order functions and operations on callable objects.

G	
gc	Interface to the cycle-detecting garbage collector.
getopt	Portable parser for command line options; support both short and long option names.
getpass	Portable reading of passwords and retrieval of the userid.
gettext	Multilingual internationalization services.
glob	Unix shell style pathname pattern expansion.
graphlib	Functionality to operate with graph-like structures
grp (Unix)	The group database (getgrnam() and friends).
gzip	Interfaces for gzip compression and decompression using file objects.

H	
hashlib	Secure hash and message digest algorithms.
heapq	Heap queue algorithm (a.k.a. priority queue).
hmac	Keyed-Hashing for Message Authentication (HMAC) implementation
html	Helpers for manipulating HTML.
http	HTTP status codes and messages

I	
imaplib	IMAP4 protocol client (requires sockets).
imghdr	Determine the type of image contained in a file or byte stream.
imp	Deprecated: Access the implementation of the import statement.
importlib	The implementation of the import machinery.
inspect	Extract information and source code from live objects.
io	Core tools for working with streams.
ipaddress	IPv4/IPv6 manipulation library.
itertools	Functions creating iterators for efficient looping.

J	
json	Encode and decode the JSON format.

K	
keyword	Test whether a string is a keyword in Python.

L	
lib2to3	The 2to3 library
linecache	Provides random access to individual lines from text files.
locale	Internationalization services.
logging	Flexible event logging system for applications.
lzma	A Python wrapper for the liblzma compression library.

M	
mailbox	Manipulate mailboxes in various formats
mailcap	Mailcap file handling.
marshal	Convert Python objects to streams of bytes and back (with different constraints).
math	Mathematical functions (sin() etc.).
mimetypes	Mapping of filename extensions to MIME types.
mmap	Interface to memory-mapped files for Unix and Windows.
modulefinder	Find modules used by a script.
msilib (Windows)	Creation of Microsoft Installer files, and CAB files.
msvcrt (Windows)	Miscellaneous useful routines from the MS VC++ runtime.
multiprocessing	Process-based parallelism.

N	
netrc	Loading of .netrc files.
nis (Unix)	Interface to Sun's NIS (Yellow Pages) library.
nntplib	NNTP protocol client (requires sockets).
numbers	Numeric abstract base classes (Complex, Real, Integral, etc.).

O	
operator	Functions corresponding to the standard operators.
optparse	Deprecated: Command-line option parsing library.
os	Miscellaneous operating system interfaces.
ossaudiodev (Linux, FreeBSD)	Access to OSS-compatible audio devices.

P	
<code>parser</code>	Access parse trees for Python source code.
<code>pathlib</code>	Object-oriented filesystem paths
<code>pdb</code>	The Python debugger for interactive interpreters.
<code>pickle</code>	Convert Python objects to streams of bytes and back.
<code>pickletools</code>	Contains extensive comments about the pickle protocols and pickle-machine opcodes, as well as some useful functions.
<code>pipes (Unix)</code>	A Python interface to Unix shell pipelines.
<code>pkgutil</code>	Utilities for the import system.
<code>platform</code>	Retrieves as much platform identifying data as possible.
<code>plistlib</code>	Generate and parse Apple plist files.
<code>poplib</code>	POP3 protocol client (requires sockets).
<code>posix (Unix)</code>	The most common POSIX system calls (normally used via module <code>os</code>).
<code>pprint</code>	Data pretty printer.
<code>profile</code>	Python source profiler.
<code>pstats</code>	Statistics object for use with the profiler.
<code>pty (Linux)</code>	Pseudo-Terminal Handling for Linux.
<code>pwd (Unix)</code>	The password database (<code>getpwnam()</code> and friends).
<code>py_compile</code>	Generate byte-code files from Python source files.
<code>pyclbr</code>	Supports information extraction for a Python module browser.
<code>pydoc</code>	Documentation generator and online help system.
Q	
<code>queue</code>	A synchronized queue class.
<code>quopri</code>	Encode and decode files using the MIME quoted-printable encoding.
R	
<code>random</code>	Generate pseudo-random numbers with various common distributions.
<code>re</code>	Regular expression operations.
<code>readline (Unix)</code>	GNU readline support for Python.
<code>reprlib</code>	Alternate <code>repr()</code> implementation with size limits.
<code>resource (Unix)</code>	An interface to provide resource usage information on the current process.
<code>rlcompleter</code>	Python identifier completion, suitable for the GNU readline

	library.
runpy	Locate and run Python modules without importing them first.
S	
sched	General purpose event scheduler.
secrets	Generate secure random numbers for managing secrets.
select	Wait for I/O completion on multiple streams.
selectors	High-level I/O multiplexing.
shelve	Python object persistence.
shlex	Simple lexical analysis for Unix shell-like languages.
shutil	High-level file operations, including copying.
signal	Set handlers for asynchronous events.
site	Module responsible for site-specific configuration.
smtpd	A SMTP server implementation in Python.
smtplib	SMTP protocol client (requires sockets).
sndhdr	Determine type of a sound file.
socket	Low-level networking interface.
socketserver	A framework for network servers.
spwd (Unix)	The shadow password database (getspnam() and friends).
sqlite3	A DB-API 2.0 implementation using SQLite 3.x.
ssl	TLS/SSL wrapper for socket objects
stat	Utilities for interpreting the results of os.stat(), os.lstat() and os.fstat().
statistics	Mathematical statistics functions
string	Common string operations.
stringprep	String preparation, as per RFC 3453
struct	Interpret bytes as packed binary data.
subprocess	Subprocess management.
sunau	Provide an interface to the Sun AU sound format.
symbol	Constants representing internal nodes of the parse tree.
symtable	Interface to the compiler's internal symbol tables.
sys	Access system-specific parameters and functions.
sysconfig	Python's configuration information
syslog (Unix)	An interface to the Unix syslog library routines.

T	
tabnanny	Tool for detecting white space related problems in Python source files in a directory tree.
tarfile	Read and write tar-format archive files.
telnetlib	Telnet client class.
tempfile	Generate temporary files and directories.
termios (Unix)	POSIX style tty control.
test	Regression tests package containing the testing suite for Python.
textwrap	Text wrapping and filling
threading	Thread-based parallelism.
time	Time access and conversions.
timeit	Measure the execution time of small code snippets.
tkinter	Interface to Tcl/Tk for graphical user interfaces
token	Constants representing terminal nodes of the parse tree.
tokenize	Lexical scanner for Python source code.
trace	Trace or track Python statement execution.
traceback	Print or retrieve a stack traceback.
tracemalloc	Trace memory allocations.
tty (Unix)	Utility functions that perform common terminal control operations.
turtle	An educational framework for simple graphics applications
turtledemo	A viewer for example turtle scripts
types	Names for built-in types.
typing	Support for type hints.
U	
unicodedata	Access the Unicode Database.
unittest	Unit testing framework for Python.
urllib	URL handling module.
uu	Encode and decode files in uuencode format.
uuid	UUID objects (universally unique identifiers) according to RFC 4122
V	
venv	Creation of virtual environments.

W	
warnings	Issue warning messages and control their disposition.
wave	Provide an interface to the WAV sound format.
weakref	Support for weak references and weak dictionaries.
webbrowser	Easy-to-use controller for Web browsers.
winreg (Windows)	Routines and objects for manipulating the Windows registry.
winsound (Windows)	Access to the sound-playing machinery for Windows.
wsgiref	WSGI Utilities and Reference Implementation.
X	
xdrlib	Encoders and decoders for the External Data Representation (XDR).
xml	Package containing XML processing modules
Xmlrpc	
Z	
zipapp	Manage executable Python zip archives
zipfile	Read and write ZIP-format archive files.
zipimport	Support for importing Python modules from ZIP archives.
zlib	Low-level interface to compression and decompression routines compatible with gzip.
zoneinfo	IANA time zone support

9.3 OS module

The OS module is one of the Python standard utility module that allows interaction between user and the operating system. It provides a number of functions to perform operating system related tasks. It also contains functions to interact with the file system to create and manipulate files and directories. Some of these functions are discussed here.

- `os.chdir(path)`: The `chdir()` method changes the current directory to the directory specified by the parameter *path*.
- `os.getcwd()`: It returns the current working directory. Let us consider the following example,

```
import os
crd = os.getcwd()
print('Current directory: ', crd)
```



```
os.chdir('D:\Temp')
crd = os.getcwd()
print('Current directory after chdir(): ', crd)
```

The output of the above code is,

Current directory: D:\pyCode
Current directory after chdir(): D:\Temp

- **os.mkdir(path, [mode]):** This command is used to create a directory. The parameter *path* supplies the name location of the directory and the parameter *mode* provides the directory permissions. The default mode is 0777 in octal numbers.

```
import os
crd = os.mkdir('D:\Temp2',0755)
print('Directory is created')
```

- **os.makedirs(path, [mode]):** This command creates directories recursively. For example,

```
import os
crd = os.makedirs('D:\Temp2\Temp3\Temp4',0755)
```

While creating directory Temp4 in above example, if any of the parent directories Temp2 and Temp3 does not exist it is also created along with directory Temp4.

- **os.remove(file):** This method is used to delete a file. It cannot delete a directory. If a directory name is supplied instead of a file then error is generated.
- **os.rmdir(directory):** It can delete an empty directory. If directory is not empty then an error message is generated.
- **os.removedirs(path):** It removes the directory recursively.
- **os.rename(src, dst):** It rename a file or directory source to destination.
- **os.renames(src, dst):** It can rename directories recursively.
- **os.listdir(path):** This methods returns a list of all entries in a directory.

In addition to above discussed methods, there are many more methods in OS module.

Check your progress

1. What is a library in a programming language?

2. How is a library loaded in a Python program?
3. How are the functions and constants loaded in a Python program?
4. What is Python standard library?
5. How many modules are there in Python standard library?
6. Name some important Python libraries.
7. In which programming language the modules of Python standard library are written?
8. What is the utility of OS module?
9. How can the current directory be changed?
10. Differentiate the `os.mkdir()` and `os.makedirs()`.
11. How can directories be removed recursively?

9.4 MATH module

The MATH module provides many useful functions to perform a number of mathematical operations. All the functions in this module work on integer or floating point numbers only. It cannot be used to perform mathematical operations on complex numbers. It also defines some standard constants to use in mathematical operations. These constants and their values are given in Table 9.1.

Table 9.1: Constants in math module.

Constant	Value	Description
E	0.718282	Returns the value of natural base.
Inf	-	Returns the infinite.
Nan	-	Not a number.
Pi	3.141592	Returns the value of pi.
Tau	6.283185	Returns the value tau.

The commonly used methods in math module are described here.

- `ceil(x)`: It returns the smallest integer greater than or equal to `x`.
- `copysign(x, y)`: It return the value of parameter `x` with sign of parameter `y`.
- `fabs(x)`: This function returns the absolute value of the number `x`.
- `factorial(x)`: This function returns the factorial of the number `x`, where `x` ≥ 0 .
- `floor(x)`: It is the largest integer that is less than or equal to the number `x`.

- `fsum(itobject)`: It determines the sum of all the elements in iterable object *itobject*.
- `gcd(x, y)`: It finds the greatest common divisor (GCD) of the numbers *x* and *y*.
- `isfinite(x)`: It returns True if the *x* is neither an infinity nor a nan.
- `isinf(x)`: It returns True if *x* is infinity.
- `isnan(x)`: It returns True if *x* is not a number.
- `remainder(x, y)`: It returns the remainder after dividing the number *x* by number *y*.

A demonstration of the above methods/constants is given in the following code.

```
import math
radius=5.8
area=math.pi*radius*radius
print('Area of the circle: ', area)
print('Ceil of area is: ', math.ceil(area))
print('Floor of area is: ', math.floor(area))
print('Absolute value of -25 is: ', math.fabs(-25))
print('Factorial of 5 is: ', math.factorial(5))
print('Remainder of 27 divide by 5 is: ', math.remainder(27,5))
print('GCD of 54 and 243 is: ', math.gcd(54,243))
print('Copysign: ', math.copysign(38,-15))

l=(2,8,5,3)
print('Sum of the elements of l is: ', math.fsum(l))
print('The number 20000000 is finite',math.isfinite(20000000))
print('The number 20000000 is infinite',math.isinf(20000000))
print('20000000 is non number',math.isnan(20000000))
```

Output:

```
Area of the circle: 105.68317686676063
Ceil of area is: 106
Floor of area is: 105
Absolute value of -25 is: 25.0
Factorial of 5 is: 120
Remainder of 27 divide by 5 is: 2.0
GCD of 54 and 243 is: 27
```

Copysign: -38.0
Sum of the elements of l is: 18.0
The number 20000000 is finite True
The number 20000000 is infinite False
20000000 is non number False

9.5 STRING module

A string is a sequence of characters. But computers deal with numbers only and not with the characters. The characters are converted to numbers. This conversion is called encoding and its reverse process is called decoding. ASCII and Unicode are two popular encodings techniques used for characters. In Python, Unicode encoding is used for characters. A string is a sequence of Unicode symbols. As discussed in previous units, a string in Python can be created using single, double, and triple quotes.

The `string` module in Python standard library provides a number of constants, functions, and classes for string manipulation. The constants in `string` module can be used to specify the categories of strings. The classes and functions in `string` module allow to create and customize the string formats. The constants defined in this module are as follows.

- `string.ascii_letters` All ASCII lowercase and uppercase characters.

abcdefghijklmnopqrstuvwxyzABCDEFGH
IJKLMNOPQRSTUVWXYZ
- `string.ascii_lowercase` All ASCII lowercase characters.

abcdefghijklmnopqrstuvwxyz
- `string.ascii_uppercase` All ASCII uppercase characters.

ABCDEFGHIJKLMNOPQRSTUVWXYZ
YZ
- `string.digits` Digits from 0 to 9.

0123456789
- `string.hexdigits` Hexadecimal digits.

0123456789abcdefABCDEF
- `string.whitespaces` Whitespace characters.

`'\t\n\r\x0b\x0c'`

- `string.punctuation`

All punctuation marks.

`!"#$%&'()*+,-./:;<?@[\\]^_`{|}~`

The important functions in string module are given here.

- `capitalize()`

This function converts the first character to uppercase.

- `casefold()`

It converts the entire string into lowercase.

It returns a space padded centered string.

- `center(width, fillchar)`

Where width: width of string, fillchar: the filler character

- `count(sub, start, end)`

It returns the number of times a substring 'sub' occurs in a string. Where start: start index, end: end of index.

- `decode(encoding, errors)`

Decodes the string using codec registered for encoding.

- `encode(encoding, errors)`

It returns an encoded version of the string.

- `endswith(suffix, beg, end)`

It returns True if the string ends with the specified value.

- `expandtabs(tabsize)`

This function is used to set the tab size of the string.

- `find(str, beg, end)`

It returns the position of a specified substring 'str' in the string.

- `index(str, beg, end)`

It returns the position of a specified substring in the string. Similar to find() but raises an exception if substring is not found.

- `isalnum()`

It returns True if all characters in the string are alphanumeric.

- `isalpha()`

It returns True if all characters in the string are in the alphabet.

• <code>isdecimal()</code>	It returns True if all characters in the string are decimals.
• <code>isdigit()</code>	It returns True if all characters in the string are digits.
• <code>isidentifier()</code>	It returns True if the string is an identifier.
• <code>islower()</code>	It returns True if all characters in the string are lower case.
• <code>isnumeric()</code>	It returns True if all characters in the string are numeric.
• <code>isprintable()</code>	It returns True if all characters in the string are printable.
• <code>isspace()</code>	It returns True if all characters in the string are whitespaces.
• <code>istitle()</code>	It returns True if the string follows the rules of a title.
• <code>isupper()</code>	It returns True if all characters in the string are upper case.
• <code>join(seq)</code>	It concatenates the elements in a sequence 'seq' with a separator.
• <code>len(string)</code>	It returns the length of the string.
• <code>ljust(width, fillchar)</code>	It is used to obtain a space padded left justified string.
• <code>lower()</code>	This function converts a string into lowercase.
• <code>lstrip()</code>	It is used to left trim a string.
• <code>maketrans()</code>	This function provides a translation table for translations.
• <code>replace(old, new)</code>	It returns a string where occurrence of a specified substring 'old' is replaced with another substring 'new'.
• <code>rfind(str, beg, end)</code>	It is same as find but works backwards.

• <code>rindex(str, beg, end)</code>	It is same as <code>index</code> but works backwards.
• <code>rjust(width, fillchar)</code>	It gives a right justified string.
• <code>rstrip()</code>	This function right trims the string.
• <code>split(sep, num)</code>	It splits the string at the specified separator 'sep'.
• <code>splitlines(num)</code>	It splits the string at line breaks.
• <code>startswith(str, beg, end)</code>	It returns True if the string starts with a specified value.
• <code>strip(chars)</code>	It trims the string.
• <code>swapcase()</code>	It causes the lower case to become upper case and vice versa.
• <code>title()</code>	It converts the first character of each word to uppercase.
• <code>translate(table, chars)</code>	It returns a translated string according to translation table.
• <code>upper()</code>	It converts a string into uppercase.
• <code>zfill(width)</code>	It fills the string with a specified number of zeros at the beginning.

The following code demonstrates the usage of some of the above functions in a Python program.

```
str = 'this is a sample string'
print('Original string: ', str)
print('Capitalize first character: ',str.capitalize())
print('Capitalize entire string: ',str.upper())
print('String padded with * and centered:\n',str.center(40, '*'))
print("Count of 'is' in string: ",str.count('is'))
print("Position of substring 'sample' in string: ",str.find('sample'))
print('Is string a title case?: ',str.istitle())
print('Is string upper case?: ',str.isupper())
print('Length of the string: ',len(str))
print("Replace 'sample' with 'good': ",str.replace('sample', 'good'))
```

```

str2 = 'Another STRING example'
print('Original string 2: ', str2)
print('Swap case: ',str2.swapcase())
print('Lower case: ',str2.lower())
seq = ('hello', 'welcome', 'home')
sep = '-'
print('Example of join: ', sep.join(seq))

```

Output:

```

Original string: this is a sample string
Capitalize first character: This is a sample string
Capitalize entire string: THIS IS A SAMPLE STRING
String padded with * and centered:
*****this is a sample string*****
Count of 'is' in string: 2
Position of substring 'sample' in string: 10
Is string a title case?: False
Is string upper case?: False
Length of the string: 23
Replace 'sample' with 'good': this is a good string
Original string 2: Another STRING example
Swap case: aNOTHER string EXAMPLE
Lower case: another string example
Example of join: hello-welcome-home

```

Check your progress

1. Write important constants provided by MATH module with their values.
2. What is the use of floor() function?
3. Which function can be used to find out remainder in division operation?
4. Write two popular encoding schemes for characters.
5. Explain the use of string.ascii_lowercase, string.whitespaces, and string.hexdigits.
6. How do you find out length of a string?

7. How do you find out the position of substring in a given string?
8. How do you find out if all the characters in a string are in lowercase?
9. Differentiate `find()` and `rfind()`.
10. Explain the use of `zfill()`.

9.6 Internet Access

In Python, a module named `urllib` provides the access to Internet. The `urllib` is a package that contains several other modules for working with uniform resource locators (URLs). These modules allow a variety of operations on URLs such as open, fetch, parse, and exception handling, etc. Some of the important modules in `urllib` package are as follows.

- `urllib.request`
- `urllib.parse`
- `urllib.error`
- `urllib.response`
- `urllib.robotparser`

The `urllib.request` module defines several classes and functions for opening URLs using a variety of protocols. It can fetch URLs using network protocols such as FTP, HTTP, and HTTPS, etc. It also offers options for handling authentication, cookies, and proxies, etc. The URLs can be fetched using a function `urlopen`. For example, the following code can fetch the url of the UP Rajarshi Tondon Open University.

```
import urllib.request

request_url = urllib.request.urlopen("https://www.uprtou.ac.in/")
```

The data of the fetched URL can be read that one can print, save, or perform other operations as follows.

```
import urllib.request

request_url = urllib.request.urlopen("https://www.uprtou.ac.in/")
print(request_url.read())
```

Here the data of the fetched URL is read and then printed. But it prints the data in byte format. In order to print the data in text format, it should be decoded before printing as follows.


```
import urllib.request
request_url = urllib.request.urlopen('https://www.uprtou.ac.in/')
print(data.decode().request_url.read())
```

There are many other methods in `urllib.request` module for performing many other different operations. Some methods are discussed as follows.

- `urllib.request.install_opener(opener)` : This method helps to install an opener if someone wants `urlopen()` to use that opener.
- `urllib.request.build_opener(handler, ...)`: It returns an `OpenDirector` instance.
- `urllib.request.pathname2url(path)`: This function converts the path name to the form used in the path component of a URL. However, it does not produce a complete URL.
- `urllib.request.url2pathname(path)`: It converts the path component from a URL to the local syntax path.
- `urllib.request.getproxies()`: This method returns a dictionary of scheme to proxy server URL mappings.

In addition to methods, there are a number of classes in `urllib.request` module that help to perform Internet related operations. The instances of these classes can deal with managing requests, handling errors and exceptions.

The `urllib.parse` module defines a number of methods and classes to manipulate the URLs and their components such as network location, addressing scheme, and path, etc. This module can be used for splitting a URL into small components as well as joining different components back into a URL string and converting a relative URL into an absolute URL. It supports

the most of the standard URL schemes. Some of the important functions of this module are briefly explained here.

- `rllib.parse.urlparse(urlstring)`: This function returns a six-tuple containing six components of a URL after splitting the given URL.
- `rllib.parse.unparse(parts)`: It constructs a URL from a tuple.
- `rllib.parse.urlsplit(urlstring)`: This function converts the path name to the form used in the path component of a URL. However, it does not produce a complete URL.
- `rllib.parse.urlunsplit(parts)`: It converts the path component from a URL to the local syntax path.
- `rllib.parse.urljoin(base, url, fragments)`: This method returns a dictionary of scheme to proxy server URL mappings.

The `urllib.error` module in `urllib` package defines some classes to handle the exceptions raised by `urllib.request`. There are two major exceptions that arise during the URL fetching: `URLError` and `HTTPError`. The `urllib.error` module helps to handle these exceptions. `URLError` exception or error is raised if there are some connectivity issues while fetching the URL. `HTTPError` is a subclass of `URLError` and it is raised due to exotic HTTP error. The `urllib.robotparser` module contains only one class that helps to identify whether a particular agent can fetch a URL. The class provides methods to read, parse, and answer the questions.

9.7 Date and Time

The Python standard library consists of a module named `datetime` that provides various classes for manipulating date and time. In Python, date and time are not built-in data types but programmers can work with date or time with the help of `datetime` module. Both date and time are considered as objects in Python. There are various classes in `datetime` module that supply a number of functions to work with date and time. The major classes in `datetime` module are *date*, *time*, *datetime*, *timedelta*, *tzinfo*, and *timezone*.

The object of *date* class represents the date in YYYY-MM-DD format. The *date* objects are considered as naïve objects not having complete information about date. The programmers need to manage the desired information related to date. The constructor of *date* class takes three arguments to create an object given as follows.

```
class datetime.class(year, month, day)
```

All the three arguments are integers with specified ranges. If any argument is non-integer or outside the range then errors are raised. The function `today()` in *date* class returns the current date. Let us consider the following example.

```
from datetime import date
dt = date(2020, 12, 26)
print('Date is: ', dt)
print('Current date: ', dt.today())
```

Output:

Date is: 2020-12-26

Current date: 2021-07-09

There is another popular class *time* in *datetime* package that provides various functions for operations related to time. The constructor of *time* has the following form.

```
class datetime.time(hour, minute, second, microsecond, tzinfo, fold)
```

All the arguments are of integer type and optional. Each argument is having its own range. The programmers can format the time as per requirements. There is an independent *time* (different from *datetime.time* module) module in Python that offers several other functions as briefly described here. It is an in-built module but some of these functions may not be available on a particular platform.

time.altzone

It is used to set local time zone.

time.asctime(tupletime)

It accepts a time-tuple and returns a 24-character string such as 'Sat Jul 10 23:48:35 2021'.

time.clock()

It returns current CPU time.

<u><code>time.ctime([secs])</code></u>	It is like <code>asctime()</code> .
<u><code>time.gmtime([secs])</code></u>	It returns a time-tuple with the UTC time.
<u><code>time.localtime([secs])</code></u>	It returns a time-tuple with the local time.
<u><code>time.mktime(tupletime)</code></u>	It returns a value with the instant expressed in seconds since the epoch.
<u><code>time.sleep(s)</code></u>	It suspends the calling thread for 's' seconds.
<u><code>time.strftime(fmt[,tupletime])</code></u>	It returns a string representing the instant as specified by string 'fmt'.
<u><code>time.strptime(str,fmt)</code></u>	It parses the string 'str' and returns the instant in time-tuple format.
<u><code>time.time()</code></u>	It returns the current time instant in seconds since the epoch.
<u><code>time.tzset()</code></u>	It is used to resets the time conversion rules.

Let us consider following example.

```
from datetime import time
t = time(23, 35, 42)
print('Time is: ', t)
```

Output:

Time is: 23:35:42

```
import time
t = time.localtime(200)
print('Local time: ',t)
print('Time in seconds: ', time.mktime(t))
print('Curent time since last epoch: ', time.time())
```

Output:

Local time: time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=5, tm_min=33, tm_sec=20, tm_wday=3, tm_yday=1, tm_isdst=0)
Time in seconds: 200.0

Check your progress

1. Write names of modules in `urllib` package.
2. How do you fetch a URL?
3. How do you get proxy server URL mappings?
4. How do you split a URL in parts?
5. How do you convert parts of a URL to local syntax?
6. What are the major classes in `datetime` module?
7. What format of data is used by date class objects?
8. How do you get current date?
9. Write syntax of the constructor of the time class.
10. How do you get local time?

9.8 Calendar

Python provides an in-built module `calendar`, which is similar to 'cal' command of UNIX/LINUX with some additional functions. These functions are used to change the appearance of the calendar and perform related operations. But it can be changed using the available functions. The functions and classes in this module use Georgian Calendar that can be extended in both directions indefinitely. By default, the first day in calendar is Monday and the last day is Sunday. Some of the important functions are briefly discussed here.

