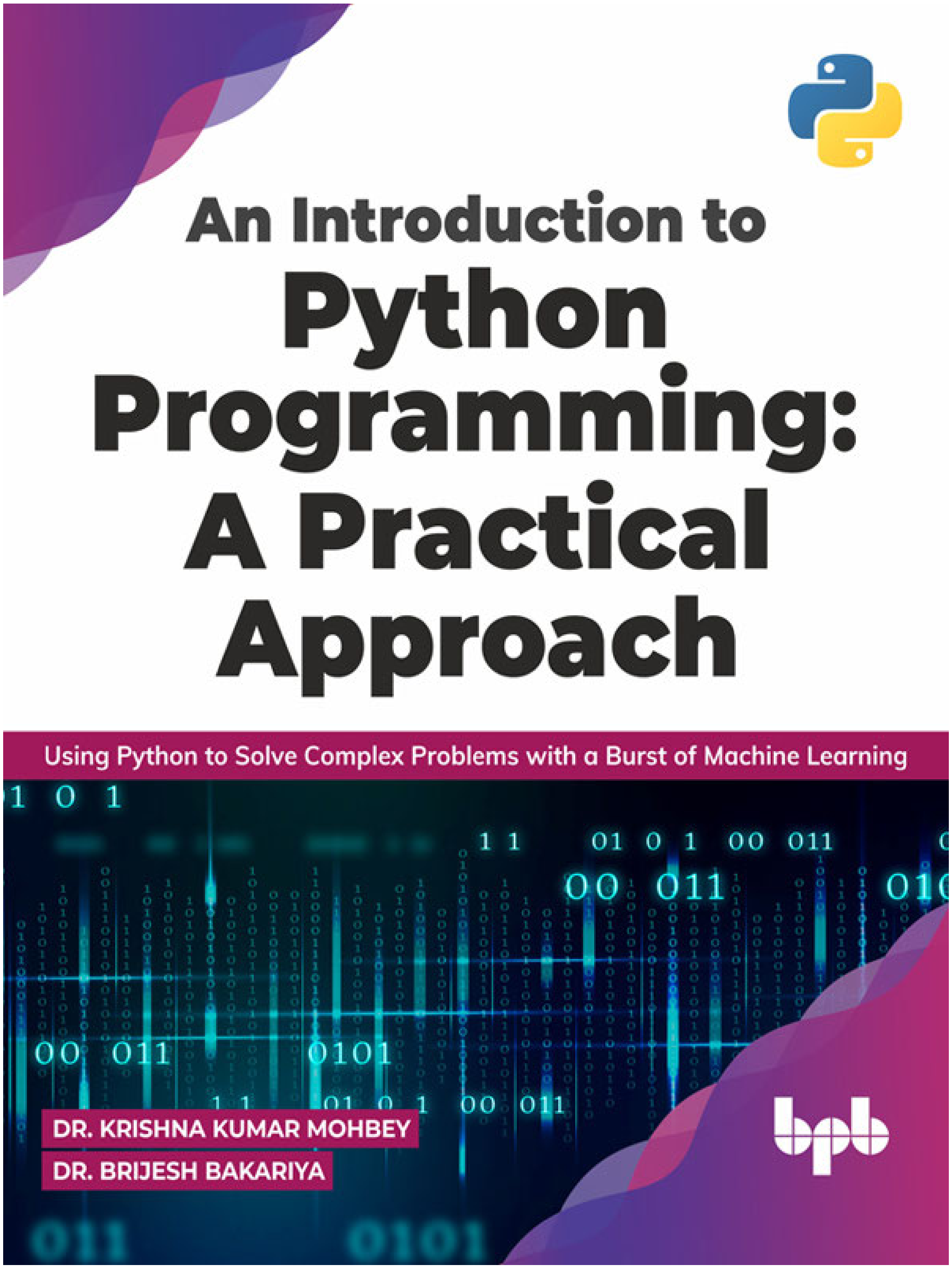


An Introduction to Python Programming: A Practical Approach

Using Python to Solve Complex Problems with a Burst of Machine Learning

DR. KRISHNA KUMAR MOHBEY
DR. BRIJESH BAKARIYA

An Introduction to Python Programming: A Practical Approach

Using Python to Solve Complex Problems with a Burst of Machine Learning

Dr. Krishna Kumar Mohbey

Dr. Brijesh Bakariya

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91392-062

Dedicated to

Our family members who ever supported us in all respects of life and career. This journey of ours proved to be a boon by following their words and experiences.

About the Authors

Dr. Krishna Kumar Mohbey is an Assistant Professor of Computer Science at the Central University of Rajasthan, India. He received his Bachelor's degree in Computer Applications from MCRPV Bhopal (2006), Master's in Computer Applications from Rajiv Gandhi Technological University Bhopal (2009) and Ph.D. from the Department of Mathematics and Computer Applications from National Institute of Technology Bhopal, India (2015). He has been teaching since 2009 and guiding Ph.D. students. His areas of interest are Machine learning, data mining, mobile web services, big data analysis and user behavior analysis. He has authored three books on different subjects and published more than 25 research articles in reputed journals and conferences.

Dr. Brijesh Bakariya is an Assistant Professor in the Department of Computer Science and Engineering, I.K. Gujral Punjab Technical University (IKGPTU) Jalandhar (Punjab). He completed his Ph. D. from Maulana Azad National Institute of Technology (NIT-Bhopal), Madhya Pradesh (2016). He received MCA Degree from Devi Ahilya Vishwavidyalaya, Indore, Madhya Pradesh (2009). He has been teaching since 2009 and guiding M.Tech/Ph.D. students. In the meantime, he has authored 1 book and published more than 15 research papers in the journals of international repute in the area of Data Mining, Image Processing, Machine Learning, and so on. Currently, three Ph.D. scholars are working with him. He has attended various short-term training programmes, refresher courses, workshops, seminars and conferences in India.

About the Reviewer

Riddhi Sahani is a strategic thinker who also happens to be an entrepreneur. She can still code like the best of them and understand what makes a product and business successful.

She is passionate about leading engineering groups that are talented, healthy and motivated to achieve great results. She is passionate about organizational health and principled leadership, believing that these are the most important factors for a team's ability to reach its full potential.

She believes that having a healthy, engaged and skilled team managed by leaders with indisputable integrity and full transparency makes success considerably more feasible and enjoyable.

She feels herself fortunate for being a part of numerous notable engineering teams, as well as having developed and led some outstanding ones. She prioritizes strong interpersonal ties and creates trust-based high-performing teams.

She is currently employed for News24 as a Senior Software Developer and Technical Leader. She has more than 8 years of software development experience with a variety of technologies, including AWS, ML, and DL. She has worked for well-known MNCs and a successful startup from stealth mode to successful exit.

Acknowledgement

This book is the culmination of few years of intense learning and research experience. We have been fortunate to interact with many people who have influenced us greatly. One of the pleasures of finally completing this book is getting an opportunity to thank them. We would like to place on record and acknowledge the works of all those great authors whose work we have referred to in the preparation of this book.

We would like to thank a few people for the continued and ongoing support they have given us during the writing of this book. First and foremost, we would like to thank our family members for continuously encouraging us to write the book — we could have never completed this book without their support.

We would like to express a deep sense of gratitude to Dr. G. S. Thakur, National Institute of Technology Bhopal, Madhya Pradesh, who constantly praised and inspired us to write books on technical subjects and whose enthusiasm and guidance led us to write this book.

Furthermore, we would like to thank the reviewers for providing helpful feedback and critical suggestions during the book's

development.

We thank our colleagues, students and research scholars who supported us by sharing their knowledge to write this book successfully. We hope this book will be a good starting point for the journey ahead.

We would extend our gratitude to the BPB Publications for bringing out the book in its present form.

Any suggestions for improving the contents would be warmly appreciated.

—Dr. Krishna Kumar Mohbey
— Dr. Brijesh Bakariya

Preface

It gives us immense pleasure to bring the book "An Introduction to Python Programming: A Practical Approach." Python is the most popular programming language and is widely used among programmers. The book is intended for the students of various courses who can use this high-level programming language as an effective tool for problem-solving. Python is used to develop applications of any stream, and it is not restricted only to computer science.

We believe that anyone with basic knowledge of computers and ability of logical thinking can quickly learn Python programming. With this motivation, we have lucidly written this book. Once you go through the book, you will learn the basics of python programming. Also, you will feel motivated to develop applications using Python.

This book has been written considering the readers have no prior knowledge of python programming. Following are some of the reasons why one should learn the python language:

Python programming is simple and easy to learn.

It has simple syntax compared to other programming languages.

It is an object-oriented programming language.

It is used to develop desktop, standalone and web-based applications.

Python is open source. Due to its open nature, one can write a program and deploy it on any platform.

This book is for everyone from the engineering and sciences background. It is also for B.Tech. B.E., BCA, BSc, M.Tech, PGDCA, M.E., MCA, M.Com., MSc, Ph.D., other UG and PG graduates. In particular, the introduction, programming skills, and learning concepts with simple examples would be good for beginners.

The book is divided into 13 chapters covering all aspects of Problem Solving with Python with a touch of Machine Learning. The details are listed below.

Chapter 1, Basics of Python Programming: This chapter gives you access to Python, from installing it to writing simple programs.

Chapter 2, Operators and Expressions: This chapter describes various operators used in Python with suitable examples.

Chapter 3, Control Flow Statements: This chapter describes the program's control flow in the order in which the code for the program is executed. It covers conditional statements, loops and function calls governing a Python program's control flow.

Chapter 4, Functions: Functions are a handy way to break the code into usable parts, enabling us to organize, make it more readable, reuse and save some time. Functions are also a crucial way of describing interfaces so that programmers can share their code. This chapter covers various functions that can be used in Python.

Chapter 5, Strings: This chapter provides descriptions and various operations on the string using Python.

Chapter 6, Lists: The list is the most flexible datatype available in Python and can be written between square brackets as a list of comma-separated values (items). This chapter covers various operations that can be applied on lists.

Chapter 7, **Tuple:** This chapter covers various operations that can be applied to tuples.

Chapter 8, Dictionaries: This chapter covers a description of dictionary data structure and various operations applied to the dictionary.

Chapter 9, File Handling: Python provides a built-in feature to build, write and read files. This chapter covers various operations that are used on files.

Chapter 10, Exception Handling, Modules, and Packages: You will learn Python's exception handling methods in this chapter. This chapter highlights exceptionally built-in Python classes and attempting and increasing excellent Python except in Python, along with Python, try-finally clause. Even this chapter concerns modules and packages.

Chapter 11, Object-oriented Programming: This chapter covers various concepts of the object-oriented paradigm. It contains multiple examples related to OOPs.

Chapter 12, Machine Learning with Python: This chapter covers various concepts of machine learning. It contains multiple models and examples of machine learning.

Chapter 13, Clustering with Python: This chapter covers concepts of unsupervised learning and clustering. It contains various models and examples of clustering approaches.

Table of Contents

CHAPTER 1

Basics of Python Programming

Nowadays, Python is one of the most popular programming languages. There are many reasons for this, but simply it is easy to read and write Python code. Python provides vast libraries for which you can efficiently make your program. In this chapter, you will see various features, installation, and the making of a simple program of Python.

Structure

In this chapter, we will cover the following topics:

Basic features and history of Python

Installing Python and running Python program to IDE

Writing and executing first Python program

Brief description about various concepts used in Python

Objective

The objective of this chapter is to introduce the concept of Python, its features, and the Installation procedure in Windows and Linux. After completing this chapter, you should be familiar with Python IDE and editors. Also, you will be able to write simple Python programs.

Understanding Python

Python is one of the high-level languages. There are various advantages of high-level languages:

Program is easier and understandable which is written in a high-level language

Program is shorter in a high-level language

The code written in a high-level language is portable, which means it can run on different computers with some modifications.

Due to these preceding advantages, almost all programs are written in high-level languages. *Guido van Rossum* designed Python in 1991; after that, Python released various versions as shown in *table*

Table 1.1: *Python versions with released year*

The compiler completes the task which is involved in the compilation process. Moreover, it converts a source code to object code, and after that, the program will be converted to machine understandable code. But an interpreter reads a high-level program and executes it. Python does not follow the compilation process; it is used only as an interpreter. That's why from a computation point of view, Python is very fast. The figures can easily understand it.

Here *figure 1.1* shows the structure of the compiler. A compiler translates source code into an object code, which a hardware executor runs:

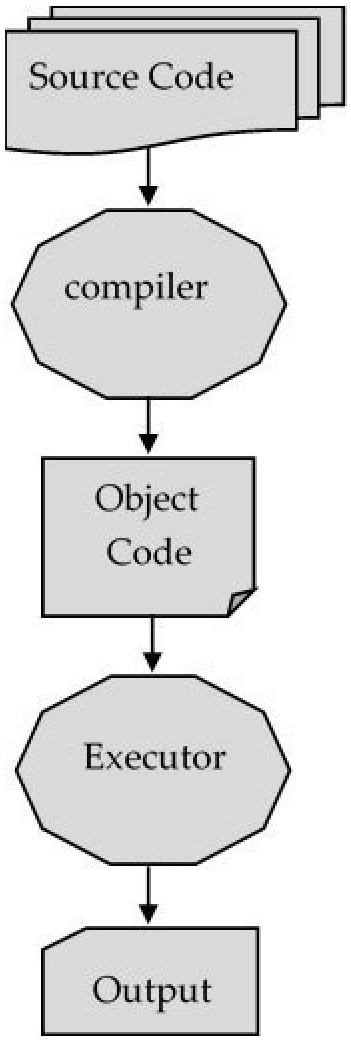


Figure 1.1: *Structure of Compiler*

Here *figure* shows the structure of the interpreter. An interpreter processes the program in less time, and it also takes less time to read lines and perform computations:

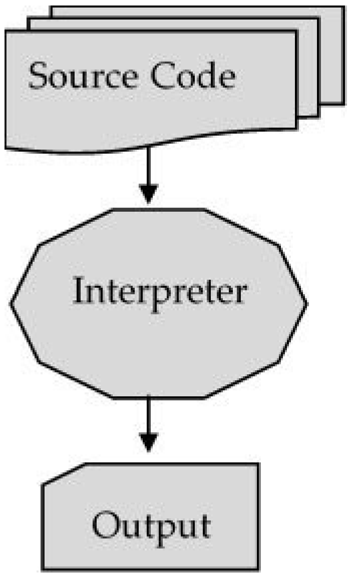


Figure 1.2: *Structure of Interpreter*

Python is an interpreted language because an interpreter executes Python programs, but the compilation process is also done in Python. The compilation part is hidden from the programmer that is why it is known as an interpreted language.

There are various reasons Python has become more popular:

It is easy to use because its coding is written in a simple English language. Users can easily understand and write code into Python, so it takes less time to write a program.

It is a very powerful language because it has many libraries where you can easily make a program. An inbuilt library has lots of inbuilt functions where you can insert as per your need.

It is an object-oriented programming language that is why it follows all the object-oriented concepts.

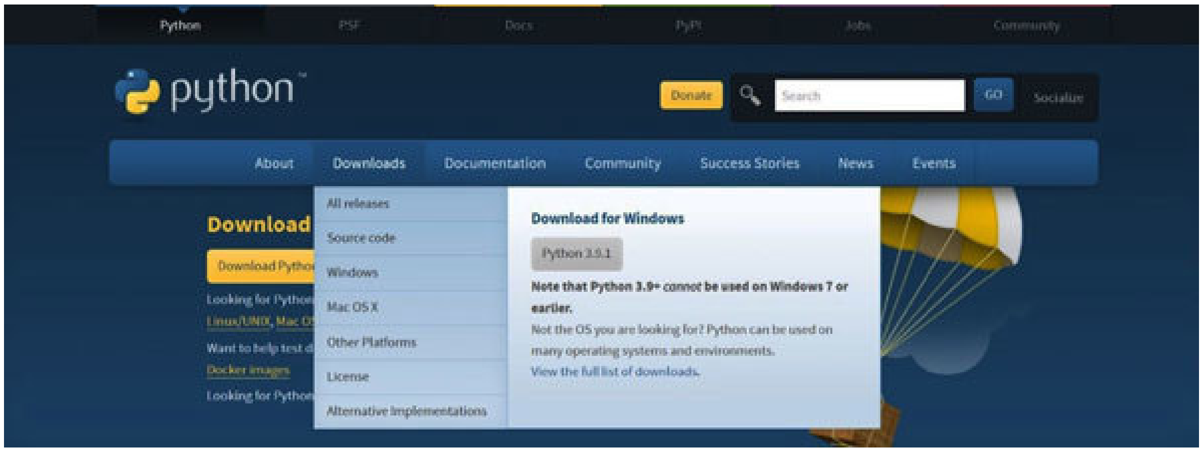
It is integrated with other programming languages such as C, C++, Java, etc.

It is platform-independent, which means if you created a program in one operating system, you could also run that program in another operating system.

It is free and open-source, which means you can install it on your computer without any cost. It can be copied or modified and resold accordingly. This is a solid reason why Python has become more popular.

Installing Python

It works with various operating systems Windows, Linux, macOS, etc. If you want to work in Python, it needs permission from the Python interpreter. Python is available on the website Download the right installer for your operating system and run it on the computer. This figure shows a clear understanding of the installer of Python. You can choose the Python installer according to the operating system:



Note that Python 3.9+ cannot be used on Windows 7 or earlier.

Not the OS you are looking for? Python can be used on many operating systems and environments.

View the full list of downloads.

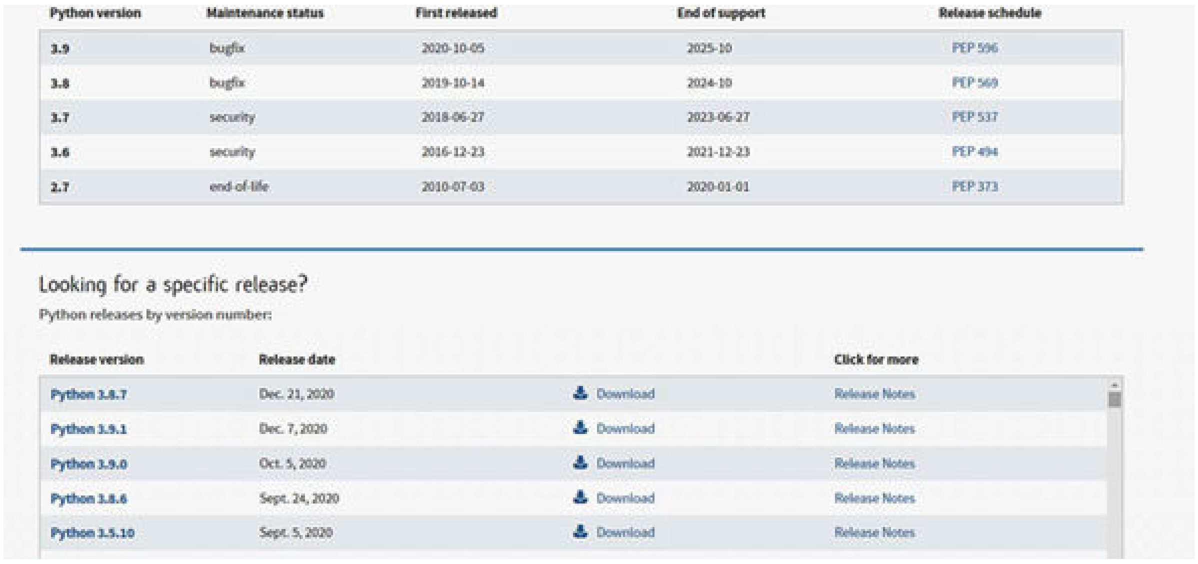
Figure 1.3: *Installation Page of Python*

Windows

There are various steps for installing Python in Windows. Python does not come preinstalled in Windows systems. However, installing Python is a straightforward process. All you need to do is to download the Python installer from the website, and then run the program. Here are the steps to install it on Windows:

Downloading the Python installer from

Select a Python version:



Python version	Maintenance status	First released	End of support	Release schedule
3.9	bugfix	2020-10-05	2025-10	PEP 596
3.8	bugfix	2019-10-14	2024-10	PEP 569
3.7	security	2018-06-27	2023-06-27	PEP 537
3.6	security	2016-12-23	2021-12-23	PEP 494
2.7	end-of-life	2010-07-03	2020-01-01	PEP 373

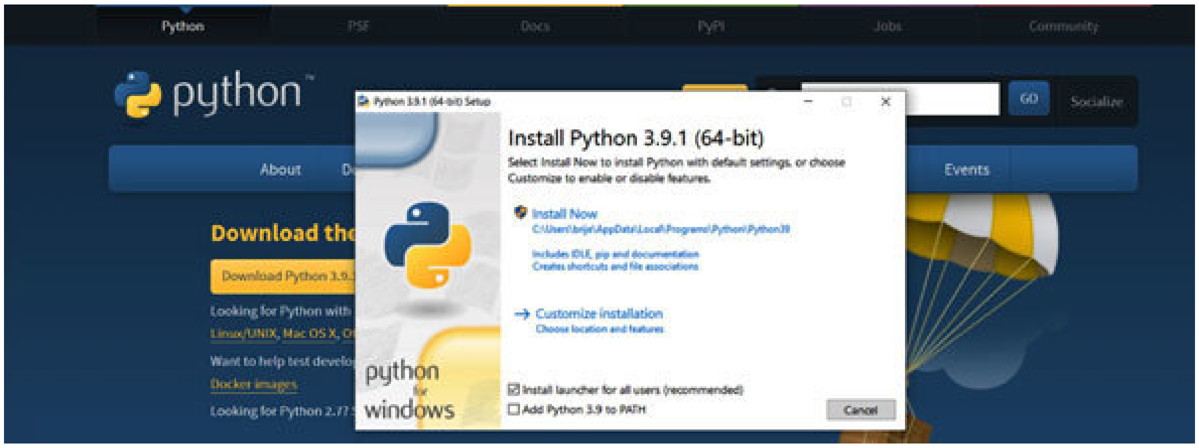
Looking for a specific release?

Python releases by version number:

Release version	Release date		Click for more
Python 3.8.7	Dec. 21, 2020	Download	Release Notes
Python 3.9.1	Dec. 7, 2020	Download	Release Notes
Python 3.9.0	Oct. 5, 2020	Download	Release Notes
Python 3.8.6	Sept. 24, 2020	Download	Release Notes
Python 3.5.10	Sept. 5, 2020	Download	Release Notes

Figure 1.4: *Python versions*

Select Windows 64-bit or 32-bit Windows:


Python 3.9.1 (64-bit) Setup

Install Python 3.9.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\brija\AppData\Local\Programs\Python\Python39

Includes IDLE, pip and documentation
Creates shortcuts and file associations

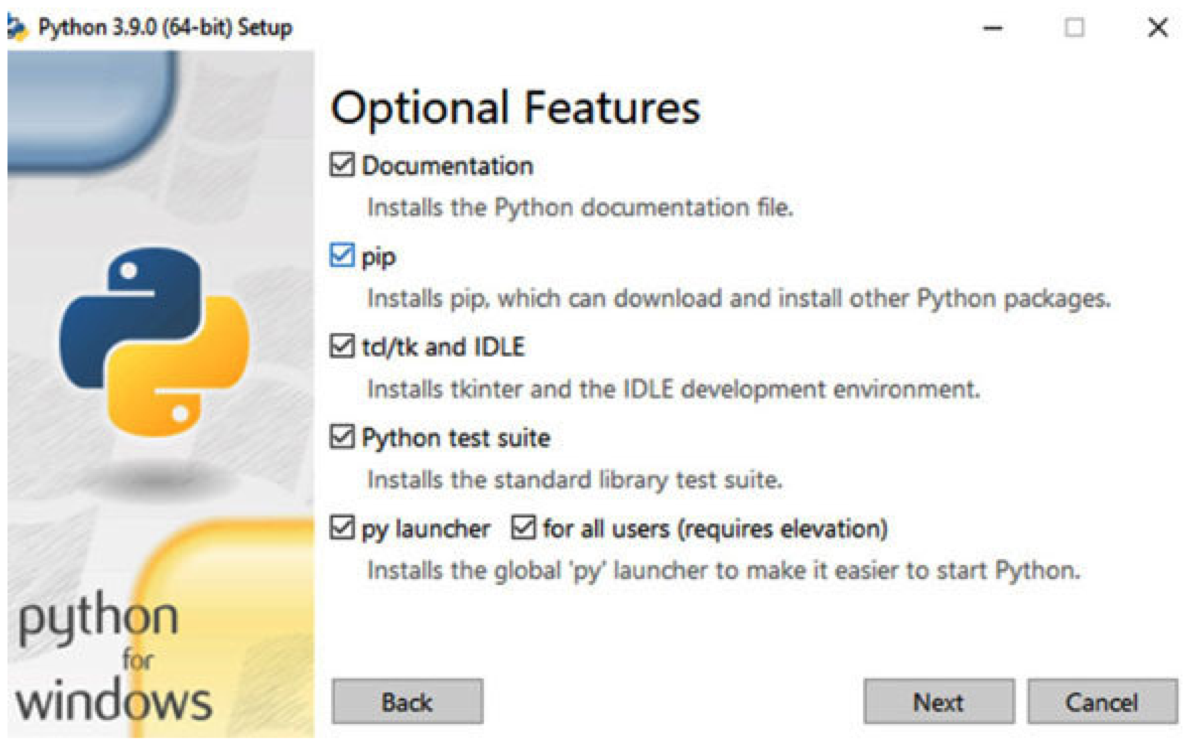
Customize installation
Choose location and features

[✓] Install launcher for all users (recommended)
[ ] Add Python 3.9 to PATH

Figure 1.5: *Installation Setup of Python*

Run the Installer.

Don't forget to click the box that indicates **Add Python Version to PATH** this will confirm that the interpreter is added to the execution path. The final step is to click **Install** and these are all the required steps:


Python 3.9.0 (64-bit) Setup

Optional Features

☑ Documentation
Installs the Python documentation file.

☑ pip
Installs pip, which can download and install other Python packages.

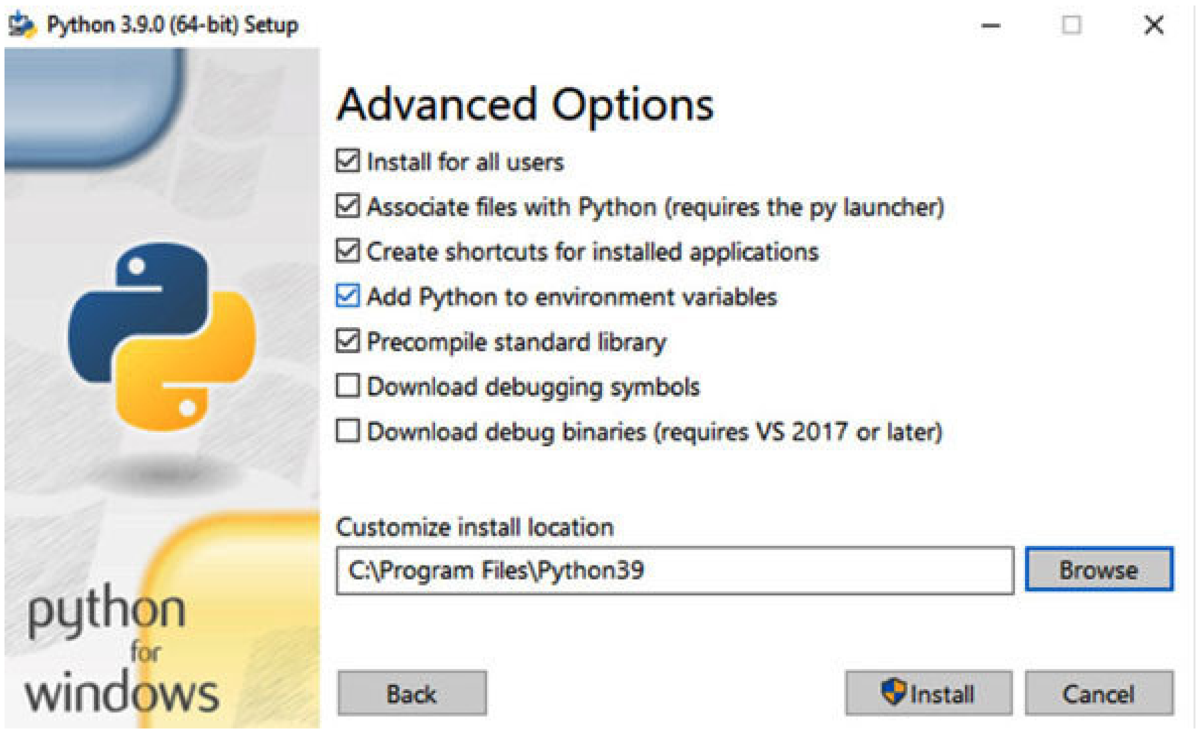
☑ tcl/tk and IDLE
Installs tkinter and the IDLE development environment.

☑ Python test suite
Installs the standard library test suite.

☑ py launcher ☑ for all users (requires elevation)
Installs the global 'py' launcher to make it easier to start Python.