<code>iterweekdays()</code>	It returns an iterator for the week day numbers for one week
<code>itermonthdates()</code>	It returns an iterator for the month in a year. All days returned as <code>datetime.date</code> objects.
<code>itermonthdays()</code>	It returns an iterator for a specified month and a year. All days are returned as numbers.
<code>itermonthdays2()</code>	This method is also used to get an iterator for a month in a year. All days are returned as tuples (day of month, day of week).
<code>itermonthdays3()</code>	This function returns an iterator for a month in a year but not restricted by the <code>datetime.date</code> range. All days are returned as tuples (year, month, day of

month), .

`itermonthdays4()`

It returns an iterator for a month in a year but not restricted by the `datetime.date` range (year, month, day of month, day of week).

`monthdatescalendar()`

It returns a list of weeks in a month of a year as full weeks. Weeks are `datetime.date` objects.

`monthdays2calendar()`

It is also used to get a list of weeks in a month of a year as full weeks. Weeks are tuples.

`monthdayscalendar`

It is also used to get a list of weeks in a month of a year as full weeks. Entries are list of day numbers.

`yeardatescalendar()`

It is also used to get a list of weeks in a month of a year as full weeks. It returns list of month rows. Entries are list of month rows.

`yeardays2calendar()`

It is used to get the data for specified year. Entries are list of day numbers. Entries are list of tuples.

`yeardayscalendar()`

It is used to get the data for specified year.

Let us consider some examples.

```
import calendar
print(calendar.month(2021,7))
```

Output:

```
July 2021
Mo Tu We Th Fr Sa Su
    1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

```
import calendar
print(calendar.calendar(2021,2,1,6))
```

Output:

2021

January

February

March

Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su						
1 2 3 1 2 3 4 5 6 7 1 2 3 4 5 6 7																				
4 5 6 7 8 9 10 8 9 10 11 12 13 14 8 9 10 11 12 13 14																				
11 12 13 14 15 16 17 15 16 17 18 19 20 21 15 16 17 18 19 20 21																				
18 19 20 21 22 23 24 22 23 24 25 26 27 28 22 23 24 25 26 27 28																				
25 26 27 28 29 30 31 29 30 31																				
April							May							June						
Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su						
1 2 3 4							1 2 1 2 3 4 5 6													
5 6 7 8 9 10 11 3 4 5 6 7 8 9 7 8 9 10 11 12 13																				
12 13 14 15 16 17 18 10 11 12 13 14 15 16 14 15 16 17 18 19 20																				
19 20 21 22 23 24 25 17 18 19 20 21 22 23 21 22 23 24 25 26 27																				
26 27 28 29 30 24 25 26 27 28 29 30 28 29 30																				
31																				
July							August							September						
Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su						
1 2 3 4							1 1 2 3 4 5													
5 6 7 8 9 10 11 2 3 4 5 6 7 8 6 7 8 9 10 11 12																				
12 13 14 15 16 17 18 9 10 11 12 13 14 15 13 14 15 16 17 18 19																				
19 20 21 22 23 24 25 16 17 18 19 20 21 22 20 21 22 23 24 25 26																				
26 27 28 29 30 31 23 24 25 26 27 28 29 27 28 29 30																				
30 31																				
October							November							December						
Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su							Mo Tu We Th Fr Sa Su						
1 2 3 1 2 3 4 5 6 7 1 2 3 4 5																				
4 5 6 7 8 9 10 8 9 10 11 12 13 14 6 7 8 9 10 11 12																				
11 12 13 14 15 16 17 15 16 17 18 19 20 21 13 14 15 16 17 18 19																				
18 19 20 21 22 23 24 22 23 24 25 26 27 28 20 21 22 23 24 25 26																				
25 26 27 28 29 30 31 29 30 27 28 29 30 31																				

9.9 Data Compression

Python provides many modules to perform data compression and archiving. The most popular modules are **zlib**, **gzip**, **bz2**, **lzma**, **zipfile**, and **tarfile**. These modules are named after the compression algorithm they use. Using the methods and classes available in these modules, the ZIP or tar-format archives can be created.

Python **zlib** library provides an interface to C **zlib** library that is based on DEFLATE lossless compression algorithm. There are several functions in the library to support compression and decompression operations. A program demonstrating compression and decompression operations is given here.

```
import zlib
import sys
str = b'a'*45 #The compress() functions takes Byte object
print('Size of the string: ', sys.getsizeof(str))
#Do compression
cstr = zlib.compress(str)
print('Size after compression: ', sys.getsizeof(cstr))
```

```
#Decompress it
dstr = zlib.decompress(cstr)
print('Size after decompression: ', sys.getsizeof(dstr))
```

Output:

```
Size of the string: 78
Size after compression: 45
Size after decompression: 78
```

Other compression modules also work in the same way. All of them include `compress()` and `decompress()` functions. Different modules are based on different compression algorithms. In addition to compression/decompression functions, there are some other functions for some other basic operations such as reading/writing data, open/close files, and `flush()` the buffer etc.

9.10 Summary

A library is a collection of modules containing many useful functions that can be used in other programs. The libraries eliminate the need of writing the entire code from scratch. The frequently used codes are included in the libraries. The Python standard library is an extensive collection of standard methods, modules, and data elements that is provided with Python itself. The built-in modules in standard library are written in C programming language, which provide solution to many standard problems. There are more than 200 modules in the standard library. The OS module is one of the Python standard utility module that allows interaction between user and the operating system. It provides a number of functions to perform operating system related tasks. The MATH module provides many useful functions to perform a number of mathematical operations. All the functions in this module work on integer or floating point numbers only. The string module in Python standard library provides a number of constants, functions, and classes for string manipulation. The constants in string module can be used to specify the categories of strings. The classes and functions in string module allow to create and customize the string formats. In Python, a module named `urllib` provides the access to Internet. The `urllib` is a package that contains several other modules for working with URLs. These modules allow a variety of operations on URLs such as open, fetch, parse, and exception handling, etc. The Python standard library consists of a module named `datetime` that provides various classes for manipulating date and time. In Python, date and time are not built-in data types but programmers can work with date or time with the help of `datetime` module. Both date and time are considered as objects in Python. Python provides an in-built module `calendar`, which is similar to 'cal' command of UNIX/LINUX with

some additional functions. These functions are used to change the appearance of the calendar and perform related operations. There are many other modules and libraries in Python that make the job of programmers easy.

9.11 Review questions

Q.1 What is library? What are the benefits of having libraries?

Q.2 How is a library used in a Python program?

Q.3 What is Python standard library? What are the major modules in this library?

Q.4 Demonstrate the methods of creating and removing directories using Python programs.

Q.5 Demonstrate the use of different types of mathematical functions in MATH module of standard library.

Q.6 Write five important functions of STRING module in Python and demonstrate their use with the help of Python program.

Q.7 Which modules are useful in the Internet access? Illustrate some important functions with the help of a Python program.

Q.8 Demonstrate the utility of DATETIME module with suitable example.

Q.9 Make use of CALENDAR module in Python program and demonstrate the use of its five functions.

Q.10 How do compress/decompress the data in Python? What are the important data compression modules? Discuss the important functions.

Unit 10: GUI Programming and Testing

Structure

- 10.0 Introduction
- 10.1 Objectives
- 10.2 Multithreading
- 10.3 Graphical user interface
- 10.4 GUI programming with Tkinter
- 10.5 Turtle graphics
- 10.6 Testing
- 10.7 Summary
- 10.8 Review questions

Unit 10: GUI Programming and Testing

10.0 Introduction

This unit is about some advanced concepts in Python programming including multithreading, graphical user interface (GUI), turtle graphics, and testing. It is hard to make efficient utilization of computer resources while performing a single task as CPU remains in idle state for a significant amount of time waiting for some events to occur. In the meantime the ideal resources can be made available for some other task. The multithreading is a technique that is based on performing multiple tasks simultaneously. Python provides a very good API to write multithreaded programs. It allows to create multiple threads in a program in some simple ways.

GUI is a user friendly interface that allows users to interact with the program or software application in an easy manner. Python provides multiple options to write GUI based programs. Tkinter is one of the most commonly used method to create GUIs in Python. It offers many different types of widgets that can be placed throughout widget panel in different arrangements. A detailed discussion on Tkinter widgets and their usage is presented in this unit.

Python offers a kind of preinstalled virtual canvas known as Turtle graphics. It allows to create graphical shapes using combinations of a number of commands. Turtle graphics provides a graphics screen with a space for drawing the figures. There are many commands that are used make desirable figure within the given space on the screen. These commands allow the programmer to move the drawing pen, set the colors, fill up the colors, etc.

Software testing is highly important for identifying bugs and errors in the programs to ensure the quality of the product. There are many different techniques for software testing with a different objectives, applicability, and usage. The most of the important testing techniques are discussed in this unit. *Unit* testing is one of the important and basic testing technique that is used to test each independent unit of the program separately. In this unit, a discussion is given on performing *unit* testing in Python.

10.1 Objectives

There are multiple objectives of this unit given as follows.

1. To provide an overview of the multithreading and to introduce the methods of writing multithreading based programs in Python.
2. To introduce the tools and methods for writing Python programs to create GUI based applications.
3. To introduce Turtle graphics library for creating graphical figure with the help of Python programs.

4. To discuss the testing techniques and to discuss how to test programs in Python.

10.2 Multithreading

Multithreading aims at better utilization of CPU or CPU cores by sharing the resources such as computing units, memory, and translation lookaside buffer, etc. among multiple threads of execution. A thread of execution is not a standalone process, instead it is a kind of sub-process that is also called lightweight process. Although any threads are not dependent on other threads but they share the resources of same process. With the help of multithreading, multiple tasks can be performed simultaneously leading to faster execution. However, it requires a careful programming to avoid any kind of conflicts.

A program is a sequence of instructions that when executed forms a process. Within a program, there could be some subsequences of instructions that can be executed independently. These independent subsequences when executed simultaneously form threads. All the global variables in a program are shared by the threads. In addition, each thread has its own set of local variables, set of registers, program counter, stack pointer, and a thread identity, etc.

Python provides a very good API to write multithreaded programs. The `threading` module provides a class *Thread* that is used to create threads. A thread is an instance of the class *Thread*. The general syntax to create a thread in Python is given as follows.

```
t = threading.Thread(target, args)
```

where, 't' is the thread, 'target' is the function to be executed by the thread and 'args' is the list of arguments to be passed to the function. There are two important methods associated with *Thread* class: `start()` and `join()`. The `start()` method is used to start the execution of the thread 't' along with the current program as follows.

```
t.start()
```

The `join()` method stops the execution of current program until the execution of thread 't' is over.

```
t.join()
```

Let us consider an example,

```
import threading
```

```
def fact(n):  
    n = int(n)  
    factorial = 1  
    if n < 0:
```

```

    factorial = 'Error'
if n >= 1:
    for i in range(1, int(n)+1):
        factorial = factorial * i
print("\nFactorial: {}".format(factorial))

def cube(n):
    v = n*n*n
    print("\nCube: {}".format(v))

if __name__ == "__main__":
    # creating threads
    thread1 = threading.Thread(target=fact, args=(5,))
    thread2 = threading.Thread(target=cube, args=(5,))
    # start thread 1
    thread1.start()
    # start thread 2
    thread2.start()
    # wait for thread 1 to completely executed
    thread1.join()
    print('Thread 1 done')
    # wait for thread 2 to completely executed
    thread2.join()
    print('Thread 2 done')
    # Execution of both threads are completely executed
    print('Parent program done')

```

Here, two threads 'thread1' and 'thread2' are created as objects of class *Thread*. The 'thread1' calls the method 'fact()' and 'thread2' calls the method 'cube()'. The threads are started with the 'start()' method. The output of the program is given here.

```

Factorial: 120
Cube: 125

Thread 1 done
Thread 2 done
Parent program done

```

This is one of the easiest way to write multithreading based programs in Python. There are some other useful methods in *threading* module as briefly described here.

- *threading.activeCount()*: It returns the number of currently active thread objects.

- `threading.currentThread()`: It returns the count of thread objects in caller thread control.
- `threading.enumerate()`: It returns the list of currently active thread objects.

The *Thread* class in *threading* module provides some more methods.

- `run()`: It is the entry point of the thread, which is overridden during the implementation of a thread.
- `getName()`: This method returns the name of a thread.
- `setName()`: This method can be used to set the name of a thread.
- `isAlive()`: This method is used to check whether a thread is still executing.

Check your progress

1. What is thread?
2. What is the difference between a thread and a process?
3. Can global variables be shared by threads?
4. Which Python module is used to create threads in a program?
5. What is *Thread* class?
6. How a thread is started?
7. How is the parent program kept in waiting until a thread is terminated?
8. How is a method associated with a thread?

10.3 Graphical User Interface

Graphical user interface or more popularly known as GUI is an interface that allows users to interact with computers and other electronic devices using small pictures called icons or widgets and menus on screen. GUI provides user friendly access to the devices. The earlier command line interfaces were quite complicated as it was difficult to learn a large number commands their syntax. Unlike command line interface, a GUI program receives user input from icons and widgets. The users interact with icons and widgets on screen using input devices such as mouse, stylus pen, or finger if it is a touch screen. The order of tasks is completely under the control of users. This style of programming is known as event driven programming. The GUI programs are event driven programs.

Python provides multiple ways of writing GUI based programs. The major options are as follows.

- **Tkinter** Tkinter is the most commonly used method for GUI development. It is an interface to Python Tk GUI toolkit that is a standard library shipped with Python.
- **wxPython** The wxPython GUI toolkit is a Python wrapper around

a C++ library wxWidgets. It is easy to work with wxPython but Tkinter is powerful and fast.

- **JPython** JPython provides seamless access to Java class libraries on the local machine. It is freely available for commercial and non-commercial use.
- **PyQt** GUI programming with PyQt is based on the communication amongst objects. It provides flexibility to deal with GUI events with the help of native platform APIs for networking and database.

In this unit, basic concepts of GUI programming with Tkinter are discussed.

10.4 GUI Programming with Tkinter

Tkinter offers an object oriented interface to Tk GUI library in Python. It provides a fast and powerful way creating GUI applications. Both Tk and Tkinter are available with most Unix, Linux, and Windows platforms. The general steps to create a GUI application with Tkinter are as follows.

1. Import Tkinter module i.e. `import tkinter`
2. Create main window of GUI application or container
3. Add widgets to the main application/window
4. Apply the main event loop to widgets that takes action against the events triggered by the user.

There are two main methods needed for creating the GUI application in Python: `Tk()` and `mainloop()`. The method `Tk()` is used to create main window of the GUI. It offers several options to customize the window. The method `mainloop()` creates an infinite loop that waits for an event to occur and take action accordingly as long as window is not closed. All the widgets are placed in between these two methods. The general approach for creating a GUI window can be given as follows.

```
import tkinter as tk
window = tk.Tk()
...
#add widgets here
...
window.mainloop()
```

Tkinter provides many different types of widgets including button, check button, list box, menu button, menu, message, radio button, etc. In addition, there are some methods for the geometry management of the window. These methods help to place the widgets throughout widget panel.

- **pack() method:** It organizes the widgets in block that can be placed

- **grid() method:** into parent widget. This method organizes the widgets in a grid in parent widget.
- **place() method:** It is used to place widgets at specific positions given by the programmer.

Some of the most important widgets are discussed here one by one.

10.4.1 Button

This widget is used to add a button to the GUI application. Usually the name conveying the purpose of the button can be displayed on the button. With each button a method or function is attached that is called automatically when user clicks the button. The button is created using the method `Button()`. The general syntax of the method is as follows.

`b = Button(window_name, option=value, ...)`

Where `window_name` is the name of main window object. The format of the button can be set with the help of a number options. Some frequently used options are explained here.

- **activebackground:** Background color when button is under the cursor.
- **activeforeground:** Foreground color when button is under the cursor.
- **bg:** Normal background color.
- **fg:** Normal foreground color.
- **bd:** Border width.
- **font:** Text font type on the button label.
- **image:** Image on the button.
- **width:** Width of the button.
- **height:** Height of the button.
- **command:** To call a function.
- **state:** State of the button: **DISABLE** or **ENABLE**
- **relief:** Type of the border.

A sample code is given here.


```

import tkinter as tk
window = tk.Tk()           #Create main window
window.title('Button Example') #Title of the window
window.geometry('400x200')  #Dimensions of the window
window['bg']='#FCE5CD'      #Background color of the window
#Create button widget
button = tk.Button(window, text='Exit', width=15, command=window.destroy)
#Place the button at specified position in window
button.place(x=140, y=25)
#Create infinite loop waiting for events
window.mainloop()

```

In the above code, first of all a blank main window is created with the help of function Tk() provided by Tkinter library. Later, window name, background color, and window size are specified. The color can be specified either by color name such as 'red', 'green', 'blue' or by HEX equivalent value of the color. As in the above code, the color is specified by HEX value 'FCE5CD'. After creating the main window, the button widget is created by method Button(). The widget is finally placed at specified position in the window. The GUI generated as output of the above program is given in Figure 10.1.

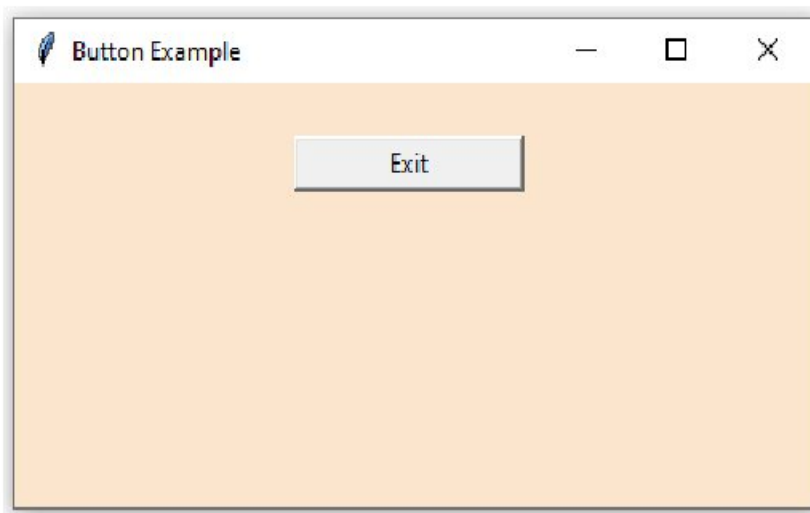


Figure 10.1: Sample window with a button

When user clicks on the button 'Exit' the window is simply destroyed without any operation as it is specified by the option 'command' in the program. However, some method or function can be associated with the button by replacing 'command=window.destroy' by 'command=function_name'. Here, 'function_name' is the name of the function/method intended to be called with a click on the button.

10.4.2 Checkbutton

The Checkbutton widget is used to provide multiple options to the user as toggle buttons. The user can select one or more options by clicking the buttons corresponding to the chosen options. A check button is different than a radio button, which allows only one selection. Each check button has two states on or off. Usually check buttons are displayed as square boxes containing white space or a tick mark. A tick mark means selected or checked. A caption or image describing the meaning of the check button is placed adjacent to check button. The check button are created with Checkbutton() method whose general syntax is given as follows.

```
cb = Checkbutton(window_name, option=value, ...)
```

The major options for Checkbutton() are as follows.

- | | |
|---------------------|---|
| • activebackground: | Background color when check button is under the cursor. |
| • activeforeground: | Foreground color when check button is under the cursor. |
| • bg: | Normal background color. |
| • bitmap: | For monochrome image on a button. |
| • bd: | Size of the border around the indicator. The default size is 2 pixels. |
| • command: | To call a function every time the state of check button is changed. |
| • cursor: | Mouse cursor changes to the specified pattern when it is over the check button. |
| • font: | Text font type. |
| • fg: | Normal foreground color. |
| • height: | Number of lines of text on the check button. |
| • image: | Display an image on the check button. |
| • justify: | Justify text- CENTER, LEFT, or RIGHT. |
| • offvalue: | A value for the off state. Default value 0. |
| • onvalue: | An alternate value for the on state. Default value is 1. |
| • relief: | Style type. Default is FLAT |

- **selectcolor:** Color of the check button when it is set. Default is 'red'.
- **selectimage:** Set image that appears when check button is set.
- **state:** State of the check button. Any of NORMAL, DISABLE, and ACTIVE
- **text:** Label next to the checkbutton.
- **width:** Width of the check button.
- **variable:** A variable that tracks current state of the check button. Usually it is IntVar.

Let us consider the following sample code for creating check buttons in the same main window used in the previous example.

```
import tkinter as tk
from tkinter import *
window = tk.Tk()      #Create main window
window.title('Sample GUI') #Title of window
window.geometry('400x200') #Size of window
window['bg']='#FCE5CD' #Background color of window
button = tk.Button(window, text='Exit', width=15, command=window.destroy)
button.place(x=140, y=25)
var1 = IntVar()
#Create check button
cb1 = Checkbutton(window, text='Travel', bg='#FCE5CD', variable=var1,\
                  onvalue=1, offvalue=0,height=3).grid(row=0, sticky=W)
var2 = IntVar()
#Another check button
cb2 = Checkbutton(window, text='Sports', bg='#FCE5CD', variable=var2,\
                  onvalue=1, offvalue=0,height=3).grid(row=1, sticky=W)
var3 = IntVar()
#One more check button
cb3 = Checkbutton(window, text='Music', bg='#FCE5CD', variable=var3,\
                  onvalue=1, offvalue=0,height=3).grid(row=2, sticky=W)
window.mainloop()
```

The above code is the modified version of the previous window with three check buttons added to it. In the existing code, additional code is added to create three check buttons cb1, cb2, and cb3 labelled as 'Travel', 'Sports', and 'Music' respectively. All the three check buttons are arranged in grid format

using the method `grid()`. The window is renamed as 'Sample GUI'. The output of the code is displayed in Figure 10.2.

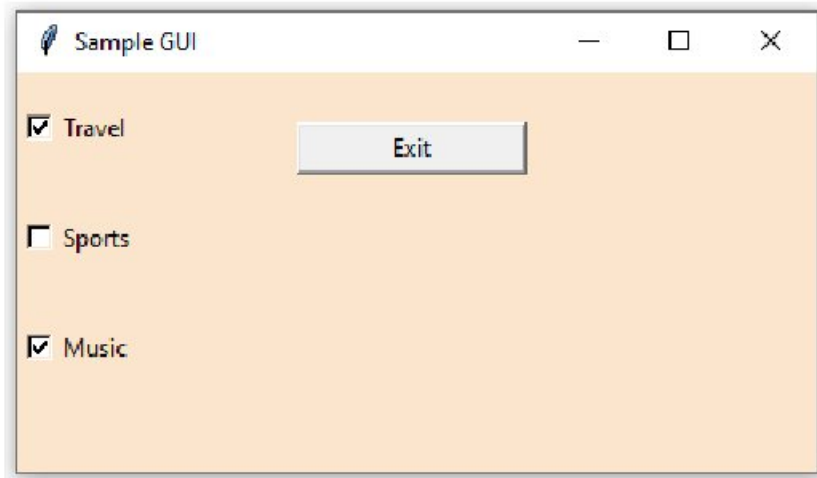


Figure 10.2: Sample GUI with check buttons.

10.4.3 Menu

This widget is used to create all kind of menus in the GUI application. A menu provides several options to execute a program function. Menus are the most commonly used widgets in GUI based software applications. There are three menu types that can be created: pop-up, top level, and pull down or drop down. A pop-up menu appears above the clicked item to prevent the user from making mistakes. A pop-up menu with only two options is known as a dialogue box. A pull down or drop down menu cause a list to display under an item when clicked by the user. In a top level menu, the items are arranged in a horizontal or vertical bar. Each item on the bar may be different sub menu or button. The general syntax of the menu is as follows.