Figure 1.6: *Running the Installer*

The *figure 1.6* shows various optional features during installation. We can select these features as required:


Python 3.9.0 (64-bit) Setup

Advanced Options

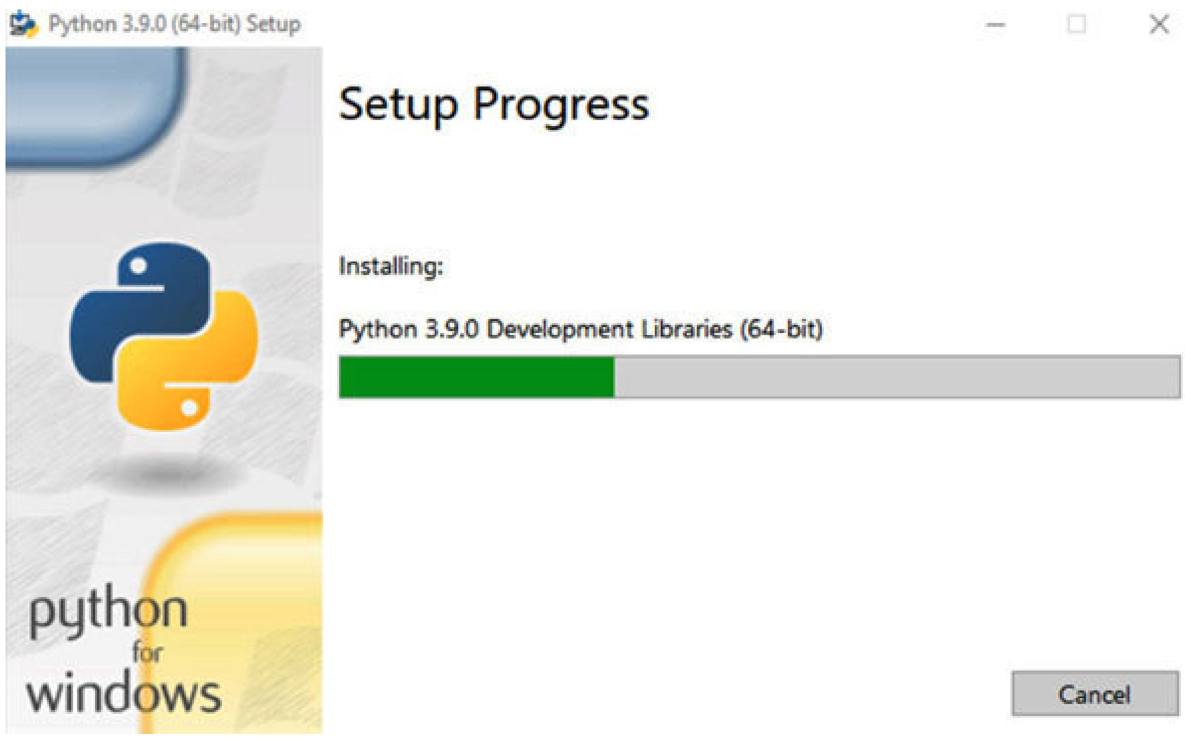
- [✓] Install for all users
- [✓] Associate files with Python (requires the py launcher)
- [✓] Create shortcuts for installed applications
- [✓] Add Python to environment variables
- [✓] Precompile standard library
- [ ] Download debugging symbols
- [ ] Download debug binaries (requires VS 2017 or later)

Customize install location

C:\Program Files\Python39

Figure 1.7: *Various Options in Python setup*

After selecting various features next window shows advanced options, which are shown in *figure* Here, installation location also needs to be selected:


Python 3.9.0 (64-bit) Setup

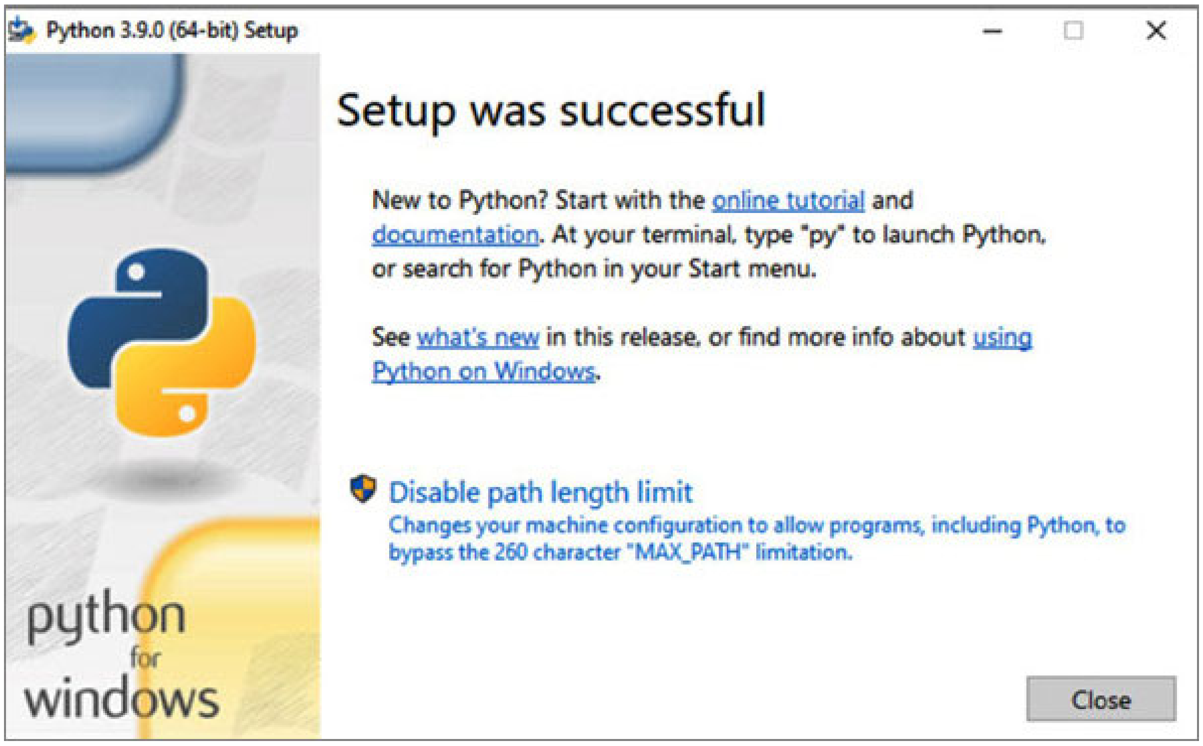
Setup Progress

Installing:

Python 3.9.0 Development Libraries (64-bit)

Figure 1.8: *Processing of Python Setup*

The *figure 1.8* shows the installation setup progress; after completion of this step, Python will be successfully installed in the system:


Python 3.9.0 (64-bit) Setup

Setup was successful

New to Python? Start with the online tutorial and documentation. At your terminal, type "py" to launch Python, or search for Python in your Start menu.

See what's new in this release, or find more info about using Python on Windows.

Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.

Figure 1.9: *Completion of Python setup*

The *figure 1.9* shows the completion of the Python setup in the system. Now the system is ready to use Python. Python application can be opened by searching from the taskbar; it is shown in *figure*

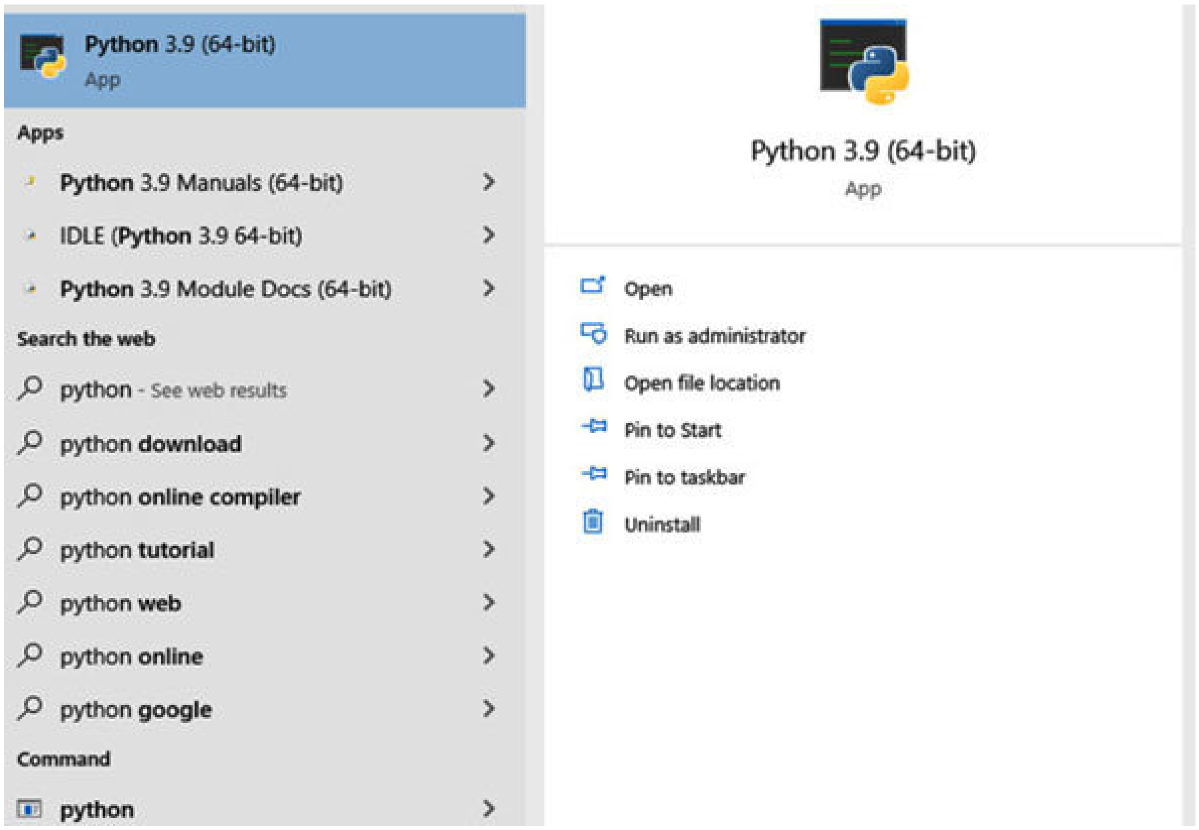


Figure 1.10: *Open Python application*

Ubuntu

There are different versions of the Ubuntu distribution, and the installations are different. Moreover, most factory versions of Ubuntu 18.04 or Ubuntu 20.04 come with Python preinstalled; first of all, check your version with the following commands and enter this command on the terminal:

```
python -version
```

Update and refresh repository lists

The next step is to update and refresh the list of the repository with the following command:

```
sudo apt update
```

Installation of supporting software

There is a following command:

```
sudo apt install software-properties-common
```

Add Deadsnakes PPA ((Personal Package Archive))

There is a following command:

```
sudo add-apt-repository ppa:deadsnakes/ppa
```

The system will prompt you to press *Enter* to continue. Do so, and allow it to finish. Refresh the package lists again:

```
sudo apt update
```

Install Python 3

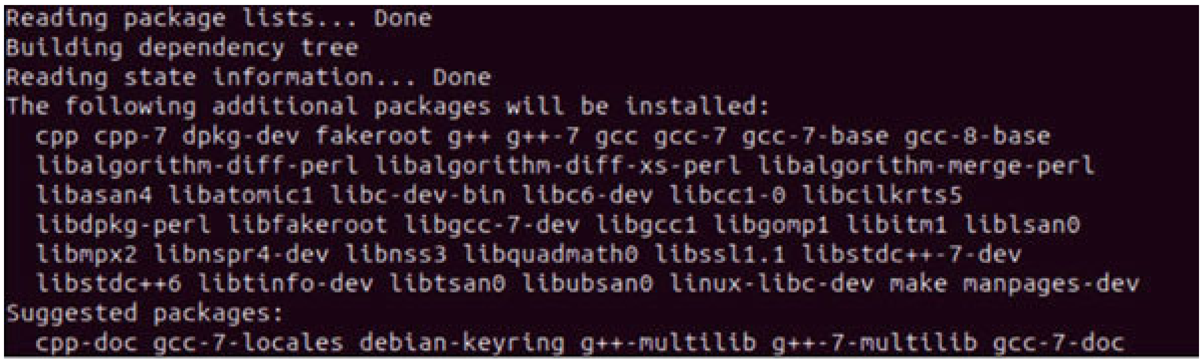
Now you can start the installation of Python 3.9 with the command:

```
sudo apt install python3.9
```

Allow the process to complete and verify if the Python version was installed successfully:

```
python -version
```

The preceding command shows the current version of Python:



```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  cpp cpp-7 dpkg-dev fakeroot g++ g++-7 gcc gcc-7 gcc-7-base gcc-8-base
  libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl
  libasan4 libatomic1 libc-dev-bin libc6-dev libcc1-0 libcilkrts5
  libdpkg-perl libfakeroot libgcc-7-dev libgcc1 libgomp1 libitm1 liblsan0
  libmpx2 libnspr4-dev libnss3 libquadmath0 libssl1.1 libstdc++-7-dev
  libstdc++6 libtinfo-dev libtsan0 libubsan0 linux-libc-dev make manpages-dev
Suggested packages:
  cpp-doc gcc-7-locales debian-keyring g++-multilib g++-7-multilib gcc-7-doc
```

Figure 1.11: *Python installation through sudo command on Ubuntu*

Linux Mint

Installation of Mint is similar to Ubuntu, and installation instruction for Ubuntu can also be used in Mint. A deadsnakes/ppa working well with Mint.

Python IDLE

Python introduces IDLE, which means **Integrated Development and Learning** By default, Python contains the IDLE module for windows, and it is not contained in Linux. A development environment provides an interactive platform that contains various tools that makes a program straightforward. There are two types of modes in IDLE which is interactive mode and script mode:

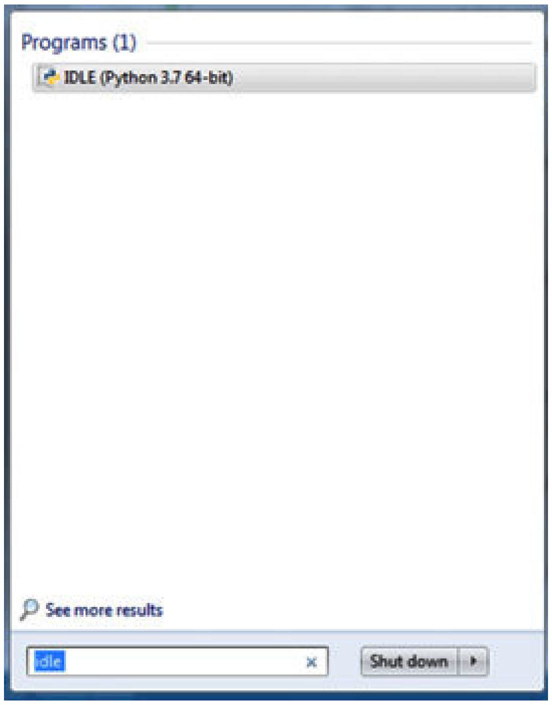
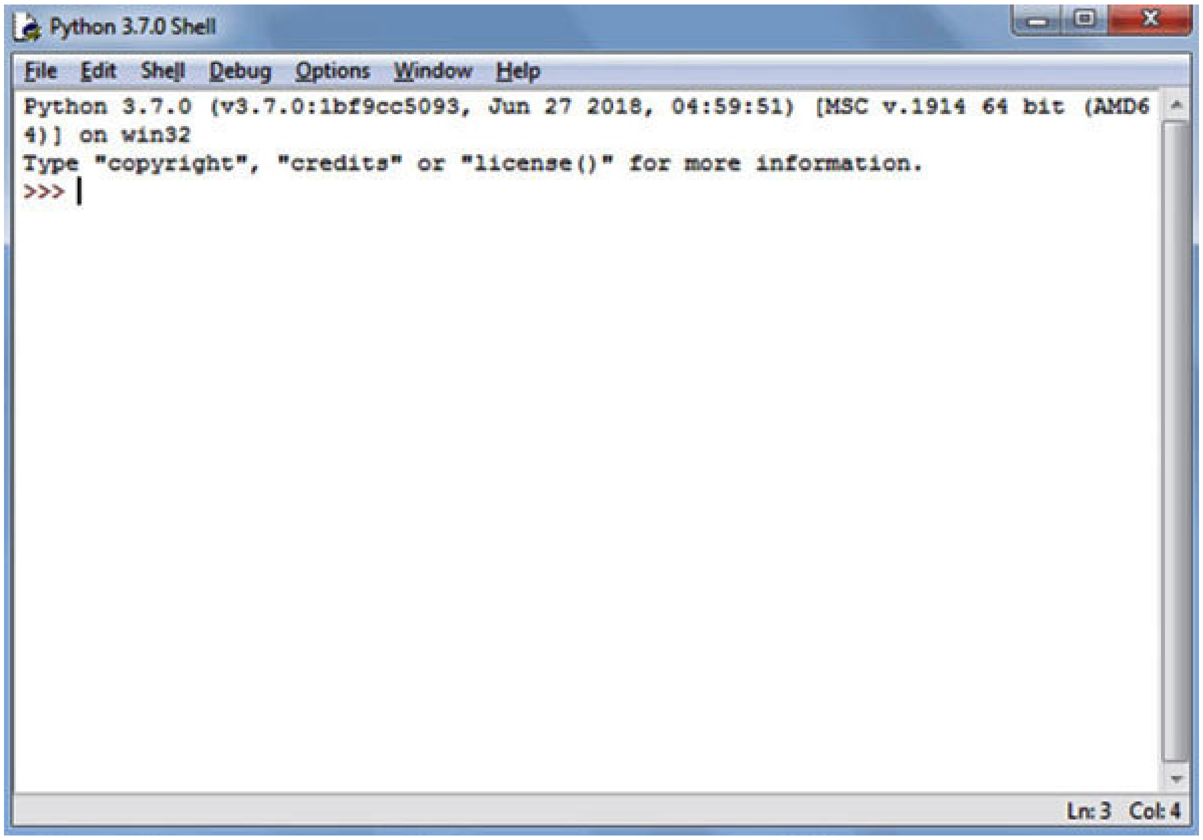


Figure 1.12: *Python on Search Bar*

The preceding figure 1.12 shows how Python IDE can search through the search bar:



```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD6
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
```

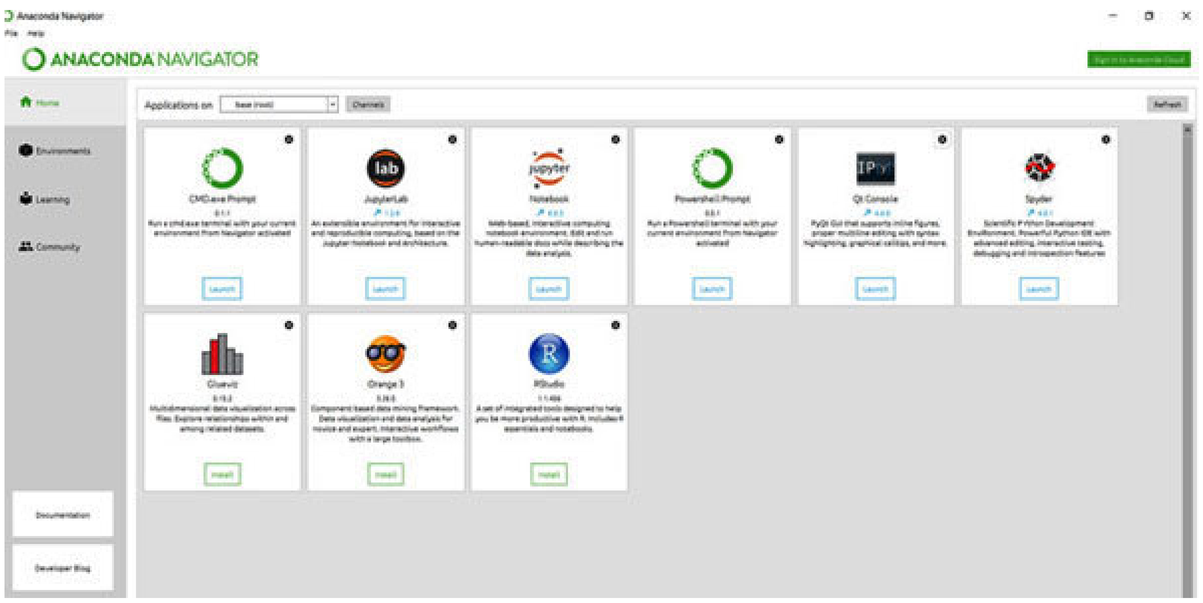
Figure 1.13: *Python Shell*

There are various types of IDEs are available for Python. Various kinds of IDEs are commercial and freely available such as PyCharm, Kite, Spyder, DLE, Visual Studio Code, Atom, Jupyter, Pydev, Thonny, and many more. Here, we will learn how to use some open-source editors to execute Python scripts or statements.

Anaconda open-source distribution

It is for Python and R programming languages. This distribution is freely available. It contains various applications such as JupyterLab, Jupyter Notebook, QtConsole, Spyder, Glue, Orange, RStudio, and Visual Studio Code.

This distribution is for large-scale data processing, predictive analytics, and scientific computing. The advantage of Anaconda is the massive amount of packages inside it. It would be very easier to make a program and handle huge data under time and space constraints.



CMD.exe Prompt
Run a cmd.exe terminal with your current environment from Navigator activated

JupyterLab
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Notebook
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Powershell Prompt
Run a Powershell terminal with your current environment from Navigator activated

Qt Console
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Spyder
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Glueviz
Multidimensional data visualization across files. Explore relationships within and among related datasets.

Orange 3
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

RStudio
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Figure 1.14: *Anaconda Navigator*

Writing and executing first Python program

There are three ways to write and execute a Python program.

Starting Python through Command Line

Starting Python IDLE

Starting Python through Spyder/Jupyter Notebook in Anaconda Prompt

Starting Python through command line

It can be writing a Python code through a command-line approach. When we can execute Python expressions, statements, or instructions from the command line then this type of mode is called interactive mode or interactive prompt. There are the following steps to start a Python code through the command line.

Click the **Start** button as shown in *figure*

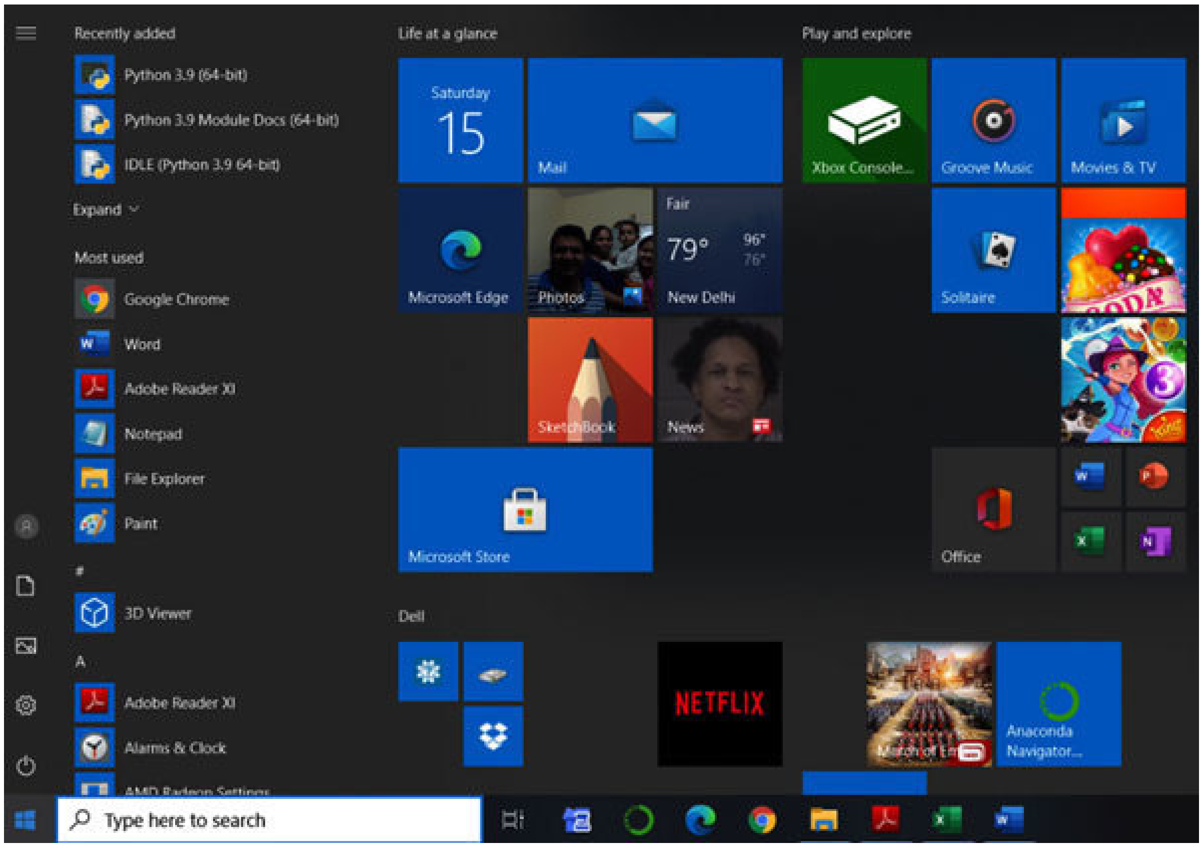


Figure 1.15: Start button

Type Python 3.9 in the search space bar beside the **Start** button, as shown in *figure*

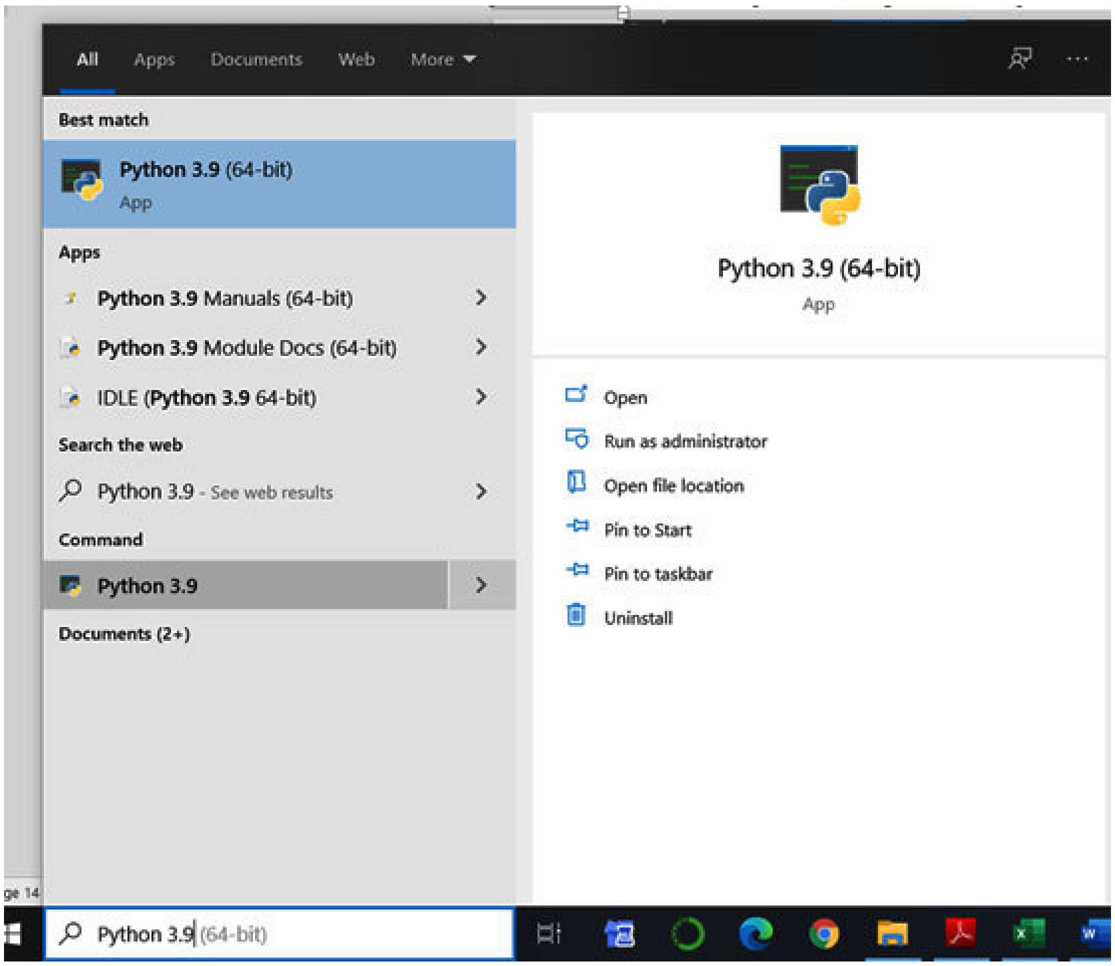
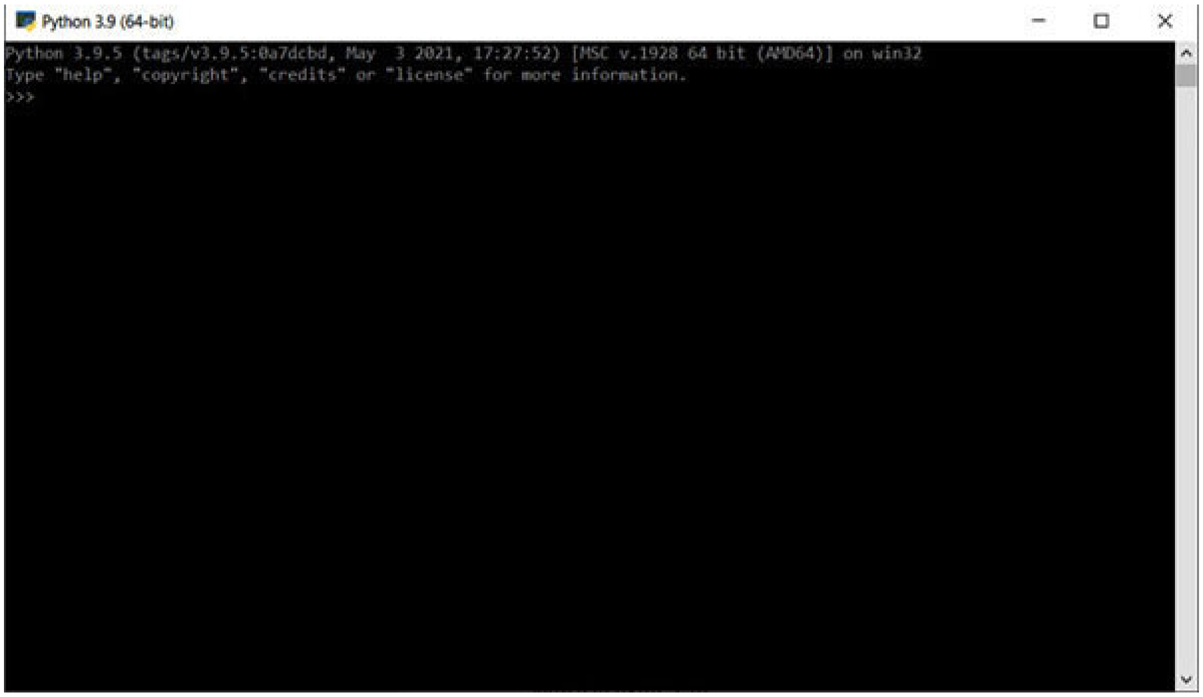