```
m = Menu(window_name, option=value, ...)
```

The major options for `Menu()` are as follows.

- **activebackground:** background color that appears on a choice when it is under the mouse.
- **activeborderwidth:** width of the border drawn around a choice when it is under the mouse. Default value is 1 pixel.
- **activeforeground:** foreground color that appears on a choice when it is under the mouse.
- **bg:** background color for the choices not under the mouse.
- **bd:** width of the border around all the choices. Default value is 1.

- **cursor**: cursor that appears when the mouse is over a choice.
- **disabledforeground**: color of the text for DISABLED items.
- **font**: default font for text.
- **fg**: foreground color for the choices not under the mouse.
- **postcommand**: used to set procedure that needs to be called.
- **image**: set image on the menu button.
- **selectcolor**: color for check buttons and radio buttons on selection.
- **tearoff**: When **tearoff=0**, the menu does not have a tear-off feature and choices are added starting at position 0.
- **title**: title of the window

There are several methods to perform various operations on menu objects as listed here.

- **add_command(options)**: adds a menu item to the menu.
- **add_radiobutton(options)**: creates a radio button in the menu.
- **add_checkbutton(options)**: creates a check button in the menu.
- **add_cascade(options)**: creates a new hierarchical menu associated to a parent menu
- **add_separator()**: adds a separator line to the menu.
- **add(type, options)**: adds a specific item to the menu.
- **delete(startindex [, endindex])**: deletes the menu items.
- **entryconfig(index, options)**: allows to modify a menu item identified by the index and change its options.
- **index(item)**: returns the index number of the given menu item label.
- **insert_separator(index)**: inserts a new separator at the specified position.
- **invoke(index)**: calls the command callback associated with the choice at specified position.
- **type(index)**: returns the type of the choice specified by index.

An example is given here for creating a sample menu in GUI application. In this example, four submenus are created and associated with a parent menu. These submenus are 'File', 'Edit', 'View', and 'Help'. In each sub menu, there are multiple items. The Python code for this menu based GUI is given as follows

```
import tkinter as tk
from tkinter import *
window = tk.Tk()
```



```

window.title('Sample Menu')           #Main window
window.geometry("400x200")
window['bg']='#FCE5CD'
menu = Menu(window)                   #Parent menu
window.config(menu=menu)
#-----
def welmsg():
    filewin = Toplevel(window)
    button = Button(filewin, text="Welcome")
    button.pack()
#-----File menu-----
filemenu = Menu(menu, tearoff=0)
menu.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New', command=welmsg)
filemenu.add_command(label='Open')
filemenu.add_command(label='Save')
filemenu.add_command(label='Save as')
filemenu.add_command(label='Close')
filemenu.add_separator() #Adding a line separating Exit from other items
filemenu.add_command(label='Exit', command=window.quit)
#-----Edit menu-----
editmenu = Menu(menu, tearoff=0)
menu.add_cascade(label='Edit', menu=editmenu)
editmenu.add_command(label='Undo')
editmenu.add_command(label='Redo')
editmenu.add_command(label='Copy')
editmenu.add_command(label='Cut')
editmenu.add_command(label='Paste')
editmenu.add_command(label='Delete')
#-----View menu-----
viewmenu = Menu(menu, tearoff=0)
menu.add_cascade(label='View', menu=viewmenu)
viewmenu.add_command(label='Zoom')
viewmenu.add_command(label='Full screen')
viewmenu.add_command(label='Show')
#-----Help menu-----
helpmenu = Menu(menu, tearoff=0)
menu.add_cascade(label='Help', menu=helpmenu)
helpmenu.add_command(label='Data types')
helpmenu.add_command(label='Operators')
helpmenu.add_command(label='Data structure')
helpmenu.add_command(label='Indentation')
helpmenu.add_command(label='Machine Learning')
helpmenu.add_command(label='GUI')
helpmenu.add_command(label='Testing')
window.mainloop()

```

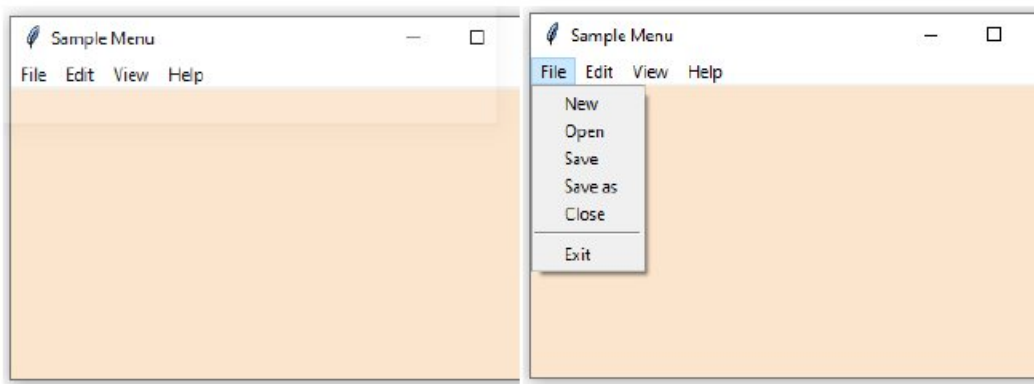
In the above code, first a main window is created then parent menu is created and added to the window. There is a function 'welmsg()', which is defined to be called by some menu items later on. After that submenus are created and associated with parent menu one by one. First, file menu 'filemenu' is created. Several menu items including 'New', 'Open', 'Save', 'Save as', 'Close', and 'Exit' are added to the menu with the help of 'add_command()' method. The output of the above code is shown in Figure 10.3 with the help of several snap shots. Figure 10.3(a) displays the main window with parent menu items 'File', 'Edit', 'View', and 'Help'. Each of these items are submenus with multiple items. The items of each submenu are shown in subsequent sub figures. The functionality of a menu item is defined with the help of a method and that method is called whenever a click is made on the item. Here one such method 'welmsg()' is defined that simply displays 'Welcome' message on a button when it is called. In this sample code, the method 'welmsg()' is called whenever there is click on the 'New' item of the 'File' submenus. It is done with the help of 'command' option. When 'New' item is added to the menu, the method 'welmsg()' is associated by setting 'command' to 'welmsg' as 'command=welmsg'. When user click on 'New' item, the associated method is called and a message is displayed in response as shown in Figure 10.3(f).

10.4.4 Message

Message is an object that is used to automatically display non-editable text. It is similar to label. This widget is created using the following syntax.

```
m = Message(window_name, option=value, ...)
```

There are a number of options similar to other widgets as discussed earlier including 'fg', 'bg', 'bd', 'font', 'image', 'text', 'relief', 'justify', 'height', 'width', 'anchor', 'padx' and 'pady', etc. Let us consider the following examples.



(a)

(b)

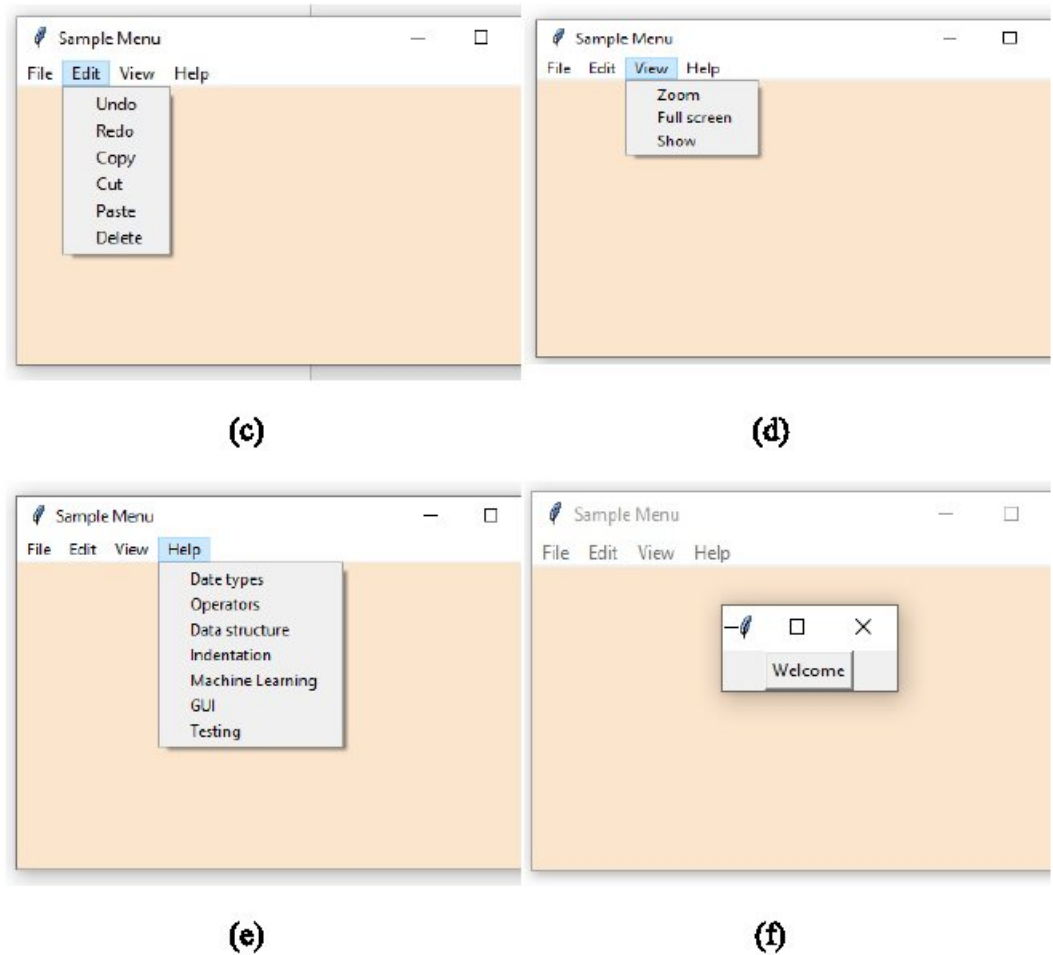


Figure 10.3: Menu creation in GUI

```
import tkinter as tk
from tkinter import *
window = tk.Tk()           #Create main window
window.title('Message Example') #Title of the window
window.geometry("400x200") #Dimensions of the window
window['bg']='#FCE5CD'     #Background color of the window
def msg():
    txtMessage = 'Hello, this GUI is developed in Tkinter'
    msgVar = Message(window, text = txtMessage, width=200)
    msgVar.config(bg='lightblue')
    msgVar.place(x=90, y=60)
#Create button widget
button = tk.Button(window, text='Message', width=15, command=msg)
#Place the button at specified position in window
button.place(x=140, y=25)
#Create infinite loop waiting for events
window.mainloop()
```

In the above example a method 'msg()' is defined where a message widget is created with message 'Hello, this GUI is developed in Tkinter'. The background of the text is set to be light blue. Later a button is created and added to the main window. The message widget is associated with button. Whenever a click is done on the button, the message widget is displayed as a response as shown in Figure 10.4. When code is executed, the main window of the GUI with a button appears as shown in Figure 10.4(a). Whenever user clicks on the button, the associated message widget is displayed with light blue background. The snapshot of the window after the click event is displayed in Figure 10.4(b). The position of the text is specified in the method 'msg()' using 'place()' method of the window. The positioning of the text can also be controlled with the help of 'anchor' option of the Message widget.

10.4.5 Radiobutton

The radio button offers multiple choice to the user. The major difference between check box and radio button is that users can make only one choice with radio button. Whereas check box allows to select multiple options. The Radiobutton widget implements the radio buttons in GUI. The general syntax is as follows.

```
rb = Radiobutton(window_name, option=value, ...)
```

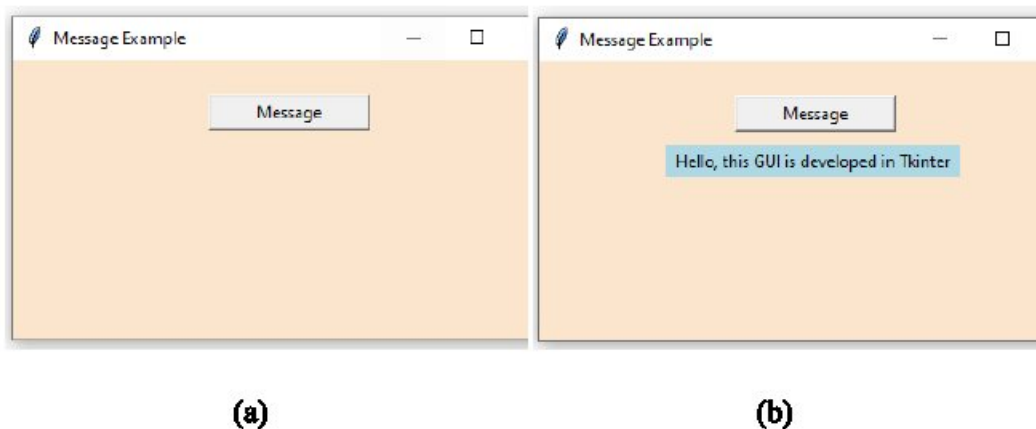


Figure 10.4: Message widget example

There are many options available for this widget similar to other widgets including 'fg', 'bg', 'font', 'image', 'text', 'relief', 'justify', 'height', 'width', 'anchor', 'value', 'relief', 'padx' and 'pady', etc. Let us consider the following example.

```
import tkinter as tk
from tkinter import *
window = tk.Tk()           #Create main window
```

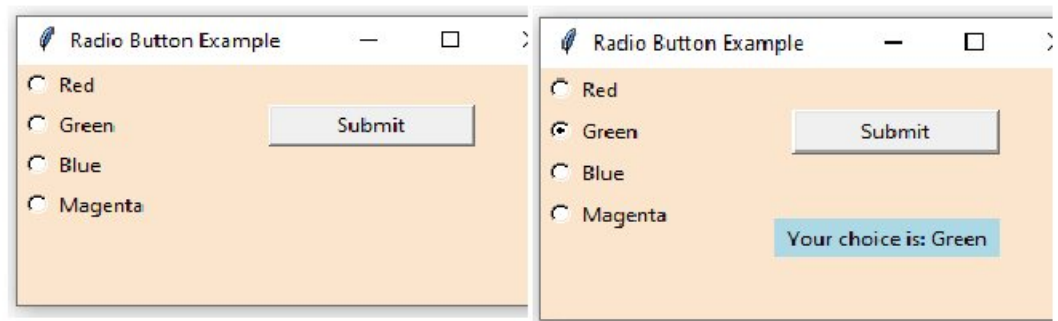


```

window.title("Radio Button Example") #Title of the window
window.geometry("310x150") #Dimensions of the window
window['bg']='#FCE5CD' #Background color of the window
var = StringVar(value='1')
#Method to be called by radio button
def rbsel():
    txtMessage = "Your choice is: " + var.get()
    msgVar = Message(window, text = txtMessage, width=200)
    msgVar.config(bg='lightblue')
    msgVar.place(x=130, y=90)
#Create first radio button
rb1 = Radiobutton(window, text="Red", bg='#FCE5CD', variable=var,
                  value='Red', command=rbsel)

rb1.pack( anchor = W )
#Create second radio button
rb2 = Radiobutton(window, text="Green", bg='#FCE5CD', variable=var,
                  value='Green',
                  command=rbsel)
rb2.pack( anchor = W )
#Create third radio button
rb3 = Radiobutton(window, text="Blue", bg='#FCE5CD', variable=var,
                  value='Blue',
                  command=rbsel)
rb3.pack( anchor = W )
#Create fourth radio button
rb4 = Radiobutton(window, text="Magenta", bg='#FCE5CD', variable=var,
                  value='Magenta',
                  command=rbsel)
rb4.pack( anchor = W )
#Create button widget
button = tk.Button(window, text="Submit", width=15, command=window.destroy)
#Place the button at specified position in window
button.place(x=140, y=25)
#Create infinite loop waiting for events
window.mainloop()

```



(a)

(b)

In the above code, four radio buttons are created. A method is also defined that is called whenever a radio button is selected by the user. The output of the program is shown in Figure 10.5. The main window is shown in Figure 10.5(a), where it can be observed that window consists of four radio button and one button. When user selects an option, the corresponding message is displayed with light blue background as shown in Figure 10.5(b). Here, for simplicity, only a message is displayed as response to selection. But programmers can define any action in the associated method.

10.4.6 Entry

This widget is used to create single line text field to take input from the user. The text entered in the Entry widget is editable. The widget *Label* is used to display no-editable text. *Text* is another widget that is used for taking editable multi-line text input from the user. The general syntax of the Entry widget is as follows.

```
ew = Entry(window_name, option=value, ...)
```

There are many options available for Entry widget including 'fg', 'bg', 'font', 'relief', 'justify', 'width', 'cursor', 'relief', and 'show', etc. Here option 'show' decides how the characters typed by the user appear in the field. By default the characters appear as typed. However in case of password or other requirements the characters entered by the user can be set to appear as asterisk '*' or any other symbol decided by the programmer. Let us consider the following code. The output is shown in Figure 10.6.

```
import tkinter as tk
from tkinter import *
window = tk.Tk()           #Create main window
window.title('Entry Widget Example') #Title of the window
window.geometry('310x150') #Dimensions of the window
window['bg']='#FCE5CD'     #Background color of the window
#Label for the first Entry widget
lb1 = Label(window, bg='#FCE5CD', text="User Name")
lb1.place(x=20,y=20)
#First Entry widget
ew1 = Entry(window, bd=5)
ew1.place(x=100, y=20)
#Label for the second Entry widget
lb2 = Label(window, bg='#FCE5CD', text="Password")
lb2.place(x=20,y=70)
#Second Entry widget. show option is set to display characters as '*'
ew2 = Entry(window, bd=5, show='*')
ew2.place(x=100,y=70)
```

#Create button widget

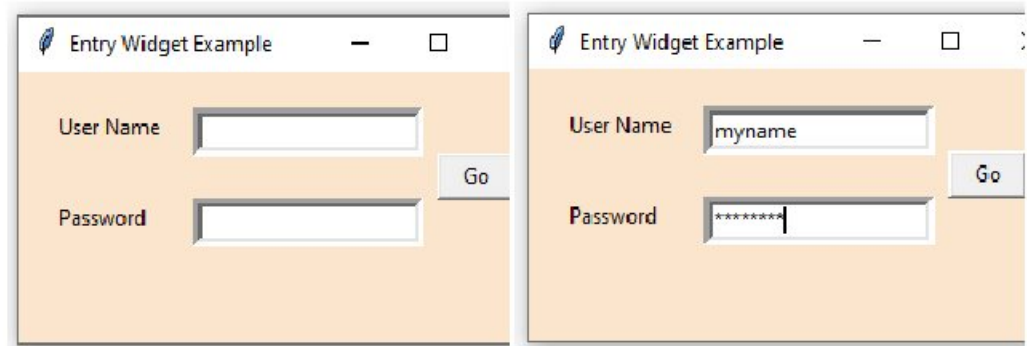
```
button = tk.Button(window, text='Go', width=5, command=window.destroy)
```

#Place the button at specified position in window

```
button.place(x=240, y=45)
```

```
window.mainloop()
```

In this example, two input text fields are created with the help of Entry widget. Each editable text field is labelled using Label widget. One field is labelled as 'User Name' and other one is labelled as 'Password'. For the 'Password' field the 'show' option of the Entry widget is set to '*' to display the input characters as '*' in the field. Whereas in 'User Name' field the characters are displayed as entered by the user.



(a)

(b)

Figure 10.6: Creating text input fields with Entry widget

10.4.7 Listbox

The Listbox widget is used to provide a number of options to the user in the form of a list. It allows the user to select multiple options at a time. The general syntax of the Listbox is as follows.

```
lb = Listbox(window_name, option=value, ...)
```

There are a number of options to format the widget that can be passed as parameter to the method Listbox(). Some important options are 'fg', 'bg', 'font', 'image', 'height', 'width', 'highlightcolor', 'xscrollcommand', 'yscrollcommand', and 'relief', etc. In addition there are several methods available for managing the list box. Some of the important methods are explained here.

- **insert(index, *elements)**- This method is used to insert one or more new elements in the list box before the position specified by the parameter 'index'.

- `delete(first, last)`- As implied by its name, this method is used to delete the items in the range [first, last] from the list box. If 'last' option is not provided then only one item referred by the 'first' index is removed.
- `see(index)`- To adjust the position of the list box so that the element referred to by index is visible.
- `size()`- It returns the number of elements in the list box.
- `get(first, last)`- Returns the elements from the list box in the range specified by indices 'first' and 'last' inclusive. If 'last' option is not provided then only one element referred by the 'first' index is returned.
- `xview()`- To make the list box horizontally scrollable.
- `yview()`- To make the list box vertically scrollable.

Let us consider the following example. The output is shown in Figure 10.7

```
import tkinter as tk
from tkinter import *
window = tk.Tk()           #Create main window
window.title('Listbox Example') #Title of the window
window.geometry("300x220")  #Dimensions of the window
window['bg']='#FCE5CD'      #Background color of the window
lb = Listbox(window, width=20)
lb.insert(1, 'Physics')
lb.insert(2, 'Chemistry')
lb.insert(3, 'Mathematics')
lb.insert(4, 'Data Science')
lb.insert(5, 'Artificial Intelligence')
lb.insert(6, 'Machine Learning')
lb.insert(7, 'Image Processing')
lb.insert(8, 'Data Visualization')
lb.insert(9, 'Big Data')
lb.insert(10, 'Python')
lb.place(x=20,y=20)
#Create button widget
button = tk.Button(window, text='Submit', width=10, command=window.destroy)
#Place the button at specified position in window
button.place(x=200, y=90)
window.mainloop()
```

In this example a list box is created with `Listbox()` method and later several elements are added to the list with the help of `insert()` method. If the list of items is too large to fit a window, then horizontal or vertical scroll options can be added to the list box with respective methods and commands.

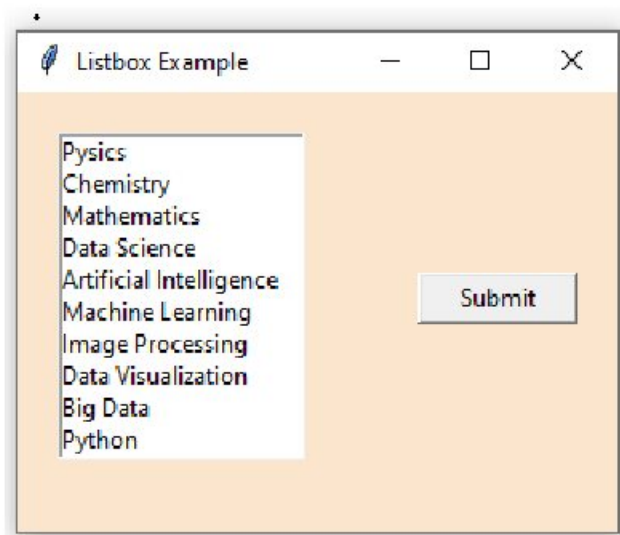


Figure 10.7: List box example.

10.4.8 Combobox

A Combobox is a combination of Entry and Listbox widgets in Tkinter that allows a user to choose from a number of options. This widget contains a down arrow. The list of items remain hidden until the user clicks on the arrow. The available options are displayed as a popup list when user clicks on the down arrow. The user can select the option from the list and the selected item is displayed in the Entry field. The general syntax of the Combobox is given as follows.

```
cb = Combobox(window_name, option=value, ...)
```

The important options are 'height', 'width', 'justify', 'values', 'state', 'postcommand', and 'textvariable', etc. The 'textvariable' option links the current value to a variable that is an instance of StringVar(). There are several methods to manage the Combobox widget. Some are briefly explained here.

- set()- This method is used to set current value.
- get()- It is used to get currently selected value.
- bind('<<ComboboxSelected>>', method)- To bind a virtual event generated by the Combobox to the method.

Let us consider the following example.

```
import tkinter as tk
from tkinter import ttk
from tkinter import *

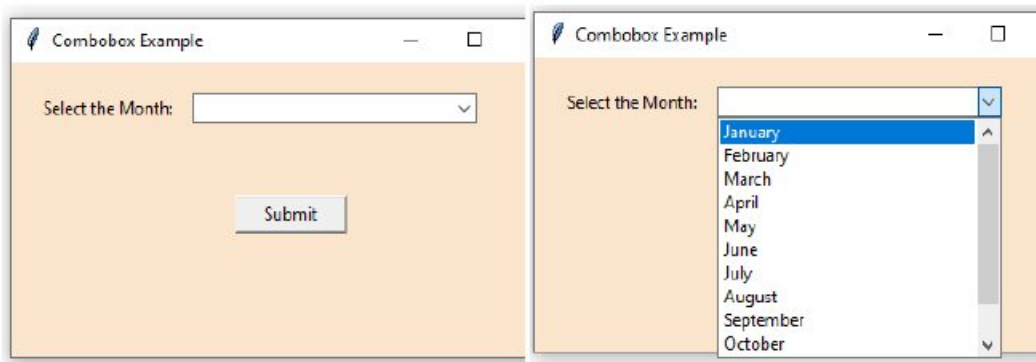
window = tk.Tk()           #Create main window
window.title('Combobox Example') #Title of the window
window.geometry("400x200") #Dimensions of the window
```

```

window['bg']='#FCE5CD'    #Background color of the window
lb = Label(window, text = 'Select the Month:', bg='#FCE5CD') #Create label
lb.place(x=20, y=20) #Place label
#Create Combobox
var = tk.StringVar()
cb = ttk.Combobox(window, width = 30, textvariable = var)
#Prepare combobox drop down list
cb['values'] = ('January',
               'February',
               'March',
               'April',
               'May',
               'June',
               'July',
               'August',
               'September',
               'October',
               'November',
               'December')
cb.place(x=130, y=20)
cb.current()
#Create button widget
button = tk.Button(window, text='Submit', width=10, command=window.destroy)
#Place the button at specified position in window
button.place(x=160, y=90)
window.mainloop()

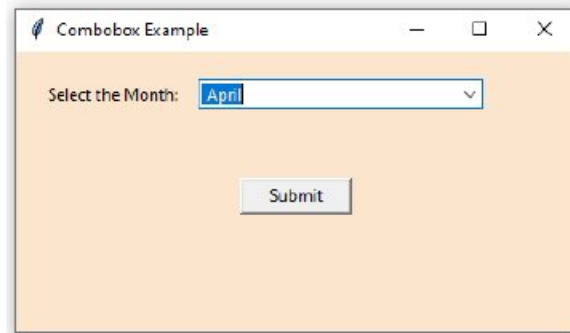
```

The output is given in Figure 10.8. A down arrow can be observed in Figure 10.8(a) with the text box. When user clicks on the arrow, the list of options appears as a drop down list as shown in Figure 10.8(b) that can be scrolled to find more options. The selected options appears in the text field as shown in Figure 10.8(c) that was blank earlier.



(a)

(b)



(c)

Figure 10.8: Creation of Combobox

There are some more widgets provided by Tkinter that could be useful for developing GUI application in Python. Some of these methods are discussed here without example code as the general procedure and syntax for creating these widgets in the program are similar to the widgets discussed already.

10.4.9 Other widgets in Tkinter

The Canvas widget provides a rectangular area for drawing pictures and other layouts, where graphics and widgets like text, frames, and labels, etc. can be placed. The general syntax for creating a canvas is as follows.

```
object_name = Canvas(window_name, option=value, ...)
```

The canvas widget offers graphics facility to place objects like line, circle, charts, and images, etc. It is good to plot graphs and draw complex layouts.

The Frame widget available in Tkinter acts like a container to hold various widgets. The main utility of the frame widget is for grouping and organization of other widgets in GUI in a friendly way. It uses a rectangular area for padding widgets. The general syntax is as follows.

```
object_name = Frame(window_name, option=value, ...)
```

A number of options are available for managing the frame. It is an important widget that can also be used as a foundation class for creating more complex widgets. Various widgets can be grouped into a compound widget with the help of Frame widget.

The Menubutton widget is the part of drop-down menu that stays on the screen all the time and it is used to display menus in the application. Each Menubutton widget is associated with a Menu widgets that displays the choices for the menubutton whenever a user clicks on it. The general syntax of Menubutton is given as follows.

```
object_name = Menubutton(window_name, option=value, ...)
```

The `menubutton` allows the user to select appropriate choice from items displayed by triggering events when clicked by the user. This widget has an option 'clicked' to track the triggered events and it reports the last clicked item.

The `Scrollbar` widget as apparent from its name provides scrolling capabilities to different types of other widgets. A scroll bar is a slide controller that is helpful to implement larger lists of items in a small window space. The general syntax is as follows.