Figure 1.16: *Python 3.9 (64-bit)*

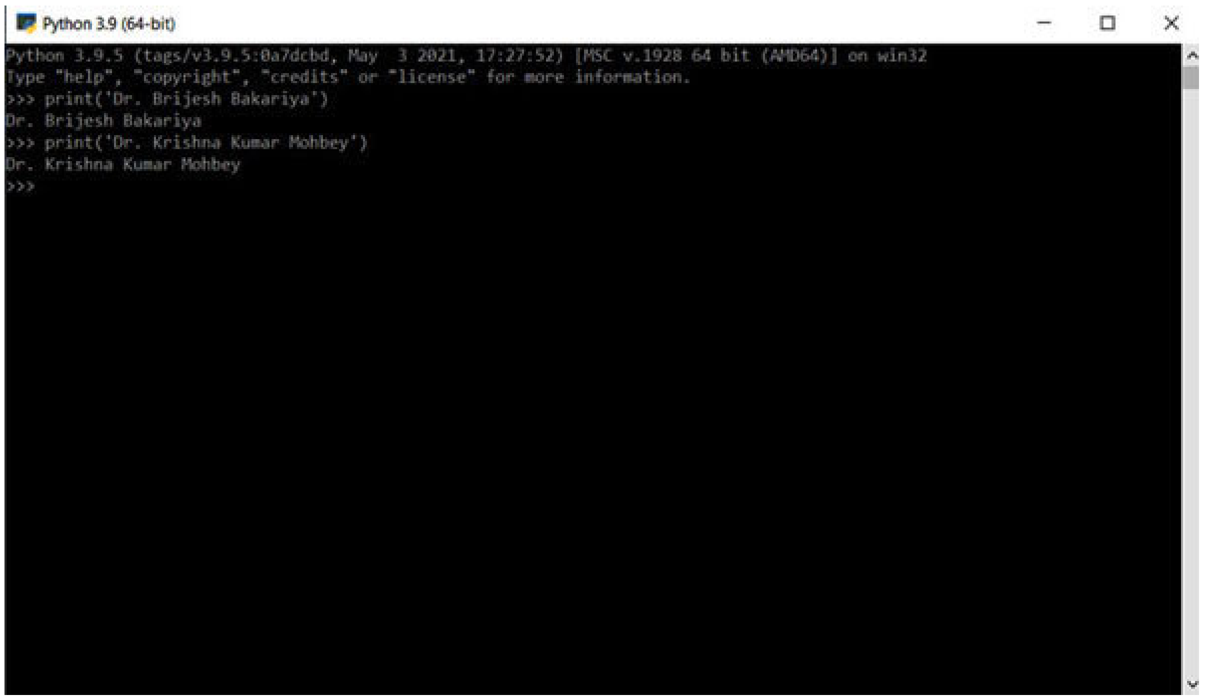
Click to open Python 3.9 (64-bit). After that, a command prompt will be open, as shown in *figure*



```
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 1.17: *Python 3.9 (64-bit) Command prompt*

We can write any expression or statements over here. This is an interactive mode of Python command prompt. Here, we are writing our names through the command prompt, as shown in *figure*


Python 3.9 (64-bit)

```
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Dr. Brijesh Bakariya')
Dr. Brijesh Bakariya
>>> print('Dr. Krishna Kumar Mohbey')
Dr. Krishna Kumar Mohbey
>>>
```

Figure 1.18: *Python code through command prompt*

To exit the command line of Python 3.9, you can press *Ctrl* + *z* and press *Enter* key or type **exit()** then press *Enter* key.

Starting Python IDLE

Python IDLE is another way to write and execute a Python program. It provides an interactive platform that contains various tools that can be used to write a program.

There are the following steps to start a Python through Python IDLE.

Type **IDLE** in the search space bar beside the start button, as shown in *figure*

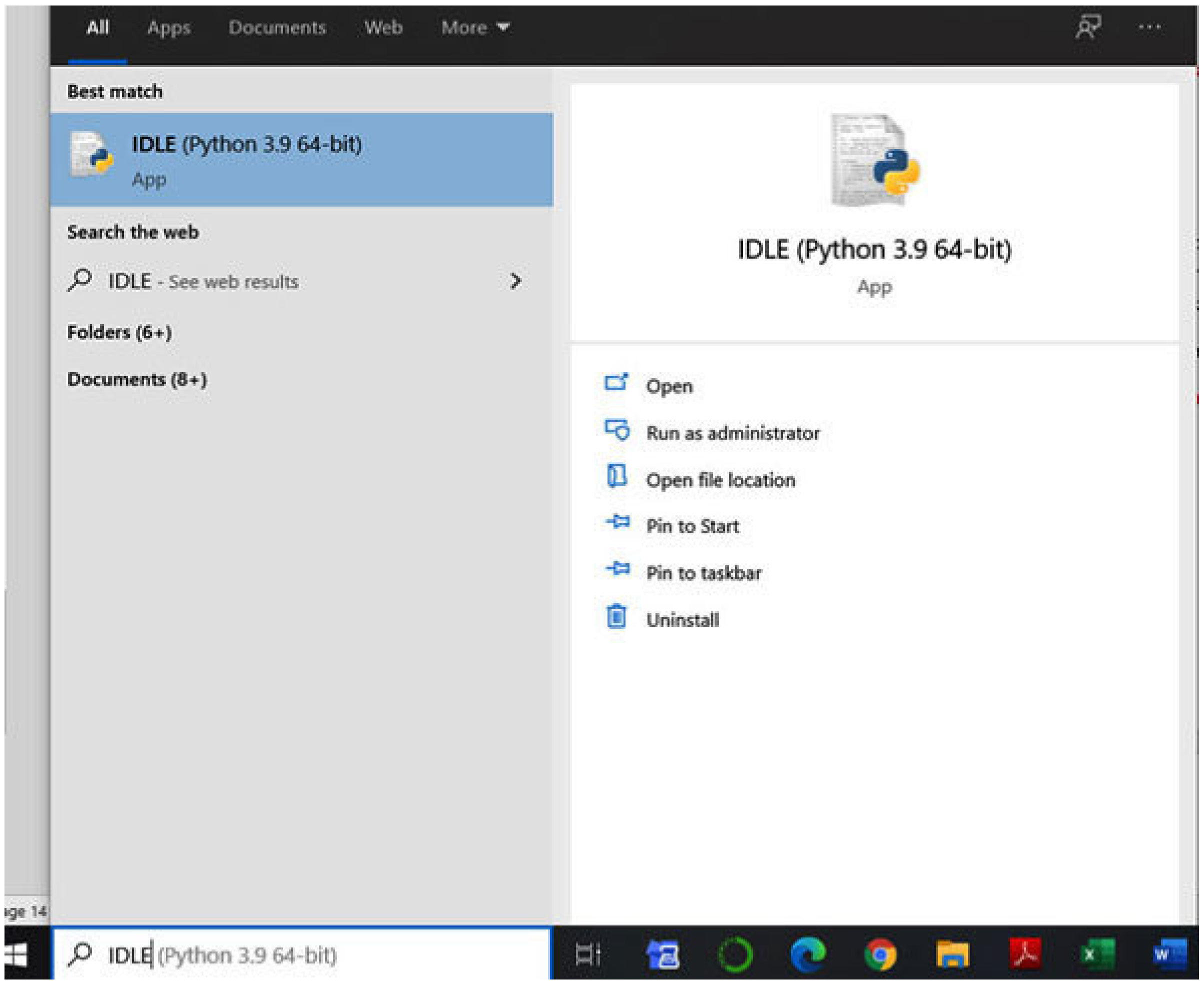
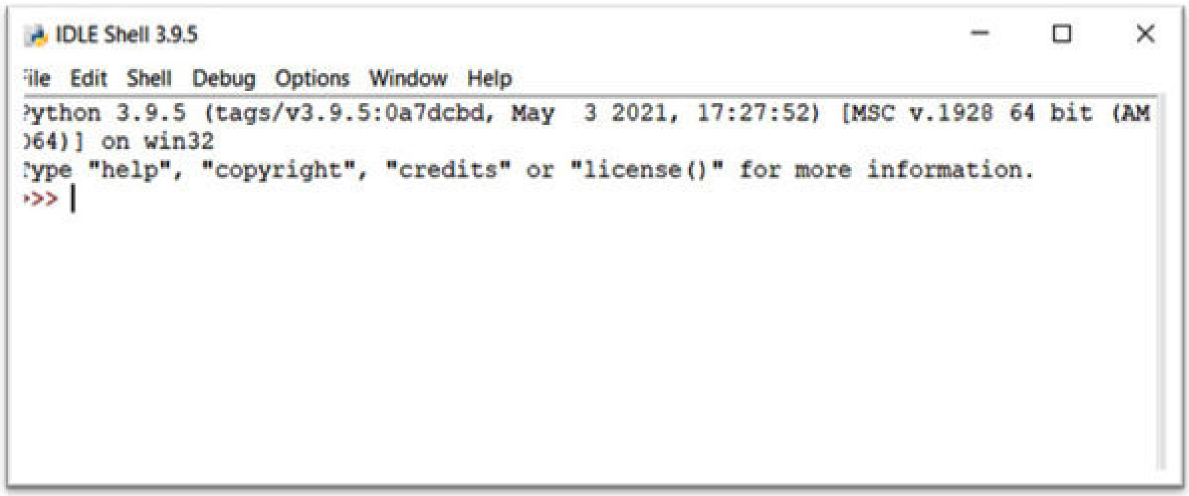


Figure 1.19: *IDLE (Python 3.9 64 bit)*

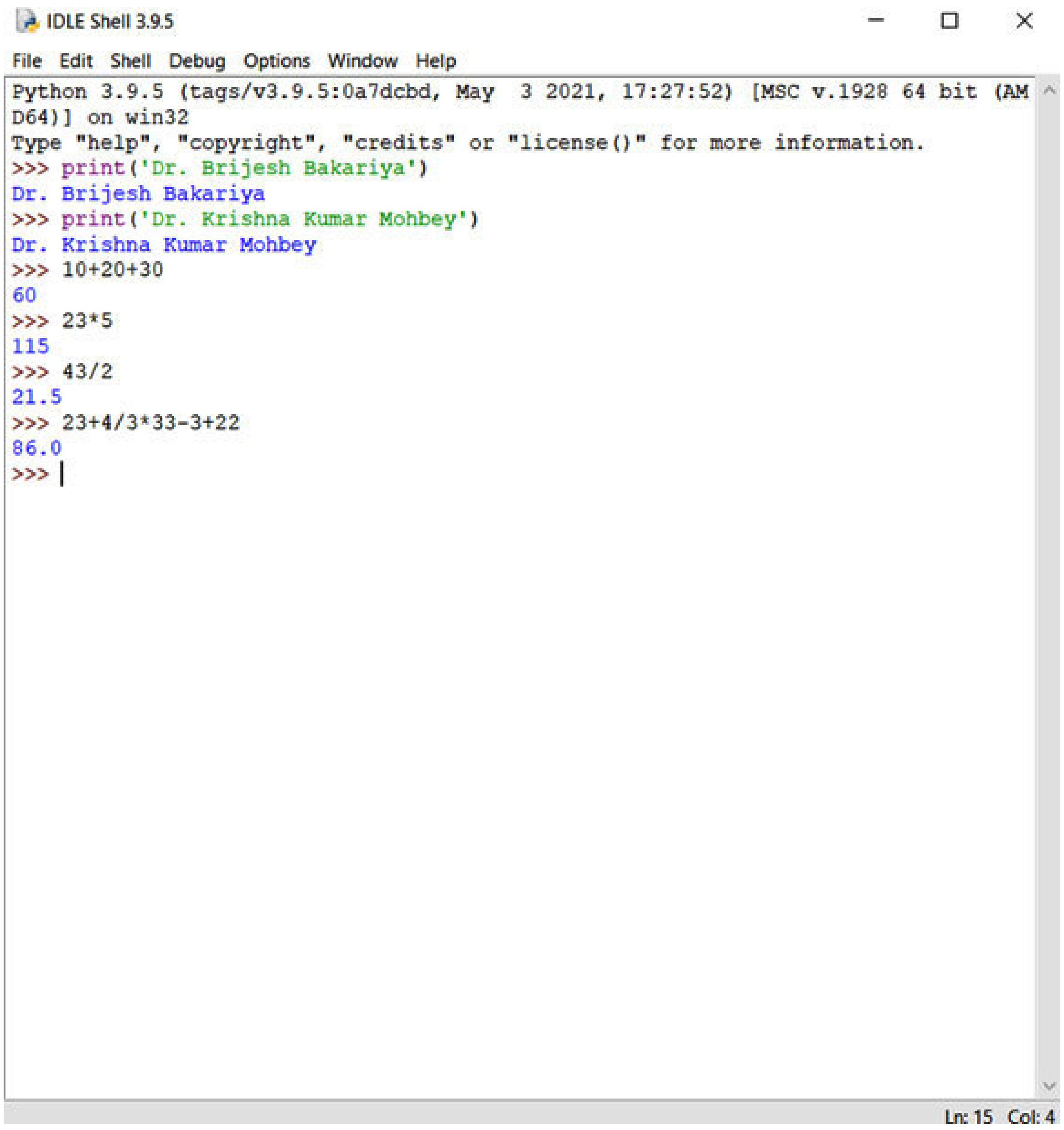
Click to open an IDLE (Python 3.9 64 bit). After that, an IDLE Shell will open, as shown in *figure*



```
ython 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AM
64)] on win32
ype "help", "copyright", "credits" or "license()" for more information.
>> |
```

Figure 1.20: *IDLE Shell*

We can write any expression or statement over here. This is an interactive mode of Python IDLE. Here, we are writing our names and some basic expressions through this prompt, as shown in *figure*



```
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('Dr. Brijesh Bakariya')
Dr. Brijesh Bakariya
>>> print('Dr. Krishna Kumar Mohbey')
Dr. Krishna Kumar Mohbey
>>> 10+20+30
60
>>> 23*5
115
>>> 43/2
21.5
>>> 23+4/3*33-3+22
86.0
>>> 
```

Figure 1.21: *Python code through IDLE Shell*

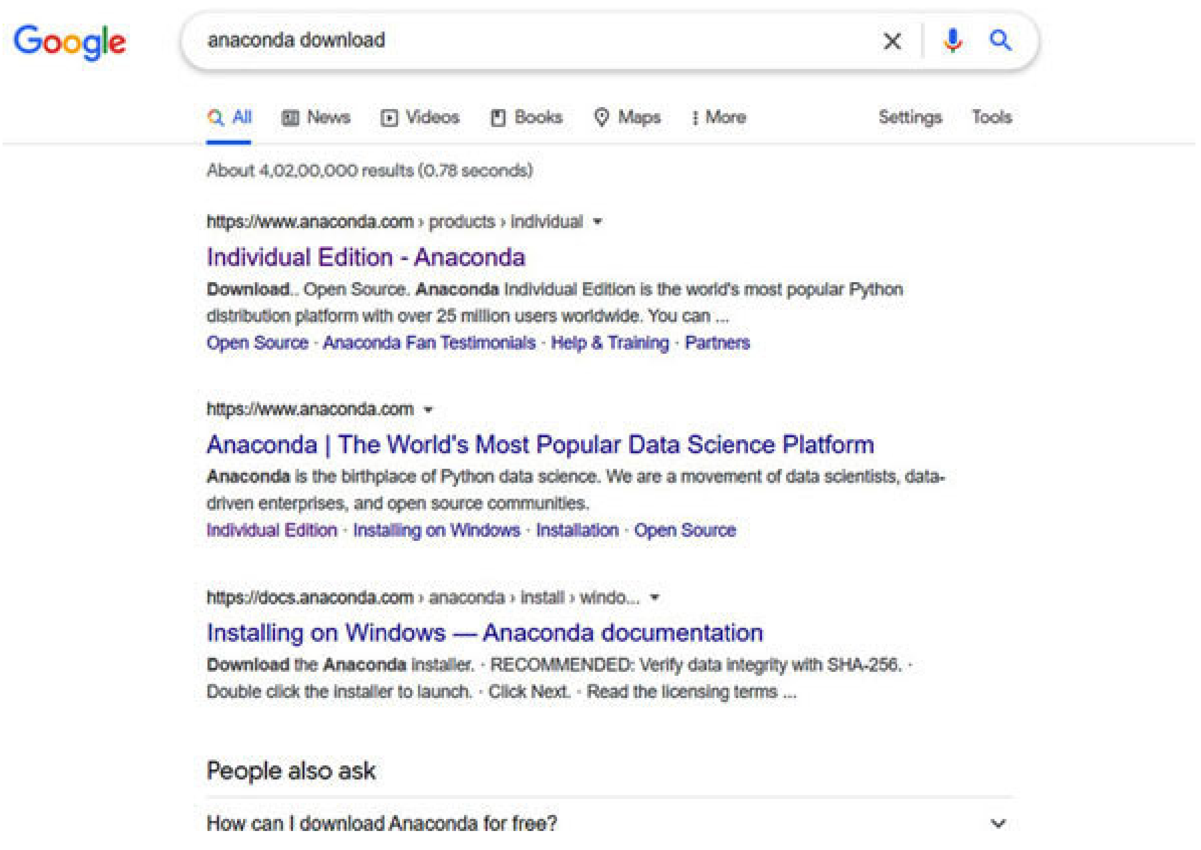
To exit from an IDLE (Python 3.9 64 bit), type **exit()** and press *Ctrl* +

Starting Python through Spyder in Anaconda Prompt

In this section, we are writing the first program on one of the IDE names Spyder. You can install Spyder separately or through an anaconda prompt. Here we are using Spyder through an anaconda prompt.

There are the following steps to start Python through Spyder in Anaconda prompt.

First of all, we have to install an anaconda prompt, then we simply type anaconda download on Google, as shown in *figure*


anaconda download

About 4,02,00,000 results (0.78 seconds)

https://www.anaconda.com › products › individual

Individual Edition - Anaconda

Download.. Open Source. **Anaconda** Individual Edition is the world's most popular Python distribution platform with over 25 million users worldwide. You can ...

Open Source · Anaconda Fan Testimonials · Help & Training · Partners

https://www.anaconda.com

Anaconda | The World's Most Popular Data Science Platform

Anaconda is the birthplace of Python data science. We are a movement of data scientists, data-driven enterprises, and open source communities.

Individual Edition · Installing on Windows · Installation · Open Source

https://docs.anaconda.com › anaconda › install › windo...

Installing on Windows — Anaconda documentation

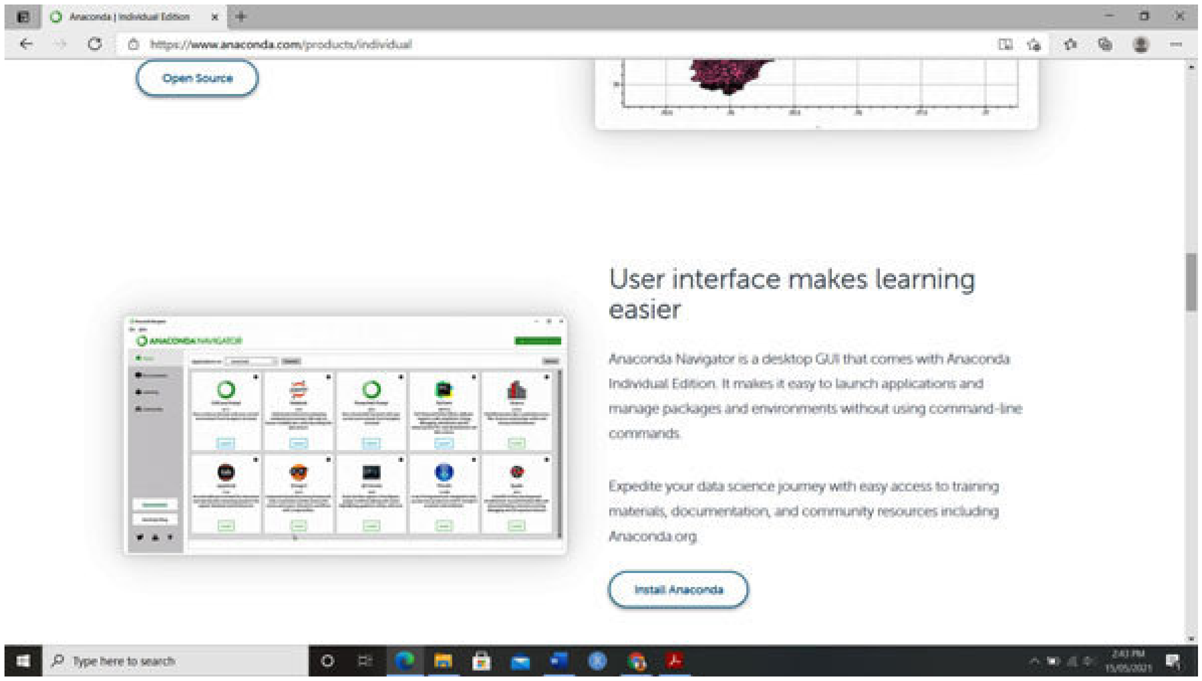
Download the **Anaconda** installer. · RECOMMENDED: Verify data integrity with SHA-256. · Double click the installer to launch. · Click Next. · Read the licensing terms ...

People also ask

How can I download Anaconda for free?

Figure 1.22: *Anaconda search on Google*

Click Individual Edition-Anaconda link, or you can go through the following URL The webpage will open, and you can download and install Anaconda, as shown in *figure*



Anaconda | Individual Edition

https://www.anaconda.com/products/individual

Open Source

User interface makes learning easier

Anaconda Navigator is a desktop GUI that comes with Anaconda Individual Edition. It makes it easy to launch applications and manage packages and environments without using command-line commands.

Expedite your data science journey with easy access to training materials, documentation, and community resources including Anaconda.org.

Install Anaconda

Type here to search

Figure 1.23: *Download Anaconda*

After installing Anaconda, you can type Anaconda Navigator in the search bar beside the start button, as shown in *figure*

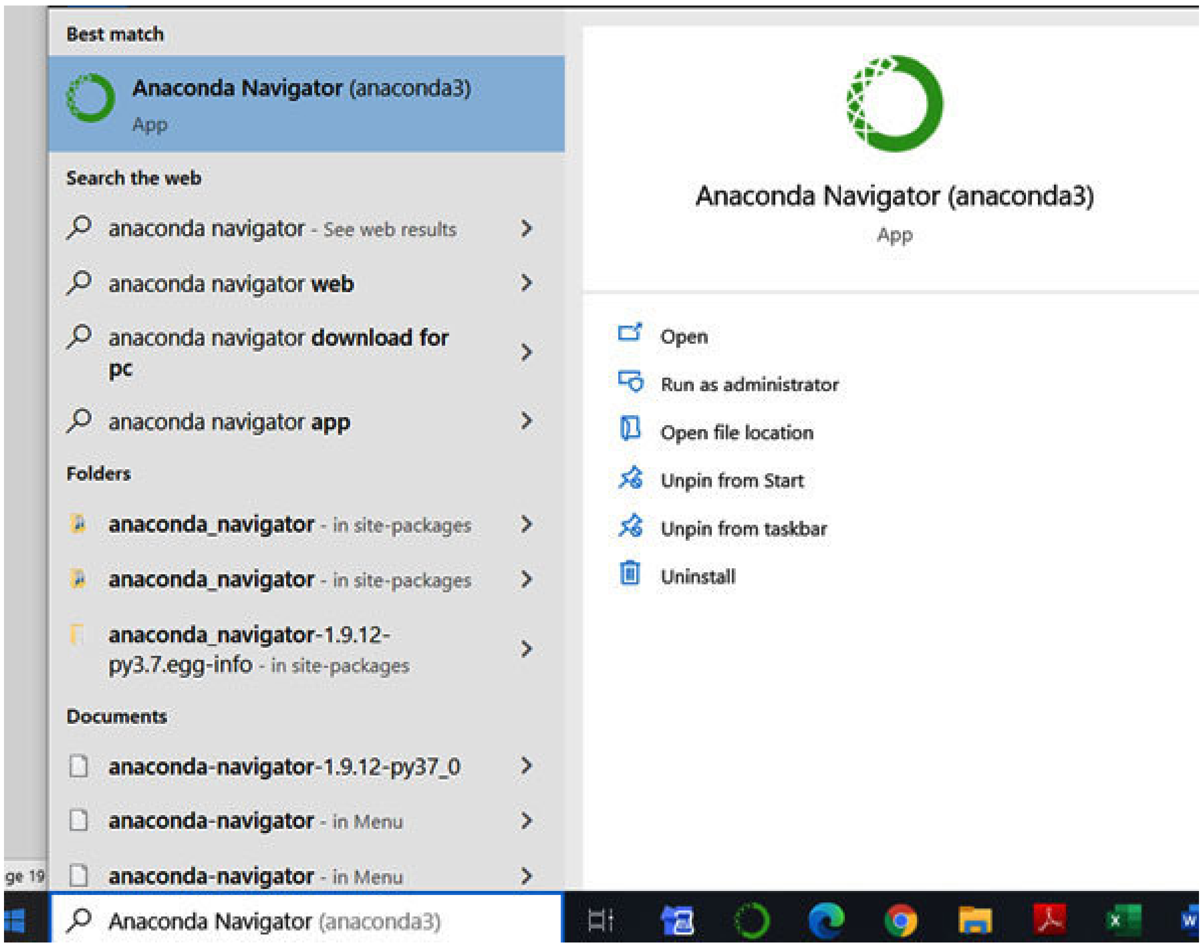

Best match
Anaconda Navigator (anaconda3)
App
Search the web
anaconda navigator - See web results
anaconda navigator web
anaconda navigator download for pc
anaconda navigator app
Folders
anaconda_navigator - in site-packages
anaconda_navigator - in site-packages
anaconda_navigator-1.9.12-py3.7.egg-info - in site-packages
Documents
anaconda-navigator-1.9.12-py37_0
anaconda-navigator - in Menu
anaconda-navigator - in Menu
Anaconda Navigator (anaconda3)
Anaconda Navigator (anaconda3)
App
Open
Run as administrator
Open file location
Unpin from Start
Unpin from taskbar
Uninstall

Figure 1.24: *Anaconda Navigator*

Click to open Anaconda Navigator. After that, the Anaconda Navigator will be open, as shown in *figure*

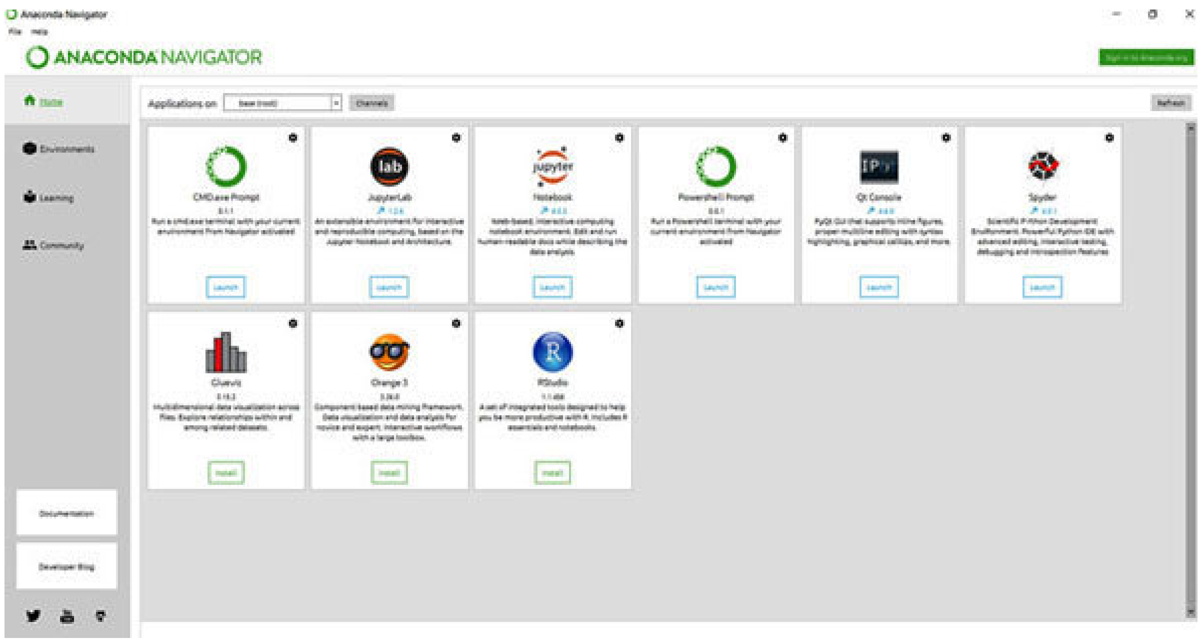


Figure 1.25: *Anaconda Navigator Snippet*

Click on the Spyder icon; after that, the Spyder application will be launched, as shown in *figure*

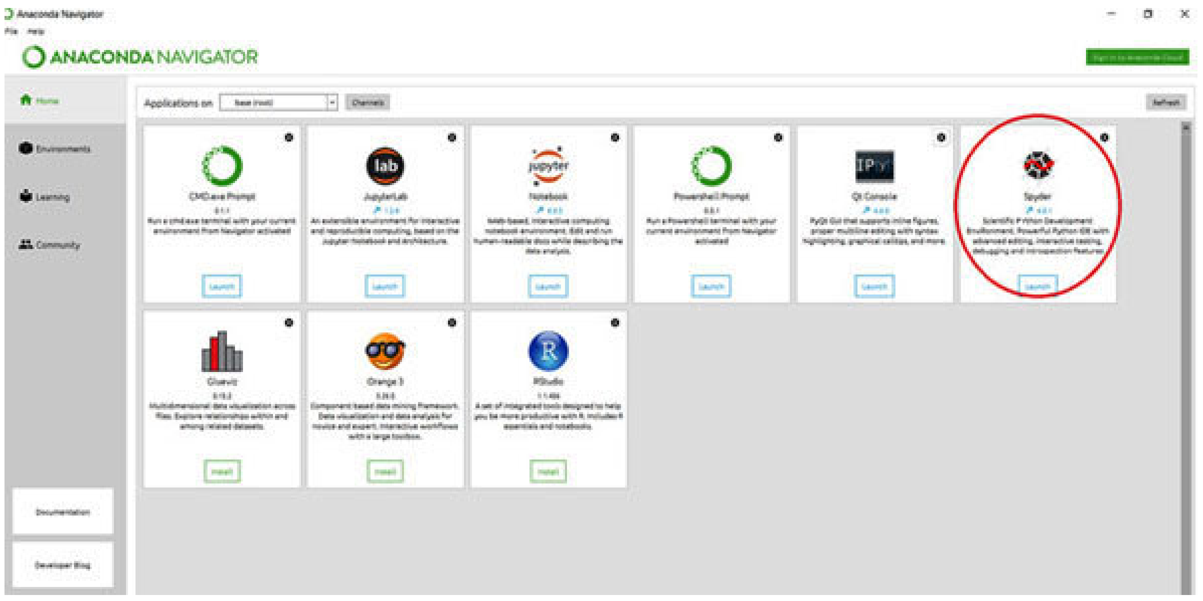
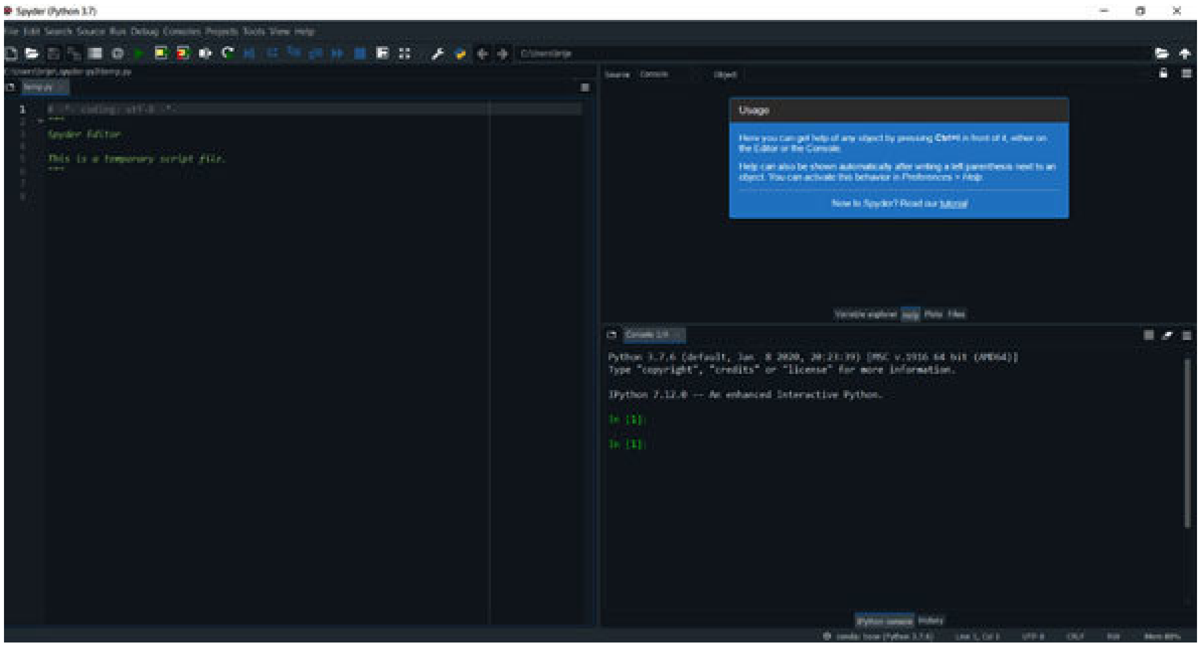


Figure 1.26: *Spyder in Anaconda Navigator*

After clicking on Spyder applications, Spyder IDE will open, as shown in [*figure*]


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
temp.py
Source Console Object
Usage
Variable explorer Help Plots Files
Console 1/A
IPython console History

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.

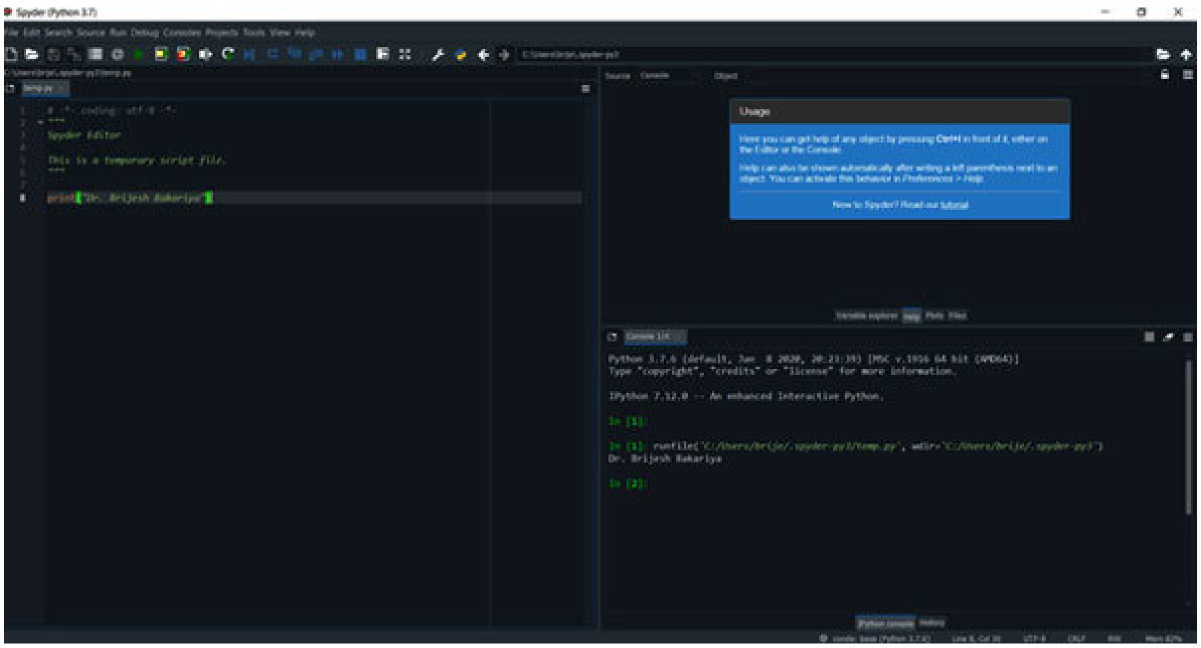
Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in Preferences > Help.

New to Spyder? Read our tutorial

```
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.12.0 -- An enhanced Interactive Python.

In [1]:

In [1]:
```

Figure 1.27: *Spyder Home*

In the Spyder IDE, the left window is used for writing codes. Here we can write python codes as shown in [*figure*]



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

print("Dr. Brijesh Bakariya")
```

```
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.12.0 -- An enhanced Interactive Python.

In [1]:

In [1]: runfile('C:/Users/brije/.spyder-py3/temp.py', wdir='C:/Users/brije/.spyder-py3')
Dr. Brijesh Bakariya

In [2]:
```

Usage

Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in Preferences > Help.

New to Spyder? Read our tutorial

Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
temp.py
Source Console Object
Variable explorer Help Plots Files
Console 1/A
Python console History

***Figure 1.28:** Code on Spyder*

Starting Python through Jupyter Notebook

The following steps will be used to write a program in Jupyter notebook.

Search Jupyter notebook from start menu as shown in the figure

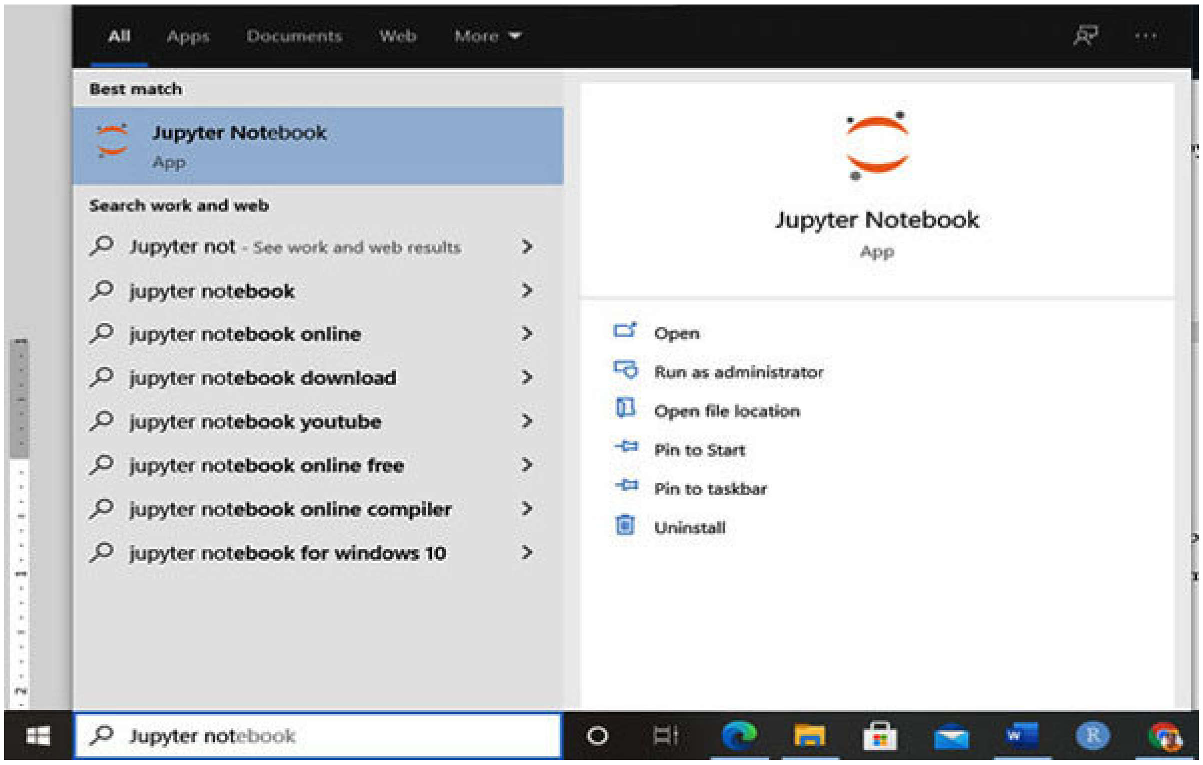
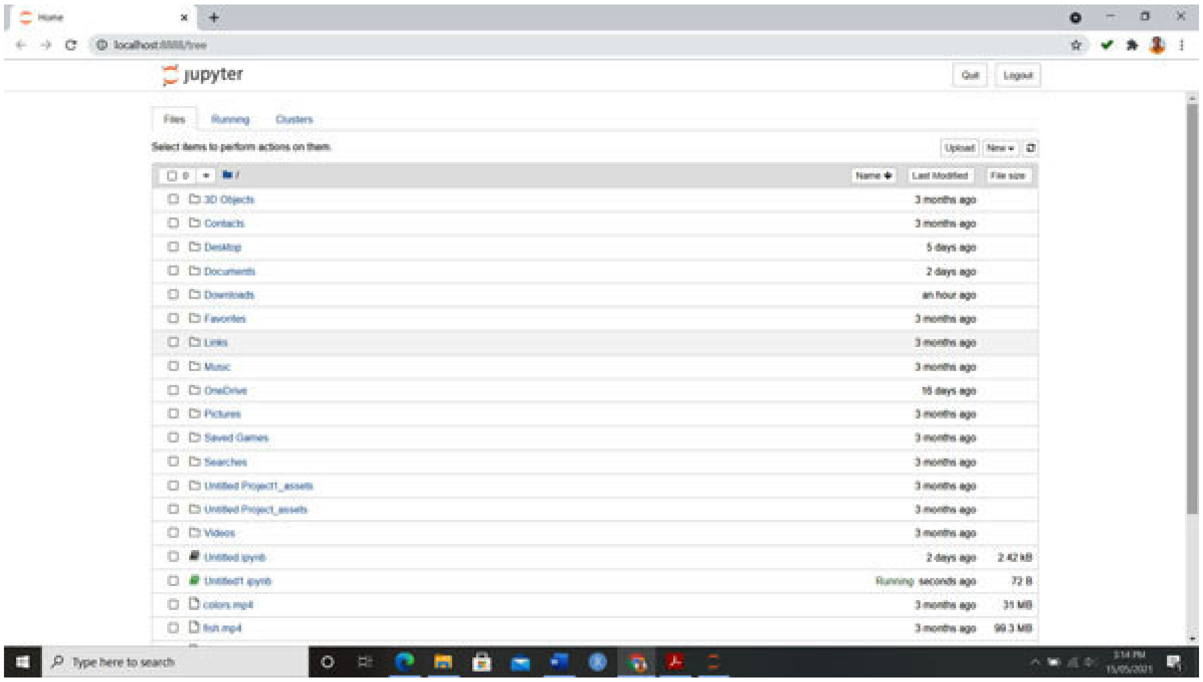


***Figure 1.29:** Searching Jupyter Notebook*

After clicking on Jupyter Notebook, it will open as shown in figure


localhost:8888/tree

Select items to perform actions on them.

Name	Last Modified	File size
3D Objects	3 months ago	
Contacts	3 months ago	
Desktop	5 days ago	
Documents	2 days ago	
Downloads	an hour ago	
Favorites	3 months ago	
Links	3 months ago	
Music	3 months ago	
OneDrive	16 days ago	
Pictures	3 months ago	
Saved Games	3 months ago	
Searches	3 months ago	
Untitled Project1_assets	3 months ago	
Untitled Project_assets	3 months ago	
Videos	3 months ago	
Untitled.ipynb	2 days ago	2.42 kB
Untitled1.ipynb	Running seconds ago	72 B
colors.mp4	3 months ago	31 MB
fish.mp4	3 months ago	99.3 MB

Figure 1.30: *Jupyter Notebook*

Now, we will start a new notebook. For this, we will click on the new Python 3 notebook. The new option is available at the top-right corner of this window. The following figure 1.31 shows an example of a new notebook:

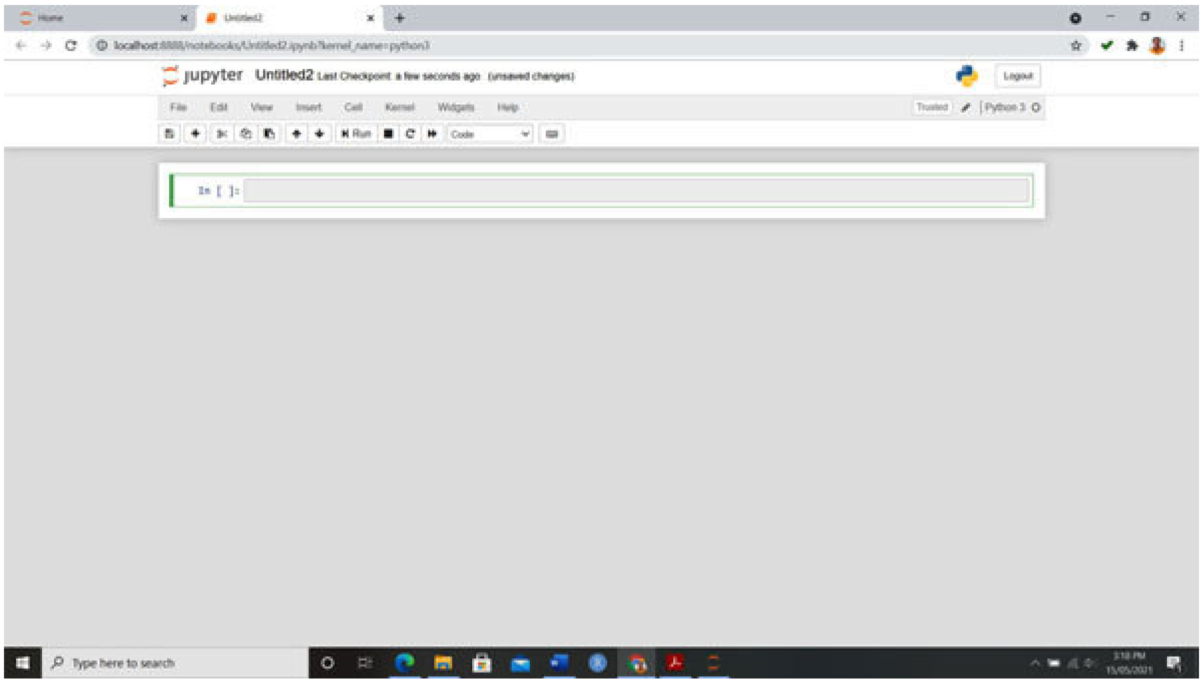
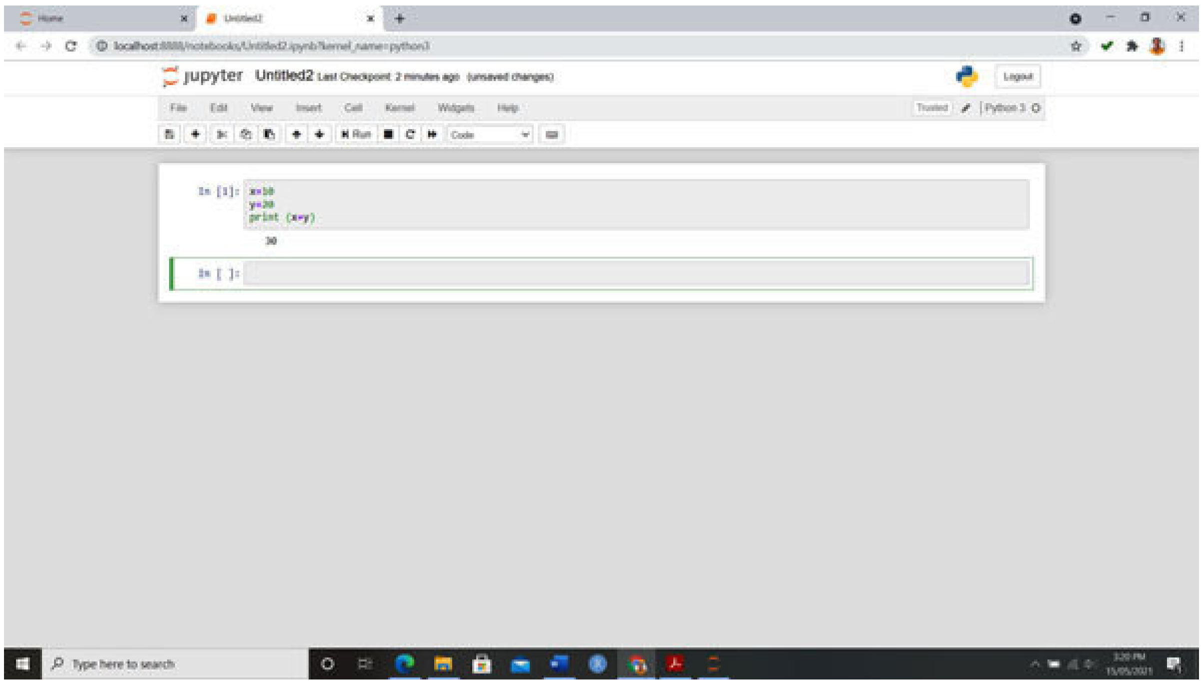

Home
Untitled2
localhost:8888/notebooks/Untitled2.ipynb?kernel_name=python3
jupyter
Untitled2 Last Checkpoint: a few seconds ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code
In [ ]:
Type here to search

Figure 1.31: *Opening a Notebook in Jupyter*

We can write Python codes in the notebook cells and execute them by clicking the **Run** button. The following *figure* gives an example of writing and executing Python codes:



```
In [1]: x=10
        y=20
        print (x+y)

        30

In [ ]:
```

Figure 1.32: *Writing codes in Jupyter*

Writing and saving Python programs

The following examples show how we create Python scripts, save and execute them in different Python editors.

Example 1.1:

```
# Display a Name
print ("Dr. Brijesh Bakariya")
#Function displaying a name
```

Output:

```
Dr. Brijesh Bakariya
```

The Python file's file extension is so it must be saved by the **.py** extension and the code should be executed.

Example 1.2:

```
# Program for the addition of two numbers
num1 = 10
```

```
num2 = 20
# Add two numbers
sum = num1 + num2
# Display the sum
print ('The sum of {0} and {1} is {2}'.format(num1, num2,
sum))
```

Output:

```
The sum of 10 and 20 is 30
```

Example 1.3:

```
# Program for addition of two numbers with user input
# input() for taking input from user
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')
# Add two numbers

sum = float(num1) + float(num2)
# Display the sum
print ('The sum of {0} and {1} is {2}'.format(num1, num2,
sum))
```

Output:

The sum of 12.6 and 66.2 is 78.8

Value and types

A value is a small part of the program. It could be a number or a letter. For example, we have an integer number 10, 20 that means this is a value and letter **Bakariya** is a letter or a string. It is also called a string of letters, this is enclosed in quotation marks.

The **type()** function tells the type of a value. Let us take some examples where it could be clearly understood.

Example 1.4:

```
a=type('Amit Kumar')
print (a)
b=type(17)
print (b)
c= type(10.5)
print (c)
```

Output:

```
'str'>
```

```
'int'>
'float'>
```

Numbers

This is one of the data types which contain numeric values. This is an immutable data type, which means you can change the value of a number data type, and the result will change.

Python supports integers, floating-point numbers, and complex numbers. There are the following representations of a number data type in Python.

The integer represents as **int**

Floating-point numbers represent **float**

Complex numbers represent **complex**

Let us take an example.

Example 1.5:

```
a = 5
print (a)
print  (type(a))
b=15.2
print  (b)
print  (type(b))
c  =  3  +  2j
print  (c  +  3)
print  (type(c))
```

Output:

```
5
'int'>
15.2
'float'>
(6+2j)
'complex'>
```

Getting input

Somewhere we have to interact with the user, the **input()** method is used to get input from the user.

When you use **input()** method, then this method uses a dialog box. It is a way of asking the user to provide some type of input. There are two types of methods for getting input from the user:

input (prompt)
raw_input (prompt)

This function first takes the input from the user and then evaluates the expression.

This method's beauty is that it is automatically identified whether the user entered a string or a number or list. If the input provided is not correct, then either syntax error or exception is raised by Python.

Let's understand how the **input()** function works?

input() function executes and waits until the user has given input. The text or message displays on the output screen. Whatever you enter as an input, the **input()** function converts it into a string. If you enter an integer value, still the **input()** function converts it into a string. You need to explicitly convert it into an integer in your code using typecasting.

Python 3.6 uses the **input()** method, and here we are taking an example of Python 3.6 or higher version.

Example 1.6:

```
# input() for taking input from user
name = input('Enter your name: ')
age = input('Enter your age: ')
print (name)
print (age)
```

Output:

```
Enter your Name: Dr. Brijesh Bakariya
Enter your age: 36
Dr. Brijesh Bakariya
36
```

Example 1.7:

```
# get input from a user
str1 = input()
# Getting input from a user without a prompt
print ('The inputted string 1 is: ', str1)
str2 = input('Enter a string: ')
#Getting input from the user with a prompt
print ('The inputted string is:', str2)
```

Output:

```
Dr. Brijesh Bakariya
The inputted string 1 is: Dr. Brijesh Bakariya
Enter a string:Dr. Krishna Kumar Mohbey
The inputted string 2is: Dr. Krishna Kumar Mohbey
```

Printing

It is the simplest way to display the output on the screen. Python has a **print()** method for output printing. We can pass zero or more **print()** function expression, but commas should separate the expressions. The **print()** function converts the expressions into a string before writing on the screen.

There is the following system of **print()** function.

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

The and **flush** are the parameters in which we can pass it. There are the following descriptions of these parameters:

Any value, and as many as you like. It will be converted to a string before printing.

(Optional) Specify how to separate the objects if there is more than

(Optional) Specify what to print at the **end.Default:**

file : (Optional) An object with a write method.
Default:sys.stdout

flush : (Optional) A Boolean, specifying if the output is flushed or buffered

It returns output to the screen.

Example 1.8:

```
# Use of print() function
print("Python is a programming language")
n1 = 10
# Two objects are passed
print ("n1 =", 10)
n2 = n1

# Three objects are passed
print ('n1 =', n1, '= n2')
```

Output:

```
Python is a programming language
n1 = 10
n1 = 10 = n2
```

Example The **print()** function with separator and end parameters:

```
n1 = 50
print ("n1 =", n1, sep='000', end='\n\n')
print ("n1 =", n1, sep='0', end='')
```

Output:

```
n1 =00050
n1 =050
```

Boolean

The **bool()** method is used to return or convert a value to a Boolean value, that is, **True** or using the standard truth testing procedure.

This method generally takes one parameter based on the truth testing procedure. By default, this returns that is, no argument is passing. It returns **True** if the parameter or value is passed.

There are various ways in which Python's **bool()** method returns Except for these, all other values return

If a False value is passed.

If None is passed.

If an empty sequence is passed, such as etc.

If Zero is passed in any numeric type, such as 0, 0.0, etc.

If an empty mapping is passed, such as

If objects of classes having **__bool__()** or **__len()__** method, returning 0 or False

Example 1.10:

```
# Different ways for printing Boolean values
# Returns False as a is False
a = False
print(bool(a))
# Returns True asa is True
a= True
print(bool(a))
# Returns False as a is not equal to b
a= 7
b = 6

print (bool(a==b))

# Returns False as a is None
a = None
print (bool(a))

# Returns False as a is an empty sequence
a = ()
print (bool(a))
```

```
# Returns False as a is an empty mapping
a = {}
print (bool(a))

# Returns False as a is 0
a = 0.0
print(bool(a))

# Returns True as a is a non-empty string
a = 'Brijesh'
print (bool(a))
```

Output:

```
False
True
False
False
False
False
False
True
```

Lists

A list is for placing all the data items in one place. Moreover, it is capable of storing different data items in a single variable. All the variables in the list are contained in square brackets separated by commas. Three more built-in data types work as a list, but there are some differences among them. Such data types are Tuple, Set, and Dictionary.

The list is a collection that is ordered and changeable. It allows duplicate members.

A tuple is a collection that is ordered and unchangeable. It also allows duplicate members.

Set is a collection that is unordered and unindexed. It does not allow duplicate members.

Dictionary is a collection that is unordered and changeable. It does allow No duplicate members.

We will discuss the details of every data type in further chapters.

Example 1.11:

```
# Creating a simple list
list_var1 = ["Brijesh", "Krishna", "Ram", "Amit", "Rohit"]
print (list_var1)

list_var2 = ["Brijesh", 35, "Krishna",36, "Ram",34, "Amit", 30,
"Rohit"]
print (list_var2)
```

Output:

```
['Brijesh', 'Krishna', 'Ram', 'Amit', 'Rohit']
['Brijesh', 35, 'Krishna', 36, 'Ram', 34, 'Amit', 30, 'Rohit']
```

Strings

The string is an array of characters. The string is created by single quotation marks, double quotation marks, and triple quotation marks. We can print the string with the help of the **print()** function.

Example 1.12:

```
# Creating a string
# Single string
str1 = "Python"
print (str1)
# Multiple string
str2 = """Python is good programming,
It is used for data analysis,
It is very faster in the computational point of view."""
print (str2)
```

Output:

```
Python
Python is good programming,
```

It is used for data analysis,

It is very faster in the computational point of view.

Variables and identifiers

A variable is a name that refers to a value. It is an essential part of a programming language because a variable can hold the value. The programming is significantly more comfortable and understandable when you use a variable for holding a value. In Python, a single value can be assigned to multiple variables at the same time. It is also possible to initialize various variables same time with different values.

Let us take some examples to understand more about the variable.

Example 1.13:

```
# multiple variables assignment with same value
x=y=10

print ('value of x is:',x)
print ('value if y is:',y)
```

Output:

```
value of x is: 10
value if y is: 10
```

Example 1.14:

```
# Initializing multiple variables
x,y,z=10,'amit',123.52

print ('value of x is:',x)
print ('value if y is:',y)
print ('value if z is:',z)
```

Output:

```
value of x is: 10
value if y is: amit
value if z is: 123.52
```

Example 1.15:

```
str='Python is very easy and efficient programming language'
num=10
float_var=34.44
```

```
print (str)
print (num)
print (float_var)
```

Output:

```
Python is very easy and efficient programming language
10
34.44
```

In this example, we have defined three variables and **float_var** for holding a string, number, and float value.