```
object_name = Scrollbar(window_name, option=value, ...)
```

It is useful with other widgets like `Canvas`, `Entry`, `Listbox`, and `Text`, etc. Both vertical and horizontal scroll bars can be implemented with `Scrollbar` widget. Among other options 'jump' and 'orient' are two important options for `Scrollbar` widget. The 'jump' controls what happens when slider is dragged. With the help of 'orient' option it can be specified the horizontal or vertical orientation of the slider.

The `Scale` widget helps to create a sliding scale that is used to select a value on the scale. It could be a horizontal or vertical scale. The general syntax to create a sliding scale is given as follows.

```
object_name = Scale(window_name, option=value, ...)
```

There are a number of options available to format the widget. Similar to `Scrollbar` widget, 'orient' option is used to set the orientation of the scale.

The `Spinbox` is a variant of the standard `Entry` widget. It allows the user to select from a fixed number of values. The spinbox has a down arrow and an up arrow to scroll the values. The general syntax to create the `Spinbox` widget is as follows.

```
object_name = Spinbox(window_name, option=value, ...)
```

There are many options similar to many other widgets to manage the `Spinbox` widget. This widget is useful only if user needs to select from a specific range of values. It is different from `Scale` widget in style and format but fulfils the same objective.

The `PanedWindow` widget is a kind of container that can hold a number of other widgets arranged in horizontal or vertical manner. It contains several panes and each pane contains only one widget. The two panes in a pair are separated by a movable sash. The moving sash causes the widgets to resize. The general syntax for creating the `PanedWindow` is as follows.

```
object_name = PanedWindow(window_name, option=value, ...)
```

There are a number of options for formatting the `PanedWindow` widget.

The `TopLevel` is a widget that does not need a parent window on top of it. This widget is directly controlled by window manager. Therefore its general syntax is different from other widgets as given here.

```
object_name = TopLevel(option=value, ...)
```

As it can be observed from the syntax that there is no need to supply window name as parameter to this widget. There could be any number of `TopLevel`

widgets in a GUI application. It is useful to provide the user additional information in a separate window. The pop-up windows can be implemented with TopLevel widget.

Check your progress

1. What are the major options for creating GUI in Python?
2. What is usage of Tk() method?
3. Why is the mainloop() method used?
4. What are major methods for managing geometry of the window?
5. Differentiate button, check button, and radio button.
6. What is a menu?
7. How is a method associated with a widget?
8. What are the major widgets in Tkinter?
9. Differentiate Text, Entry, and Spin box.
10. What is a combo box?
11. What is use of bind() method in combo box?
12. What is scroll bar?

10.5 Turtle Graphics

Turtle is a pre-installed Python library that provides a kind of virtual canvas for drawing pictures and shapes. It provides an onscreen pen known as turtle for the drawing. There are various commands such as backward, forward, left, and right, etc. to draw different kind of shapes. With the combination of various commands, many different types of simple or complex shapes can be created. The major Turtle commands are discussed here.

- | | |
|---------------------|--|
| • Turtle(): | This is the primary method to create an object of type Turtle That is required to use the commands provided by the Turtle library. |
| • left(angle): | Turns the counter clockwise by the angle given by parameter 'angle' in degrees. |
| • right(angle): | Turns the counter anticlockwise by the angle given by parameter 'angle' in degrees. |
| • forward(amount): | This command moves the turtle arrow forward by number of pixels specified by 'amount'. |
| • backward(amount): | This command moves the turtle arrow |

backward by number of pixels specified by 'amount'.

- **up():** Used to pick up the turtle pen.
- **down():** Used to put down the turtle pen.
- **position():** Returns the current position of the turtle pen.
- **color(color_name, color_name):** Used to change the colors of turtle pen and fill. First argument is for pen color and second one for fill color.
- **fillcolor(color_name):** Used to fill color in turtle pen.
- **stamp():** Leaves the impression of turtle shape at current position.
- **heading():** Returns the current heading.
- **goto(x, y):** Moves the turtle to the position (x, y).
- **dot():** Leaves a dot at the current position.
- **shape(shape_name):** Changes the way turtle looks.
- **begin_fill()** Starting point for a filled polygon.
- **end_fill()** Closing point for a filled polygon.
- **bgcolor(color_name):** Changes background color. Default is White.
- **pensize(thickness):** Changes the size of the turtle pen.
- **pencolor(color_name):** Changes the color of turtle pen.
- **getscreen():** To open the turtle screen.
- **width(pixels):** To change the width of drawing.

Let us consider some examples.

```
import turtle
#Create turtle object
t = turtle.Turtle()
#Open turtle screen
sc = t.getscreen()
```

The above example simply creates a turtle object and open ups the turtle graphics screen as shown in Figure 10.9. This screen provides the space or a kind of virtual canvas for drawing the figures. The turtle pen appears in the middle of the screen.

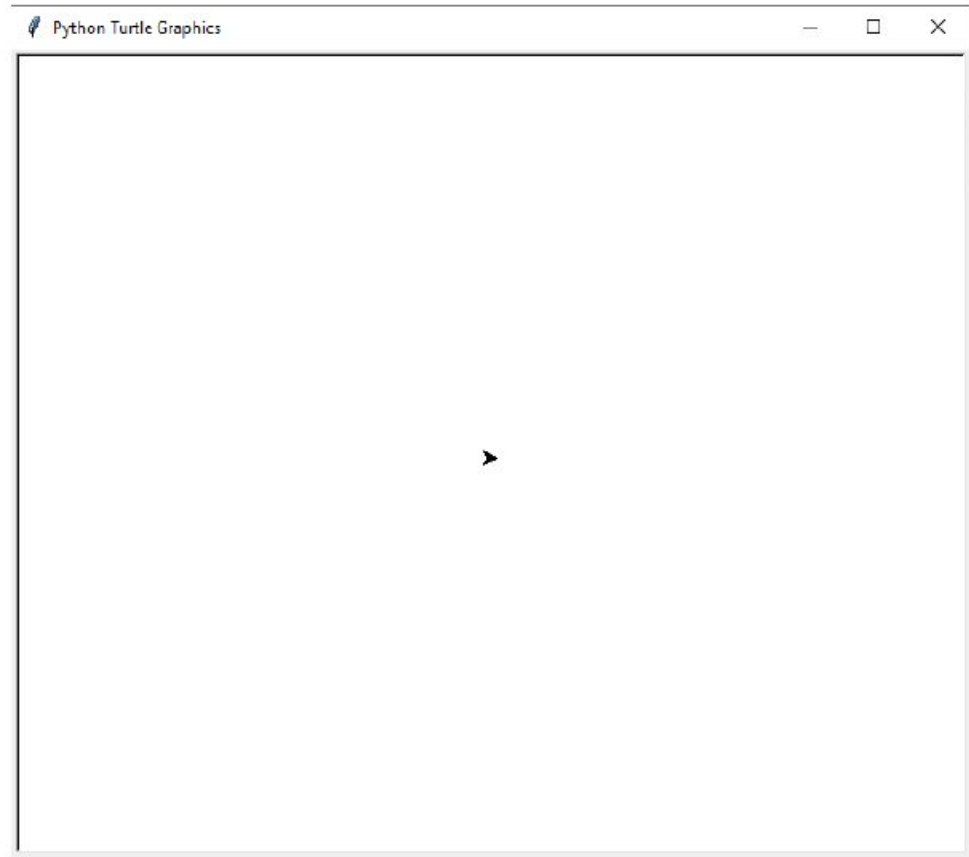


Figure 10.9: The turtle screen.

By moving the turtle pen, a variety of simple and complex figures can be drawn. Let us consider the following code. The output is shown in Figure 10.10.

```
import turtle
#Create turtle object
t = turtle.Turtle()
#Open turtle screen
sc = t.getscreen()
#Move the turtle pen
t.forward(100)
t.right(90)
t.forward(100)
```

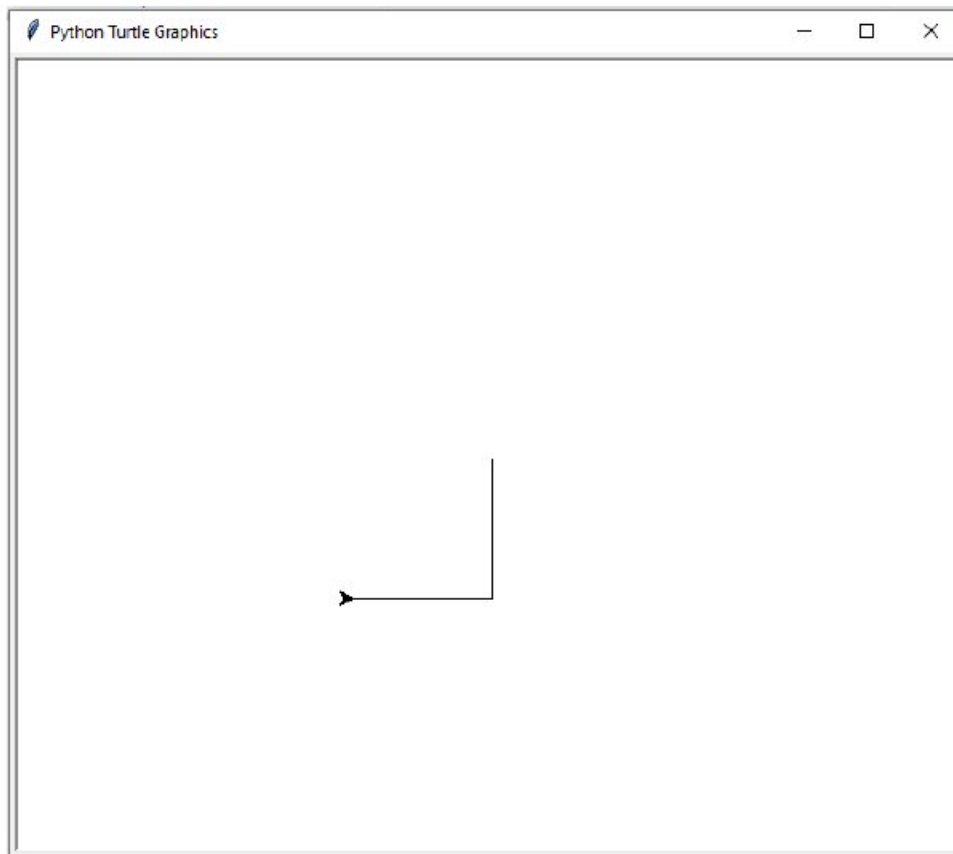



Figure 10.10: Output due to turtle pen movement.

It can be observed from the Figure 10.10 that as turtle pen moves, it draws the lines. These lines form the desirable figures or graphics as it is done in the following code.

```
import turtle
#create turtle object
t = turtle.Turtle()
#open turtle screen
sc = t.getscreen()
#draw a figure
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
t.goto(100,-100)
t.goto(-100,-100)
```

```
t.goto(100,-0)
```

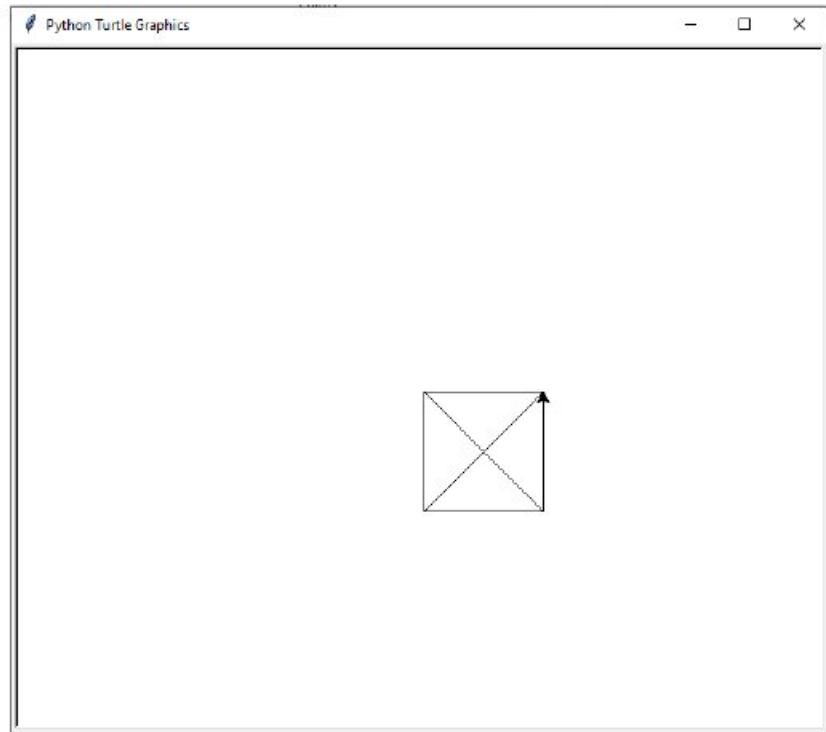


Figure 10.11: Figure drawn with turtle pen movements.

The output of the above code is shown in Figure 10.11. It draws a figure with turtle pen movements. As pen moves and changes position, the lines are drawn that create a figure. In this way many desired figures can be created. Using the available methods, the colors of the figures drawn can be changed. Consider the following code, where color of the figure and turtle pen is changed to red as shown in Figure 10.12.

```
import turtle
#create turtle object
t = turtle.Turtle()
#open turtle screen
sc = t.getscreen()
#draw a square
t.color('red') #change the colors
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
```

```

t.right(90)
t.forward(100)
t.goto(100,-100)
t.goto(-0,-100)
t.goto(100,-0)

```

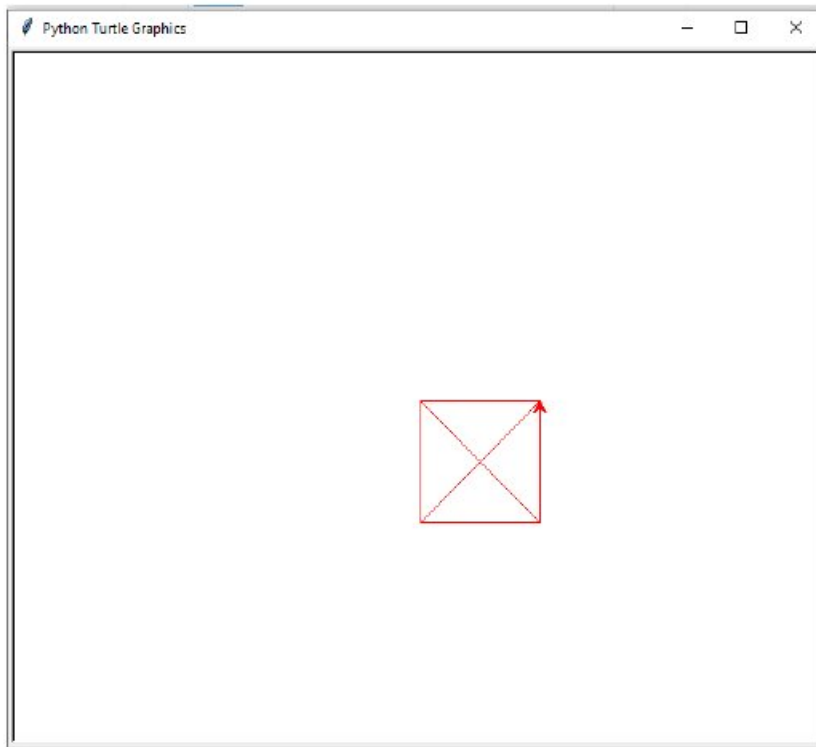


Figure 10.12: Changing the color of the figure.

Using the appropriate logic, many simple and complex figures can be drawn with the help of turtle programming. Let us consider some examples.

```

import turtle
#create turtle object
t = turtle.Turtle()
#open turtle screen
sc = t.getscreen()
#draw a star
colors = ['red', 'purple', 'blue', 'green', 'magenta']
turtle.bgcolor('lightgray')
t.width(5)
for i in range(5):
    t.color(colors[i%5])
    t.forward(200)
    t.right(144)

```

Output is shown in Figure 10.13.

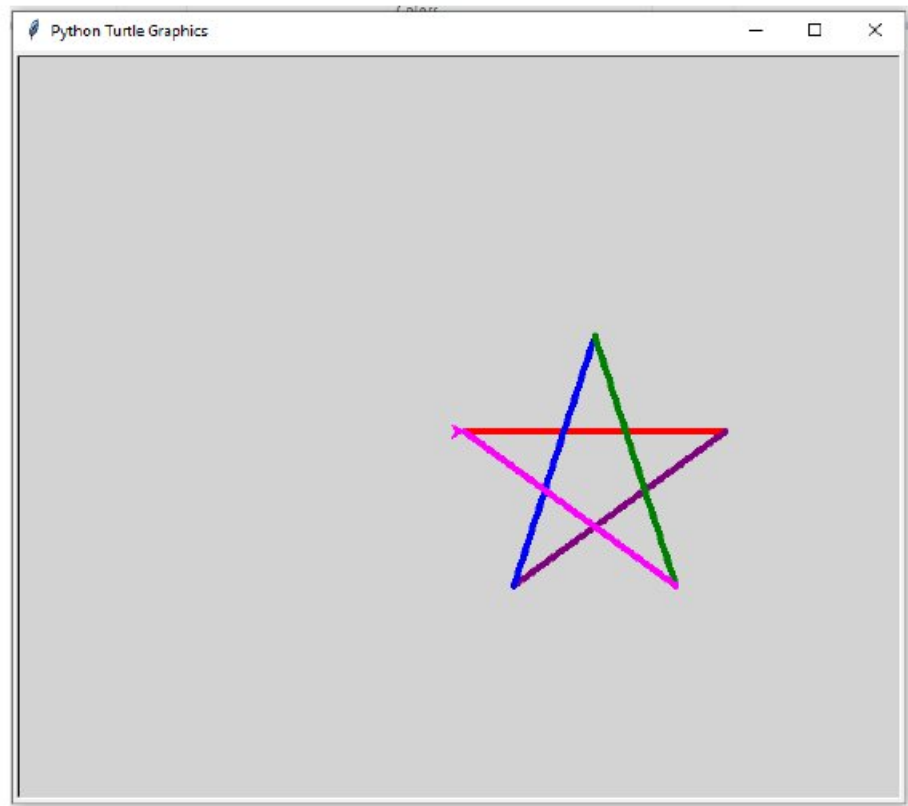


Figure 10.13: Drawing a star.

```
import turtle
#create turtle object
t = turtle.Turtle()
#open turtle screen
sc = t.getscreen()
#draw a star
colors = ['red', 'purple', 'blue', 'green', 'magenta']
turtle.bgcolor('lightgray')
for i in range(360):
    t.pencolor(colors[i%5])
    t.width(i/100 + 1)
    t.forward(i)
    t.left(60)
```

The output is shown in Figure 10.14.

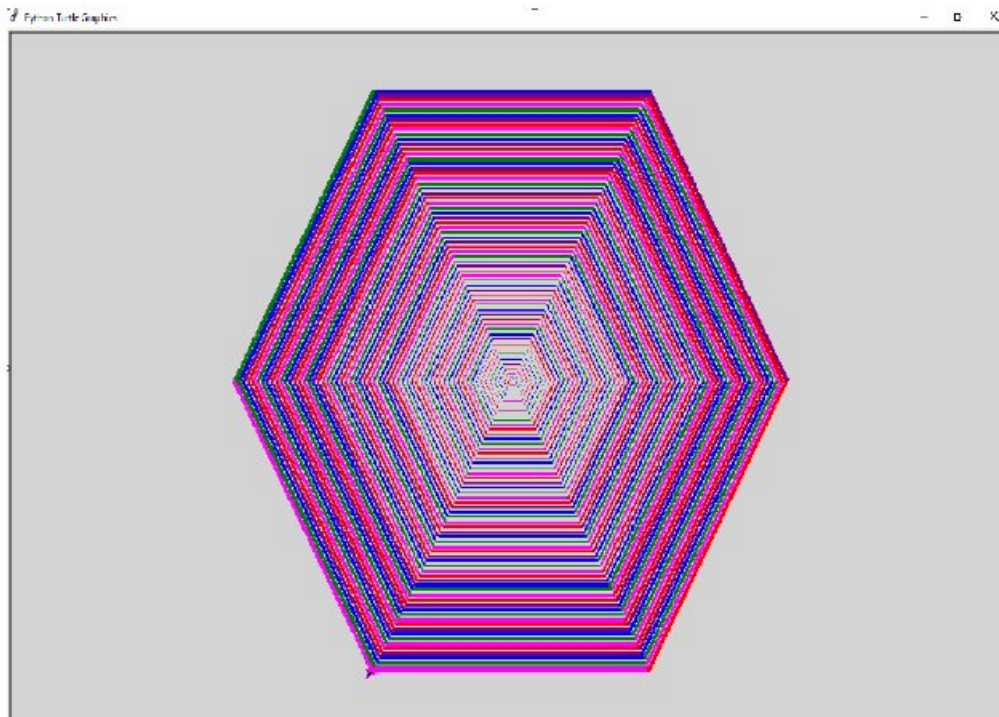


Figure 10.14: Drawing a complex pattern.

```
import turtle
#create turtle object
t = turtle.Turtle()
#open turtle screen
sc = t.getscreen()
colors = ['red', 'purple', 'blue', 'green', 'magenta']
turtle.bgcolor('lightgray')
t.width(5)
for i in range(135):
    t.pencolor(colors[i%5])
    t.forward(i)
    t.left(20)
```

The output is shown in Figure 10.15.

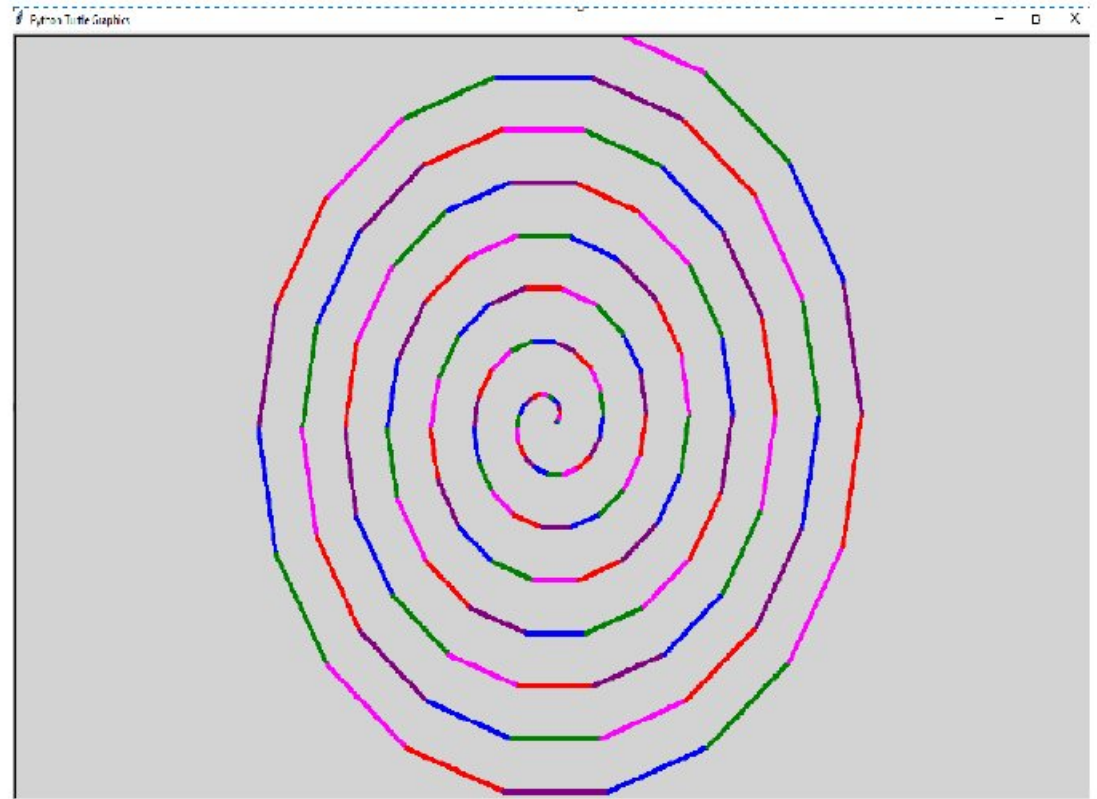


Figure 10.15: Drawing a coil.

```
import turtle
#create turtle object
t = turtle.Turtle()
#open turtle screen
sc = t.getscreen()
colors = ['red', 'purple', 'blue', 'green', 'magenta']
turtle.bgcolor('lightgray')
for i in range(350):
    t.width(i/100 + 1)
    t.pencolor(colors[i%5])
    t.forward(i)
    t.left(59)
```

The output of the code is shown in Figure 10.16.

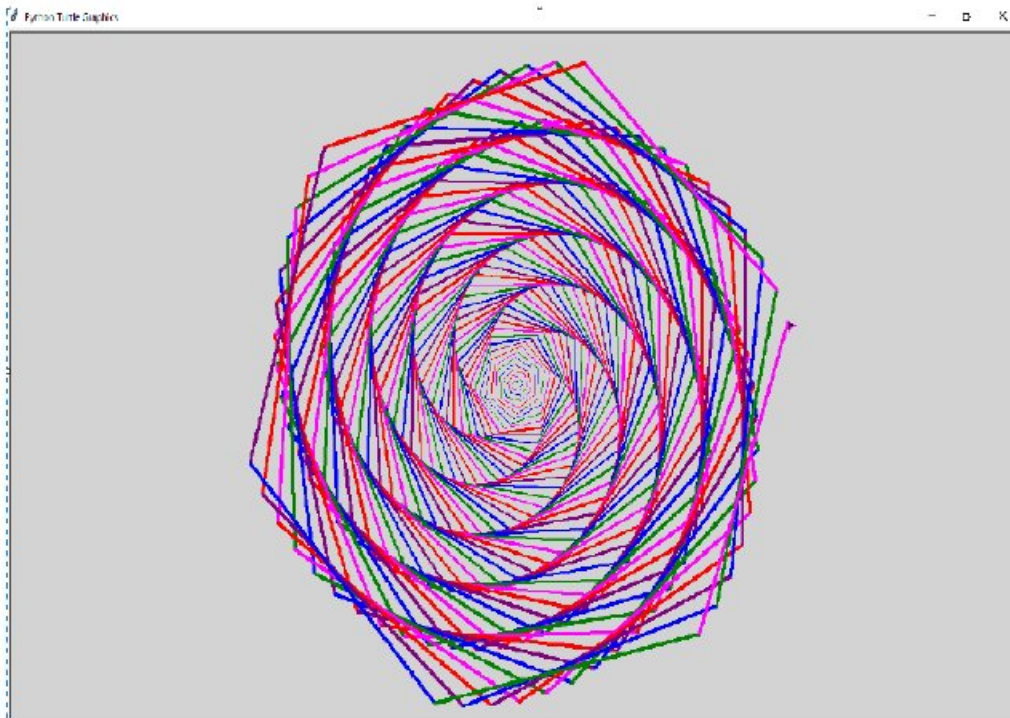


Figure 10.16: Drawing a complex figure.

With minor changes in the code, many different patterns can be drawn as given here whose output is shown in Figure 10.17.

```
import turtle
#create turtle object
t = turtle.Turtle()
#open turtle screen
sc = t.getscreen()
colors = ['red', 'purple', 'blue', 'green', 'magenta']
turtle.bgcolor('lightgray')
for i in range(350):
    t.width(i/100 + 1)
    t.pencolor(colors[i%5])
    t.forward(i)
    t.left(80)
```

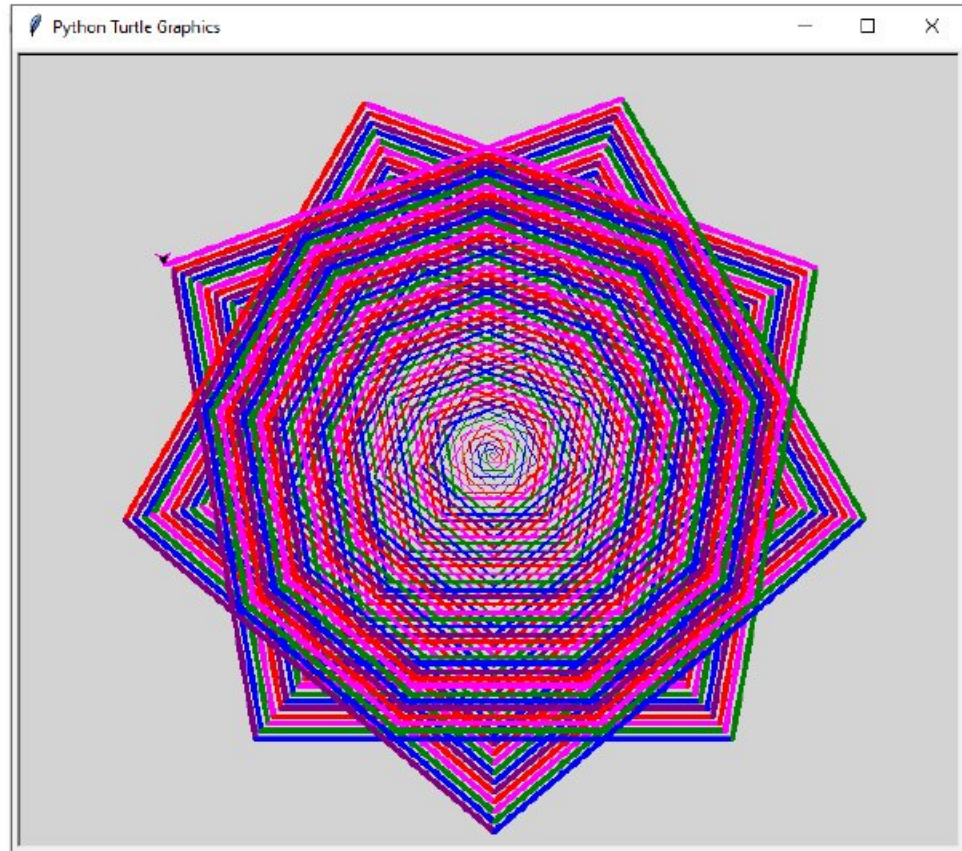


Figure 10.17: Another pattern.

10.6 Testing

Testing is a method of identifying errors and bugs in the program. It is primary used to identify logical errors, to verify the correct behaviour of the program, and to find out any gaps or missing requirements. In other words, testing is the process of verifying if the program or software is working according to the expectations and doing nothing that is not expected. Testing is important to ensure the quality of the end product in terms of performance, security, and effectiveness.

The testing is done at various levels of software development process. Accordingly, there are different types of testing approaches and techniques. Primarily testing techniques can be divided into two major categories: functional testing and non-functional testing. These testing types can further be divided into sub-categories depending on the approach and goals.

10.6.1 Functional Testing

Functional testing validates the program or software against the functional requirements of the application or system. The purpose of the functional testing is test or validate each individual function of the application by providing

appropriate input. Each function in the program is tested against a given input to find out if the actual output is matching to the expected output. There are several different types of functional testing.

- **Unit testing-** This testing technique aims at testing each individual unit of the source code to verify its appropriate behaviour. Each unit is tested to check its correctness and its expected performance. These units include functions, methods, subroutines, and any smallest piece of code that can be logically isolated.
- **Integration testing-** In this technique, individual units or modules are logically combined as a group and tested. Integration testing is useful when different modules are developed by different programmers. It exposes the defects and issues in the integration of the modules to make the software. Although each module is unit tested individually but still there may exist some defects.
- **Interface testing-** A connection that integrates two software components is known as interface. Interface testing checks whether the software components are communicating correctly to transfer data and control messages. The interface could be an API, web service, or connection strings etc. and interface testing involves these components.
- **System testing-** System testing is performed on the fully integrated software system to detect end-to-end flaws as a user. It involves integrated software and hardware system to evaluate all the requirements specified by the client. The major purpose of this technique is to test the system compliance with specific requirements.
- **Smoke testing-** It consists of minimal set of test runs to evaluate the important features of the software. After smoke testing, confirmation is given to proceed for other testing. Smoke testing is done for quality assurance and to verify the stability of the end product.
- **Regression testing-** Regression is done when changes are made to an existing software or program. It uses the old test cases to check whether the existing features are still working properly after the new code changes. Once the code changes are made, the regression testing is performed to ensure that there are no side effects.
- **Sanity testing-** Sanity testing is considered as a subset of the regression testing. It is not a detailed testing technique and only used to evaluate the changes made to the code. Suppose a bug is reported in the software and some changes are made to the code to fix it. In such cases, sanity testing is done before deploying the upgraded software. The focus of the sanity is a section of code to check if the software is working as expected after the minor changes.
- **Beta/Acceptance testing-** This testing is based on the user responses. A nearly finished product is made available to a group of real users with an intention to uncover any bugs or undesirable issues related to

usability, reliability, functionality, and compatibility, etc. The beta testing is an important technique where real users not only evaluate the product but also provide useful information for future use.

All these techniques are applied according to product and scope. The basic goal of the testing techniques is to eliminate the bugs, errors, and enhance different aspects of the code. Not every techniques suits to every product and each one having its own merits and demerits.

10.6.2 Non-Functional Testing

The non-functional testing techniques work on non-functional aspects of the software such as usability, reliability, portability, and efficiency, etc. It helps to optimize setup, installation, monitoring, and management of the product. The non-functional testing helps to reduce the production risk. There are various parameters involved in non-functional testing including availability, efficiency, flexibility, interoperability, integrity, portability, reliability, reusability, scalability, security, survivability, and usability. Based on these parameters, there are various kind of non-functionality testing.

- **Performance testing-** The performance testing is done to check how the program or software application performs under the expected workload. The parameters of the testing are speed, response time, reliability, resource usage, and stability, etc. The performance of the application is measured under the varying workload.
- **Load testing-** The load testing evaluates the software application under heavy workload especially when multiple users are working simultaneously. The major difference between performance testing and load testing is that the performance testing is done under the normal expected load, whereas the load testing is performed for extreme load.
- **Stress testing-** The stress testing is used to evaluate the robustness of the software application. It is done under the conditions beyond the normal operational capabilities. The goal of stress testing is to verify the stability and reliability of the system.
- **Security testing-** The security testing is done to test the safeguard of application against sudden or deliberate security attacks and threats. The objective of the security testing is to identify the flaws in security mechanisms of the application and vulnerabilities based on the elements such as authentication, confidentiality, integrity, authorization, and non-repudiation.
- **Install testing-** The most software applications use some installation procedures to make the software functional at the machine. The install testing is used to verify that installation procedures install all the necessary components and application works properly as expected.

- **Volume testing-** It is also known as flood testing, which is done for the huge amount of data. The objective of the volume testing is to measure the throughput to check the performance with increasing data volume. It helps to identify the system capacity under normal and high data volume.
- **Compatibility testing-** The compatibility testing is performed to check the compatibility of the software application on different platforms under different conditions. The testing is done on different hardware, operating systems, network settings, and network devices, etc.
- **Usability testing-** The usability testing is concerned about how easy it is easy to use the application. It checks the user friendliness of the application, flexibility, user controls, and user satisfaction, etc. It is also known as user experience testing and it is done with group of end users of the application.
- **Reliability testing-** The reliability testing aims at checking the failure free functioning of the software under specific conditions. It helps to discover problems in software design and ensure better quality.
- **Recovery testing-** The recovery testing measures the software ability to recover from failure. The kind of failures that are tested include software crashes, hardware crashes, and network failures, etc. It aims to check whether the application can continued under such conditions.
- **Localization testing-** Localization testing checks the quality and performance of a software for its local version to test its ability to perform under customized conditions. It is done under localized conditions with specific hardware and application.
- **Compliance testing-** The compliance testing is performed to certify the software compliance with the standards and regulations. It is important to validate the compliant state of the product for its life. The compliance testing is a kind of audit according to some rules, regulations, and policy, etc.

The non-functional testing is important to ensure security and reliability of the software. It is also a kind of measure of quality of the product. Both functional and non-functional testing are needed to develop an optimized and refined application.

10.6.3 Unit Testing in Python

As discussed earlier, unit testing is performed to validate each testable individual units of the program irrespective their size. It is the first level of testing that can identify bugs and errors during early stage of development. Python provides the `unittest` module to perform unit testing of source code. Python supports various important concepts related to unit testing with object oriented approach.

- **Test fixture-** Test fixtures are used to make preparation for the unit test. It is used to create a fixed environment for running the tests. It includes creation of temporary or proxy database, starting server process, clean-up process, and optionally setup methods, etc.
- **Test case-** A test case defines a set of conditions to check the response of the system for a specific set of inputs. It finds out whether a particular feature or functionality of the software works properly under the given conditions. A test case may include test data, preconditions, expected results, and postconditions. In Python, the `unittest` framework provides a class `TestCase` to create the new test cases.
- **Test suit-** A test suit is collection of various test cases or test suits to execute multiple tests together under same conditions.
- **Test runner-** A test runner is a component that executes test code, checks the assertions, and generates a report containing outcomes of the test code. It could be a GUI based program or text interface.

Let us discuss how unit testing is done in Python. There are various ways of carrying out testing in Python. One of the most basic approach makes use of `unittest` module available in Python standard library. The fundamental requirement of this approach is that each test unit must be completely independent and able to execute alone. The source code and test code must be saved in different files. Let us first write a test unit in the form of a function.

```
def vol(x,y,z):
    v = x*y*z
    return v
```

In the above code, a function is defined for the volume of a figure that takes three arguments and simply returns the multiplication of three numbers. Suppose this code is saved in a file named 'sourceCode.py'. In order to perform unit testing we need a set of inputs and corresponding expected outputs. For example, if input is (2, 4, 5) the expected output is 40. A test code is then written using `unittest` framework as follows that is saved in a separate file.

```
from sourceCode import *
import unittest
class testVol(unittest.TestCase):
    def test_vol(self):
        self.assertEqual(vol(2,4,5),40)
        self.assertEqual(vol(3,3,3),27)
        self.assertEqual(vol(4,8,2),64)
if __name__ == '__main__':
    unittest.main()
```

In the above test code, the test case is written by subclassing *unittest.TestCase*. The above code defines only a single test. The test is defined as a method whose name conventionally starts with a string 'test'. Although it may also be named differently, the naming convention is helpful to identify the methods representing the test. The test method calls a function *assertEqual()* to verify the expected results. A special method *unittest.main()* is also used in the code that provide the command line interface to run the test script. The test runner accumulates the results to produce the report. The test code be executed from Python IDLE or it can be executed from command line interface. The output of the above test case is obtained as given here.

```
.
-----
Ran 1 test in 0.021s
OK
```

The source code has qualified through the unit test given above. Suppose the programmer knowingly or unknowingly makes a mistake in the program as given here. Instead of returning *v* it returns *v+1*.

```
def vol(x,y,z):
    v=x*y*z
    return v+1
```

```
F
-----
FAIL: test_vol (__main__.testVol)
-----
Traceback (most recent call last):
  File "D:/pyCode/testCode1.py", line 5, in test_vol
    self.assertEqual(vol(2,4,5),40)
AssertionError: 41 != 40
-----
Ran 1 test in 0.015s
FAILED (failures=1)
```

When the same test case is executed, the following result is obtained.

The source code is unable to qualify the unit test. Apart from *assertEqual()*, the *unittest* framework provides some other functions including *assertTrue()*, *assertFalse()*, and *assertRaises()*. Let us consider another source code,

```
def isPalindrome(s):
    return s == s[::-1]
```

The test case is written as follows. It makes use *assertTrue()* and *assertFalse()* functions.

```
from sourceCode import *
import unittest
class testPalin(unittest.TestCase):
    def test_palindrome(self):
        self.assertTrue(isPalindrome('aba'))
        self.assertTrue(isPalindrome('malayalam'))
        self.assertFalse(isPalindrome('abaa'))

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

The output of the unit test is as follows.

```
.
-----
Ran 1 test in 0.015s
OK
```

It is also possible to write multiple test cases in the same file. Suppose there are multiple methods in the file 'sourceCode.py'.

```
def fact(n):
    n = int(n)
    factorial = 1
    if n < 0:
        factorial = 'Error'
    if n >= 1:
        for i in range(1, int(n)+1):
            factorial = factorial * i
    return factorial
def vol(x,y,z):
    v = x*y*z
    return v+1

def isPalindrome(s):
    return s == s[::-1]

def maxThree(x,y,z):
    if(x>y):
        if(x>z):
```



```

        return x
    else:
        return z
    else:
        if(y>z):
            return y
        else:
            return z

```

Let us write test cases for all these methods in the same file. A separate class may written for each test case. Alternatively, it is also possible to write multiple test cases within the same class. Let us consider following test code, where each test case is written in a separate class.

```

from sourceCode import *
import unittest

class testVol(unittest.TestCase):
    def test_vol(self):
        self.assertEqual(vol(2,4,5),40)
        self.assertEqual(vol(3,3,3),27)
        self.assertEqual(vol(4,8,2),64)

class testFac(unittest.TestCase):
    def test_fac(self):
        self.assertEqual(fact(5),120)
        self.assertEqual(fact(-1),'Error')
        self.assertEqual(fact(1),1)

class testPalin(unittest.TestCase):
    def test_palindrome(self):
        self.assertTrue(isPalindrome('aba'))
        self.assertTrue(isPalindrome('malayalam'))
        self.assertFalse(isPalindrome('abaa'))

class testMax3(unittest.TestCase):
    def test_max(self):
        self.assertEqual(maxThree(30,65,21),65)
        self.assertEqual(maxThree(12,25,124),124)
        self.assertEqual(maxThree(86,49,12),86)

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

```

The output of the above code is given as follows.

```

.....
-----
Ran 5 tests in 0.025s
OK

```


All the test cases qualified the unit test. The above test script can also be executed with an option `-v` to obtain the result with a higher verbosity. If the above script is executed with option `-v`, the following output is obtained.

It is shown in the above report that all the methods qualified the unit test. Let us write all the test cases in the same class named as 'testCode'.

```
from sourceCode import *
import unittest
class testCode(unittest.TestCase):
    def test_vol(self):
        self.assertEqual(vol(2,4,5),40)
        self.assertEqual(vol(3,3,3),27)
        self.assertEqual(vol(4,8,2),64)

    def test_fac(self):
        self.assertEqual(fact(5),120)
        self.assertEqual(fact(-1),'Error')
        self.assertEqual(fact(1),1)

    def test_palindrome(self):
        self.assertTrue(isPalindrome('aba'))
        self.assertTrue(isPalindrome('malayalam'))
        self.assertFalse(isPalindrome('abaa'))

    def test_max(self):
        self.assertEqual(maxThree(30,65,21),65)
        self.assertEqual(maxThree(12,25,124),124)
        self.assertEqual(maxThree(86,49,12),86)

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

It provides the same output.

At command line interface following command is used to execute the test code named as testFile.py.

```
python -m unittest testFile
```

```
python -m unittest -v testFile
```

If only a particular class suppose 'testVol' from testFile.py is needed to execute, then

```
python -m unittest testFile.testVol
```

If only a particular function suppose 'test_max()' from a class 'testCode' from testFile.py is needed to execute, then

```
python -m unittest testFile.testCode.test_max
```

10.7 Summary

Some advanced concepts such as multithreading, testing, GUI, and Turtle Graphics were discussed in this unit. Multithreading is used to perform several tasks simultaneously in a program. Python provides a simple and efficient way to write multithreading based programs using **threading** module. The threads are created as an instance of *Thread* class. There are several methods offered by **threading** module and *Thread* class to manage the threads.

GUI provides a user friendly interface to interact with the computers through various widgets and menus. There are several methods in Python to create GUIs. Tkinter is a commonly used powerful method for developing GUI based applications. The programmers can create different types of widgets and manage the geometry of the GUI using a number of in-built methods. Turtle graphics is a Python library for creating graphics by drawing different types of figures on a virtual canvas. It provides a pen with options to change its color, thickness, and shape. By moving the pen with the help of different types of commands various kind of simple or complex figures can be drawn.

Software testing is important for the quality of the product. There are many different types of testing techniques. The testing techniques can be divided into two major categories: functional and non-functional testing techniques. The purpose of the functional testing is test or validate each individual function of the application by providing appropriate input. The non-functional testing techniques work on non-functional aspects of the software that helps to optimize setup, installation, monitoring, and management of the product. The non-functional testing helps to reduce the production risk. Both functional and non-functional techniques include various testing methods. Unit

testing is the first level of functional testing techniques that is used to validate each testable individual units of the program irrespective their size. Python provides the `unittest` module to perform unit testing of source code with object oriented approach.

10.8 Review Questions

Q.1 What is a thread? What are the advantages of having multiple threads in a program?

Q.2 How are threads created in Python? Write a program to demonstrate multithreading in Python.

Q.3 What are the major options in Python for creating GUIs? Briefly describe each option.

Q.4 Write general steps to create GUI in Tkinter.

Q.5 How is the placement widgets is done? Explain different methods with their syntax.

Q.6 Explain following widgets with methods of their creation in Python:

(i) Button (ii) Check box (iii) Radio button (iv) Entry (v) Text

Q.7 Write a program to create a Menu in GUI in Python. There should be several sub-menus in the Menu.

Q.8 What is turtle graphics? Write methods with syntax to:

(i) Change color of the pen

(ii) Change position of the pen

(iii) Fill the color

(iv) Change thickness of the pen

(v) Change background color

(iv) Change shape of the turtle pen

Q.9 What is functional testing? Briefly explain different types of functional testing techniques.

Q.10 Briefly describe the important concepts related to unit testing in Python.

Q.11 Define some functions in a program and then write test code to perform unit testing of the source code in Python.



Uttar Pradesh Rajarshi Tondon
open University

SBSCS-02

Python Programming

Block

4

MACHINE LEARNING IN PYTHON

Unit 11	
Machine Learning using Python	280
Unit 12	
Regression and Classification in Machine Learning	306

Course Design Committee

Prof. Ashutosh Gupta Director (In-charge) School of Computer and Information Science, UPRTOU Prayagraj	Chairman
Prof. R. S. Yadav Department of Computer Science and Engineering MNNIT Prayagraj	Member
Dr. Marisha Assistant Professor (Computer Science), School of Science, UPRTOU Prayagraj	Member
Mr. Manoj Kumar Balwant Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Member

Course Preparation Committee

Dr. Brajesh Kumar Associate Prof. Dept. of CSIT, MJPRU, Bareilly, 243006	Author Block – 01 to 04 Unit – 01 to 12
Prof. Manu Pratap singh Prof. Department of Computer Science Dr. Bhimrao Ambedkar University, Agra - 282002	Editor Block – 01 to 04 Unit – 01 to 12
Dr. Chandrakant Kumar Singh Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	Coordinator

© UPRTOU , Prayagraj

ISBN :

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar pradesh Rajarshi Tandon Open University.

OVERVIEW:

Machine learning techniques have become highly important in recent times. These techniques have applications in many different areas including pattern recognition, data analytics, and natural language processing, etc. This block deals with machine learning support in Python. It contains two units: Unit 11 and Unit 12. Different types of machine learning techniques with their merits and demerits are discussed in Unit 11. The focus of Unit 12 is mainly on regression and classification techniques. In both units, Python libraries for machine learning techniques and their implementation are also explained.

Unit 11: Machine Learning Using Python

Structure

- 11.0 Introduction
- 11.1 Objective
- 11.2 Features and Labels
- 11.3 Supervised Learning
- 11.4 Regression
- 11.5 Classification
- 11.6 Merits and Demerits of Supervised Learning
- 11.7 Unsupervised Learning
- 11.8 Unsupervised Learning Algorithms
- 11.9 Difference between Supervised and Unsupervised
- 11.10 Summary
- 11.11 Review questions

Unit 11: Machine Learning Using Python

11.0 Introduction

Machine learning is the process of recognizing data structure and fitting it into a model that people can understand and use. Despite the fact that machine learning is a branch of computer science, it differs from standard computing methodologies. Existing computing algorithms are a set of explicit instructions that computers utilize to solve problems. Machine learning techniques, on the other hand, allow computers to train input data and then utilize statistical analysis to produce values that fall inside a specific range. As a result, machine learning enables computers to automate the decision-making process based on inputted data by creating models from sample data. Machine learning is a technique for converting data into knowledge. You can use hidden patterns and knowledge of a problem to predict future events and make all kinds of complex decisions. Most of us are unaware that we are already interacting with machine learning on a daily basis. Whether you're researching for something on Google, listening to music or taking a picture, machine learning is the part of the engine behind it that's always learning and improving in every interaction. It is also behind world-changing advances such as optical character recognition (OCR) technology, facial recognition, cancer detection drugs, and recommendation engines. Working with machine learning generally falls into a large category.

These classifications depend on how learning is received, or how response on learning is provided to the developed system. The machine learning techniques can be divided into three main categories: supervised learning, unsupervised learning, and reinforcement learning as shown in Figure 11.1. Supervised learning is task-based that trains a model based on input and output data classified by humans. Here, the machine is trained using appropriately labeled training material and predicts the output according to the trained classified data. Label data means input data that is already tagged on the correct output. Here, the training data acts as a machine cycle supervisor to predict the correct output. Supervised learning deals with task-based problems such as classification problems, evaluation problems and regression analysis.

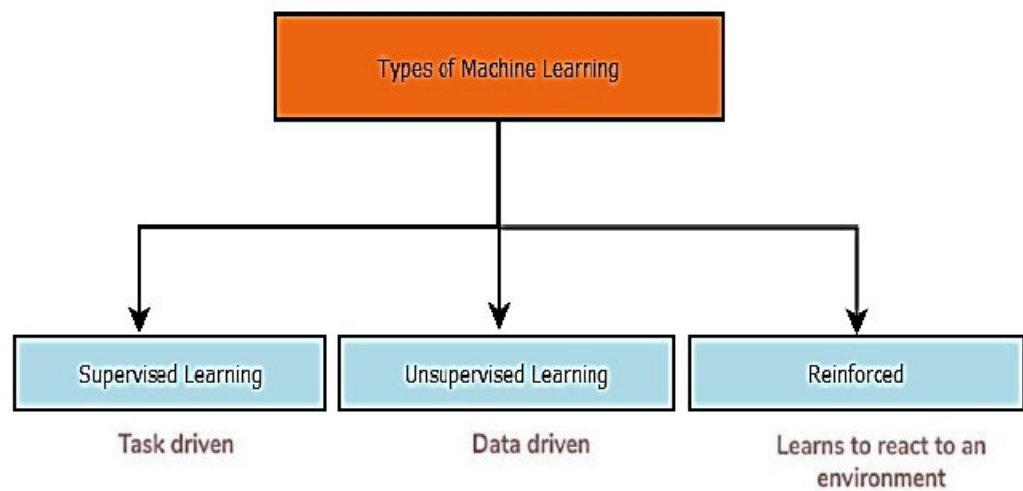


Figure 11.1: Types of Machine Learning algorithms

Unsupervised learning is a type of data-driven learning in which the algorithm is given no labeled data to help it to detect structure in the data it is given. Unsupervised learning techniques, in which classification models are not supervised using training data, are required. Models, on the other hand, find the hidden patterns in the provided dataset. Unsupervised learning used to handle data driven problems such as clustering, segmentation, associate mining or dimension reduction. Reinforcement learning learns to react to an environment. Main task here is training a model to make a series of decisions. Agents learn how to reach goals in uncertainties, potentially complex environments. It refers to the process of teaching machine learning models to make a series of judgments. The agent learns to achieve a goal in an unpredictable, potentially complex environment. Reinforcement learning is used to handle reward system, recommendation system and decision problem.

11.1 Objectives

The objectives of this unit are:

1. To discuss basics concepts of machine learning.
2. To understand the structure of data that used as input to the machine learning models.

11.2 Features and Labels

A **feature** is a value representing some characteristics that can distinguish the objects or entities of interest. The training dataset consists of feature values for each label. There can be a single or multiple features depending on the complexity of the problem. A simple machine learning project might use a

single feature, while a more sophisticated machine learning project could use hundreds or thousands of features. Usually features are kept in the form of a vector as follows.

$$\{\text{features, label}\}: ((x_1, x_2, \dots, x_n), y)$$

where, x is the feature vector and y is the label. Suppose someone wants make decision to sell a car based on car mileage, year, etc. Let Yes and No are the labels then mileage and year would be the features for this problem. Data used as features can be divided in two categories:

- Labeled data
- Unlabeled data

A **labeled data** includes both features and corresponding labels i.e.

$$\text{Labeled data: } \{\text{features, label}\}: (x, y)$$

For example: Let consider the **calories** dataset, which is available in 'csv' format. The some sample from the dataset are given in Table 11.1. The columns Gender, Age, and Height, etc. in the dataset are the features. While the column Calories are the class labels. Each column represents distinct features and labels given as follows.

Table 11.1: Labeled data

User ID	Gender	Age	Height	Weight	Calorie
333	Male	69	191	80	220
619	Female	23	165	59	78
853	Male	60	178	75	28

Here, features such as Gender, Age and Weight are used for predicting the Calories burnt during exercise.

An **unlabeled data** contains features but not the label i.e.

$$\text{Unlabeled data: } \{\text{features, ?}\}: (x, ?)$$

Here is the example of unlabeled data given Table 11.2. There are columns in this table that represent values of different features but labels are not given for Calories.

Table 11.2: Unlabeled data

User ID	Gender	Age	Height	Weight
333	Male	69	191	80
619	Female	23	165	59
853	Male	60	178	75

11.3 Supervised machine learning

Machine learning is a system that learns to associate outputs with input based on some labeled data to produce useful predictions on an unseen data. This involves the use of features and labels in form of x (independent features) and y (dependent labels) variables. Features are the fields used as inputs and labels are used as output. This applies to both classification and regression problems. In supervised machine learning, the machines are trained using well labeled training data, and predict the output based on trained labeled data.

Labeled data means some data inputs that has already been tagged with the exact outcome. Here the training data acts as a supervisor to teach the machine to predict the correct outcome. For this type of model, the model needs to find a mapping function to map the input variable (x) to the output variable (y).

$$y = f(x)$$

To train the map learning model, which is equivalent to a student learning something in front of a teacher, a direction is required. To train the model, this type of machine learning method requires supervision, similar to how a student learns in the presence of a teacher. Supervised learning is a method of feeding input data to a machine learning model, which then predicts the proper output data. A supervised learning algorithm's goal is to appropriately transfer the input variable(x) to the output variable(y). Weather forecasting, image categorization, housing price prediction, spam filtering, and other applications of supervised learning can be found in the real world.

11.3.1 Working of Supervised machine Learning

The following example and diagram in Figure 11.2 illustrates the concept of supervised learning. Let us assume you have a dataset with a variety of shapes (Square, circle, and triangle). The initial stage is to train the model for various shapes found in the dataset. The supervised training model is trained using a labeled dataset, where the model trains for each data type. When the training process is complete, it tests the model based on test data which is a subset of the training dataset and predicts the output. Let us consider a dataset of different types of shapes (rectangle, square, and triangle). The first step involved is to train models of the various shapes of the dataset. Model will be trained using certain features of different shapes.

- If shape side=0, then labeled shape is a circle.
- If shape side=3, then labeled shape is a triangle.
- If shape side=4 and are all equal, then labeled shape is a square.

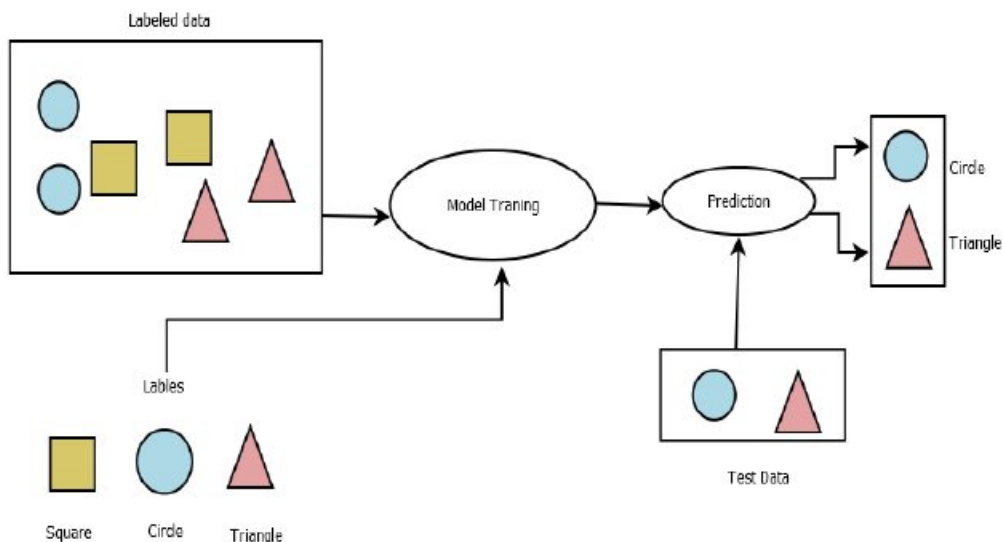


Figure 11.2: Supervised machine learning

Once the model is trained, the next step is to test the model using the test set, and the model's job is to figure out the correct shape. Because the model is trained on any given type of shape, it can classify new shapes based on the number of sides and predict the correct output. There are many different ways teachers can use machine learning algorithms, including linear regression, logistic regression, multi-class classification, decision trees, and support vector machines.

After the model has been trained, the next stage is to put it to the test with the test set, with the model's objective being to identify the correct shape. The model can now classify the new shape based on a number of sides and predict the proper output because it has been trained for all of the specified types of shapes. Supervised Machine Learning methods include linear regression, logistic regression, multi-class classification, Decision Trees, and support vector machines, and some others.

11.3.2. Steps Involved in Supervised Learning

There are many different machine learning techniques but the general methodology followed them is similar. The major steps involved in supervised learning are as follows.

1. Determine the training dataset's type.
2. Gather the training data that has been labeled.
3. Divide the training dataset into three parts: training, test, and validation (Sometimes needed as the control parameters, which are the subset of training datasets).

4. Determine the training dataset's input properties that allow the model to accurately predict the output.
5. Select an appropriate algorithm for the model (e.g., linear regression, support vector machine, decision tree, etc.).
6. Run the algorithm on the practice data.
7. Provide the test set to evaluate the model's correctness. If the model correctly predicts the outcome, then our model is accurate.

Supervised learning can be grouped into two types of categories regression and classification as shown in Figure 11.3. Any machine learning problem has the same goal of building a model that can calculate the value of dependent variables from a feature variable. The difference between the two problems is the fact that the dependent variable is numeric for regression and category for classification.

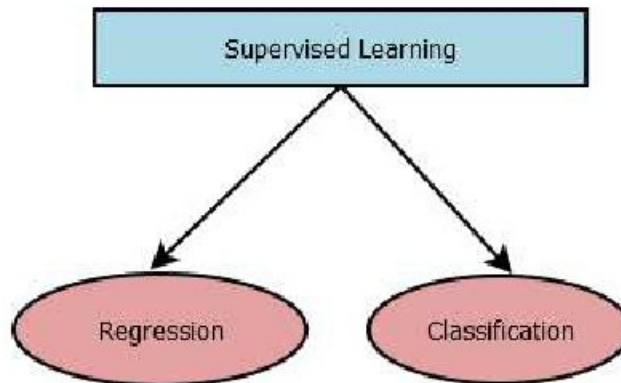


Figure 11.3: Types of Supervised learning

11.4 Regression

Regression analysis is one of the most important fields in statistics and machine learning. The regression algorithm is used when the dependent variable is a continuous value such as "payment", "height", or "weight". It attempts to align the data in the best hyper plane to pass through the point. The regression algorithms for supervised learning are as follows:

1. Linear Regression
2. Multiple Regression
3. Polynomial Regression
4. Regression trees

11.4.1 Linear Regression

Linear Regression is a form of predictive modeling method which explores the relationship between a **dependent variable** and **independent variables**. This

type of regression involves only one predictor. Linear regression is mostly used for weather forecasting, time series models and etc. between the variables. Regression analysis is a significant tool for modeling and exploring the data. Here, a curve/lines are fitted to the data points, in such a way that the variances between the distances of data points from the curve should be minimized. The following code shows important classes needed to implement linear regression from `sklearn.linear_model` library. Once you have created model object the next step is to fit the model using `fit` method passing `x_train` and `y_train` as a parameters.