Example 1.16:

```
# Showing variable and its type
str='Python is very easy and efficient programming language'
num=10
float_var=34.44
print (str)
print (num)
print (float_var)
print (type(str))
print (type(num))
print (type(float_var))
```

Output:

```
Python is very easy and efficient programming language
10
34.44
'str'>
'int'>

'float'>
```

A variable name should be meaningful while creating a variable because it is more understandable when you use it to create a meaningful variable. Variable names can be long and contain letters and numbers, but they should start with a letter; special characters are not allowed. Underscore is also allowed. Let's take an example to understand the variable name.

Example 1.17: Variable name

```
This_is_for_storing_Name ="Dr. Brijesh Bakariya"
_My_Qualification = "Ph.D."
print (This_is_for_storing_Name)
print (_My_Qualification)
```

Output:

Dr. Brijesh Bakariya
Ph.D.

Python identifiers

An identifier is a name given to entities such as variables, functions, etc.

A combination of upper-case letters (A to Z), Lower case letters (a to z), and digits (0 to 9), underscore are allowed in an identifier, but it cannot start with digits.

Example 1.18:

```
# Identifier
Name1='Amit'
NAME='Raj'
N12='Ramesh'
_Name='Rohit'
print (Name1)
print (NAME)
print (N12)
print (_Name)
```

Output:

Amit

Raj

Ramesh

Rohit

Data types

Python has a data type for every value. Moreover, everything is an object in Python, and data types can be classes and objects of these classes. There are various categories of built-in data types in Python:

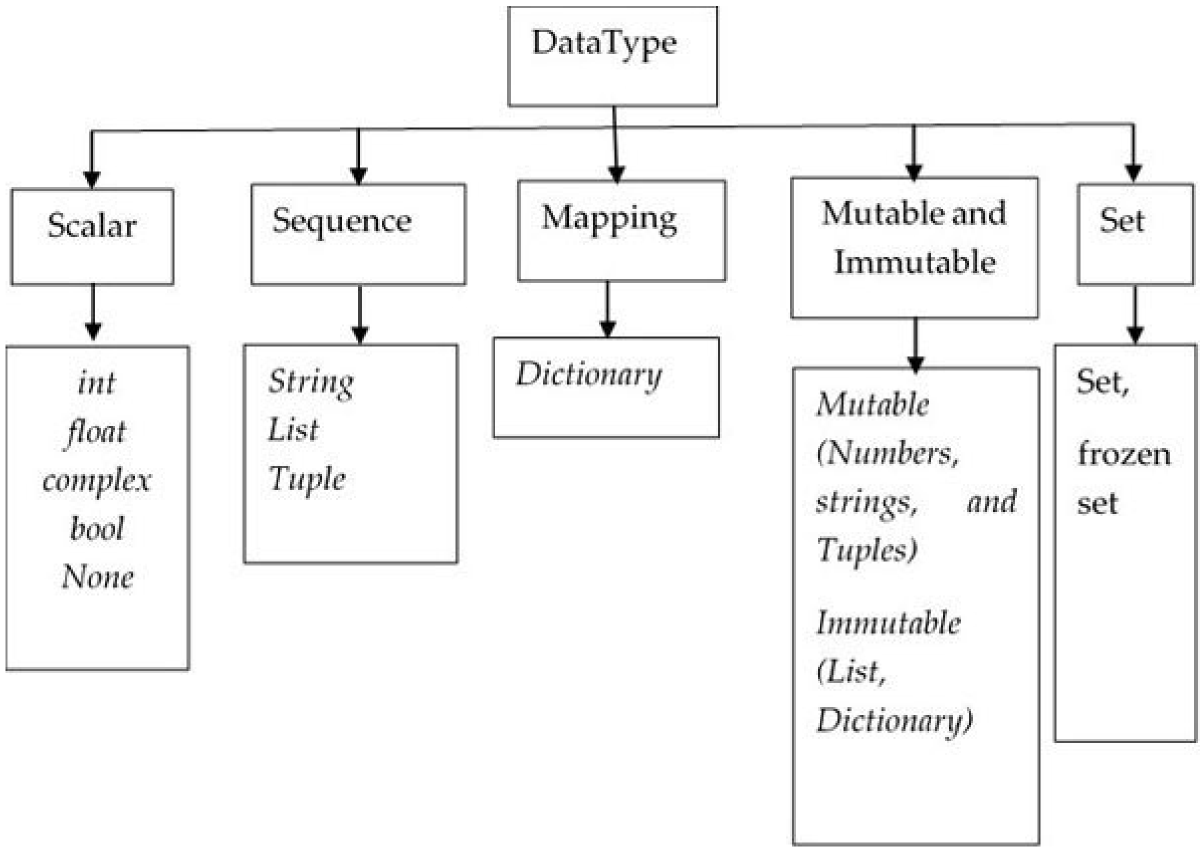


Figure 1.33: *Categories of Data Type*

Example 1.19:

```
# data types
#str
x1 = "Dr. Brijesh Bakariya"
#int
x2= 35
#float
x3 = 205.2
#complex
x4 = 3j

#list
x5 = ["mango", "cherry", "apple"]
#tuple
x6 = ("mango", "cherry", "apple")
#range
x7 = range(3)
#dict
x8 = {"name" : "Brijesh", "age" : 35}
#set
x9 = {"mango", "cherry", "cherry"}
#frozenset
x10 = frozenset({"apple", "banana", "apple"})
#bool
x11 = True
```

```
print (x1)
print (x2)
print (x3)
print (x4)
print (x5)
print (x6)
print (x7)
print (x8)
print (x9)
print (x10)
print (x11)
```

Output:

```
Dr. Brijesh Bakariya
35
205.2


3j
['mango', 'cherry', 'apple']
('mango', 'cherry', 'apple')
range(0, 3)
{'name': 'Brijesh', 'age': 35}
{'cherry', 'mango'}
frozenset({'banana', 'apple'})
True
```

Statements

The statement is an instruction that can be executed by the Python interpreter. There are various types of statements that we use, such as assignment statements, **while** statements, **if** statements, etc.

Most of the statements use indentation for defining the scope of a code. Here, indentation means putting whitespace at the beginning of a line, similar to curly brackets for other programming languages.

Example 1.20: Statement

```
a = 10
b = 20
if b > a:
print ("b is greater than a")
elif a == b:
print ("a and b are equal")
else:
print ("a is greater than b")
```

Output:

```
b is greater than a
```

Keywords

Keywords are not allowed in a variable name, function name, or any other identifier. In Python 3.7, there are 35 keywords. Those are given below:

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

Dictionary

Dictionary is a collection of data values. It uses a two-element key and value. It is an unordered collection of data because it depends on the element defined in the dictionary. Each key is separated from its value by a colon the items are separated by commas, and it is enclosed in curly braces. An item can be changeable.

Example 1.21: Dictionary

```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Brijesh', 2: 'Krishna', 3: 'Amit'}
print ("\nDictionary with the use of Integer Keys: ")
print (Dict)
```

Output:

```
Dictionary with the use of Integer Keys:
{1: 'Brijesh', 2: 'Krishna', 3: 'Amit'}
```

Conclusion

In this chapter, we discussed the basics and various features of Python. We also discussed installing and executing the Python programs. Various IDEs that are used to write Python code. Briefly description about various concepts of Python with examples. Based on these examples, you can write your code for a specific task. In the next chapter, we will discuss operators as the next topic used in Python.

Points to remember

Python is a high-level and interpreted language

Python is a free and open-source language

There are 35 keywords in Python 3.7

Multiple choice questions

Who developed the Python language?

Zim Den

Guido van Rossum

Niene Stom

Wick van Rossum

In which language is Python written?

English

PHP

C

Java

Python was developed in which year?

1972

1995

1991

1981

Python 3.0 was released in which year?

2000

2008

2011

2016

By the use of which character, a single line is made a comment in Python?

*

@

#

!

Which of the following statements is true?

Python is a high-level programming language.

Python is an interpreted language.

Python is an object-oriented language

All of the above

What is used to define a block of code in Python?

Parenthesis

Indentation

Curly braces

None of the above

What is the extension of a Python file?

.p

.py

.python

None of the above

Answers

b

c

c

b

c

d

b

b

Questions

What is Python? Describe various features of Python.

What do you mean by Python Interpreter?

Write Python code to display your name.

How to get input in Python? Write a code for getting input.

What do you mean by indentation? Write a code for it.

What do you mean by keywords? How many keywords are there in Python 3?

Discuss variables and identifiers.

Discuss data type in Python.

CHAPTER 2

Operators and Expressions

Operator is a particular concept in Python that is carried out from the computational point of view. Here, there are two types of the terminology used - operator and operands. The operator is a symbol that is performed with operands' help, and the operand is the value that the operator operates.

Structure

In this chapter, we will cover the following topics:

Types of operators

Writing expression in Python

Use comments during a program

Working with function and modules

Objective

The objective of this chapter is to introduce Python operators, expressions, functions, and modules. After completing this chapter, you should be familiar with various Python operators and their working. Also, you will be able to write Python programs using operators and expressions.

What is an operator?

It is a particular type of symbol that performs some operations on operands and returns the result. Let us take an example for better understanding, 10+20 is an expression. Here, 10 and 20 are operands, and + is an operator. So, in this example, a simple arithmetic expression is evaluated, and the result will be the addition of the two values.

There are various types of operators used in Python. The *table* 2.1 shows the hierarchy of the operators available in Python:

Python:
Python: Python: Python: Python:
Python: Python:
Python: Python:
Python: Python:
Python:
Python: Python: Python:
Python: Python:

Python: Python: Python:

Table 2.1: Hierarchy of the operators

Arithmetic operators

It performs the basic mathematical operation on the numeric operands. The arithmetic operators return the result, and the outcome depends on which type of operands you are using in programming.

Addition The symbol **+** is used while using this operator. It uses between two operands and this operator adds the two value. For example, a=4, b=2 then **a + b** returns 6.

Subtraction The symbol – (minus) is used while using this operator. It uses between two operands, and this operator subtracts the value and returns the result. For example, a=4, b=2, then **a - b** returns 2.

Multiplication The symbol * is used while using this operator. It operates between two operands, and this operator multiplies these values and returns the result. For example, a=4, b=2, then **a*b** returns 8.

Division The symbol **/** is used while using this operator. It uses two operands, and this operator divides the value and returns the result. For example, a=4, b=2 then **a / b** returns 2.

Modulus The symbol % is used while using this operator. It uses between two operands, and this operator returns the remainder. For example, a=4, b=2 then **a % b** returns 0.

Exponent The symbol ** is used while using this operator. It operates between two operands. It performs exponential (power) calculation on operators. For example, a=4, b=2, then **a ** b** means (4)2. The result will be 8.

Floor Division Floor division is like regular division, but it returns the highest possible integer. This integer is less than or equal to the outcome of normal division. For example, **7//2 = 3** and **7.0//2.0 = -11//3 = -11.0//3 = -4.0**

Example 2.1:

```
a=4
b=2
c=a+b      #Addition (+)
print (c)
d=a-b      #Subtraction (-)
print (d)
e=a*b       #Multiplication (*)
```

```
print  (e)
f=a/b                  #Division  (/)
print  (f)
g=a%b                   #  Modulus  (%)
print  (g)
h=a**b                 #Exponent  (**)
print  (h)
i=a//b              #Floor  Division  (//)
print  (i)
```

Output:

```
6
2
8
2.0
0

16
2
```

Comparison operators

These operators compare the two values and also decide the relation between them. It is also called a relational operator because it shows the relation between two operands. This type of operator returns a Boolean value There are various types of relational operators.

Greater than It returns true if the left operand is greater than the right. Example,

Greater than or equal to It returns true if the left operand is greater than or equal to the right. Example,

Less than It returns true if the left operand is less than the right. Example,

Less than or equal to It returns true if the left operand is less than or equal to the right. Example,

Equal to It returns true if both operands are equal. Example,

Not equal to (! It returns true if operands are not equal. Example, **!=**

Example 2.2:

```
x = 10
y = 20
print ("x is equal to y:", x == y)
print ("x is not equal to y:", x != y)
print ("x is less than y:", x < y)
print ("x is greater than y:",  x > y)
x = 100
y = 200

print ("x is either less than or equal to y:", x <= y)
print ("y is either greater than or equal to y:", y >= x)
```

Output:

```
x is equal to y: False
x is not equal to y: True
x is less than y: True
x is greater than y: False
x is either less than or equal to  y: True
y is either greater than or equal to y: True
```

Assignment and shortcut operators

It is used in Python to assign values to variables. Suppose **x = 10** is a simple assignment operator that assigns the value 10 on the right to the variable **x** on the left. There are various compound operators in Python, such as **x += 10** that adds to the variable and later assigns the same. It is equivalent to **x = x +** It is also called a **shortcut** There are various types of assignment and shortcut operators.

Assignment (=) This operator assigns values from right side operands to left side operand **z = x + y** assigns **x + y** into

Add AND (+=) This operator adds the right operand to the left operand and assigns the result to the left operand **z += x** is equivalent to **z = z +**

Subtract AND (-=) This operator subtracts the right operand from the left operand and assigns the result to the left operand **z -= x** equals **z = z –**

Multiply AND (*=) This operator multiplies the right operand with the left operand and assigns the result to the left operand **z *= x** is equivalent to **z = z ***

Divide AND (/=) This operator divides the left operand with the right operand and assigns the result to the left operand **z /= x** is equivalent to **z = z /**

Modulus AND (%=) This operator takes modulus using two operands and assigns the result to left operand **z %= x** is equivalent to **z = z %**

Exponent AND (=)** This operator performs exponential (power) calculation on operators and assign value to the left operand **z **= x** is equivalent to **z = z ****

Floor division (//=) This operator performs floor division on operators and assign value to the left operand **z //= x** is equivalent to **z = z //**

Example 2.3:

```
x = 10
y = 20
z = 0
z = x + y
print (" 1 - Value of z is ", z)
```

```
z += x
print (" 2 - Value of z is ", z)
z *= x
print (" 3 - Value of z is ", z)
z /= x
print (" 4 - Value of z is ", z)
z = 2
z %= x
print (" 5 - Value of z is ", z)
z **= x
print (" 6 - Value of z is ", z)
z //= x
print (" 7 - Value of z is ", z)
```

Output:

```
1 - Value of z is 30
2 - Value of z is 40
3 - Value of z is 400

4 - Value of z is 40.0
5 - Value of z is 2
6 - Value of z is 1024
7 - Value of z is 102
```

Unary operators

It is used in Python, and it is performed on a single operand. There are two types of unary operators such as unary and unary Let us take an example to understand unary operators.

Example 2.4:

```
x = 10
print ("Unary positive operator", +x)
print ("Unary Negative operator", -x)
```

Output:

```
Unary positive operator 10
Unary Negative operator -10
```

Bitwise operators

Bitwise operator works on bits and performs bit-by-bit operation.

Suppose **x = 26** and **y =** then the binary conversion of **x** and **y** is 11010, 10010. Let us understand the various bitwise operations on **x** and **y** with their results:

x= 11010
y= 10010
x & y = 10010
x | y = 11010
x ^ y = 01000
~x = 00101
~y = 01101

There are the following Bitwise operators:

Binary AND This operator copies a bit to the result if it exists in both operands.

Binary OR This operator copies a bit if it exists in either operand.

Binary XOR This operator copies the bit if it is set in one operand but not both.

Binary One's Complement It is unary and has the effect of *flipping* bits.

Binary Left Shift In this operator, the left operand's value is moved left by the number of bits specified by the right operand.

Binary Right Shift In this operator the left operands value is moved right by the number of bits specified by the right operand.

Let us take an example to understand all bitwise operators.

Example 2.5:

```
x = 26                  # 26 = 11010
y = 18                  # 18 = 10010
z = 0
z = x & y,
print ("1 - Value of z is ", z)
z = x | y,
print ("2 - Value of z is ", z)
z = x ^ y,
print ("3 - Value of z is ", z)
z = ~x,
print ("4 - Value of z is ", z)
z = x << 2,
print ("5 - Value of z is ", z)
z = x >> 2,
print ("6 - Value of z is ", z)
```

```
1 - Value of z is  18
2 - Value of z is  26
3 - Value of z is  8
4 - Value of z is  -27
5 - Value of z is  104
6 - Value of z is  6
```

Logical operators

Logical operators are used to perform arithmetic and logical computations. These are special symbols and for accomplishing the operations.

Logical This operator returns true when both the conditions become true.

Logical This operator returns true when two operands are non-zero.

Logical This operator returns the reverse result, or it can say true condition becomes false and false condition becomes true.

Example 2.6:

```
x = 7
print (x > 3 and x < 10) # returns True because 7 is greater than 3 AND 7 is less than 10
print (x > 3 or x < 4) # returns True because one of the conditions are true (7 is greater than 3, but 7 is not less than 4)
print (not(x > 3 and x < 10)) # returns False because not is used to reverse the result
```

Output:

```
True
True
False
```

Membership operators

A membership operator is used to test if a sequence is presented in an object. Here, an object may be strings, lists, or tuples. There are two types of membership operators in and not. Let us take an example of using these operators.

Operator This operator evaluates to true if it finds a variable in the specified sequence and false otherwise.

Operator This operator evaluates to true if it does not find a variable in the specified sequence and false otherwise.

Example 2.7:

```
x= 10
y = 20
list = [10, 12, 23, 44, 35],
if (x in list):
print (" 1 - x is available in the given list")
else:
print (" 1 - x is not available in the given list")
if (y not in list):
```

```
print (" 2 - y is not available in the given list")
else:
print (" 2 - y is available in the given list")
x = 2
if (x in list):
print (" 3 - x is available in the given list")
else:
print (" 3 - x is not available in the given list")
```

Output:

```
1 - x is available in the given list
2 - y is not available in the given list
3 - x is not available in the given list
```

Identity operators

This operator is used to compare the objects, not if they are equal, but if they are the same object, with the exact memory location. There are two types of identity operators **is** and **is** Let us take an example of using these operators.

Operator This operator returns true if both variables are the same object.

Operator (is This operator returns true if both variables are not the same object.

Example 2.8:

```
x = ["Amit", "Raj"]
y = ["Amit", "Raj"]
z = x
print (x is z) # returns True because z is the same object as
x
print (x is y) # returns False because x is not the same object
as y, even if they have the same content
```

```
print (x == y) # This comparison returns True because x is
equal to y
print (x is not z) # returns False because z is the same object
as x
print (x is not y) # returns True because x is not the same
object as y, even if they have the same content
print (x != y) # This comparison returns False because x is
equal to y
```

Output:

```
True
False
True
False

True
False
```

Operators precedence and associativity

Operator precedence means which operator should be executed first. It is based on the priority of an operator. If two or more operators have the same precedence, it will check the associativity of an operator. Associativity means how to operate the value either left to right or right to left.

Suppose an expression is **2*4+6** then first of all **2*4** will be executed because * has higher precedence between * and + operator.

Suppose an expression is then the operator's priority is the same, it will check the associativity of an operator. Here associativity of **/** and * is left to right, then 5/2 will be executed first.

The following table lists all operators from the highest precedence to the lowest precedence:

precedence:
precedence:

precedence:
precedence:
precedence:
precedence:
precedence:
precedence:
precedence:

precedence:
precedence:
precedence:
precedence:
precedence:
precedence:
precedence:

Table 2.2: *Operators description and its associativity*

Example 2.9:

```
x = 12
```

```
y = 14
x = 15
p = 6
q = 0
q = (x + y) * x / p          #(30 * 15) / 5
print ("Value of (x + y) * x / p is ", q)
q = ((x + y) * x) / p        # (30 * 15) / 5
print ("Value of ((x + y) * x) / p is ", q)
q = (x + y) * (x / p),       # (30) * (15/5)
print ("Value of (x + y) * (x / p) is ", q)
q = x + (y * x) / p,         # 20 + (150/5)

print ("Value of x + (y * x) / p is ", q)
```

Output:

```
Value of (x + y) * x / p is 72.5
Value of ((x + y) * x) / p is 72.5
Value of (x + y) * (x / p) is 72.5
Value of x + (y * x) / p is 50.0
```

Expressions in Python

It is a representation of value. Moreover, it is a combination of values, variables, operators, and calls to functions. Expressions need to be evaluated. If you ask Python to print an expression, the interpreter evaluates it and displays the desired result.

Python expressions only contain three things, identifiers, literals and, operators.

It can be any name that is used to define a class, function, variable module, or an object.

It can be string literals, byte literals, integer literals, floating-point literals, and imaginary literals.

Based on the token or symbol, it can evaluate an operation.

There are following operators, and their corresponding token can be used to implement any expression:

add +

subtract –

multiply *

power **

Integer Division /

remainder %

decorator @

Binary left shift <<

Binary right shift >>

and &

or \

Binary XOR ^

Binary ones complement ~

Less than <

Greater than >

Less than or equal to <=

Greater than or equal to >=

Check equality ==

Check not equal !=

Example 2.10:

```
a = 9
b = 12
c = 3
x = a - b / 3 + c * 2 - 1
y = a - b / (3 + c) * (2 - 1)
z = a - (b / (3 + c) * 2) - 1
print ("X = ", x)
print ("Y = ", y)
print ("Z = ", z)
```

Output:

```
X = 10.0
Y = 7.0
Z = 4.0
```

Operations on strings

A string is an essential concept in Python because in many places it can be used in coding. The string can be created either in single quotation marks or double quotation marks. Let us take an example of string creation:

```
string1= 'Python Programming.'
string2="Python Programming."
```

Both are the ways to create a string.

Accessing values in strings

An assignment is much easier than another programming language because Python does not support character type. It is treated as strings of length one. If we want to take a subpart of a string, then we have to take a substring. We have to use the square brackets for slicing, and index or indices to obtain your substring.

String can also update and reassign the value into another string. Let us take an example.

Example 2.11:

```
str1='Python Programming'
str2="Computer Science"
print ("str1[0]: ", str1[0])
print ("str2[0:5]: ", str2[0:5])
print ("Updated String :- ", str1[:7] + 'Book')
```

Output:

```
str1[0]: P
```

str2[0:5]: Compu
Updated String :- Python Book

There are various types of special symbol used in the string:

Concatenation It adds values on either side of the operator.

Repetition It creates a new string, concatenating multiple copies of the same string.

Slice It gives the character from the given index.

Range slice It gives the characters from the given range.

Membership It returns true if a character exists in the given string.

Membership (not It returns true if a character does not exist in the given string.

There are various types of formatted string operators which decide the formatting of string:

%c character

%s string conversion

%i signed decimal integer

%d signed decimal integer

%u unsigned decimal integer

%o octal integer

%x hexadecimal integer (lowercase letters)

%X hexadecimal integer (Uppercase letters)

%e exponential notation (with lowercase 'e')

%E exponential notation (with Uppercase 'E')

%f floating point real number

%g the shorter of %f and %e

%G the shorter of %f and %E

Let us take an example.

Example 2.12:

```
print ("My name is %s and age is %d years" % (Amit, 30))
```

Output:

```
My name is Amit and age is 30 years
```

Triple quotes

It is used for writing a string in multiple lines. It can be written in triple quotes to three consecutive single or double quotes.

Example 2.13:

```
Bio_Auth = """ Brijesh Bakariya received Graduation degree from Barkatullah University Bhopal M.P. in 2005, and Post-Graduation Degree in Computer Applications from Devi Ahilya Vishwavidyalaya Indore M.P. in year 2009. He has done Ph.D. Degree in the Department of Computer Applications, Maulana Azad National Institute of Technology Bhopal M.P"""
print (Bio_Auth)
))
```

Output:

Brijesh Bakariya received Graduation degree from Barkatullah University Bhopal M.P. in 2005, and Post-Graduation Degree in Computer Applications from Devi Ahilya Vishwavidyalaya Indore M.P. in year 2009. He has done Ph.D. Degree in the

Department of Computer Applications, Maulana Azad National Institute of Technology Bhopal M.P bbbbb

There are various types of built-in functions which manipulate strings:

It capitalizes the first letter of the string.

center(width, It returns a space-padded string with the original string cantered to a total of width columns.

count(str, beg= It counts how many times **str** occurs in string or in a substring of string if starting index beg and ending index end are given.

It decodes the string using the codec registered for encoding. It defaults to the default string encoding.

It returns encoded string version of string, on error, default is to raise a **ValueError** unless errors is given with '*ignore*' or

endswith(suffix, beg=0, It determines if a string or a substring of string (if starting index beg and ending index end are given) ends with suffix, returns true if so, and false otherwise.

expandtabs Expands tabs in the string to multiple spaces, defaults to 8 spaces per tab if tab size is not provided.

find (str, beg=0 It determines if **str** occurs in string or a substring of string if starting index beg and ending index end are given, returns index if found, and -1 otherwise.

index(str, beg=0, It is same as but raises an exception if **str** not found.

It returns true if the string has at least one character and all characters are alphanumeric and false otherwise.

It returns true if the string has at least one character and all characters are alphabetic and false otherwise.

It returns true if the string contains only digits and false otherwise.

It returns true if the string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

It returns true if a Unicode string contains only numeric characters and false otherwise.

It returns true if the string contains only whitespace characters and false otherwise.

It returns true if the string is properly "title case" and false otherwise.

It returns true if the string has at least one cased character and all cased characters are uppercase and false otherwise.

It merges (concatenates) the string representations of elements in sequence **seq** into a string, with separator string.

It returns the length of the string.

ljust(width[, It returns a space-padded string with the original string left-justified to a total of width columns.

It converts all uppercase letters in a string to lowercase.

It removes all leading whitespace in a string.

It returns a translation table to be used in the translate function.

It returns the **max** alphabetical character from the string

It returns the **min** alphabetical character from the string

replace(old, new [, It replaces all old in string with new or at most max occurrences if max given.

rfind(str, It is same as but search backwards in string.

rindex(str, beg=0, It is same as but search backwards in string.

rjust(width,[, It returns a space-padded string with the original string right-justified to a total of width columns.

It removes all trailing whitespace of a string.

split(str="", It splits a string according to delimiter **str** (space if not provided) and returns list of substrings, split into at most **num** substrings if given.

It splits a string at all (or NEWLINEs and returns a list of each line with NEWLINEs removed.

startswith(str, It determines if a string or a substring of a string (if starting index beg and ending index end are given) starts with a substring returns true if so and false otherwise.

It performs both **lstrip()** and **rstrip()** on string.

It returns a string where all the upper case letters are lower case and vice versa.

It returns version of a string, that is, all words begin with uppercase and the rest are lowercase.

translate(table, It translates a string according to translation table **str(256** removing those in the del string.

It converts lowercase letters in a string to uppercase.

zfill It returns original string left padded with zeros to a total of width characters, intended for numbers, **zfill()** retains any sign given (less one zero).

Returns true if a Unicode string contains only decimal characters and false otherwise.

Let us take an example to understand all the above built-in functions.

Example 2.14:

```
text = "dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print ("text.capitalize() : ", text.capitalize())
print ("text.center(40, 'a') : ", text.center(40, 'a'))
sub = "i";
print ("text.count(sub, 4, 40) : ", text.count(sub, 4, 40))
sub = "wow";
print ("text.count(sub) : ", text.count(sub))
text1 = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
text2 = "NIT";
print (text1.find(text2))
print (text1.find(text2, 10))
print (text1.find(text2, 40))

text1 = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
text2 = "Ph.D.";
print (text1.index(text2))
print (text1.index(text2, 30))
print (text1.index(text2, 30))
```

```
text = "this2009"; # No space in this texting
print (text.isalnum())



text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.isalnum())

text = "this"; # No space & digit in this texting
print (text.isalpha())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.isalpha())

text = "123456"; # Only digit in this texting
print (text.isdigit())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.isdigit())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.islower())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.islower())
```

```
text = u"this2009";
print (text.isnumeric())

text = u"23443434";
print (text.isnumeric())

text = "  ";
print (text.isspace())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.isspace())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.istitle())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.istitle())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal";
print (text.isupper())

text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"

print (text.isupper())
s = "-";
seq = ("a", "b", "c"); # This is sequence of textings.
```

```
print (s.join(seq))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print ("Length of the texting: ", len(text))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.ljust(50, '0'))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.lower())
text = " Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print ("Max character: " + max(text))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print ("Max character: " + max(text))
text = "this-is-real-texting-example....wow!!!"
print ("Min character: " + min(text))
text = "this-is-a-texting-example....wow!!!"
print ("Min character: " + min(text))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal
this is really texting"
print (text.replace("is", "was"))
print (text.replace("is", "was", 3))
text1 = "this is really a texting example....wow!!!"
text2 = "is"
print (text1.rfind(text2))
print (text1.rfind(text2, 0, 10))
print (text1.rfind(text2, 10, 0))
print (text1.find(text2))
print (text1.find(text2, 0, 10))


print (text1.find(text2, 10, 0))
```

```
text1 = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
text2 = "from"
print (text1.rindex(text2))
print (text1.index(text2))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.rjust(50, '0'))
text = "Line1-abcdef \nLine2-abc \nLine4-abcd"
print (text.split())
print (text.split(' ', 1))
text = "1-a b c d e f\n 2-a b c\n\n 4-a b c d"
print (text.splitlines())
print (text.splitlines(0))
print (text.splitlines(3))
print (text.splitlines(4))
print (text.splitlines(5))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.startswith('this'))
print (text.startswith('is', 2, 4))
print (text.startswith('this', 2, 4))
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.swapcase())
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.swapcase())
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print (text.title())
text = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
print ("text.capitalize() : ", text.upper())
var = "Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal"
```

```
print (var.zfill(40))
print (var.zfill(50))
var = u"this2009";
print (var.isdecimal())
text = u"23443434";
print (text.isdecimal())
```

Output:

```
text.capitalize() : Dr. brijesh bakariya received ph.d. from nit
bhopal
text.center(40, 'a') : dr. Brijesh Bakariya received Ph.D. from
NIT Bhopal
text.count(sub, 4, 40) : 3
text.count(sub) : 0
41
41
41
30
30
30
True
False
True
False
True
False
```

```
False
False
False
True

True
False
False
False
False
False
a-b-c
Length of the texting: 51
Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal
dr. brijesh bakariya received ph.d. from nit bhopal
Max character: y
Max character: y
Min character: !
Min character: !
Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal thwas was
really texting Dr. Brijesh Bakariya received Ph.D. from NIT
Bhopal thwas was really texting
5
5
-1
2
2
-1
```

```
36
36
Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal
['Line1-abcdef', 'Line2-abc', 'Line4-abcd']
['Line1-abcdef', '\nLine2-abc \nLine4-abcd']
['1-a b c d e f', ' 2- a b c', '', ' 4- a b c d']


['1-a b c d e f', ' 2- a b c', '', ' 4- a b c d']
['1-a b c d e f\n', ' 2- a b c\n', '\n', ' 4- a b c d']
['1-a b c d e f\n', ' 2- a b c\n', '\n', ' 4- a b c d']
['1-a b c d e f\n', ' 2- a b c\n', '\n', ' 4- a b c d']
False
False
False
dR. bRIJESH bAKARIYA RECEIVED pH.d. FROM nit bHOPAL
dR. bRIJESH bAKARIYA RECEIVED pH.d. FROM nit bHOPAL
Dr. Brijesh Bakariya Received Ph.D. From Nit Bhopal
text.capitalize() : DR. BRIJESH BAKARIYA RECEIVED PH.D.
FROM NIT BHOPAL
Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal
Dr. Brijesh Bakariya received Ph.D. from NIT Bhopal
False
True
```

Type conversion

It is the process of converting the value from one data type to another data type. The data type may be an integer, string, float, etc. There are two types of type conversion as implicit type conversion and explicit type conversion.