```
from sklearn.linear_model import LinearRegression
model=LinearRegression()
model.fit(x_train,y_train)
```

11.4.2 Multiple Regression

Multiple regression is similar to linear regression, but it includes more than one independent value, implying that we are attempting to predict a value using two or more factors. The dependent or outcome variable is the one that needs to be predicted. The variables used in predicting the dependent variable value are called the independent or predictor variables (explanatory or regressor variables). The following code shows an important class needed to implement multiple regression from `sklearn.linear_model` library. Once you have created model object next step is to fit the model using `fit` method passing `x_train` and `y_train` as a parameters.

```
from sklearn.linear_model import LinearRegression
model=LinearRegression()
model.fit(x_train,y_train)
```

11.4.3 Polynomial Regression

This regression is a kind of linear regression in which there is the n th degree polynomial relation between independent variable x and dependent variable y and fits a nonlinear relationship among them where the conditional mean of y is denoted as $E(y|x)$. This Regression technique is generally applied when the hypothesis is curvilinear including only polynomial terms. Some basic examples of non-linear phenomena where polynomial regression can be used are Progression of disease epidemics, growth rate of tissues or Distribution of carbon isotopes in lake sediments, etc. The main aim of this type of regression is to model the expected value of a dependent variable y in terms of an independent variable x . The following code shows an important class needed to

implement polynomial regression from `sklearn.linear_model` library. Once you have created a model object the next step is to fit the model using the `fit` method passing `x_train` and `y_train` as parameters. Poly object is created to set the degree to be used for polynomial features using the `PolynomialFeatures` class. Next to scale the features `fit_transform()` method will be used to scale the training data.

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
model=LinearRegression()
model.fit(x_train,y_train)
poly = PolynomialFeatures(degree = 4)
X_poly = poly.fit_transform(X)
poly.fit(X_poly, y)
lin2 = LinearRegression()
lin2.fit(X_poly, y)
```

11.4.4. Regression Tree

All of the above regression methods used one dependent (response) variable and one or more independent (predictive) variables. The response variable is numeric. In a regression tree, the independent (input) variables are categories and continuous mixtures. It can be considered as a kind of decision tree designed to approx imatereal-valued functions rather than a regression tree classification method. Regression trees use the concept of binary recursive partitioning. This is an iterative process that divides the data into branches and continues to divide these partitions into smaller groups as each point moves. All training dataset records are usually grouped in the same partition. This algorithm chooses a partition that, on average, minimizes the sum of squared deviations in two separate partitions. This same split rule applies in each new branch and the process carries on until each node becomes the terminal node. The following below code shows the important classes needed to implement a regression tree (`DecisionTreeRegressor`) imported from the `sklearn.tree` library, `train_test_split` imported from the `sklearn.model_selection` library for training and splitting test data. Similarly to calculate error metrics import `r2_score` and `mean_squared_error` from `sklearn.metrics` library. Once the Decision Tree Model object is created, fit the model using the `fit` method passing `x_train` and `y_train` as a parameters.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error
DecisionTreeModel= DecisionTreeRegressor()
DecisionTreeModel.fit(x_train, y_train)
```


11.5 Classification

Classification refers to a predictive modeling problem where a class label is predicted for a given example of input data. Classification algorithms can be used when the output variable is binary, multi-class, multi-Label or imbalanced etc. For example, Yes-No, Male-Female, True-false, etc. Some classification algorithms commonly used in machine learning are given below:

1. Decision Trees
2. Random Forest
3. K nearest neighboring
4. Support vector Machines

11.5.1 Decision Trees

Decision Tree is a tree structure like a flowchart, with inner nodes representing functions (or properties), branches representing decision rules, and each leaf node representing an outcome. The top node in a decision tree is called the root node. We need to partition our data into subsets containing instances with the same value, called entropy, built top-down from the decision tree root node. The information gain is based on a decreased value in entropy after the data set is partitioned on attributes. The construction of a decision tree will find the attribute (i.e., the most homogeneous point) that will return the best information gain, called the information gain. Generally, decision trees are used to visually represent and communicate decisions. The decision tree is used as the predictive model when working with machine learning and data mining. This model maps to the conclusions about the target value of the observed data in the data. The goal of decision tree learning is to create a model that predicts target values based on input variables. The following code shows the important class (DecisionTreeClassifier) needed to implement the decision tree algorithm imported from the sklearn. tree library. Once you have created model object next step is to fit the model using fit method passing x_train and y_train as parameters.

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(x_train, y_train)
DecisionTreeClassifier(class_weight=None, criterion='gini ',
                        max_depth=None, max_features=None,
                        max_leaf_nodes=None, min_impurity_decrease=0.0,
                        min_impurity_split=None, min_samples_leaf=1,
                        min_samples_split=2,
                        min_weight_fraction_leaf=0.0,
                        presort=False, random_state=None, splitter='best')
```

Example of the Decision Tree algorithm is given as follows. Here, the benchmark IRIS dataset is considered. Some samples from IRIS dataset are given in Table 1. The sepal length, sepal width, petal length and petal width are the independent variable and class is the dependent variable. Firstly, important libraries are imported, then read the dataset using read_csv(). Next is to split the dataset into train and test. Import Decision Tree Classifier class, fit the model and predict the model.

Table 1: Samples from IRIS dataset

	sepal length	sepal width	petal length	petal width	class
0	4.9	3.0	1.4	0.2	Iris-setosa
1	4.7	3.2	1.3	0.2	Iris-setosa
2	4.6	3.1	1.5	0.2	Iris-setosa
3	5.0	3.6	1.4	0.2	Iris-setosa
4	5.4	3.9	1.7	0.4	Iris-setosa

The Python program to classify the data is given as follows.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
df = pd.read_csv('iris.data')
col=['sepal length','sepal width', 'petal length', 'petal width', 'class']
df.columns=col
x=df.iloc[:,0:4]
y=df.iloc[:,4:]
from sklearn.model_selection import train_test_split
x_train , x_test , y_train , y_test = train_test_split(x,y,test_size=0.4, random_state= 5)
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(x_train, y_train)
```

```
pred= model.predict(x_test)
```

```
pred= model.predict(x_test)
pred
```

```
array(['Iris-virginica', 'Iris-virginica', 'Iris-virginica',
      'Iris-setosa', 'Iris-virginica', 'Iris-versicolor', 'Iris-setosa',
      'Iris-virginica', 'Iris-setosa', 'Iris-virginica',
      'Iris-virginica', 'Iris-virginica', 'Iris-virginica',
      'Iris-versicolor', 'Iris-setosa', 'Iris-setosa', 'Iris-virginica',
      'Iris-virginica', 'Iris-setosa', 'Iris-setosa', 'Iris-versicolor',
      'Iris-versicolor', 'Iris-setosa', 'Iris-versicolor',
      'Iris-versicolor', 'Iris-virginica', 'Iris-virginica',
      'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
      'Iris-setosa', 'Iris-versicolor', 'Iris-virginica', 'Iris-setosa',
      'Iris-versicolor', 'Iris-versicolor', 'Iris-setosa',
      'Iris-virginica', 'Iris-setosa', 'Iris-versicolor',
      'Iris-virginica', 'Iris-virginica', 'Iris-setosa', 'Iris-setosa',
      'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
      'Iris-versicolor', 'Iris-virginica', 'Iris-setosa',
      'Iris-versicolor', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
      'Iris-setosa', 'Iris-versicolor', 'Iris-virginica',
      'Iris-versicolor', 'Iris-versicolor', 'Iris-virginica'],
      dtype=object)
```

```
my_pred= model.predict([[4.9,3.0,1.4,0.2]])
my_pred
```

```
array(['Iris-setosa'], dtype=object)
```

11.5.2. Random Forest

Random forest is a method proposed in the 2000s by Leo Breimanto build a predictor ensemble on a decision tree set that grows in a randomly selected subspace of this data. Despite growing interest and practical use, the search for the statistical attributes of RandomForest is little known to the mathematical force that drives the algorithm. Random Forest is a kind of supervised machine learning algorithm. Machine learning algorithms in the field of data mining are widely used to analyze data and generate predictions based on this data. The ensemble algorithm Random Forest creates a multi-decision tree in the base categorizer and applies a majority to combine the results of the default tree. A key issue in determining a generalization error in a Random Forest classifier is the correlation between the strength of individual decision trees and the base tree. Below code shows an important class (RandomForestClassifier) needed to implement random forest algorithm imported from sklearn.ensemble library. Once the model object is created the next step is to fit the model using the fit method passing x_train and y_train as parameters.


```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(x_train, y_train)
```

The sample Python program for classification using Random Forest classifier is given as follows. In the example, the same IRIS dataset is used as described earlier. Firstly, important libraries are imported, then read the dataset using `read_csv()`. Next is to split the dataset into train and test. Import `RandomForestClassifier` class, fit the model and predict the model.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
df = pd.read_csv('iris.data')
col=['sepal length','sepal width', 'petal length', 'petal width', 'class']
df.columns=col
x=df.iloc[:,0:4]
y=df.iloc[:,4:]
from sklearn.model_selection import train_test_split
x_train , x_test , y_train , y_test = train_test_split(x,y,test_size=0.4, random_state= 5)
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(x_train, y_train)

my_pred= model.predict([[4.9,3.0,1.4,0.2]])
my_pred

array(['Iris-setosa'], dtype=object)
```

11.5.3 K-nearest neighboring

K-nearest neighbor or K-NN classification algorithm is basically used to create an imaginary boundary for classifying the data. Whenever any new data entry comes as an input, this algorithm tries to predict data which is nearest to the boundary line. Therefore, larger the value of *k* smoother the curves of separation resulting in less complex models. Whereas, smaller *k* values tend to overfit the data and result in complex models. It is very important to choose the right *k*-value while analyzing the dataset so as to avoid dataset overfitting and under-fitting. On the basis of historical data, models can easily be trained and can predict the future using KNN classifiers. Below code shows an important class (`KNeighborsClassifier`) needed to implement KNN algorithm imported from `sklearn.neighbors` library. Once the model object is created the next step is to fit the model using the `fit` method passing `x_train` and `y_train` as parameters. Some of the hyperparameter used are given below.

```
from sklearn.neighbors import KNeighborsClassifier
model= KNeighborsClassifier(n_neighbors=5,metric='minkowski',p=2)
model.fit(x_train, y_train)
```

There are following hyper parameters involved:

- **n_neighbors = 5** (Explanation : Number of neighbors to use by default for kneighbors queries.)
- **metric='minkowski'** , 'manhattan','euclidean' , 'hamming' (Explanation : different distance formula)
- **p=2** (Explanation : power only for minkowski)

Example of the KNN algorithm is given as follows. A dataset that contains records for diabetic patients is used here. The samples from the dataset are given in Table 2. Where glucose, blood pressure, skin thickness, insulin, BMI, DiabeticsPedigreeFunction, and Age are the independent variables and outcome is the dependent variable. Firstly, important libraries are imported, then read the dataset using read_csv(). Next is to split the dataset into train and test. Import KNeighborClassifier class, fit the model, predict the model, and at last confusion matrix and accuracy score for evaluating the model performance.

Table 2: Sample from Diabetics dataset

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
2	183	64	0	0	23.3	0.672	32	1
3	89	66	23	94	28.1	0.167	21	0
4	137	40	35	168	43.1	2.288	33	1

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
df = pd.read_csv('diabetes.csv')
x=df.iloc[:,0:7]
y=df.iloc[:,7:]
from sklearn.model_selection import train_test_split
x_train , x_test , y_train , y_test = train_test_split(x,y,test_size=0.4, random_state= 5)
from sklearn.neighbors import KNeighborsClassifier
model= KNeighborsClassifier()
model.fit(x_train, y_train)
pred= model.predict(x_test)
```



```

from sklearn.metrics import confusion_matrix, accuracy_score
confusion_matrix(y_test, pred)

array([[163, 44],
       [ 39, 62]], dtype=int64)

accuracy_score(y_test, pred)

0.7305194805194806

```

11.5.4 Support vector Machine

SVM is an influential machine learning method developed in statistical learning that has achieved considerable success in several areas. The foundation of SVM method has been developed and is gaining popularity in machine learning field due to its many amazing features. The SVM method has no limitations with data dimensions or with limited samples. Users generally need to perform a wide range of cross-validation to understand the optimal parameter settings. This process is typically known as model selection. The following code shows an important class (SVC) needed to implement SVM algorithm imported from *sklearn.svm* library. Once the model object is created the next step is to fit the model using the fit method passing *x_train* and *y_train* as parameters. Here, gamma ranges from 1 to 10.

```

from sklearn.svm import SVC
model = SVC(kernel='linear', gamma=1)
model.fit(x_train, y_train)

```

Let us classify the diabetes dataset using the SVM classifier. Firstly, important libraries are imported, then read the dataset using *read_csv()*. Next, split the dataset into train and test. Import SVC, fit the model, predict the model, and at last determine the accuracy score for evaluating the model performance.

```

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
df = pd.read_csv('diabetes.csv')
x = df.iloc[:, 0:7]
y = df.iloc[:, 7:]
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4, random_state=5)
from sklearn.svm import SVC
model = SVC(kernel='linear', C=100)
model.fit(x_train, y_train)

```

```
pred= model.predict(x_test)
from sklearn.metrics import confusion_matrix, accuracy_score
accuracy_score(y_test,pred)
```

0.7727272727272727

11.6 Merits and Demerits of Supervised learning

The performance of supervised learning algorithms is usually found good.

There are several advantages of supervised learning given as follows.

1. In supervised learning, the model can predict performance of the model based on the previous experience.
2. In supervised learning, we can have a detailed idea of classes of objects.
3. This type of machine learning algorithm helps us to solve different real world problems such as spam filtering, fraud detection, etc.

Despite above mentioned advantages of supervised learning methods, there are certain demerits of these algorithms given as follows.

1. Supervised machine learning algorithm is not suitable for handling complex tasks.
2. These models cannot predict exact output if the test data differs from the training data.
3. A supervised learning model requires lot of calculation time.
4. For applying supervised machine learning algorithms, we should have sufficient knowledge about the classes of objects.

Check your progress

1. Why was Machine Learning Introduced?
2. Give real examples of features and labels.
3. What are Different Types of Machine Learning algorithms?
4. Explain Supervised Learning and its types.
5. Explain the working of supervised learning with examples.
6. Explain the Difference Between Classification and Regression?
7. What are the different types of regression techniques in machine learning?
8. What are the different types of classification techniques?
9. Differentiate between linear regression and multiple regression.
10. What are the advantages and disadvantages of supervised learning?

11.7. Unsupervised Machine Learning

In case of supervised learning, models were trained using labeled data under the supervision of training dataset. However, if there is no labeled data available and you need to find hidden patterns in a specific data set. Therefore,

to deal with this type of problem in machine learning, we need an unsupervised learning method using the training data set. Instead, the model itself looks for hidden patterns in the given data. This process of learning can be compared to the learning that takes place in the human brain while learning new things. This technique cannot be directly applied to a regression or classification problem because there is the input data but no corresponding output data. For example, suppose that the unsupervised algorithm provides an input data that contains images of different types of cats and dogs. Here, Algorithms are not trained on a given input data. That is, it doesn't think about any thing about the features of the data set. The task of the unsupervised algorithm is to uniquely identify the features of an image itself by using the concept of clustering the images into the similar groups based on certain similar features. Compared to other learning algorithms, unsupervised learning algorithm is important due to the following reasons:

1. Unsupervised machine learning algorithm is used to find useful insights from the input data.
2. Unsupervised learning algorithm is very similar to learning that humans think through their own experiences and is closer to real AI.
3. Unsupervised learning algorithm is more important because it works with unlabeled and unclassified data.
4. In the real world, the output and input data are not always available. To solve such a case, it is necessary to adopt unsupervised learning techniques.

11.7.1 Working of Unsupervised Learning

The work flow of unsupervised learning algorithm can be understood from Figure 4 below. The input data is not labeled here. That is, it is not classified and its output is not displayed either. Currently, this unlabeled input data is sent to a machine learning model for training. It first interprets the raw data, finds hidden patterns in the data, and then applies appropriate algorithms such as decision trees, k-means clustering, and random forests, etc. When the appropriate algorithm is applied, the algorithm divides the data object into groups. Based on the similarities and differences between the objects, it gives an output as shown in Figure 11.4.

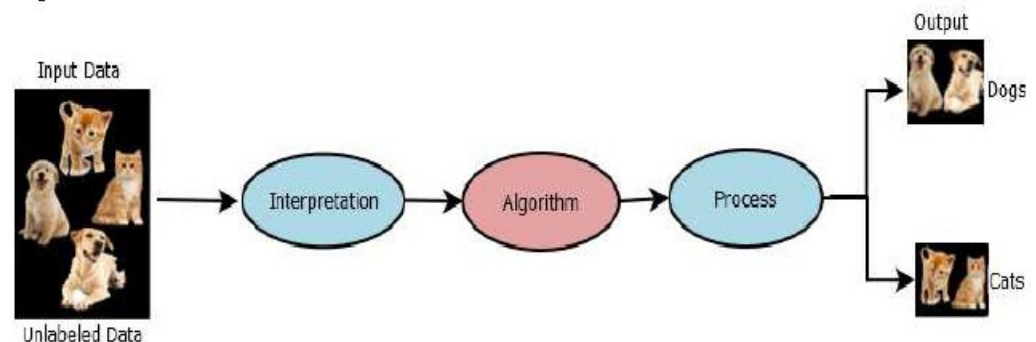


Figure 11.4: Unsupervised learning

11.7.2 Types of Unsupervised Learning Algorithms

The unsupervised learning algorithms are categorized into two types of problems as shown in Figure 11.5.

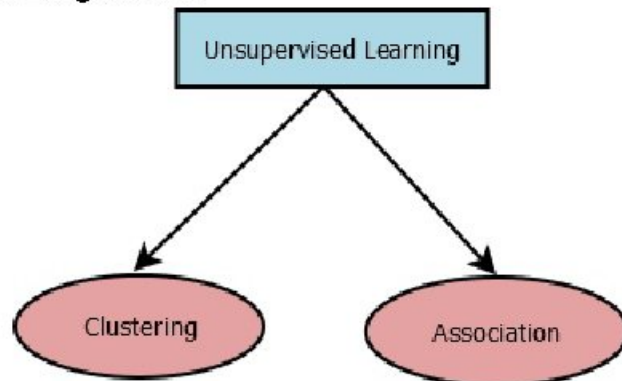


Figure 11.5: Types of Unsupervised learning

1. **Clustering:** Clustering is a method of grouping objects into clusters so that the objects with the highest similarity are moved to one group, whereas the objects with little or no similarity moved to another groups. Cluster exploration finds commonalities between the data objects and classifies them according to whether they have commonalities or not.
2. **Association:** An association rule is a method in unsupervised learning that is used to find relationship or associations between variables in large dataset. Association basically defines a set of items that occur together in a data set. For example, Anomaly detection or detection of bot activities, grouping images based on some similar patterns, Inventory management and etc.

11.8. Unsupervised Learning algorithms

Some popular unsupervised learning algorithms are discussed here. The major well known unsupervised algorithms are listed as follows.

1. K-means clustering
2. Mean Shift Clustering
3. Hierarchical clustering
4. Affinity propagation

11.8.1 K-means clustering

This algorithm is the simplest unsupervised machine learning. In K-means clustering inferences are made using input vectors without referring to outcomes. The 'k' in K-means is the number of clusters and 'means' is averaging of data or finding the centroid. Here, underlying patterns are

discovered using grouping of related data points. To do this, K-means observe for a fixed number of clusters (k) in the dataset. A cluster is a collection of data points combined together on the basis of certain similarities. Defining the number of clusters k is referred to as the number of centroids needed in the dataset. A centroid is an imaginary location signifying the center of a cluster. K-means clustering algorithm identifies the number of centroids and accordingly allocates the data points to its nearest cluster and sideways keeping centroids as small as possible. The following code shows k-Means clustering example by importing KMeans class from sklearn.cluster library and selecting random centroids which are used as the beginning point for clusters. Number of clusters taken here is 2.

```
from sklearn.cluster import KMeans
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0], [10, 2], [10, 4], [10, 0]])
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
kmeans.labels_

array([1, 1, 1, 0, 0, 0])

kmeans.predict([[0, 0], [12, 3]])

array([1, 0])

kmeans.cluster_centers_

array([[10.,  2.],
       [ 1.,  2.]])
```

The methods available in K-Mean clustering module in Python are as follows:

<code>fit(X[, y, sample_weight])</code>	Compute k-means clustering.
<code>fit_predict(X[, y, sample_weight])</code>	Compute cluster centers and predict cluster index for each sample.
<code>fit_transform(X[, y, sample_weight])</code>	Compute clustering and transform X to cluster-distance space.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, sample_weight])</code>	Predict the closest cluster each sample in X belongs to.
<code>score(X[, y, sample_weight])</code>	Opposite of the value of X on the K-means objective.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>transform(X)</code>	Transform X to a cluster-distance space.

11.8.2 Mean Shift Clustering

This category of clustering algorithm is also known as mode-seeking algorithm that assigns data points to closest clusters centroid by shifting points towards

the mode and direction towards this closest cluster is determined by points nearest by it. So, following this again and again each data point will move closer to the most points are at. This process will lead to the cluster center. Where, a mode has the highest density of data points in a region. This algorithm will stop when each and every point is assigned to a cluster. This algorithm has many applications in image processing and computer vision. Mean-shift does not need to specify the number of clusters in advance. The following code shows the Mean Shift clustering example by importing the MeanShift class from sklearn.cluster library and selecting random centroids which are used as the beginning point for clusters.

```
from sklearn.cluster import MeanShift
import numpy as np
x = np.array([[1, 1], [2, 1], [1, 0], [4, 7], [3, 5], [3, 6]])
clustering = MeanShift(bandwidth=2).fit(x)
clustering.labels_

array([1, 1, 1, 0, 0, 0], dtype=int64)

clustering.predict([[0, 0], [5, 5]])

array([1, 0], dtype=int64)

clustering

MeanShift(bandwidth=2)
```

The methods available in Mean Shift clustering module in Python are as follows.

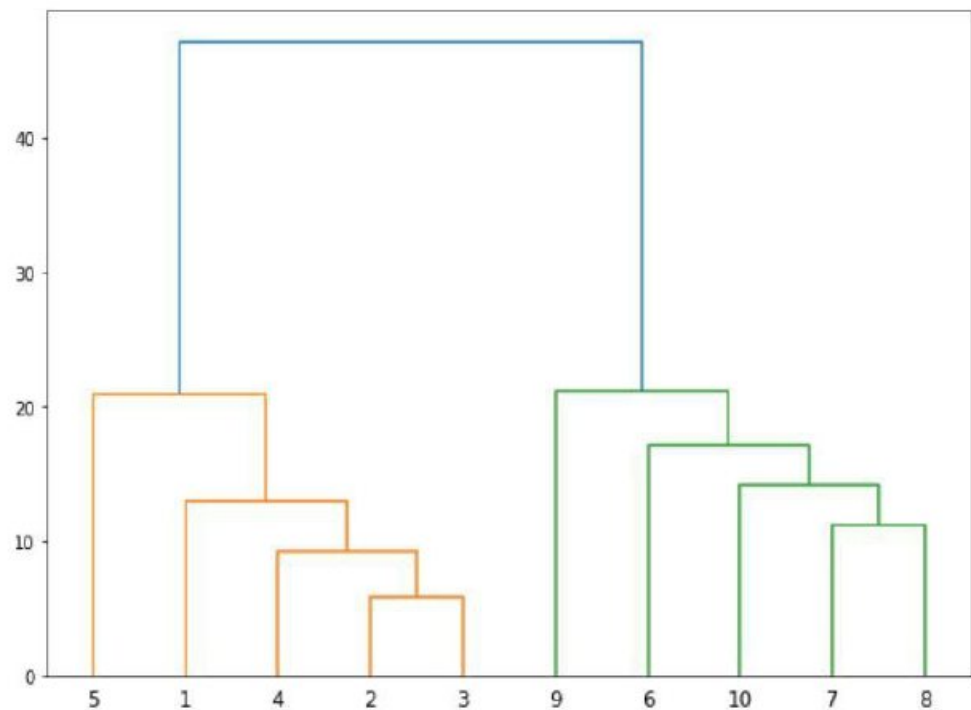
<code>fit(X[, y])</code>	Perform clustering.
<code>fit_predict(X[, y])</code>	Perform clustering on X and return cluster labels.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X)</code>	Predict the closest cluster each sample in X belongs to.
<code>set_params(**params)</code>)	Set the parameters of this estimator.

11.8.3 Hierarchical clustering

Hierarchical clustering is a kind of unsupervised machine learning algorithm and is used to group the unlabeled datasets in a cluster form. This algorithm is also known as **hierarchical cluster analysis**. Generally this algorithm is used to develop the hierarchy of clusters so as to form a shape of a tree which is known as dendrogram. The hierarchical clustering algorithm has two following approaches:

1. **Agglomerative:** This approach is a **bottom-up**. Where, the algorithm starts by considering all data points as single clusters and then keeps on merging them until one cluster is left.
2. **Divisive:** This algorithm is the converse of the agglomerative algorithm as it follows a **top-down approach**.

```
from scipy.cluster.hierarchy import dendrogram, linkage
from matplotlib import pyplot as plt
linked = linkage(X, 'single')
labellist = range(1, 11)
plt.figure(figsize=(10, 7))
dendrogram(linked,
            orientation='top',
            labels=labellist,
            distance_sort='descending',
            show_leaf_counts=True)
plt.show()
```



The above code shows hierarchical clustering and agglomerative clustering examples by importing dendrogram and linkage from sklearn.cluster.hierarchy. Where, a dendrogram is a figure that helps to know relationships between the hierarchical objects and is normally created as an output from hierarchical clustering. The main use of a dendrogram diagram is to know the best way to allocate objects to clusters. Where, Linkage or also known as single linkage hierarchical clustering, merges two clusters in each step whose two closest cluster members have the smallest minimum pairwise.

The following code shows agglomerative clustering example by importing AgglomerativeClustering from sklearn.cluster. In this type of

clustering, Clustering begins by calculating a distance between every pair of units that you want as a cluster as it is a bottom up approach and starts with smaller clusters, then merge them to create to bigger cluster.

```
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
import numpy as np
from sklearn.cluster import AgglomerativeClustering
X = np.array([[5,3],
              [10,15],
              [15,12],
              [24,10],
              [30,30],
              [85,70],
              [71,80],
              [60,78],
              [70,55],
              [80,91]],)
cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')
cluster.fit_predict(X)

array([1, 1, 1, 1, 1, 0, 0, 0, 0, 0], dtype=int64)

print(cluster.labels_)

[1 1 1 1 1 0 0 0 0 0]
```

11.8.4 Affinity propagation: Affinity Propagation was first published in 2007 by Brendan Frey and Delbert Dueck in Science. This algorithm does not need to specify the number of clusters in advance. Here, each data point sends messages to all other data points notifying its targets of each target's relative attractiveness to the sender. In response to this, each target responds to all senders of its readiness to associate with sender, given the attractiveness of the messages received from all other senders. The sender responds to each target, informing them of the target's revised relative attractiveness to the sender, based on the sender's access to all messages received from all targets. This message-passing procedure continues until a consensus is obtained. Once the sender is linked with its targets, then that target becomes the point's exemplar. All points with the same exemplar are located in the same cluster. Below code shows Affinity propagation example by importing AffinityPropagation from sklearn.cluster library. Where, a random state is taken as 5.


```

from sklearn.cluster import AffinityPropagation
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
clustering = AffinityPropagation(random_state=5).fit(X)
clustering

```

```
AffinityPropagation(random_state=5)
```

```
clustering.labels_
```

```
array([0, 0, 0, 1, 1, 1], dtype=int64)
```

```
clustering.predict([[0, 0], [4, 4]])
```

```
array([0, 1], dtype=int64)
```

```
clustering.cluster_centers_
```

```
array([[1, 2],
       [4, 2]])
```

The methods available in AffinityPropagation module in Python are as follows.

<code>fit(X[, y])</code>	Fit the clustering from features, or affinity matrix.
<code>fit_predict(X[, y])</code>	Fit the clustering from features or affinity matrix, and return cluster labels.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X)</code>	Predict the closest cluster each sample in X belongs to.
<code>set_params(**params)</code>	Set the parameters of this estimator.

11.8.5 Merits and Demerits of Unsupervised Learning

Like other techniques the unsupervised learning techniques also have some merits and demerits. The major advantages of unsupervised learning are as follows:

1. Unsupervised learning algorithm is used for more complex tasks than supervised learning, as we do not have input data labels in case of unsupervised learning.
2. Unsupervised Learning algorithm is the best to obtain unlabeled data compared to the labeled data.

The most important limitations of unsupervised learning are as follows:

1. Unsupervised machine learning algorithm is more difficult than supervised learning because it does not have the corresponding output.
2. The results of unsupervised machine learning algorithms may be less accurate, as the input data are unlabeled and algorithms do not know the output in advance.

11.9 Difference between Supervised and Unsupervised Learning

Supervised learning and unsupervised learning are two methods of machine learning. However, both methods are also used in different data sets in different

scenarios. Figure 11.6 shows a simple general example of supervised learning and unsupervised learning. Where in case of supervised learning, learning of students takes place in the supervision of a teacher whereas in case of unsupervised learning, learning is done by student itself without any such supervision of teacher.



Figure 11.6: Supervised Learning and Unsupervised

Another example of supervised learning is let's suppose, that there are images of different types of fruit and task of supervised learning algorithm is to identify and classify the different fruits images accordingly. In order for a supervised algorithm to recognize correctly we need to provide the input data as well as output data well in advance. To make the supervised model learn the fruit images we need to train the model by providing new set of fruits, so that, the model is trained according to the color, taste, shape, and size of each fruit. After the completion of training, we need to provide a new set of fruits in order to test the model. The model recognizes the fruit and uses the appropriate algorithm to predict the output. On the other hand if we talk the same thing in unsupervised learning, then there will be no corresponding output data for each input data. Unsupervised learning algorithm learns itself on the basis of similar patterns of images and on the basis of those similar patterns it recognizes the correct output.

11.10 Summary

The purpose of machine learning is to recognize the data structure and fitting that data into a model that people can understand and use. In machine learning algorithms, features and labels play an important role. Where, a feature is an input variable—the x variable Or a column name in the training dataset. There can be a single or multiple features depending on the complexity of the project. Data used in different machine learning algorithms can be labeled or unlabeled. Where, a labeled data includes both features and the label, whereas, an unlabeled data contains features but not the label. Machine learning

algorithms can be supervised, unsupervised and reinforcement. To train the model, supervised learning requires supervision, similar to how a student learns in the presence of a teacher. A supervised learning algorithm's goal is to appropriately transfer the input variable to the output variable. There are two sorts of issues in supervised learning: regression and classification. One of the most significant fields in statistics and machine learning is regression analysis. When the output variable is a continuous value, such as "height," "salary," or "weight," regression procedures are applied. It attempts to fit data using the best hyper-plane that passes through the points. And classification is a predictive modeling issue in which a class label for a given example of input data is anticipated. Classification algorithms can be used when the output variable is binary, multi-class, multi-Label or imbalanced etc. Some classification algorithms commonly used in machine learning are decision trees, random forest, K nearest neighboring and support vector machines. A decision tree is a tree structure that looks like a flowchart, with an internal node representing a feature (or attribute), a branch representing a decision rule, and each leaf node representing the outcome. Random Forest is a prediction ensemble that consists of a set of decision trees that grow in randomly selected data subspaces. The K-nearest neighbor algorithm is used to create an imaginary boundary for classifying the data. Whenever any new data entry comes as an input, this algorithm tries to predict data which is nearest to the boundary line. SVM is an influential machine method developed from statistical learning and has made substantial achievements in some fields. SVM method does not suffer the limitations of data dimensionality and limited samples. In case of supervised learning, models are basically trained using labeled data under the supervision of machine learning training dataset. However, there are many cases where there is no labeled dataset and there is a need to find hidden patterns in a specific dataset. Therefore, to handle these types of problems, we need unsupervised learning methods in which the models are unsupervised using training dataset. The unsupervised learning algorithm can be further categorized such as K-means clustering, Mean Shift Clustering, Hierarchical clustering, and Affinity propagation. K-means clustering is the simplest unsupervised machine learning. Where, inferences are made using input vectors without referring to outcomes. The 'k' in K-means is the number of clusters and 'means' is averaging of data or finding the centroid. The Mean Shift Clustering algorithm assigns data points to closest clusters centroid by shifting points towards the mode and direction towards this closest cluster is determined by points nearest by it. Hierarchical clustering is used to group the unlabeled datasets in a cluster form. Affinity Propagation was first published in 2007 by Brendan Frey and Delbert Dueck in Science. This algorithm does not need to specify the number of clusters in advance.

Review questions

- Q1. Explain Unsupervised Learning and its types?
- Q2. Explain the Difference between Clustering and association.
- Q3. What are the strengths of unsupervised learning?
- Q4. What are the difference between k-Means clustering and mean shift clustering?
- Q5. Explain hierarchical clustering and its types.
- Q6. Differentiate between supervised and unsupervised learning.
- Q7. Explain the working of unsupervised learning with example.
- Q8. What are the advantages and disadvantages of unsupervised learning?

Unit 12: Regression and Classification in Machine Learning

Structure

- 12.1 Simple Linear Regression Introduction
- 12.2 Objective
- 12.3 Multiple Regression
- 12.4 Data collection for Machine Learning
- 12.5 Classification
- 12.6 Features and Types
- 12.7 Summary

Unit 12: Regression and Classification in Machine Learning

12.0 Introduction

One of the most popular statistical method is regression that is useful in finance, investing and in many other disciplines. This method determines the strength as well as the relationship between the independent and dependent variables. The variables used to obtain the output are independent variable and variable which are dependent on independent variables are dependent variables. Regression algorithms are something that are most commonly used for training process of machine learning models. For example, simple linear regression, multiple regression, polynomial regression, lasso, ridge, etc. When a set of machine learning algorithms allowed to predict an outcome variable (y) based on one or more predictor variables (x) then this process is known as regression analysis. The main aim of regression algorithms is to build a mathematical equation that can define y (dependent variable) as a function of x (independent variable) in such a way that this equation can predict the outcome variable (y) on the basis of predictor variables or input variables (x). Regression algorithm can be of different types depending upon the relationship between dependent and independent variables. Simple linear regression is a regression method is used when the regression model is linear or sloped straight line. The key point in this type of Regression is that the dependent variable (y) must be a continuous value whereas; the independent or input variable (x) can be measured on continuous or categorical values. This type of regression is the simplest technique used for predicting a continuous variable assuming linear relation between outcome and predictor variables.

When we get output using more than one independent variable then it is known as multiple linear regression. Multiple regression indicates that there is linear relationship between the features and output, which is its limitation. There can be many features such as $x_1, x_2, x_3, \dots, x_n$ and output feature as y . If data points are clearly not fitted in a straight line or the relationship between the outcome and the predictor variables is not linear, then it might be case of polynomial regression and spline regression. Similar to linear regression, polynomial regression uses the relationship between the variables independent variable x and independent variable y to find the best way to draw a line through the data points.

12.1 Objectives

The major objectives of this unit are as follows.

1. To model the relationship between the two variables.
2. To implement major machine learning algorithms.

3. To make predictions for sting new observations using machine learning techniques.

12.2 Simple Linear Regression Model

A simple linear regression model is generally expressed using the following given below equation.

$$y = a_0 + a_1x + \varepsilon$$

where a_0 is the regression line's intercept (which is determined by setting $x=0$), a_1 is the regression line's slope, which indicates whether the line is rising or falling, and ε = the error term. Let's imagine a dataset has two variables: Numbers (dependent variable) and Result (independent variable) (Independent variable). The major goal of this issue are:

- To see if there is a link between the two variables.
- To change the independent variable to watch how the dependent variable changes.

To implement a Simple Linear regression model in machine learning using Python, we need to consider the following steps.

- 1 Firstly, import the required libraries needed for loading the dataset, plotting graphs, and creating a Regression model. Use ctrl+ENTER to execute the line of code.

```
import pandas as pd
import matplotlib as plt
import seaborn as sns
import numpy as np
%matplotlib inline
```

- 2 Next, load and read the dataset csv file using pandas library:

```
df= pd.read_csv('data.csv')
```

```
df.head()
```

	Number	Result
0	1	2
1	2	4
2	3	6
3	4	8
4	5	10

The above result shows the first five records in the dataset, with two variables: Number and Result using head () method.

3 After reading the dataset, extract the dependent and independent variables from the dataset. The following code is given for extracting the independent variable is x (Result), and the dependent variable is y (Result) using iloc as shown here.

```
x= df.iloc[:, :1]
```

x

Number	
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
10	11
11	12
12	13
13	14

```
y= df.iloc[:, 1:]
```

Above code shows the *iloc* function used for extracting features and response vectors. Where, is used to extract all rows from the dataset and 1: is used to extract the response column. By executing the above line of code, we can see the output for x variable and y variable as:

y

Result	
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20
10	22
11	24
12	26
13	28

Now, the x (independent) variable and y (dependent) variable are available.

4 Next step is to split both variables x and y into the training set and test set in order to train the model using a training dataset and then test the model using a test dataset. This task is done using `train_test_split` imported from `sklearn.model_selection`. The code for this execution is given as follows.

Syntax

- `from sklearn.model_selection import train_test_split`
- `X_Train, X_Test, y_train, y_test= train_test_split(X,y,test_size=0.4,random_state=3)`

```
from sklearn.model_selection import train_test_split

x_train , x_test, y_train, y_test= train_test_split(x,y,test_size=0.4, random_st

x_test
```

Number	
5	6
1	2
7	8
2	3
10	11
13	14

By executing the above code, the dataset is splitted into x-test, x-train and y-test, y-train. Now, Fit the Linear Regression to the training dataset. To do so, import the `LinearRegression` class from `sklearn.linear_model` library i.e. `scikit learns`. Once the required class is imported, an object of the class named as a model will be created. The code for this is shown below:

```
from sklearn.linear_model import LinearRegression
```

```
model=LinearRegression()
```

syntax : `model.fit(X_Train,y_train)`

- Model Parameter
- `fit_intercept` is a Boolean (True by default) that decides whether to calculate the intercept b_0 (True) or consider it equal to zero (False).
- `normalize` is a Boolean (False by default) that decides whether to normalize the input variables (True) or not (False).
- `copy_X` is a Boolean (True by default) that decides whether to copy (True) or overwrite the input variables (False).
- `n_jobs` is an integer or None (default) and represents the number of jobs used in parallel computation. None usually means one job and -1 to use all processors.

```
model.fit(x_train,y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                  normalize=False)
```

In the above code, to fit the Simple Linear Regression object to the training set a `fit()` method is used. Pass the `x_train` and `y_train` as parameters to the `fit()` function, which is the training dataset for the dependent and an independent variable. Now, fit the model object to the training set as shown in above code.

```
pred = model.predict(x_test)
```

5 Next step is the prediction of the test set (new observations) result from the dataset, by providing the test dataset to the model it can be checked whether it can predict the correct incorrect output. Now, Create `y_pred`, and `x_pred` vectors as a prediction vectors, that contains predictions of test dataset, and prediction of training set respectively.

```
pred
array([[12.],
       [ 4.],
       [16.],
       [ 6.],
       [22.],
       [28.]])
```

Above output shows the prediction vector using the 'pred' variable.

```
model.intercept_
array([-3.55271368e-15])

model.coef_
array([[2.]])
```

The model intercept and model coefficient are calculated when the above line of code is run. The coefficient is a number that describes the relationship between two variables that are unknown. If x is a variable. The number 2 represents the coefficient, while x is the unknown variable. Using `y_test` and prediction as parameters, the error metrics mean absolute error, mean squared error, and root mean square errors are calculated. Where, the mean absolute error (MAE) is the average of the absolute values of the differences between the forecast and the related observation over the verification sample. Mean squared error (MSE) is used to know how much the regression line is close to a set of points by calculating the distances from the points to the regression line and squaring them. This squaring is done in order to remove the negative signs. These calculations give the average set of errors. Therefore, MSE should be low for getting better forecasting results. Root mean square error (RMSE) is used to measure the average magnitude of the set of errors. RMSE is also used to express the difference between square values of forecast and corresponding observed values and then finding the average values over the sample. At last,

the square root of the calculated average is taken. This is used where large errors are undesirable. For this, import metrics class from *sklearn* library to find out mean absolute error, mean squared error and root mean square error as shown in below code with their respective formulae.

1. Mean Absolute Error

Mean Absolute Error (MAE) is the average of the absolute values of the differences between the predicted and the related true observation. It is defined as follows.

$$MAE = \frac{1}{n} \sum_{i=1}^n |Y_i - y_i|$$

Where Y_i is predicted value and y_i is true observation for i^{th} sample.

2. Mean Squared Error

Mean squared error (MSE) is the average of the square of the difference between predicted value and true value of a sample. The mathematical formula to calculate the MSE is given as follows.

$$MSE = \frac{1}{N} \sum_{i=1}^n (Y_i - y_i)^2$$

Where Y_i is predicted value and y_i is true observation for i^{th} sample.

3. Root Mean Square Error

The root mean square error (RMSE) is the standard deviation of the prediction errors. It is defined as follows.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (Y_i - y_i)^2}$$

Where Y_i is predicted value and y_i is true observation for i^{th} sample.

Code for linear regression:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
df=pd.read_csv("data.csv")
x=df.iloc[:, :1]
y=df.iloc[:, 1:]
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size=0.4, random_state=3)
from sklearn.linear_model import LinearRegression
model=LinearRegression()
model.fit(x_train, y_train)
prediction=model.predict(X_Test)
```


Check your progress

1. Apply linear regression on the following dataset.
 $X = (1, 2, 3, 4, 5, 6, 7, 8, 8, 9)$
 $Y = (2, 3, 4, 5, 5, 6, 6, 7, 8, 9)$
Calculate the following.
 - (a) Coefficient of Determination
 - (b) Intercept
 - (c) Mean absolute error
 - (d) Mean squared error
 - (e) Root mean square error
2. Check whether linear regression is suitable here or not?
 $x = [88, 44, 35, 37, 96, 11, 67, 35, 39, 21, 27, 30, 49, 65, 7, 6, 37, 67, 73, 41]$
 $y = [22, 47, 4, 36, 68, 96, 54, 73, 59, 11, 27, 35, 91, 34, 39, 21, 57, 3, 48, 16]$
3. Apply the regressor.
 $x = [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19, 21, 22]$
 $y = [100, 90, 80, 60, 60, 55, 60, 65, 70, 70, 75, 76, 78, 79, 90, 99, 99, 100]$

12.3. Multiple Regressions

Multiple regression is quite similar to simple linear regression but the only difference is that in Simple linear regression there is only one dependent variable and one independent variable. If “Person job” is explained by the “education” of an individual then the simple regression is expressed in terms of linear regression as follows:

$$\text{PersonJob} = b_0 + b_1 * \text{Education} + e$$

When variables are one on both the sides then simple linear regression can be used but if there is a need to go one step further and there are more than two or more independent variables, use multiple regression. For example, if experience is added as an additional variable in calculating the Person job variable then the above equation will be framed as below:

$$\text{PersonJob} = b_0 + b_1 * \text{Education} + b_2 * \text{Experience} + e$$

Problem domains of Regression-based machine learning:

1. Quantifying relationships: Generally, correlation is used finding the strength (in terms of a number between 0 and 1) and direction (positive or negative) of the relationship between two variables. But multiple regression is used to quantify this relationship.
2. Prediction: Everything in Machine learning is prediction. Simple regression can quantify the relationships; this capability of regression can be useful in solving prediction problems. That means, if the education level and number of years the person experiences an individual is known, then it is possible to predict the level of salary of an individual.
3. Forecasting: This domain for multiple regression is forecasting with time series analysis. For forecasting domain problems, Vector auto regression and panel data regression are used using the concept of principles of multiple regression.