Implicit type conversion

In this conversion, Python automatically converts one data type to another data type. A user does not get involved in this type of conversion. Let us take an example of implicit type conversion.

Example 2.15:

```
num_int = 234

num_float = 23.4
num_new= num_int + num_float
print ("datatype of num_int:", type(num_int))
print ("datatype of num_flo:", type(num_float))
print ("Value of num_new:", num_new)
print ("datatype of num_new:", type(num_new))
```

Output:

```
datatype of num_int: 'int'>
datatype of num_flo: 'float'>
Value of num_new: 257.4
```

```
datatype of num_new: 'float'>
```

Explicit type conversion

In this conversion, users convert the data type of an object to the required data type. It uses some predefined functions like etc. to perform explicit type conversion. Moreover, it is also called typecasting because the user casts (changes) the objects' data type.

Typecasting can be done by assigning the required data type function to the expression. There are following syntax of explicit type casting:

(expression)

Example 2.16:

```
num_int = 345
num_str = "867"
print ("Data type of num_int:", type(num_int))
print ("Data type of num_str before Type Casting:",
type(num_str))

num_str = int(num_str)
```

```
print ("Data type of num_str after Type Casting:",
type(num_str))
num_sum = num_int + num_str
print ("Sum of num_int and num_str:", num_sum)
print ("Data type of the sum:", type(num_sum))
```

Output:

```
Data type of num_int: 'int'>
Data type of num_str before Type Casting: 'str'>
Data type of num_str after Type Casting: 'int'>
Sum of num_int and num_str: 1212
Data type of the sum: 'int'>
```

Python avoids the loss of data in implicit type conversion.

Explicit type conversion is also called type casting. The data types of objects are converted using predefined functions by the user.

In type casting, loss of data may occur as we enforce the object to a specific data type.

Python comments

If we want to explain or understand the particular code, Python provides the facility in comments. Moreover, comments can be used to describe Python code. Comments do not execute, or it is free from execution. Comments start with a **#** symbol, and it can be placed at the end of a line. There are two types of comments in Python. These are:

Single line comments

Multiline comments

Single line comments

It starts with a # symbol that means you have to write **#** symbol in every line.

Multiline comments

A multi-line comment in Python is a block of text enclosed by a delimiter at the beginning and end of the comment. There should be no white space between the delimiters

Let us take an example to understand comments.

Example 2.17:

```
print ("Python Programming") # Single line comment
"""


This is a comment
written in
more lines
"""


print ("Book for Computer Science.")
```

Output:

Python Programming
Book for Computer Science.

Functions in Python

A function is a block of statements that performs a specific task. It is used for code reusability because if you are using some lines then in the place of this, you can use function and function will do the same thing multiple times. There are two types of function, as built-in function and user-defined function.

Built-in A built-in function is already defined, and you have to call the types of function and function that will perform the task.

User-defined We can create our function to perform a particular task. This type of function will be created according to user needs.

Defining a function

It starts with **def** keywords followed by the function name and parentheses

```
def function_name()
```

An argument can be passed within parentheses of the function. The code block within every function starts with a colon and is indented.

Example

```
def fun_name (str): # Function definition
print (str)
return;
fun_name ("Python Programming") # Function calling
```

Output:

```
Python Programming
```

Module

It is used for understanding and organizing a code. It is a group related to code that is called a module. Moreover, a module is a file consisting of Python code. A module can define functions, classes, and variables.

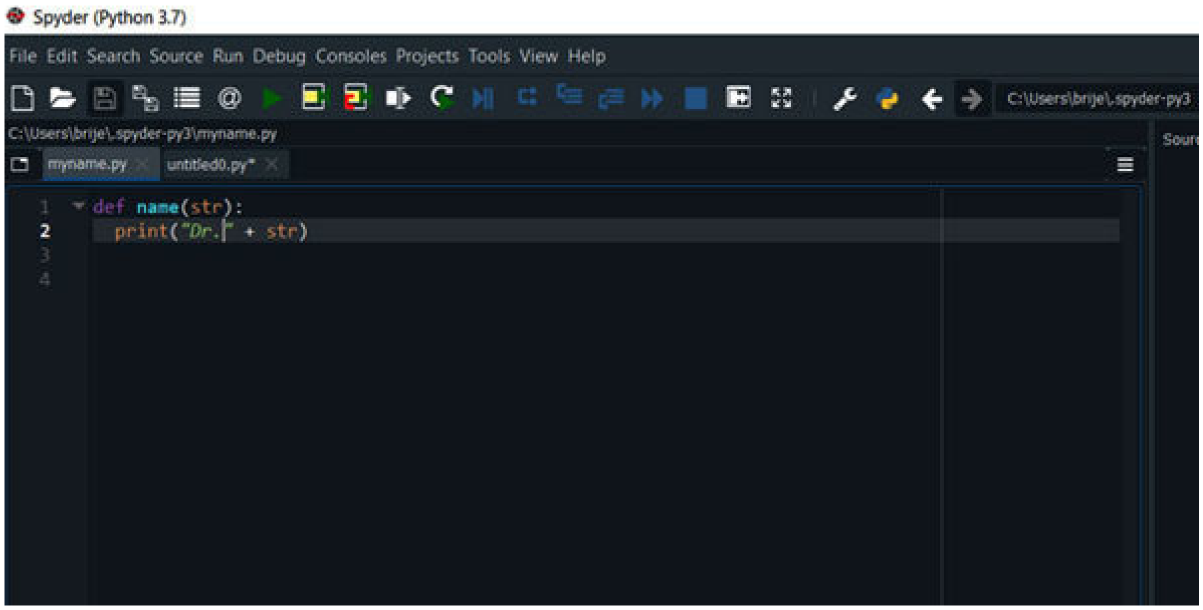
You have to save the code with the file extension This is a module.

Example

Save this code in a file named

```
def fname(str):
print ("Dr.", " + str)
```

The following screenshot shows the code window for saving a module file:


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\brije\.spyder-py3
C:\Users\brije\.spyder-py3\myname.py
myname.py
untitled0.py*
Sour

```
def name(str):
  print("Dr." + str)
```

Figure 2.1: Module file

Use a module

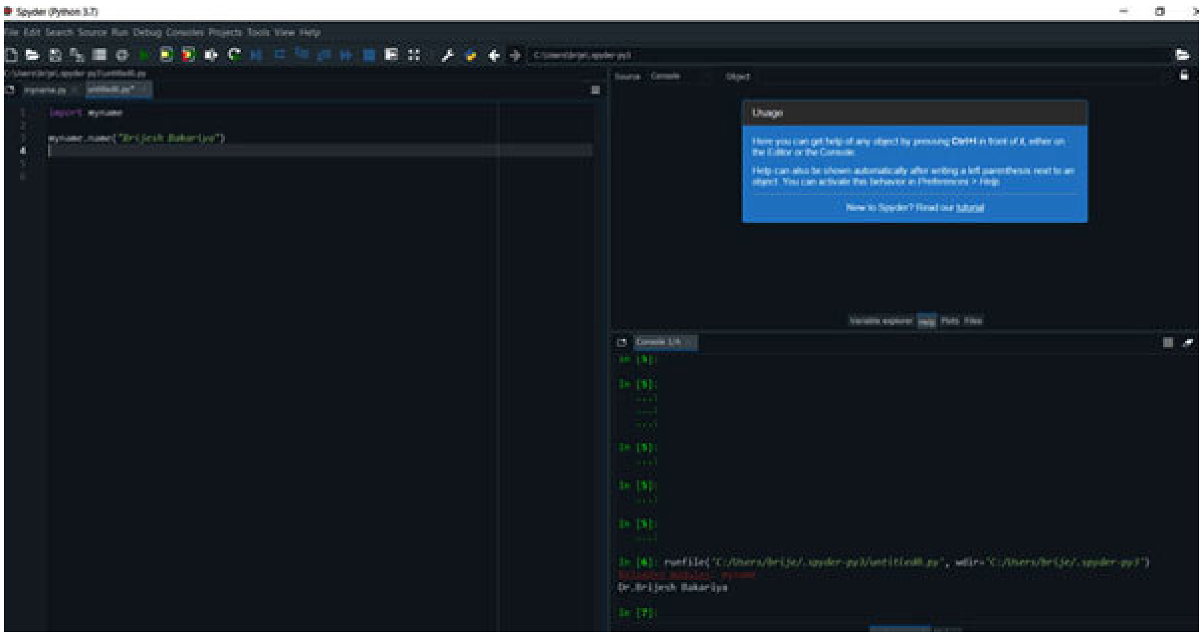
Now we can use the module by using the **import** statement:

Example

Import the module named **myname** and call the name function:

```
import myname
myname.fname("Brijesh Bakariya")
```

The following screenshot shows a code window in which **myname** module is imported:



```
import myname

myname.name("Brijesh Bakariya")
```

Usage

Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in Preferences > Help.

New to Spyder? Read our tutorial

```
In [6]: runfile('C:/Users/brije/.spyder-py3/untitled8.py', wdir='C:/Users/brije/.spyder-py3')
Reloaded modules: myname
Dr.Brijesh Bakariya

In [7]:
```

Figure 2.2: Import and use of module

Parameters or arguments

It is a piece of information that is passed into a function. A parameter is the variable listed inside the parentheses in the function definition. An argument is a value that is sent to the function when it is called. The number of parameters should be matched with several arguments with its data type.

Let us take an example to understand parameters and arguments.

Example 2.21:

```
def name_fun(fname, lname): # Two Parameters
print (fname + " " + lname)
name_fun("Python", "Book") # Two arguments
```

Output:

```
Python Book
```

Conclusion

In this chapter, we discussed various types of operators. We also discussed the operator's precedence and associativity for the evaluation of an expression. We discussed a brief description of various concepts of string operations, function, and module creations. It is discussed how to pass parameters in function and module. In the next chapter, we will discuss control flow statements as the next topic used in Python.

Points to remember

Operator return actual or Boolean values.

The expression will be executed according to the operator's precedence and associativity

Python avoids the loss of data in implicit type conversion.

Comments do not execute, or it is free from execution.

Multiple choice questions

print 9//2

4.5

4.0

4

Error

Which is the correct operator for power(xy)?

x^y

x**y

x^^y

None of the mentioned

Which one of these is floor division?

/

//

%

None of the mentioned

What is the output of this expression, 3*13?**

27

9

3

1

Which one of the following has the same precedence level?

Addition and Subtraction

Multiplication, Division and Addition

Multiplication, Division, Addition and Subtraction

Addition and Multiplication

Which one of the following has the highest precedence in the expression?

Exponential

Addition

Multiplication

Parentheses

Operators with the same precedence are evaluated from?

Left to Right

Right to Left

Depends on Compiler

None of the above

Is "in" an operator in Python?

True

False

Neither true nor false

None of the above

str1="6/4"

print("str1")

1

6/4

1.5

str1

str1="Information"

print(str1[2:8])

format

formatio

orma

ormat

Which keyword is used for function?

Fun

Define

Def

Function

Answers

b

b

b

c

a

d

a

a

d

a

C

Questions

What is an operator? Explain in detail.

How many types of operator are there? Explain each operator in detail.

What do you mean by operators' precedence and associativity? Explain in detail.

Discuss identifiers and literals.

Explain various types of operations performed on strings.

Discuss type conversion with example.

What do you mean by calling and called function?

What do you mean by comments in Python?

CHAPTER 3

Control Flow Statements

The control flow of a program is the order in which the code for the program is executed. Conditional statements, loops, and function calls govern the control flow of a Python program.

Structure

In this chapter, we will cover the following topics:

Types of control flow statements

Types of loops

Use of **break** and **continue** statements

Use of **pass** statements

Objective

The objective of this chapter is to introduce Python control flow statements. After completing this chapter, you should be familiar with various Python control statements such as **break** and Also, you will be able to write Python programs using these conditional and looping statements.

Understanding control flow statement

A control flow statement shows a program's control flow. Moreover, it is the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

There are three types of control structures

Sequential control structures

Selection control structures

Repetition control structures

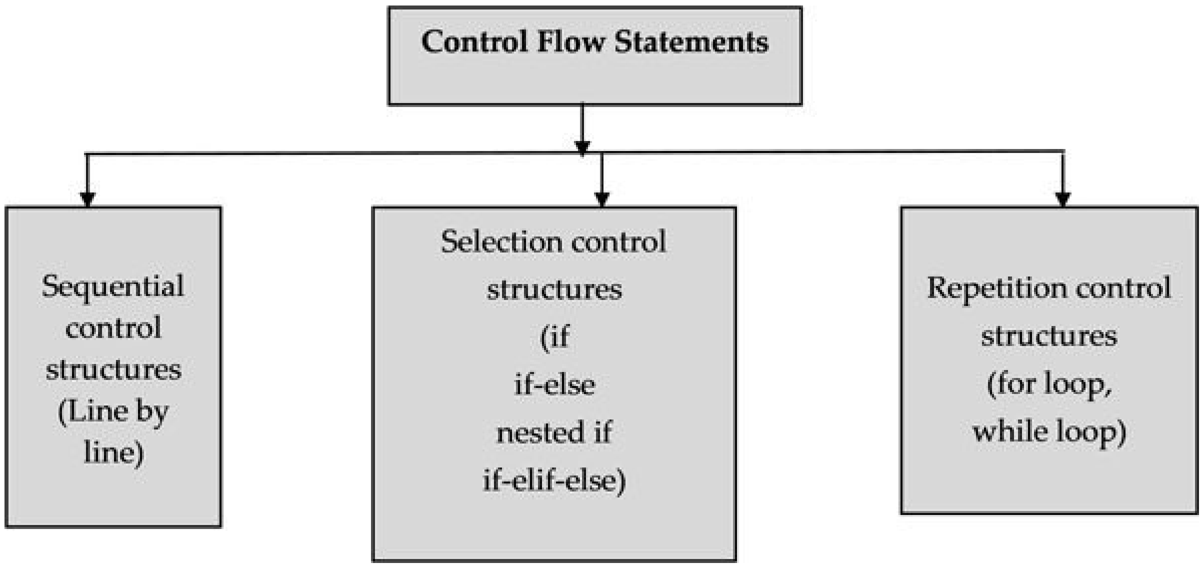

Control Flow Statements
Sequential control structures (Line by line)
Selection control structures (if if-else nested if if-elif-else)
Repetition control structures (for loop, while loop)

Figure 3.1: *Types of control flow statements*

Sequential control structures

It uses a default mode because control moves line by line in a program. Moreover, it is a series of statements that is executed in a sequence.

Example

```
# Python Program to find the area of triangle
a = 5
b = 6
c = 7
# calculate the semi-perimeter
s = (a + b + c) / 2
# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print ('The area of the triangle is %0.2f' %area)
```

Output:

```
The area of the triangle is 14.70
```

Selection control structures

A selection control statement is also known as **decision control statements** or branching statements. The selection statements are based on condition. If a condition is true, then it will be executed. It is also used for checking and testing purposes.

There are various types of decision control statements:

if

if-else

nested if

if-elif-else

if statements

If statements are used for running a particular code based on the condition. If a condition is true, then it will work; otherwise, it won't. The following *figure 3.2* shows the flow diagram of the **if** statement:

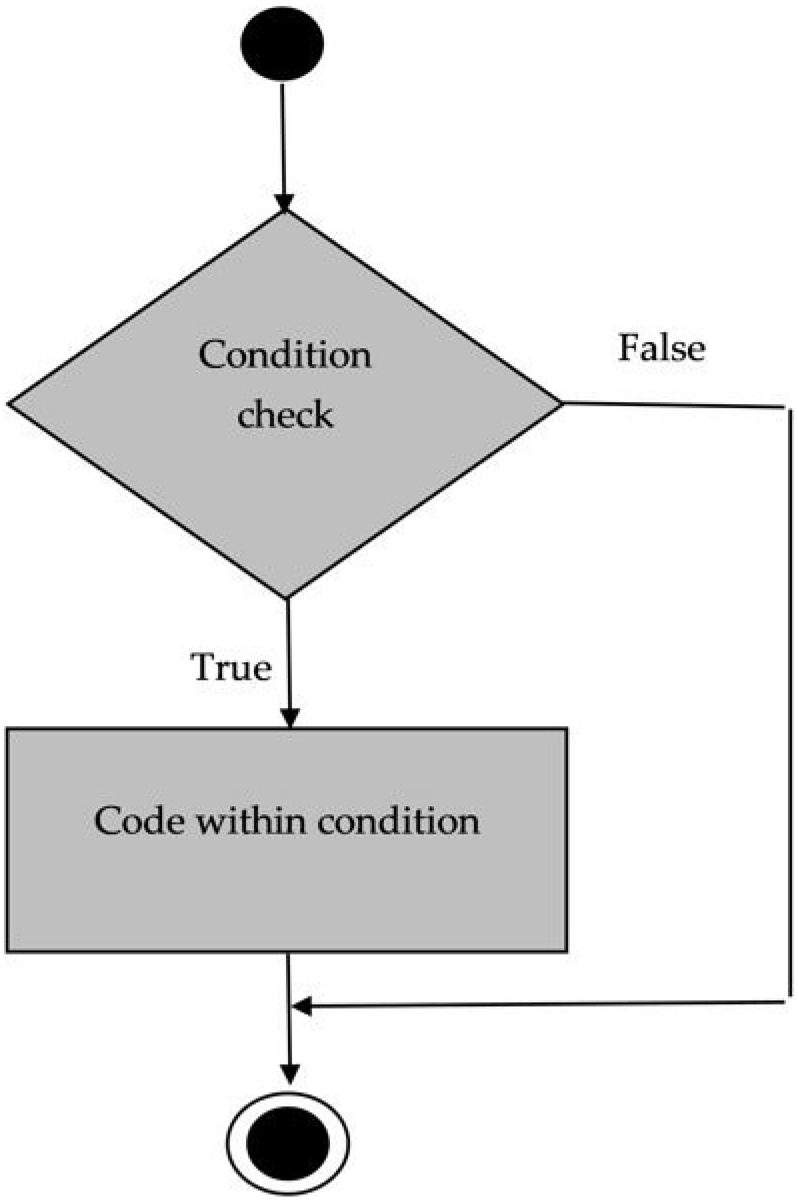


Figure 3.2: *The if statements flow diagram*

Example

```
# If the number is positive and Negative
```

```
n = 17
if n > 0:
print (n, "is a positive number.")
n1 = -12
if n1<0:
print (n1, "is a Negative number.")
```

Output:

```
17 is a positive number.
-12 is a Negative number.
```

Example

```
n1 = 23
n2 = 56
if n2 > n1:
print ("n2 is greater than n1")

if n2 < n1:
print ("n2 is lesser than n1")

if n1 == n2:
print ("n1 and n2 are equal")
```

Output:

n2 is greater than n1

if-else statements

The **if-else** statement is used for running a particular code based on the **if** and **else** statements. If a condition is true, then it will execute the code within **if** block. If a condition is not true, then it will execute the code within the **else** block. The following *figure 3.3* shows the flow diagram of the **if-else** statement:

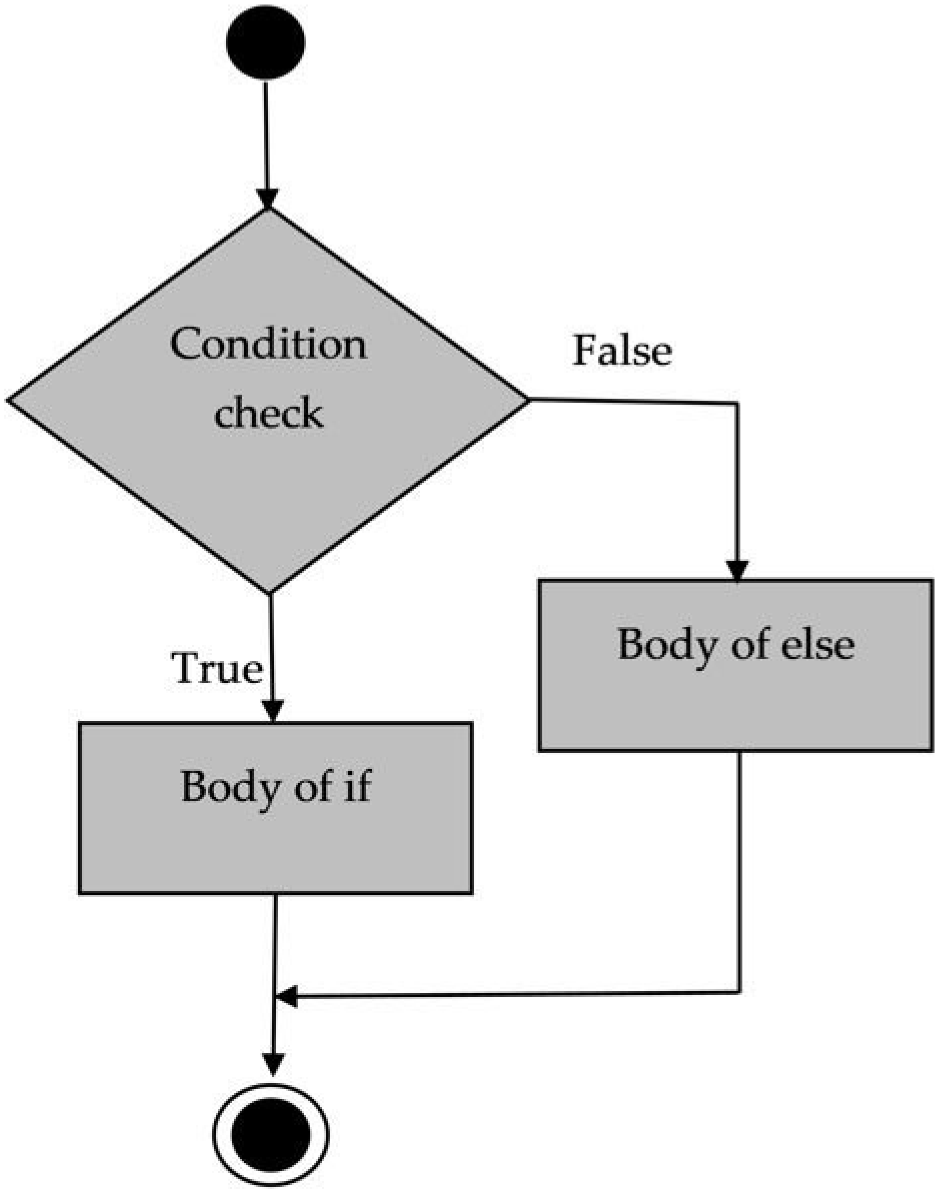


Figure 3.3: *The if-else statements flow diagram*

Example

```
# Program to checks whether the number is positive or
negative
n = 7
if n >= 0:
print ("Positive or Zero")
else:
print ("Negative number")
```

Output:

```
Positive or Zero
```

Example 3.5:

```
price = 500
if price >= 1000:
print ("price is greater than 1000")
else:
print ("price is less than 1000")
```

Output:

```
price is less than 1000
```

nested if

A nested **if** statement means an **if** statement inside another **if** statement. The following *figure 3.4* shows the flow diagram of nested **if** statement:

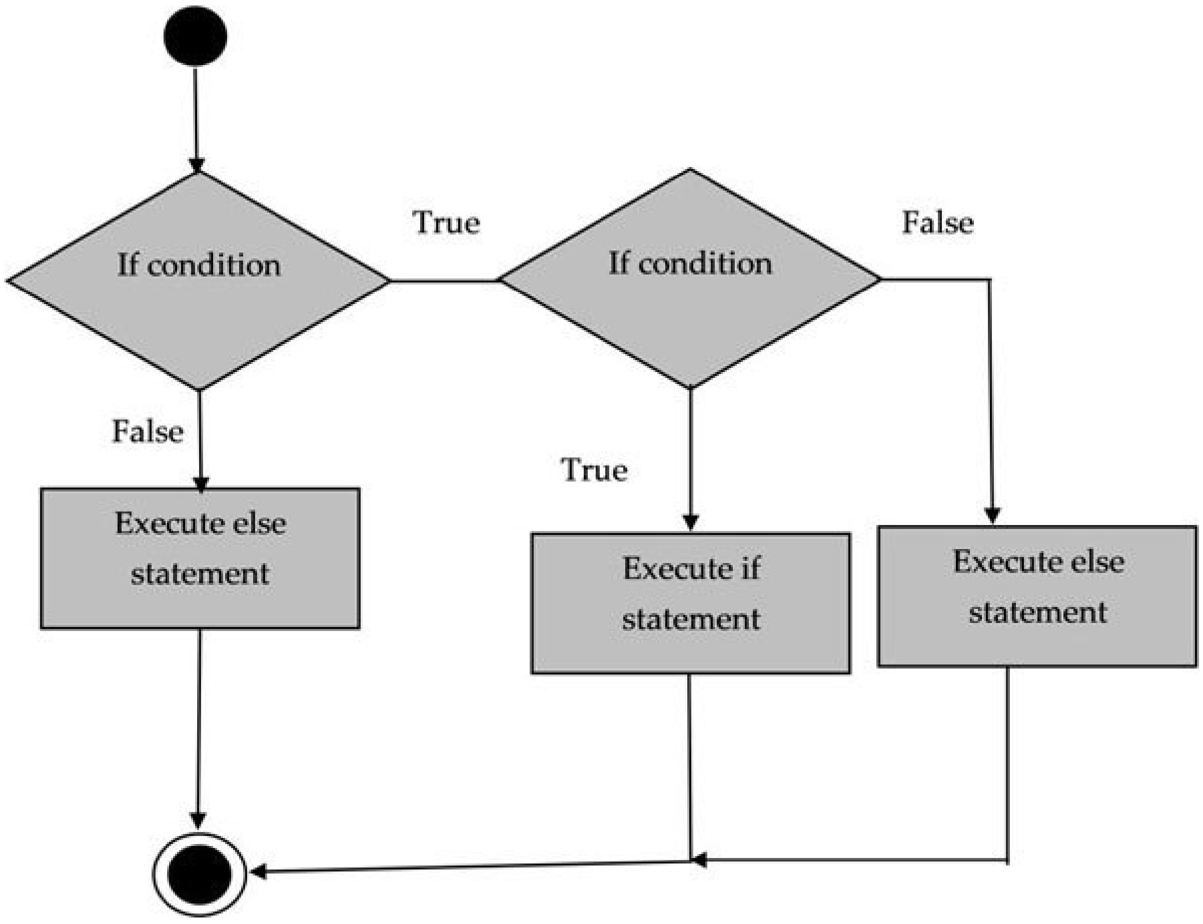


***Figure 3.4:** The nested if flow diagram*

Example 3.6:

```
n = 23
if n >= 0:
if n == 0:
print("Zero")
else:
print ("Positive number")
else:
print ("Negative number")
```

Output:

```
Positive number
```

Example 3.7:

```
x = 32
if x > 10:
print ("Above ten")
if x > 20:
print ("and also above 20")
else:
print ("but not above 20.")
```

Output:

```
Above ten
and also above 20
```

if-elif-else

The **if-elif-else** statement is used to execute a statement or a block of statements conditionally. The following *figure 3.5* shows the flow diagram of **if-elif-else** statement:

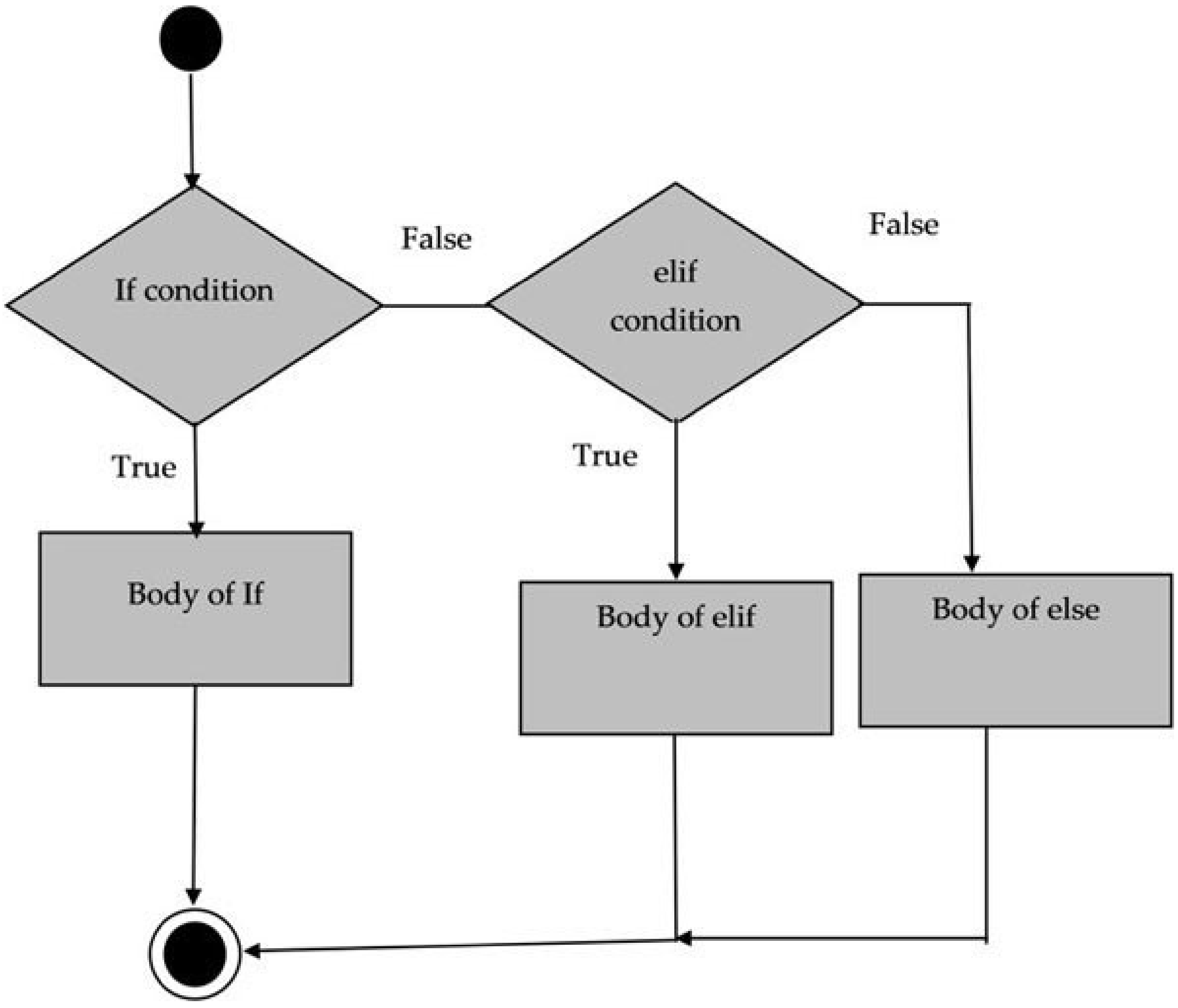


Figure 3.5: The if-elif-else flow diagram

Example 3.8:

```
n = 3.4
if n > 0:
print ("Positive number")
elif n == 0:
print ("Zero")
else:
print ("Negative number")
```

Output:

```
Positive number
```

Example 3.9:

```
price = 50

if price > 100:
print ("price is greater than 100")
elif price == 100:
print ("price is 100")
else :
```

```
print ("price is less than 100")
```

Output:

```
price is less than 100
```

Repetition control structures

It is used to repeat a group of code multiple times, that is, looping. If we want to repeat some instructions according to a particular condition, this control structure will be used.

There are two types of loops in Python:

for loop

while loop

for loop

A **for** loop is used to repeat over a sequence that is either a list, tuple, dictionary, or a set. It is essential to note that the **in** keyword is part of the **for** statement's syntax and is functionally unrelated to the in operator used for membership testing. The following *figure 3.6* shows the flow diagram of **for** loop:

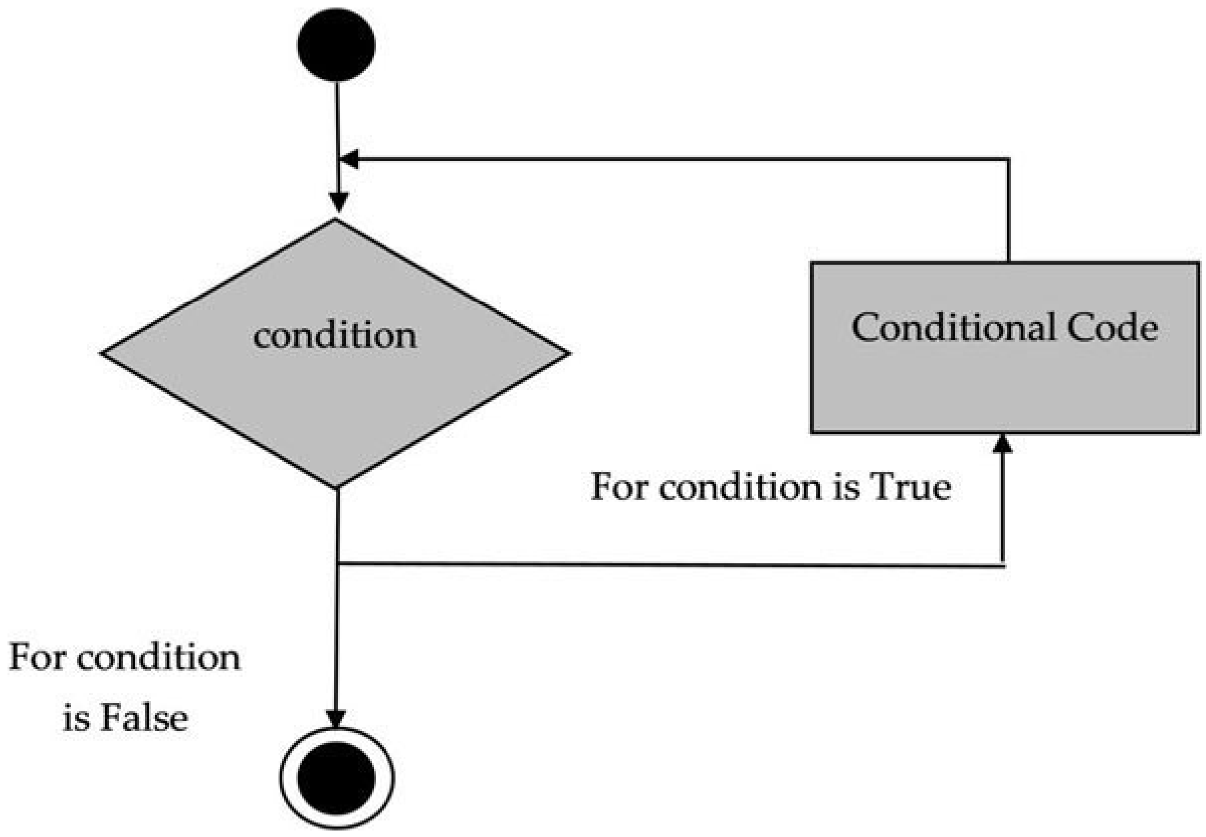


Figure 3.6: *The for-loop flow diagram*

Example 3.10:

```
# Python program to repeating over range 0 to n-1
n = 10
for i in range(0, n):
print (i)
```

Output:

```
0
1
2


3
4
5
6
7
8
9
```

Example 3.11:

```
list = [1,2,3,4,5,6,7,8,9,10]
```

```
n = 5
for i in list:
c = n*i
print (c)
```

Output:

```
5
10
15
20
25
30
35
40
45
50
```

Example 3.12:

```
for x in range(5): #print numbers from 0 to 4
print(x)
for x in range(3, 6): #print numbers from 3 to 5
print(x)

for x in range(3, 8, 2): #print numbers from 3 to 7, step by 2
print(x)
```

Output:

```
0
1
2
3
4
3
4
5
3
5
7
```

while loop

It repeats the statement until a given condition is satisfied. If the condition is true, the **while** loop will work; otherwise, it won't. The flow diagram of the **while** loop is shown in *figure*

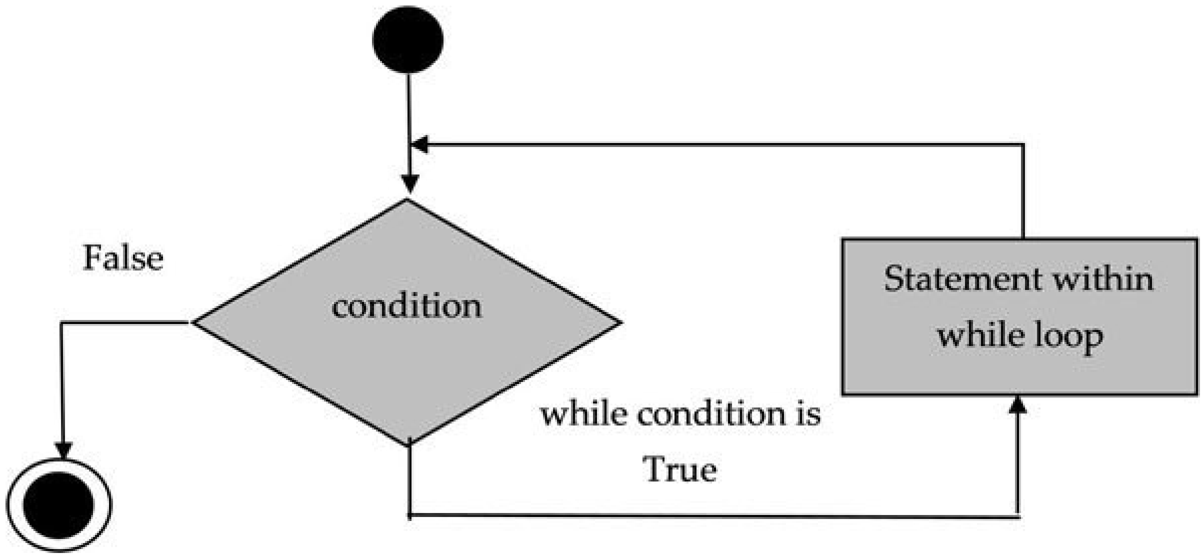


***Figure 3.7:** The while loop flow diagram*

Example 3.13:

```
cnt = 0
while (cnt < 10):
print ("The count is:", cnt)
```

```
cnt = cnt + 1
```

Output:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8

The count is: 9
```

Example 3.14:

```
n = 2
while n < 10:
print (n)
n = n + 3
```

Output:

```
2
```

5
8

Nested loop

A nested loop means one loop inside another loop. We can put a **for** loop inside a or a while inside a or **for** inside or a **while** inside a Let us take an example to understand the nested loop.

Example 3.15:

```
# This program uses a nested for loop to find the prime
numbers from 2 to 20
i = 2
while(i < 20):
j = 2
while(j <= (i/j)):
if not(i%j): break
j = j + 1
if (j > i/j) :
print (i, " is prime")
i = i + 1
```

Output:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

Example 3.16:

```
#program to print a pattern
for i in range (1, 5):

for j in range(i):
print (j + 1, end = ' ')
print ()
```

Output:

```
1
1 2
1 2 3
1 2 3 4
```

Example 3.17:

```
#program to print a pattern
for i in range (1, 5):
for j in range(i):
print (j + 1, end = ' ')
print ()
```

Output:

```
1
1 2
1 2 3
1 2 3 4
```

Break statement

This statement is used to terminate a loop. The control will switch out of the loop when the **break** statement is executed. The flow diagram of the **break** statement is shown in *figure* If the **break** statement is inside a nested loop, then the **break** statement will terminate the innermost loop:

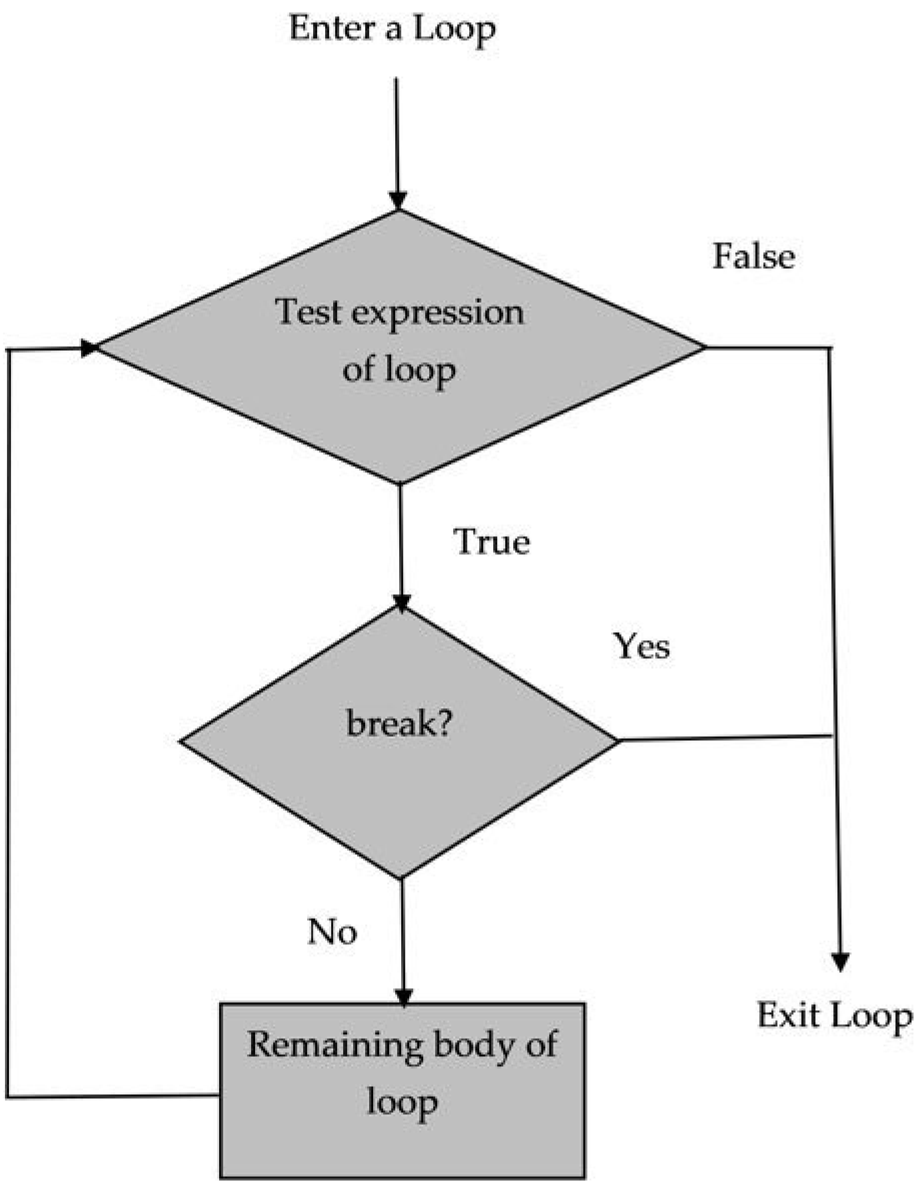


Figure 3.8: *The break statement flow diagram*

Example 3.18:

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9) # Declaring the tuple
num_sum = 0
```

```
count = 0
for x in numbers:
num_sum = num_sum + x
count = count + 1
if count == 6:
break
print ("Sum of first ",count,"integers is: ", num_sum)
```

Output:

```
Sum of first 6 integers is: 21
```

Example 3.19:

```
for i in range(100):
print(i)
if i == 7:
break
print ('Loop exited')
```

Output:

```
0
1
2
3
```

```
4
5
6
7
Loop  exited
```

Continue statement

It is used to skip the rest of the code inside a loop for the current iteration only. In a **continue** statement, the loop does not terminate but continues with the next iteration. The flow diagram of the **continue** statement is shown in *figure*

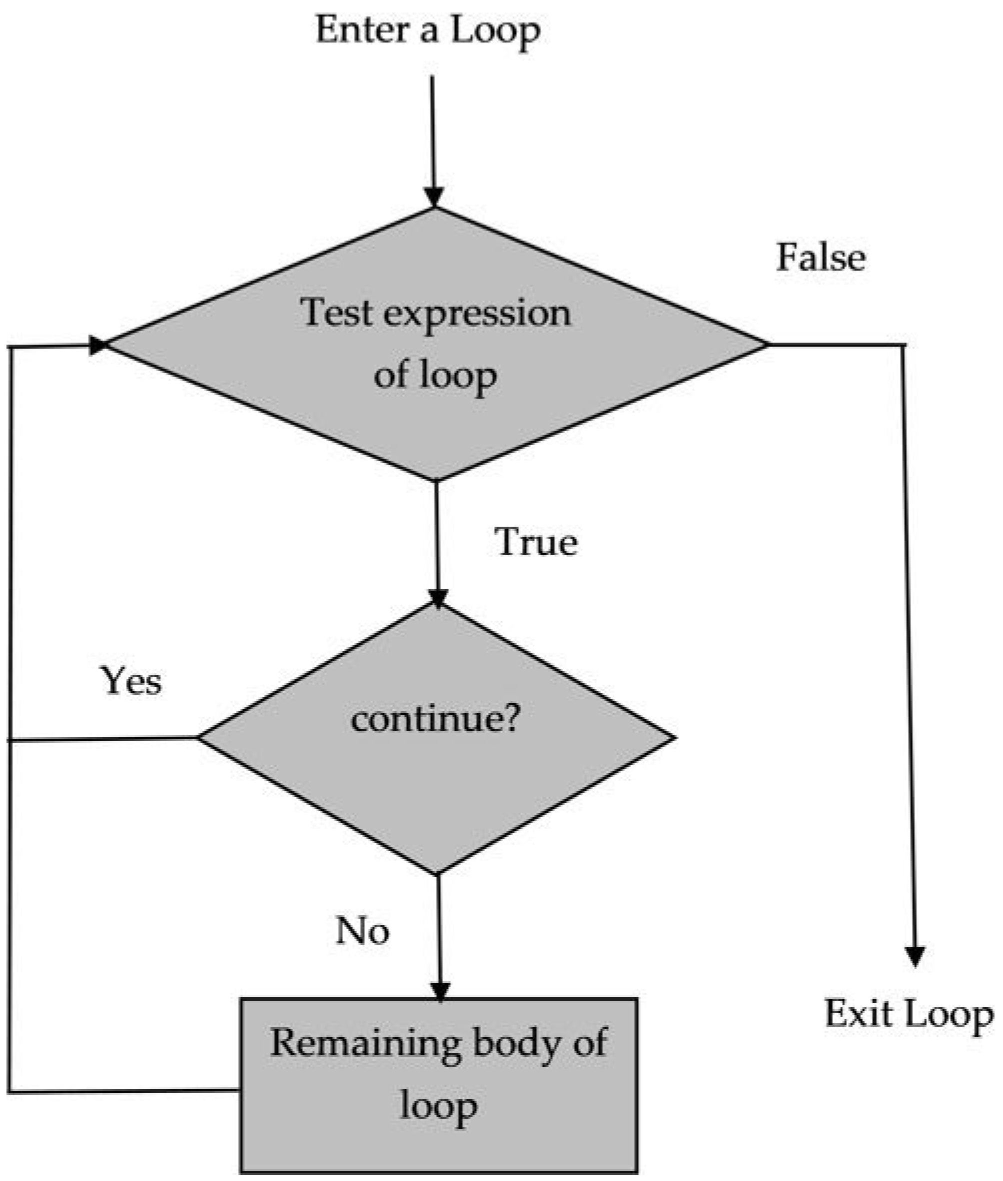


Figure 3.9: *The continue statement flow diagram*

Example 3.20:

```
for i in range(10):

if i==5:
```

```
continue
print (i)
```

Output:

```
0
1
2
3
4
6
7
8
9
```

Example 3.21:

```
for x in range(7):
if (x == 4 or x==6):
continue
print (x)
```

Output:

```
0
1
```

2

3

5

The pass statement

The pass statement means a null statement in Python. The major difference between a comment and a pass statement is that while the interpreter ignores a comment entirely, the pass is not ignored. Moreover, nothing happens when the **pass** is executed. It results in no operation (NOP). Let us take an example to understand the **pass** statement.

Example 3.22:

```
a = 33
b = 200
if b > a:
pass
```

Output:

No output

Example 3.23:

```
sequence = {'b', 'g', 'd', 'k'}
for val in sequence:
```

```
pass
```

Output:

No output

Conclusion

This chapter discussed various types of control statements, that is, sequential control structures, selection control structures, and repetition control structures. It discusses each control statement in detail. It is discussed how the loop will work? So, this chapter contains every control statement with a suitable example and diagram. In the next chapter, we will discuss function as the next topic used in Python.

Points to remember

Sequential control structures use a default mode.

Selection control statements are also known as decision control statements or branching statements.

Repetition control structures are also called loops.

A **break** statement is used for terminating a loop.

The **pass** statement means null statement.

Multiple choice questions

Which of the following is False regarding loops in Python?

Loops are used to perform certain tasks repeatedly.

While loop is used when multiple statements are executed repeatedly until the given condition becomes False

While loop is used when multiple statements are executed repeatedly until the given condition becomes true.

for loop can be used to iterate through the elements of lists.

Can we write if/else into one line in Python?

Yes

No

if/else not used in Python

None of the above

In a Python program, a control structure:

Defines program-specific data structures

Directs the order of execution of the statements in the program

It dictates what happens before the program starts and after it terminates

None of the above

for loop in Python works on

range

iteration

Both of the above

None of the above

In which of the following loops in Python can we check the condition?

for loop

while loop

do-while loop

None of the above

To break the infinite loop, which keyword do we use?

continue

break

exit

None of the above

What do we put at the end of the loop?

semicolon

colon

comma

None of the above

Answers

b

a

b

c

b

b

b

Questions

What do you mean by control flow statement? Discuss its types.

Discuss all the decision control statements with suitable examples.

What do you mean by **for** and **while** loop? Discuss in detail.

Explain nested loop with example.

Write the difference between **break** and **continue** statements? How are they used in programming?

Why **pass** statement is a null statement in Python?

CHAPTER 4

Functions

Functions are a useful way to divide code into manageable pieces, allowing us to organize, make it more readable, reuse, and save time. Furthermore, a function is a code block that executes when it is called. It accepts data as a parameter or an argument and returns the outcome.

Structure

In this chapter, we will cover the following topics:

Use of predefined and user-defined function

Use of local and global variables

Types of functions (lambda functions, recursive functions, etc.)

Use of parameter passing method into a function

Objectives

After studying this chapter, the concepts and working of function in Python should be understood. You should be familiar with parameter passing and returning values from functions. Also, you will be able to write Python programs by using functions.

Introduction to Function

A function contains a block of a statement that performs a specific task. There are some essential rules for defining a function:

Function blocks start with the **def** keyword. After that, the function name and parentheses are used.

An argument or parameter should be **pass** inside the parentheses.

Any function's code block begins with a colon and is indented.

Syntax

```
def function_name(parameters):
"Statements 1"
"Statements 2"
"Statements 3"
-
-
-
"Statements n"
```

return [expression]

There are two types of functions as shown in *figure*

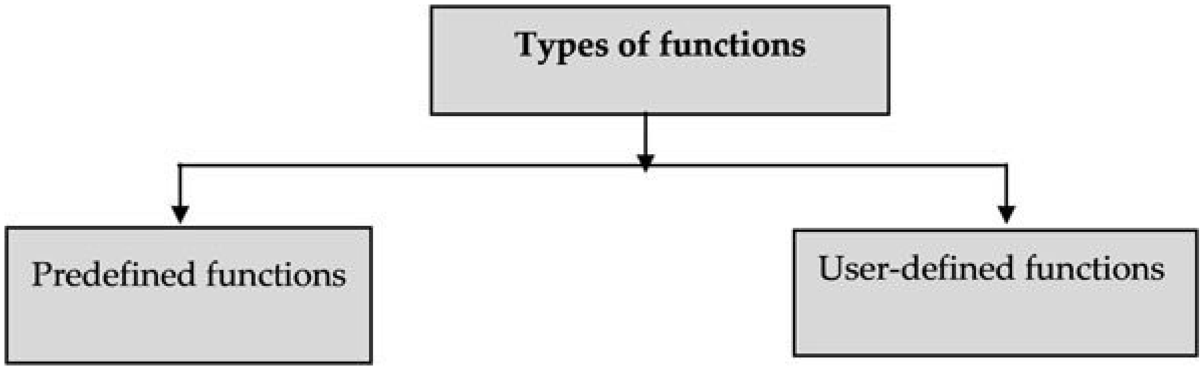


Figure 4.1: *Types of functions*

Predefined functions

It is also called as built-in function because its functionality is predefined in Python. The Python interpreter has several functions that are always present for use. There are various types of built-in functions. Some of the built-in functions are:

It returns the absolute value of a number.

It returns the binary version of a number.

It returns a floating-point number.

It converts a number into a hexadecimal value.

It returns an integer number.

It returns the length of an object.

It returns a list.

It returns the largest item in an iterable.

It returns the smallest item in an iterable.

It converts a number into an octal.

It returns the value of **x** to the power of

It prints to the standard output device.

It returns a sequence of numbers, starting from 0 and increments by 1 (by default).

It rounds a numbers.

Let us take an example of the above built-in functions to clearly understand their usage.

Example

```
#Return the absolute value of a number
x1 = abs(-5.37)
print (x1)

# Return the binary version of 24
x2 = bin(24)
```

```
print (x2)



#Convert the number 7 into a floating-point number
x3 = float(7)
print (x3)

#Convert 321 into hexadecimal value
x4 = hex(321)
print (x4)

#Convert the number 7.2 into an integer
x5 = int(7.2)
print (x5)

#Return the number of items in a list
list1 = ["Brijesh", "Krishna", "Amit", "Raj"]
x6 = len(list1)
print (x6)

#Return the largest number
x8 = max(245, 433)
print (x8)

#Return the lowest number
x9 = min(343, 434)
```

```
print (x9)

#Convert the number 12 into an octal value
x10 = oct(32)
print (x10)

x11 = pow(5, 5) #Return the value of 5 to the power of 5
(same as 5 * 5 * 5 * 5)
print (x11)

#Print a message onto the screen
Print ("It is a message")

#Create a sequence of numbers from 0 to 7, and print each
item in the sequence

x12 = range(8)
for n in x12:
print (n)

#Round a number to only two decimals
x13 = round(6.66543, 2)
print (x13)
```

Output:

```
5.37
0b11000
7.0
0x141
7
4
433
343
0040
3125
It is a message
0
1
2
3
4
5
6
7
6.67
```

User-defined functions

When a function is written for a specific task, those functions are called as user-defined function. Moreover, it is a function created or written by the user. Let's take an example of user-defined functions to clearly understand their working procedure.

Example

```
# Declaring a show function
def show():
print ("Inside function")

# Calling function
show()
```

Output:

```
Inside function
```

Example 4.3:

```
# A simple Python function for checking even and odd
```

```
def evenOdd(var):
if (var%2 == 0):
print ("Number is Even")
else:
print ("Number is Odd")

# Calling a function
evenOdd(12)
evenOdd(37)
```

Output:

```
Number is Even
Number is Odd
```

Function call

We may call a function to perform a task once it has been specified. For calling a function, we type the name of the function and the necessary parameters.

Syntax:

```
function_name (argument_1, argument_2)
```

Example 4.4:

```
# Function Definition
def my_function():
print ("This function has created")
my_function()   # Function Calling
```

Output:

```
This function has created
```

Example 4.5:

```
# Function Definition
def my_name(name):
print (name)
my_name('Ankit')   # Function Calling with argument
```

Output:

```
Ankit
```

Example 4.6:

```
# Function for sum of three numbers
def sum_three_numbers(num1, num2, num3):
return num1 + num2 + num3

# Function calling with three arguments
x= sum_three_numbers (10,20,30)
print (x)
```

Output:

```
60
```

Function parameters and arguments

A piece of information can be passed into a function based on function arguments and parameters. Arguments are a specific value which passes into a function. We can give many arguments, but they should be separated with a comma. Both the terms, parameter and argument can be used interchangeably. Furthermore, data is passed through a function. There is a slight distinction between the function parameters and arguments.

The variable mentioned within the parentheses in the function description is referred to as a parameter, while an argument is a value passed to the function when it is called. Let's look at an example to help you understand the meaning of function arguments and parameters.

Example

```
# Two parameters F_name and L_name
def Name_function(F_name, L_name):  # Function Definition
print (F_name + " " + L_name)
```

```
# Two arguments value (Brijesh, Bakariya)
Name_function("Brijesh", "Bakariya")   # Function Calling
```

Output:

```
Brijesh  Bakariya
```

Default arguments

In Python, default values for function arguments are allowed. The assignment operator can be used to provide a default value for an argument. Consider the following scenario.

Example

```
def my_name(name, msg="how are you?"):    # Function
Definition
print ("Hello", name + ', ' + msg)

my_name("Dr. Brijesh Bakariya")   # Function Calling
my_name("Brijesh", "Welcome")     # Function Calling
```

Output:

```
Hello Dr. Brijesh Bakariya, how are you?
Hello Brijesh, Welcome
```

Let's look at some other examples related to function.

Example 4.9:

```
#Python Arbitrary Arguments
def some_name(*names):
print("Hello", names)
some_name("Brijesh", "Krishna", "Ram", "Amit")

#Arguments passing
def fun1(food):
for n in food:
print(n)
var = ["apple", "banana", "cherry"]
fun1 (fruits)
```

Output:

```
Hello ('Brijesh', 'Krishna', 'Ram', 'Amit')
apple
banana

cherry
```

Variable scope and lifetime

Variable cannot be accessible from any part of the program. Some of the variables may not even exist for the entire duration of the program. The existence and accessibility depends on the declaration of the variable.