12.3.2 Implementation of Multiple Regression Algorithm using Python

Let's suppose a housing dataset has 13 variables consisting of one dependent variable and remaining as an Independent variable. The goal of this problem is:

- 1 To find out if there is any correlation among the variables.
- 2 How the dependent variable is changing by changing the independent variable.

To implement the multiple linear regression model in machine learning the following below steps are followed using Jupyter notebook as an IDE in python. Firstly, import the required libraries needed for loading the dataset, plotting graphs, and creating multiple regression models as shown in the following code. Use CTRL+ENTER to execute the line of code.

```
# import all libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')

# read the dataset
df = pd.read_csv('housing.data', header=None, delim_whitespace=True)

df.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

The description of dataset used is as follows.

1. Dataset Name: Boston Housing Data

2. Sources:

(a) Origin: From the StatLib library maintained at Carnegie Mellon University.

(b) Creator: Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics & Management, vol.5, 81-102, 1978.

(c) Date: July 7, 1993

3. Relevant Information: Concerns housing values in suburbs of Boston.

4. Number of Attributes: 13 continuous attributes

7. Attribute Information:

1. CRIM Per capita crime rate by town
2. ZN Proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS Proportion of non-retail business acres per town
4. CHAS Charles River dummy variable (= 1 if tract bounds river; 0

otherwise)

- 5. NOX Nitric oxides concentration (parts per 10 million)
- 6. RM Average number of rooms per dwelling
- 7. AGE Proportion of owner-occupied units built prior to 1940
- 8. DIS Weighted distances to five Boston employment centres
- 9. RAD Index of accessibility to radial highways
- 10. TAX Full-value property-tax rate per \$10,000
- 11. PTRATIO Pupil-teacher ratio by town
- 12. B $1000(B_k - 0.63)^2$ where B_k is the proportion of blacks

by town

- 13. LSTAT Percentage of lower status of the population
- 14. MEDV Median value of owner-occupied homes in \$1000's

Here, df is the dataframe consist of the csv file which is loaded using read_csv function as shown in above executed code. Using head () function first five records can be seen that doesn't have any header assigned initially. To apply the column names to the dataframe the following code is used.

```
# applying the column name to data frame
```

```
col = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
```

```
df.columns= col
```

```
df.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

- After reading the dataset, extract the dependent and independent variables from the dataset. Below is the code for extracting independent, and the dependent variable using the ilocfunction.

```
x = df.iloc[ :, 0:13]
x.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33

```
y = df.iloc[ :,13:]
y.head()
```

	MEDV
0	24.0
1	21.6
2	34.7
3	33.4
4	36.2

- **Next step is to split both variables x and y into the training set and test set in order to train the model using a training dataset and then test the model using a test dataset. This task is done using the train_test_split class imported from sklearn. model_selection library. The code for this execution is given as follows.**

```
from sklearn.model_selection import train_test_split
```

```
x_train ,x_test, y_train , y_test = train_test_split(x,y, test_size=0.4,random_state=7)
```

```
x_train.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
140	0.29090	0.0	21.89	0	0.624	6.174	93.6	1.6119	4	437.0	21.2	388.08	24.16
194	0.01439	60.0	2.93	0	0.401	6.604	18.8	6.2196	1	265.0	15.6	376.70	4.38
102	0.22876	0.0	8.56	0	0.520	6.405	85.4	2.7147	5	384.0	20.9	70.80	10.63
258	0.66351	20.0	3.97	0	0.647	7.333	100.0	1.8946	5	264.0	13.0	383.29	7.79
174	0.08447	0.0	4.05	0	0.510	5.859	68.7	2.7019	5	296.0	16.6	393.23	9.64

```
y_train.head()
```

	MEDV
140	14.0
194	29.1
102	18.6
258	36.0
174	22.6

By executing the below code, the dataset is splitted into x-test, x-train and y-test, y-train. Now, Fit the Linear Regression to the training dataset. To do so, import the LinearRegression class from sklearn.linear_model library i.e. scikit learns. Once the required class is imported, an object of the class named as a model will be created. The code for this is shown as follows.


```
from sklearn.linear_model import LinearRegression
```

```
# step 1 : create the variable which act as your linear regression model  
model = LinearRegression()
```

```
# step 2 : fit training data in model variable  
model.fit(x_train, y_train)
```

```
# step 3 : get prediction on testing data  
pred = model.predict(x_test)
```

- Next step is the prediction of the test set (new observations) result from the dataset, by providing the test dataset to the model it can be checked whether it can predict the correct incorrect output. Now, Create **y_pred**, and **x_pred** vectors as a prediction vectors, that contains predictions of test dataset, and prediction of training set respectively.

```
pred
```

```
array([[22.56741764],  
       [18.85005445],  
       [19.89979188],  
       [18.85482469],  
       [ 2.42498435],  
       [10.89840921],  
       [20.98572821],  
       [29.12340185],  
       [29.11245011],  
       [13.54398291],  
       [ 5.67176253],  
       [33.27601599],  
       [18.58717034],  
       [20.51612589],  
       [37.57373449],  
       [22.94627462],  
       [29.40260562],  
       [33.44937425],  
       [10.76534814],
```

Above output shows the prediction vector using 'pred' variable.

```
from sklearn import metrics
```

```
metrics.mean_absolute_error(y_test, pred)
```

```
3.5197077872895233
```

```
metrics.mean_squared_error(y_test, pred)
```

```
27.82287608224572
```

```
## root mean squared error  
np.sqrt(metrics.mean_squared_error(y_test, pred))
```

```
5.274739432639846
```

Mean absolute error, Mean squared error and Root mean square error is used to find out the error metrics using `y_test` and `prediction` as parameters. For this, import `metrics` class from `sklearn` library to find out mean absolute error, mean squared error and root mean square error as shown in above code. The code for multiple regression is given as follows.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
df = pd.read_csv('housing.data', header=None, delim_whitespace=True)
df.head()
col=['CRIM','ZN','INDUS','CHAS','NOX','RM','AGE','DIS','RAD','TAX','PTRATIO','B','LSTAT','MEDV']
df.columns=col
x = df.iloc[:, 0:13]
x.head()
y = df.iloc[:,13:]
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size=0.4, random_state=3)
from sklearn.linear_model import LinearRegression
model=LinearRegression()
model.fit(x_train, y_train)
prediction=model.predict(x_test)
from sklearn import metrics
metrics.mean_absolute_error(y_test,pred)
metrics.mean_squared_error(y_test,pred)
np.sqrt(metrics.mean_squared_error(y_test,pred))
metrics.r2_score(y_test,pred)
model.coef_
```

12.3.3. Implementation of Polynomial Regression

Firstly, import the required libraries needed for loading the dataset, plotting graphs, and creating multiple regression models as shown in the following code. Use `ctrl+ENTER` to execute the line of code.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
datas = pd.read_csv('Data.csv')
datas.head()
```

	S.no.	weight	height
0	1	40	80
1	2	41	78
2	3	42	72
3	4	43	82
4	5	44	88

```
x = datas.iloc[:, 1:2].values
y = datas.iloc[:, 2:].values
```

Here, `data` is the data frame consisting of the `csv` file which is loaded using `read_csv` function as shown in above executed code. Using `head()` function first five records can be seen. After reading dataset, extract the dependent and

independent variables from the dataset. X and y are variables in code for extracting independent, and the dependent variable using iloc function.

By executing the following code, Fit the Linear Regression to the training dataset. To do so, import the Linear Regression class from sklearn.linear_model library i.e. scikit learns. Once the required class is imported, create an object named as lin. After fitting linear regression to the dataset now fit polynomial regression to the dataset using Polynomial Features from sklearn.preprocessing library. The code for this is given as follows.

```
# Fitting Linear Regression to the dataset
from sklearn.linear_model import LinearRegression
lin = LinearRegression()
lin.fit(X, y)

LinearRegression()

# Fitting Polynomial Regression to the dataset
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree = 4)
X_poly = poly.fit_transform(X)
poly.fit(X_poly, y)
lin2 = LinearRegression()
lin2.fit(X_poly, y)

LinearRegression()
```

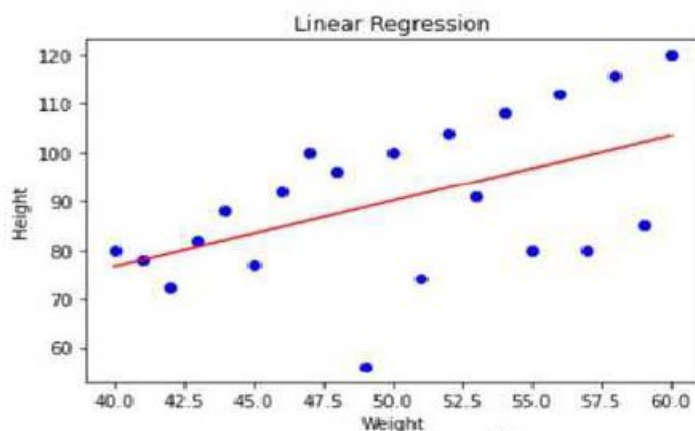
The result of the above code can be visualized as follows.

```
# Visualising the Linear Regression results
plt.scatter(X, y, color = 'blue')

plt.plot(X, lin.predict(X), color = 'red')
plt.title('Linear Regression')
plt.xlabel('Weight')
plt.ylabel('Height')

plt.show()
```

The graph plotted using the above code is shown as follows.



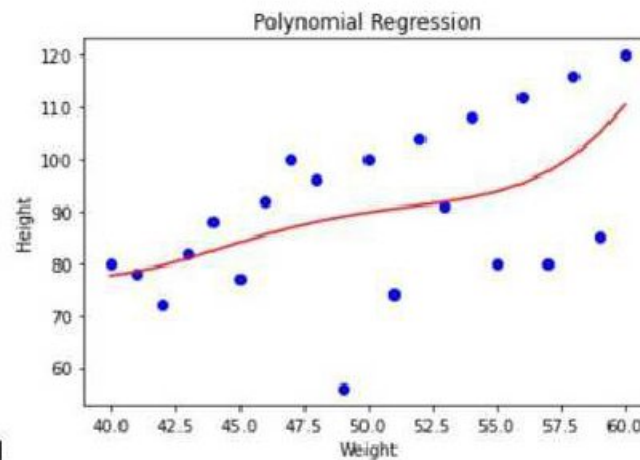
The Python code for polynomial regression is given as follows.

```

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
# Importing the dataset
datas = pd.read_csv('Data.csv')
datas.head()
X = datas.iloc[:, 1:2].values
y = datas.iloc[:, 2:].values
# Fitting Linear Regression to the dataset
from sklearn.linear_model import LinearRegression
lin = LinearRegression()
lin.fit(X, y)
# Fitting Polynomial Regression to the dataset
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree = 4)
X_poly = poly.fit_transform(X)
poly.fit(X_poly, y)
lin2 = LinearRegression()
lin2.fit(X_poly, y)
# Visualising the Linear Regression results
plt.scatter(X, y, color = 'blue')
plt.plot(X, lin.predict(X), color = 'red')
plt.title('Linear Regression')
plt.xlabel('Weight')
plt.ylabel('Height')
plt.show()

```

The output of the above code is given as follows.



12.4 Data Collection and Feature Extraction for Machine Learning

In order to enable machines to behave like humans, it is very important to focus on basic activities of machines that can be started from normal walking and talking. Just as the babies learn by absorbing similar patterns of information (data) and analyzing it to identify those patterns, machines also need the information (data) to enable them to learn using some pattern

information because without any relevant data there can be no machine learning. There are certain questions that emerge when we think about the data collection for training machines to learn patterns. Some of such question can be:

1. What kind of data should be collected to train the machine?
2. From where the data is collected?
3. How much data is required?
4. What are the data patterns or features?
5. How can the collected data be used to identify and extract features for the Machine Learning process?
6. How to prepare a dataset in order to get the most out of it?

12.4.1. Preparing and Collecting Data

Preparing and collecting the useful or relevant data there are certain steps needed to be followed. Some of the steps are given below:

1. Step 1: **Gathering the data.**
2. Step 2: **Handling missing data.**
3. Step 3: **Taking your data further with feature extraction.**
4. Step 4: **Deciding which key factors are important.**
5. Step 5: **Splitting the data into training & testing sets.**

Step 1: Gathering the data

The type of problem decides the choice of picking the right data. Both quantity and quality of dataset is important. Datasets can be created by your own or can be gathered from publically available dataset. Some datasets are public and free. Some of the popular websites for gathering dataset are:

1. Kaggle: Here the detailed dataset are available along with are the important details of features, data types, number of records, etc.
2. Reddit
3. Google Dataset Search
4. UCI Machine Learning Repository

The amount of data depends on the number of features existing in the dataset. More and quality data is always recommended for achieving good results. Feature selection technique can be used to filter the relevant data from big data. If the data is relevant and big enough with maintaining quality and quantity then important features can easily be identified and extracted. Features are basically the patterns found in the gathered dataset and these patterns help machines to be trained properly. Gathered data can be categorized in two types: Structured and Unstructured. Structured data is a well-defined type of data stored in a search friendly database and unstructured data is basically the data you collect and is not search friendly. Examples of structure data can be numbers, strings, dates, etc. and unstructured data can be data in text files, in emails, in videos, in music, in photos, etc.

Step 2: Handling missing data

This step includes the task of handling the missing. Handling of missing data should be done in the proper way otherwise it can cause disasters. Some of the

ways to handle missing data are:

1. Replacement of null value
2. deleting the whole missing record
3. Model based imputation
4. Interpolation \ Extrapolation
5. Forward filling \ Backward filling

Step 3: Taking data further with feature extraction

Feature extraction is one of the most important points that makes the dataset unique.

Step 4: Deciding which key factors are important

This step is the model's job. Dump the whole dataset and let the artificial intelligence be intelligent. Here, artificial intelligence decides true features that affect the output and which doesn't affect. Discard the unrelated data which is not relevant for extracting features.

Step 5: Splitting the data into training & testing sets

The well-known rule of splitting the data is 80 percent training & 20 percent testing sets.

12.5. Classification Algorithm in Machine Learning

The Supervised Machine Learning algorithms are divided into Regression and Classification. In Regression algorithms, the outputs are predicted for continuous values, whereas Classification algorithms are used to predict the categorical values. On the basis of trained data, Classification algorithms identify the new observation category. This is possible only when a program learns from the given observation and classifies new observations into a number of groups. Such as, identifying whether the mail is spam mail or not, whether the image is of cat or dog, Drugs Classification, Speech Recognition, Biometric Identification, Identifications of Cancer tumor cells etc. These classes can be called as categories or labels. Classification algorithm considers labeled input data that contains input data with the corresponding output data. The main aim of this algorithm is to recognize the class of a given dataset, and basically these algorithms are used to predict the output for the categorical data. Example of classification is shown in the figure below where there are two classes, class A and class B. Class A has all circle images and class B has all triangle images

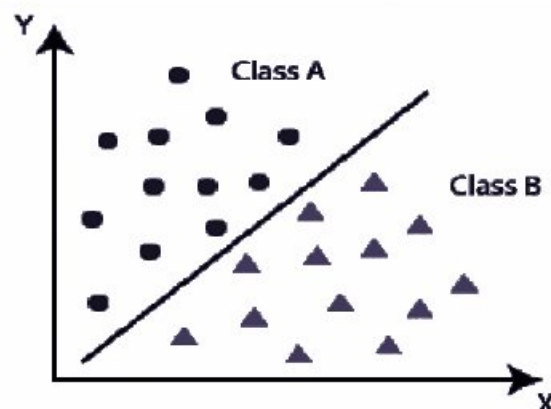


Figure 12.1: Classification in machine learning

The major classification techniques and their implementation in Python is already discussed in Unit 11.

12.6. Classification model Evaluation:

After the Classification or Regression model is completed, it is required to assess the model's performance. To do so, the following ways are there:

1. Cross-Entropy Loss or Log Loss: It is used for calculating the performance of a classifier, where the output is a probability value between 0 and 1. Value near to 0 is always considered as a good classification model. If in case the predicted value diverges from the actual value then the value of cross-entropy increases otherwise decreases. The model can predict more accurately if the loss is low.

2. Confusion Matrix: This matrix provides a confusion matrix or confusion table as an output and helps in describing the model performance. This matrix is also known as the error matrix. It consists of a summarized form of predictions with a total number of correct and incorrect predictions. Below is the confusion matrix in Table 12.1 form:

Table 12.1: Confusion matrix

	Actual Positive	Actual Negative
Predicted Positive	True Positive(TP)	False Positive(FP)
Predicted Negative	False Negative(FN)	True Negative(TN)

To calculate the model's accuracy below given formulae is used:

$$Accuracy = \frac{TP + TN}{Total\ Population}$$

3. AUC-ROC curve: Receiver Operating Characteristics Curve (ROC) and Area Under the Curve (AUC) is a graph used to show the performance of multi-class classification model at different thresholds. These curves are plotted using FPR (False Positive Rate) on X-axis and TPR (True Positive Rate) on Y-axis.

12.7. Features and Types

Features can be defined as the individual independent variables or columns of data that are found in the given dataset or problem set as the input. Features are

also known as attributes or dimensions. With the help of strong features, an accurate predictive model can be built. For example, in predicting the sale price of any house, size or locality can be the features of a house. A problem dataset can have several features tagged to it. To improve the accuracy of the model, it is essential to select the most relevant features which will also reduce the complexity of the model by avoiding the unnecessary feature data. This process of selecting only the relevant feature data is known as feature selection or feature engineering and becomes a crucial step of pre-processing. This process of feature engineering or feature selection in machine learning helps in obtaining new features from old features in machine learning. With the help of feature selection, a lot of time will be saved in calculating and collecting unusable patterns that in the later stage needed to be removed. This technique also simplifies the particular ML model and enables faster and more effective training by removing unused data and thus reducing overfitting problems. There are different types of features:

- In Simple Supervised learning algorithms, feature selection can be simple values like characters (locality) or numbers (size of house).
- In unsupervised learning algorithms, the model is self-trained to identify the features and work on it.
- For example, building a model that will tell whether to give a loan to a particular customer or not. Now, here the feature data can have many features attached to it. Such as Loan id, Customer id, Customer Name, Customer Address, Annual Income, credit score, current balance and so on. Using the concept of feature selection it can be identified that for a particular Customer, Annual Income, credit score, current balance can significantly contribute to the model accuracy rather than the other features. Therefore, these features become the main features for building the model for this particular domain.

Features are basically categorized in two major types, i.e. **structural and statistical features**. Statistical features involves of moments, zoning, characteristic loci and n-tuples. Process of extracting such features from images is known as feature extraction. Some of the commonly used feature extraction techniques are Zoning, Geometric moment invariants, Template matching, Graph description, shape features, histogram features, and etc. **Shape feature extraction** methods are further categorized in two groups, contour based methods and region based methods.

The contour-based technique only calculates the shape feature from the boundary of any shape, whereas, the region-based technique extracts the feature from the entire area. The spatial features are categorized by its spatial distribution, gray scale and amplitude. Where, amplitude is the most important characteristics of any object for example, in X-ray images, amplitude denotes the absorption feature of the body mass, which can distinguish bones and tissues.

There are other types of features, such as boundary features, edge features, histogram features, and etc. Where, histogram is basically the intensity value of the pixel that shows number of pixels for each and every intensity value. Besides this it also tells scattering of pixels among the grayscale values. An 8-bit grayscale image has 256 possible intensity values. Narrow histograms indicate low-contrast areas. Some common histogram characteristics are mean, energy, variance, skewness, median, and kurtosis. The transformation characteristics generally provide the frequency domain information of the data by transforming the image. Use area filtering to extract transformation characteristics from the image. This is also called a feature mask.

High frequency components are usually used for edge and boundary detection. Angle grooves can be used for direction detection for example, in signature recognition to know whether the signature direction is slightly in upward direction or downwards. When the input data comes from transformed coordinates, transformation feature extraction is also very important. Edge and boundary features are one of the most difficult tasks, making them an essential problem in image processing. The edge of an image is an area with strong intensity contrast, and the intensity jump from one pixel to the next can cause a significant change in image quality. Image edge detection significantly decreases the amount of data as well as it filters non-essential information while retaining important image attributes. Edges vary in size and the edges may contain other edges, but at certain sizes the edges are not yet wide. Accurately identifying the edges of an image makes it easy to locate all such objects and measure basic attributes such as area, perimeter, and shape. Therefore, edges are used in the extraction of boundary estimation and splitting in the scene. Some of the more features may include General features, color features, texture-based features, and etc. List of commonly used general features are described below.

1. **Area:** The area feature is basically used to calculate the actual number of pixels in the region.
2. **Eccentricity:** Eccentricity calculates the eccentricity of an ellipse with the same second-order moment as the functional area. Eccentricity is the ratio of the distance between the focal point of an ellipse and the length of its major axis. Values are 0-1. This property is only supported on 2D input images.
3. **MajorAxisLength:** Major Axis Length is the length (in pixels) of the major axis of the ellipse with a second moment equal to the area. This property is only supported on 2D input images.
4. **MinorAxisLength:** Minor Axis Length is the length, in pixels, of the major axis of the ellipse with a second moment equal to the area. This property is only supported on 2D input images.
5. **Orientation:** The direction is the angle (degrees) between the x-axis and the

- major axis of the ellipse that has the same moment of inertia as the region. This attribute is only supported for 2D input images.
6. **ConvexArea:** Number of pixels in 'Convex Image'. This property is only supported on 2D input images.
 7. **FilledArea:** Filled Area is used to calculate the number of pixels in Filled Image.
 8. **EulerNumber:** The number of entities in the Euler Number area minus the number of holes in that object. This attribute is only supported for 2D input images.
 9. **EquivDiameter:** Equiv Diameter is calculated over the diameter of a circle the same size as the area. It is calculated as $\sqrt{4 * \text{Area} / \pi}$. This property is only supported on 2D input images.
 10. **Centroid-Radii:** The basic idea is to use the centroid radii model to signify shape feature.

12.8. Summary

A type of regression method known as simple linear regression models the relationship between dependent and independent variables. Because the simple linear regression model reveals a linear or slanted connection, it is referred to as simple linear regression. The dependent variable must be continuous in order for simple linear regression to work. Multiple regression is quite similar to simple linear regression, with the exception that there is only one dependent variable and one independent variable in simple linear regression. Using y test and prediction as parameters, the error metrics mean absolute error, mean squared error, and root mean square error are calculated. The mean absolute error (MAE) is the average of the absolute values of the differences between the forecast and the related observation over the verification sample. The mean squared error (MSE) is used to determine how near a regression line is to a set of points by squaring the distances between the points and the regression line. Root mean square error (RMSE) is used to measure the average magnitude of the set of errors. In order to enable machines to behave like humans, it is very important to focus on basic activities of machines that can be started from normal walking and talking. Just as the babies learn by absorbing similar patterns of information (data) and analyzing it to identify those patterns, machines also need the information (data) to enable to learn using some pattern information because without any relevant data there can be no machine learning. For preparing and collecting the useful or relevant data there are certain step need to follow. Some of the steps are Gathering the data, handling missing data, taking your data further with feature extraction, deciding which key factors are important, splitting the data into training & testing sets. Once the data is prepared the next step is to extract features from the collected data. Feature extraction is one of the most important points that makes the dataset

unique. The Supervised Machine Learning algorithm is classified into Regression and Classification Algorithms. In Regression algorithms, the outputs are predicted for continuous values, whereas Classification algorithms are used to predict the categorical values. On the basis of trained data, Classification algorithms identify the new observation category. This is possible only when a program learns from the given observation and classifies new observations into a number of groups. The main aim of this algorithm is to recognize the class of a given dataset, and basically these algorithms are used to predict the output for the categorical data. For calculating the performance of a classifier, Cross-Entropy Loss is used where the output is a probability value between 0 and 1. Value near to 0 is always considered as a good classification model. For describing the performance confusion Matrix is used that provides a confusion table as an output and helps in describing the model performance. This matrix is also known as the error matrix. The AUC-ROC curve is used to show the performance of multi-class classification models at different thresholds. These curves are plotted using FPR (False Positive Rate) on X-axis and TPR (True Positive Rate) on Y-axis. Features are also known as attributes or dimensions. With the help of strong features, an accurate predictive model can be built. For example, in predicting the sale price of any house, size or locality can be the features of a house. A problem dataset can have several features tagged to it. To improve the accuracy of the model, it is essential to select the most relevant features which will also reduce the complexity of the model by avoiding the unnecessary feature data.

Review Questions

Q1. Why is linear regression useful and how are they used in real-world data science?

Q2. What is the importance of using linear and multiple regression?

Q3. Differentiate between linear and multiple regression with examples.

Q4. List all the steps of Preparing and collecting the data.

Q5. List the implementation step of linear and multiple regression.

Q6. Explain classification algorithms in detail.

Q7. Using the IRIS dataset evaluates the decision tree and random forest model's accuracy score. Source: <https://www.kaggle.com/uciml/iris>

Q8. Implement KNN and SVM algorithms on diabetics and evaluate the model's accuracy score.

Source: <https://www.kaggle.com/saurabh00007/diabetescsv>

Q9. Explain features and its types in detail.

Q10. Differentiate between structural and statistical features in detail.

Bibliography

1. Luciano Ramalho, "Fluent Python," O'Reilly Media, 2014.
2. Gowrishankar S. and Veena A., "Introduction to Python Programming," CRC Press, 2019.
3. H. Bhasin, "Python Basics," Mercury Learning and Information, 2019.
4. Nicholas Lacey, "Python By Example," Cambridge University Press, 2019.
5. Mark Lutz, Learning Python, Fifth Edition, O'Reilly, 2013.
6. David Beazley, Python Essential Reference, Third Edition, Sams Publishing, USA, 2006.
7. Wesley J. Chun, Core Python Programming, First Edition, Prentice Hall PTR, 2000.
8. Peter Harrington, Machine Learning in Action, Manning Publishing Company, 2012.
9. <https://www.python.org/>
10. <https://www.geeksforgeeks.org/>
11. <https://www.javapoint.com/>
12. <https://tutorialspoint.com/>
13. <https://en.wikipedia.org/>
14. <https://document.pub/>
15. <https://www.realpython.com/>