Scope of the variable

It is a part of the program in which a variable is accessible. There are two types of scope of the variable.

Local scope

Global scope

Local scope

A variable created in the nested function then the scope exists in that function. Suppose the variable is declared in the innermost function, then the scope of that variable is only in the innermost function.

Global scope

A global variable is one that is generated in the main body of the software and belongs to the global scope. Global variables of this kind are available in any scope. Furthermore, the global variable is easily understood by any function, method, or code.

The lifetime of the variable

It is the duration for which a variable exists. Let's take an example to understand the concept of variable scope and lifetime.

Example

```
# Here a variable created inside a function i.e. local scope
def show():
x = 234   # Local variable
print (x)
show()

x = 234   # Global Variable

def show():

print (x)
show()

print (x)   # Global scope
```

Output:

```
234
234
234
```

Local and global variables

Local variables are declared inside the method or block. Local variables are assessable only within the declared method or block and not outside that method or block.

Example

```
def my_fun():
x=10          #Local variable
print (x)

my_fun()
```

Output:

```
10
```

Example 4.12:

```
def my_fun():
x=10          #Local variable
```

```
my_fun()
print (x)   # Here x is not local variable
```

Output:

```
name 'x' is not defined
```

Global variables

This type of variable is defined in the main body of a file. It is available throughout the program. All kinds of functions or blocks can easily understand and access a global variable.

Example 4.13:

```
x = 100
def my_fun():
print (x) # Calling variable ‘x’ inside my_fun()

my_fun()

print (x) # Calling variable ‘x’ outside my_fun()
```

Output:

```
100
100
```

Example 4.14:

```
x=100   # Global Variable and global scope
def my_fun():
x=50   # Local scope within my_fun()
print (x) # Print local variable

my_fun()

print (x)   # Print Global variable
```

Output:

```
50
100
```

Global statement

The local variable's scope is limited to the block in which it is specified. Furthermore, when we construct a variable within a function, that variable is local. It can only be used within that function, but we can construct a **global** variable within a function using the **global** keyword. Let's look at an example of a **global** statement using the **global** keyword to better understand the definition.

Example

```
def my_func():
global x
x = 100   # x is not local it is global because of global
keyword

my_func()
print (x)   # print a global variable
```

Output:

```
100
```

Suppose we have a variable with the same name as that of a global variable in the program. In such a case, a new local variable of that name is created, different from the global variable.

Example 4.16:

```
x=100
def show():
x=50
print ("In Function x is=",x)
show()
print ("Outside function is=",x)
```

Output:

```
In Function x is= 50
Outside function is= 100
```

Return statement

It's used to finish the function call's execution and return the result (value or expression). There are a few key points to remember when it comes to **return** statements:

Return None is the same as a **return** statement with no arguments.

The statements following the **return** statements are skipped.

If there is no expression in the return declaration, the unique value None is returned.

You can't use a **return** statement outside of a function.

Syntax:

```
def fun_name():
function body
.
.
```

```
.
return [expression]
```

Example 4.17:

```
def fun():
return 10+20
print (fun())
```

Output:

```
30
```

Example 4.18:

```
# Add both the parameters and return them
def addition(n1, n2):
total = n1+ n2

return total;

total = addition(10, 20);
print ("Addition is =", total)
```

Output:

```
Addition is = 30
```

Lambda functions

An anonymous function or function defined without a name is called lambda functions. Generally, functions are defined using the **def** keyword, but anonymous functions are defined using the **lambda** keyword. Lambda functions have the following syntax:

lambda arguments: expression

The following are some essential points about lambda functions:

The number of arguments for a lambda function is infinite, but there is only one expression.

If we need an anonymous function for a short time, we use lambda functions.

The meaning of Lambda does not include a statement; instead, it always includes an expression that is returned.

Example

```
# Add 50 to argument n, and return the result:
res = lambda a : a + 50
print (res(15))
```

Output:

```
65
```

Example 4.20:

```
# sum of three numbers
res = lambda n1, n2, n3 : n1 + n2 + n3
print (res(10, 20, 30))
```

Output:

```
60
```

Recursive functions

When a function called itself is called recursion. When we define a function and call the same function in their definition that means that code is recursive. Recursion is a very efficient technique for programming but one thing is to be remembered that the recursion should be terminated. If the terminating condition is there in recursion that means the code is properly working. If the terminating condition is not there, that means the recursive code is going to infinite times.

Syntax:

```
def func_name(): <-
        |
        | (recursive call)
        |
func_name()  ----
```

The following are the advantages and disadvantages of a recursive function.

Advantages

Using recursion, this function can be broken down into smaller sub-problems.

Often creating a sequence is simpler than nested iteration.

The use of recursive functions makes the code appear simple and efficient.

Disadvantages

Recursive calls use a lot of memory and time, which is why they are costly to use.

Debugging recursive functions is difficult.

The logic behind recursion can be difficult to understand at times.

Example 4.21:

```
# Program for fibonacci series up to k terms
def fib(k):         # Recursive function
if k <= 1:
return k
else:
return(fib(k-1) + fib(k-2))

term=9
if term <= 0:
print ("Invalid input ! Please input a positive value")
else:
```

```
print ("Fibonacci series:")
for i in range(term):
print (fib(i))
```

Output:

```
0
1
1
2
3
5
8
13

21
```

Example 4.22:

```
# Recursive program for factorial of a number
def fact(k):     # Recursive function
if k == 1:
return k
else:
return k * fact(k-1)
```

```
n = 7
# check if the input is valid or not
if n < 0:
print ("Invalid input ! Please enter a positive number.")
elif n == 0:
print ("Factorial of number 0 is 1")
else:
print ("Factorial of number", n, "=", fact(n))
```

Output:

```
Factorial of number 7 = 5040
```

Function redefinition

Python provides a facility for the redefinition of function because Python is dynamic. Redefinition of function means redefining an already defined function. Let's take an example to understand the concept of function redefinition.

Example

```
# Program for function redefinition
from time import gmtime, strftime

def show(msg):        # Function definition
print (msg)
show("Ready.")

def show(msg):        # Function redefinition
print (strftime("%H:%M:%S", gmtime()))
print (msg)
show ("Processing.")
```

Output:

Ready.

11:20:20

Processing.

Conclusion

In this chapter, we discussed function and how they are important for modular programming. They are concerned about the parameter passing method into a function. We have also discussed local and global variable, statements, and their scope. So, this chapter contains every variation of function and its working procedure. In the next chapter, we will discuss strings and its related features as the next topic used in Python.

Points to remember

Function blocks start **def** keyword after the function name and parentheses ().

Each function has an indented code block that begins with a colon

Return None is equivalent to a return statement with no arguments.

The number of arguments in a lambda function is unlimited, but there is only one expression.

Multiple choice questions

Which keyword is used for function definition?

fun

define

def

function

Which of the following encloses the input parameters or arguments of a function?

brackets

parentheses

curly braces

quotation marks

What is called when a function is defined inside a class?

class

function

method

Module

If the return statement is not used inside the function, the function will return:

None

0

Null

Arbitrary value

What is a recursive function?

A function that calls other functions.

A function which calls itself.

Both A and B

None of the above

How is a function declared in Python?

def function function_name():

declare function function_name():

def function_name():

declare function_name():

What will be the output of the following Python code?

```
x = 50
def func():
```

```
global x
print ('x is', x)
x = 2
print ('Changed global x to', x)
func()
print ('Value of x is', x)
```

```
x is 50

Changed global x to 2

Value of x is 50

x is 50

Changed global x to 2

Value of x is 2

x is 50

Changed global x to 50

Value of x is 50
```

Error

What will be the output of the following Python code?

```
def show(message, times = 1):
print(message * times)
show('Hello')
show('World', 5)
```

Hello

WorldWorldWorldWorldWorld

Hello

World 5

Hello

World,World,World,World,World

Hello

HelloHelloHelloHelloHello

Answers

c

b

c

a

b

c

b

a

Questions

What is a function? Discuss its types.

What do you mean by function call? Discuss with an example.

Explain function parameters and arguments with examples.

What do you mean by the scope of the variable? Explain with an example.

Discuss the **global** statement with an example

Discuss **return** statement with an example.

What do you mean by lambda functions? Explain with a suitable example.

Write all the advantages and disadvantages of recursive functions.

CHAPTER 5

Strings

Strings are arrays of bytes in Python that represent Unicode characters. However, since Python lacks a character data type, a single character is simply a string with one length. Square brackets may be used to access the string's elements. Single quotes ('...') or double quotes ("...") may be used to enclose strings in Python.

Structure

In this chapter, we will cover the following topics:

Use of various types of string operations

Use of built-in string functions

Use of string formatting operator

Use of immutable strings

Objective

After studying this chapter, you should be able to understand the working of strings and various strings operations in Python. Also, you will be able to write programs with strings, including string comparison, iterations, and immutable.

Concepts of String

A string is a sequence of characters. It is an essential concept in Python. There are some important points about strings.

Strings are amongst the most popular types in Python.

Python can create strings simply by enclosing characters in quotes (single, double, or triple).

Python treats single-quotes the same as double-quotes.

A string is a sequence of Unicode characters and a character is simply a symbol.

If we want to create a string of names, then it has to write the name within quotes like **Write name** The string can also assign a variable for further operation and usage of that string. Let us take a basic example to understand this concept of string.

Example

```
str = "This is a sting."
print (str)
```

Output:

```
This is a string
```

The character data type is not available in Python. A single character is considered as a string and its length would be one. The individual elements of any string can be accessed using square brackets

Example 5.2:

```
str = "Brijesh"
print (str[0])
print (str[1])
print (str[2])

print (str[3])
print (str[4])
print (str[5])
print (str[6])
```

Output:

```
B
r
i
j
e
s
h
```

The string can use **for** loop, character by character.

Example

```
for i in "Krishna":
print (i)
```

Output:

```
K
r
i
s
h
n
a
```

Other concepts of strings such as **len()** function for getting the length of a string, keyword **not in** for checking the substring or character is present or not in a string. It can also use in an **if** statement.

Example

```
# print a length of a string
str = "Brijesh"
print (len(str))   #String Length

# print "simple" in a string
str1= "Python is so simple"
print ("simple" in str1)

#Print only if "simple" is present in a string:
str2 = "Python is so simple"
if "simple" in str2:
print ("Yes, 'simple' is present.")

# print "simple" is not in a string
str3 = "Python is so simple"
print ("programming" not in str3)
```

Output:

```
7
True
Yes, 'simple' is present.
True
```

String concatenation

String concatenation means combining two strings. If we have two strings and those strings have been assigned into two variables **str1** and they can be concatenated or combined **str1** and

There are four ways to concatenate the strings.

Using + operator

Using **join()** method

Using % operator

Using **format()** function

Using + operator

It is a straightforward way to concatenate two strings. It can add multiple strings together. Let us take an example to understand this concept.

Example

```
# Defining strings
fname = "Brijesh"
lname = "Bakariya"

# + Operator is used to combine strings
name = fname + lname
print (name)
```

Output:

```
BrijeshBakariya
```

Using join() method

The **join()** method is a string method that returns a string in which the sequence elements have been joined using an **str** separator.

Example

```
fname = "Brijesh"
lname = "Bakariya"

print ("".join([fname, lname]))

# join() method string with a separator Space(" ")
name = " ".join([fname, lname])
print (name)
```

Output:

```
BrijeshBakariya
Brijesh Bakariya
```

Using % operator

It can use a string formatting operator for concatenating the strings.

Example 5.7:

```
fname = "Brijesh"
lname = "Bakariya"
# % operator
print ("% s % s" % (fname, lname))
```

Output:

```
Brijesh Bakariya
```

Using format() function

It's an example of a string formatting method. Multiple substitutions and value formatting are possible. To concatenate the strings, it must use This method uses positional formatting to concatenate elements inside a series. The **format()** function in the following example combines the string stored in the **fname** and **lname** variables and stores it in another variable name. The curly braces are used to position strings in a specific order. The first variable is stored in the first set of curly braces, while the second set of curly braces is stored in the second set of curly braces.

Example

```
fname = "Brijesh"
lname = "Bakariya"
print ("{} {}".format(fname, lname))

# result in name
name = "{} {}".format(fname, lname)
print (name)
```

Output:

```
Brijesh  Bakariya
Brijesh  Bakariya
```

Appending strings

Appending to string means adding one string to other. Suppose we have two strings and those strings have been assigned into two variables **str1** and then it can be appended or add **str1** to There are two ways to append one string to another.

Using += operator

Using join()

Using += operator

It is a more straightforward way than the traditional way employed in other languages, like using a dedicated function to perform this particular task. Let us take an example to understand this concept.

Example

```
# initializing string
str1 = "Brijesh"
# initializing add_string
str2 = "Bakariya"
# printing original string
print ("The original string : " + str(str1))
# printing original add string
print ("The add string : " + str(str2))

# adding one string to another
str1 += str2                # Using += operator
# print result
print ("The appended string is : " + str1)
```

Output:

```
The original string: Brijesh
The add string: Bakariya
The appended string is: BrijeshBakariya
```

Using join()

This method is used to join the strings. This method's advantage over the above method can be realized when we have many strings to concatenate rather than just two. Let us take an example to understand this concept.

Example

```
# initializing string
str1 = "Brijesh"

# initializing add_string
str2 = "Bakariya"
# printing original string
print ("The original string : " + str(str1))
# printing original add string
print ("The add string : " + str(str2))

# adding one string to another
result = "".join((str1, str2))          # Using join()
# print result
print ("The appended string is : " + result)
```

Output:

```
The original string : Brijesh
The added string : Bakariya
The appended string is : BrijeshBakariya
```

Multiplying strings

Multiplication of strings in Python is much easier because it is similar to another data type (number value, etc.). There are various methods that we can go to multiply strings. Let us take an example to understand this concept.

Example

```
str1= "Python Programming"
str2= 2*str1      # Method 1
print (str2)

str3=4*(str1) # Method 2
print (str3)

str4= 3*('Python', 'Programming') # Method 3
print (str4)
```

Output:

```
Python ProgrammingPython Programming
Python  ProgrammingPython  Programming
('Python',  'Programming',  'Python',  'Programming',  'Python',
'Programming')
```

Immutable strings

An immutable string means that the string values cannot be updated. Moreover, once we assign the string's value, it cannot be reassigned when we perform this, it will produce an error. Let us take an example to understand this concept.

Example

```
str= "Brijesh"
print (str)
str[0] = "V"     # Cannot reassign
```

Output:

```
str[0] = "V"     # Cannot reassign
TypeError: 'str' object does not support item assignment
```

String formatting operator

It uses the % operator for string formatting operations. Various symbols can be used along with as shown in *table*

Table 5.1: *String formatting symbols and their description*

Let us take an example to understand the concept of string formatting operator.

Example

```
# Initialize variable as a string
var = '27'
```

```
string = "Variable as string = %s" %(var)
print (string)

# Printing as raw data
print ("Variable as raw data = %r" %(var))

# Convert the variable to integer

# And perform check other formatting options
var = int(var)
string = "Variable as integer = %d" %(var)
print (string)
print ("Variable as float = %f" %(var))

print ("Variable as printing with special char = %c Ram" %
(var))

print ("Variable as hexadecimal = %x" %(var))
print ("Variable as octal = %o" %(var))
```

Output:

```
Variable as string = 27
Variable as raw data = '27'
Variable as integer = 27
Variable as float = 27.000000
Variable as printing with special char = Ram
Variable as hexadecimal = 1b
Variable as octal = 33
```

Built-in string functions

There are various types of built-in string functions. We just use those functions and get the desired results. One crucial point is that these functions return new strings. It cannot change the original string. Here are some essential built-in string functions.

It returns the number of times a specified value occurs in a string.

It returns **True** if all characters in the string are in lower case.

It splits the string at the specified separator and returns a list.

It returns **True** if the string ends with the specified value.

It returns a centered string.

It returns **True** if all characters in the string are alphanumeric.

It returns **True** if all characters in the string are numeric.

It converts the first character of each word to upper case

Let us take an example to understand the above built-in string functions.

Example

```
# built-in string functions
txt = "Python is a good prog., Python is easy to learn, Python
is easy to understand."
cnt = txt.count("Python")
print (cnt)



low = txt.islower()  # check if all the characters in the text are
in lower case
print (low)

spl= txt.split()  #Split a string into a list where each word is a
list item
print (spl)

ew = txt.endswith(".") #Check if the string ends with a (.)
print (ew)

cen = txt.center(10) # space of 10 characters
```

```
print (cen)

an = txt.isalnum()  #Check if all the characters in the text are
alphanumeric
print (an)

isn = txt.isnumeric()  #Check if all the characters in the text
are numeric
print (isn)

tit = txt.title() # first letter in each word upper case
print (tit)
```

Output:

```
3
False
['Python', 'is', 'a', 'good', 'prog.,', 'Python', 'is', 'easy', 'to', 'learn,',
'Python', 'is', 'easy', 'to', 'understand.']
True
Python is a good prog., Python is easy to learn, Python is
easy to understand.
False
False
Python Is A Good Prog., Python Is Easy To Learn, Python Is
Easy To Understand.
```

Slice operation

The **slice()** function returns a slice object that can be used to slice strings, lists, tuple, etc. Following is the syntax with parameters description of **slice()** function.

```
slice(stop)
slice(start, stop, step)
```

A **start** parameter specifies the index at which an object's slicing begins. A stop parameter sets the index at which an object's slicing comes to a halt. The increment between each index for slicing is determined by the step parameter, an optional statement.

Following are some important points about the **slice()** function

The **slice()** function returns a sliced object that only contains elements from the defined set.

If only one parameter is passed, then the start and step are considered to be

Let us take an example to understand the concept of **slice()** functions.

Example

```
# It start the slice object at position 2, and slice to position 4, and return the result:
var = ("b", "r", "i", "j", "e", "s", "h")
sl = slice(2, 4)
print (var[sl])

#It use the step parameter to return every second item:
var = ("b", "r", "i", "j", "e", "s", "h")
sl = slice(0, 7, 2)
print (var[sl])
```

Output:

```
('i', 'j')
('b', 'i', 'e', 'h')
```

```
# String slicing
str ='BrijeshBakariya'
sl1 = slice(2)
sl2 = slice(1, 4, 3)
print ("String slicing")
print (str[sl1])
print (str[sl2])

# List slicing
lst = [5, 6, 7, 8, 9]
sl1 = slice(2)
sl2 = slice(1, 4, 2)
print ("\nList slicing")
print (lst[sl1])
print (lst[sl2])

# Tuple slicing
tpl = (4, 5, 6, 7, 8)
sl1 = slice(3)
sl2 = slice(1, 5, 2)
print ("\nTuple slicing")
print (tpl[sl1])
print (tpl[sl2])
```

Output:

```
String slicing
Br
r

List slicing


[5, 6]
[6, 8]

Tuple slicing
(4, 5, 6)
(5, 7)
```

The ord() and chr() functions

The ord() function

It's a built-in feature that returns the single Unicode character's integer as an integer. The returned integer represents the Unicode code point. Furthermore, the **ord()** function returns an integer representing the character's Unicode code point when passed a string of length 1. A **TypeError** will be raised if the string length is greater than one. Let us take an example to understand the concept of **ord()** functions.

Example

```
res1 = ord('A')
res2 = ord('a')
print (res1, res2)     # prints the unicode value
```

Output:

```
65 97
```

The chr() function

From an integer, the **chr()** method returns a character or string. The integer represents the character's Unicode code point. Let us take an example to understand the concept of **chr()** functions.

Example

```
res1 = chr(65)      #return a character of unicode valve 65
res2 = chr(97)     #return a character of unicode valve 97
print (res1, res2)    # prints the character value
```

Output:

A

Comparing strings

Comparing strings means identifying whether the two strings are equivalent to each other or not. There are three ways to compare the strings.

Using relational operators

Using is and is not

User-defined function

Using relational operators

The relational operators compare the Unicode values of the characters of the strings. It returns a Boolean value according to the operator used. Following are the relational operators available in Python.

Operator This operator checks whether two strings are equal.

Operator This operator checks if two strings are not equal.

Operator This operator checks if the string on its left is smaller than that on its right.

Operator This operator checks if the string on its left is smaller than or equal to that on its right.

Operator This operator checks if the string on its left is greater than that on its right.

Operator This operator checks if the string on its left is greater than or equal to that on its right.

Let us take an example to understand the concept of relational operators for string.

Example

```
print ("Brijesh" == "Brijesh")
print ("Brijesh" < "brijesh")
print ("Brijesh" > "brijesh")
print ("Brijesh" <= "brijesh")
print ("Brijesh" >= "brijesh")
print ("Brijesh" != "Brijesh")
```

Output:

```
True
True

False
True
False
False
```

Using is and is not

This type of operator checks whether both the operands refer to the same object or not. Let us take an example to understand this concept.

Example

```
str1 = "Brijesh"
str2 = "Brijesh"
str3 = str1

print (str1 is not str1)
print (str1 is not str2)
print (str1 is not str3)

print (str1 is str1)
print (str1 is str2)
print (str1 is str3)
```

Output:

```
False
False
False
True
True
True
```

User-defined function

We can make user-defined functions for string comparison. A user-defined function will compare the strings based upon the number of digits. Let's take an example

Example

```
def str_cmp(str1, str2):    #comparison of string based on the
number of digits
count1 = 0
count2 = 0

for i in range(len(str1)):
if str1[i] >= "0" and str1[i] <= "9":
count1 += 1

for i in range(len(str2)):
if str2[i] >= "0" and str2[i] <= "9":
count2 += 1
return count1 == count2

print (str_cmp("234", "5672"))
print  (str_cmp("1246",  "Brijesh"))
print  (str_cmp("22Brijesh",  "Brijesh22"))
```

Output:

```
False
False
True
```

Iterating strings

Iterating over a string means accessing each of its characters one at a time. There are various ways to iterate over the characters of a string in Python. Let's take an example.

Example

```
# Using simple iteration and range()
str1 = "Brijesh"

for var in str1:
print (var, end=' ')
print ("\n")

str2 = "Bakariya"

for i in range(0, len(str2)):  # Iterate over index
print (str2[i])
```

Output:

```
B r i j e s h

B
a
k
a
r
i
y
a
```

Example

```
#Using enumerate() function
str3 = "Brijesh"

# Iterate over the string
for i, v in enumerate(str3):
print (v)
```

```
B
r
i
j
```

```
e
s
h
```

Example 5.23:

```
#Using enumerate() function
str3 = "Brijesh"

# Iterate over the string
for i, v in enumerate(str3):
print (v)
```

Output:

```
B
r
i
j
e
s
h
```

In Python, we can also iterate over the words of a string. A string of several words separated by spaces is given. In

Python, there are many methods for iterating over words in a string.

Using split()

We can split the string into a list of words, but this method fails if the string contains punctuation marks. Let us take an example.

Example

```
str1 = "Python is a good prog., Python is easy to use, Python
is easy to understand"
print ("The original string is: " + str1)

result = str1.split()    # using split() to extract words from
string

print ("\nThe words of string are")
for i in result:
print (i)
```

Output:

```
The original string is: Python is a good prog., Python is easy
to use, Python is easy to understand

The words of string are
Python
is
a
good
prog.,
Python
is
easy
to
use,
Python
is
easy
to

understand
```

Using re.findall()

This approach necessitates the use of regular expressions to complete the job. After filtering the string and extracting words while ignoring punctuation marks, the **findall()** function returns the list. Let's look at an example.

Example

```
#findall() example
import re

str1= "Python is a good prog., Python is easy to use, Python
is easy to understand"
print ("The original string is: " + str1)

result = re.findall(r'\w+', str1)

print ("\nThe words of string are")
for i in result:
print (i)
```

Output:

```
The original string is: Python is a good prog., Python is easy
to use, Python is easy to understand
The words of string are Python
is
a
good
prog
Python
is
easy
to
use
Python

is
easy
to
understand
```

The string module

The string module is a built-in module for using classes and constants. First of all, we have to import it so that it can be used. Let us take an example.

String module constants

There are various types of string constants defined in this module are:

The following constants are concatenated: **ascii_lowercase** and

The lowercase letters This value is not locale-dependent and will not change.

The uppercase letters This value is not locale-dependent and will not change.

The string

The string

String of ASCII characters which are considered punctuation characters in the C locale:

Example 5.26:

```
# import string library function
import string

# Storing the value in variable result variables
result1 = string.ascii_letters
result2 =string.ascii_lowercase
result3 =string.ascii_uppercase
result4 =string.digits
result5 =string.hexdigits
result6 =string.punctuation
# Printing the value
print (result1)

print (result2)
print (result3)
print (result4)
print (result5)
print (result6)
```

Output:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
0123456789abcdefABCDEF
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

String capwords() function

This function splits the specified string into words and capitalizes each word in it.

Example

```
# import string library function
import string
str= 'My name is Dr. Brijesh Bakariya'
result= string.capwords(str)
print (result)
```

Output:

```
My Name Is Dr. Brijesh Bakariya
```

String module classes

There are two types of string module classes:

Formatter

Template

Formatter

This class becomes useful if we want to subclass it and define your format string syntax. Moreover, it is same as **str.format()** function. Let us take an example to understand the concept of **Formatter** class.

Example

```
from string import Formatter

formatter = Formatter()
print (formatter.format('{Faculty}', Faculty='Dr. Brijesh Bakariya'))
print (formatter.format('{} {Faculty}', 'Hello', Faculty='Dr. Brijesh Bakariya'))

print ('{} {Faculty}'.format('Hello', Faculty='Dr. Brijesh Bakariya'))
# it is same as format()
```

Output:

Dr. Brijesh Bakariya

Hello Dr. Brijesh Bakariya

Template

The **Template** class enables us to construct output specification syntax that is easier to understand. The format combines **$** with valid Python identifiers, such as alphanumeric characters and underscores, to create placeholder names. Let's look at an example to understand the idea of the **Template** class.

Example

```
# template example
from string import Template

# Creating a template
temp = Template('P is $P')

# Substitute value
print (temp.substitute({'P' : 1}))

# List of faculty stores the name and their id
Faculty = [('Dr. Brijesh Bakariya',11), ('Dr.Krishna K. Mohbey
',22), ('Dr. Amit Kumar',33)]
```

```
temp = Template('Hello $name, $id is your employee id')

for var in Faculty:
print (temp.substitute(name = var[0], id = var[1]))
```

Output:

```
P is 1
Hello Dr. Brijesh Bakariya, 11 is your employee id
Hello Dr.Krishna K. Mohbey, 22 is your employee id
Hello Dr. Amit Kumar, 33 is your employee id
```

Regular expression

A regular expression is a special sequence of characters that uses a specialized syntax to help you fit or locate other strings or sets of strings. It also defines a set of strings (pattern) that corresponds to it. The following metacharacters are available for use in functions:

Metacharacters It used to drop the special meaning of a character.

Metacharacters It represents a character class.

Metacharacters It matches the beginning.

Metacharacters It matches the end.

Metacharacters It matches any character except newline.

Metacharacter It matches zero or one occurrence.

Metacharacters It matches any of the characters separated by it.

Metacharacters It shows any number of occurrences (including 0 occurrences)

Metacharacters It is for one or more occurrences.

Metacharacters It indicates several occurrences of a preceding regular expression to match.

Metacharacters It encloses a group of regular expressions.

Let us take an example to understand the concept of regular expression.

Example

```
import re

p = re.compile('[a-m]')


print (p.findall("My name is Brijesh"))
```

Output:

```
['a', 'm', 'e', 'i', 'i', 'j', 'e', 'h']
```

The metacharacter plays a crucial function in signaling different sequences. Following are the uses of metacharacter which is shown in *table*

Table 5.2: *Metacharacters along with backslash (\)*

Example

```
import re

p1 = re.compile('\d')   # \d is equivalent to [0-9].
print (p1.findall("My age is 36 year and my year of Birth is
1985"))



p2 = re.compile('\d+')   # \d+ will match a group on [0-9],
group of one or greater size
```

```
print (p2.findall("My age is 36 year and my year of Birth is
1985"))

p3 = re.compile('\w')     # \w is equivalent to [a-zA-Z0-9_].
print (p3.findall("Welcome to the world of python. *"))

p4 = re.compile('\w+')   # \w+ matches to group of
alphanumeric character.
print (p4.findall("My age is 36 year and my year of Birth is
1985."))

p5 = re.compile('\W')     # \W matches to non alphanumeric
characters.
print (p5.findall("Welcome to the world of python. *"))

p6 = re.compile('br*')   # '*' replaces the no. of occurrence of
a character.
print (p6.findall("brbrbrbrbrbrbrbbbrrrrrr"))
```

Output:

```
['3', '6', '1', '9', '8', '5']
['36', '1985']
['W', 'e', 'l', 'c', 'o', 'm', 'e', 't', 'o', 't', 'h', 'e', 'w', 'o', 'r', 'l', 'd',
'o', 'f', 'p', 'y', 't', 'h', 'o', 'n']
```

```
['My', 'age', 'is', '36', 'year', 'and', 'my', 'year', 'of', 'Birth', 'is',
'1985']
[' ', ' ', ' ', ' ', ' ', '.', ' ', '*']
['br', 'br', 'br', 'br', 'br', 'br', 'br', 'b', 'b', 'brrrrrr']
```

Example 5.32:

```
import re

# Upon matching, 'y' is replaced by '!!' and CASE has been
ignored

print (re.sub('y', '~*', 'My age is 36 year and my year of Birth
is 1985', flags = re.IGNORECASE))

# Consider the Case Sensitivity, 'y' in "year", will not be
replaced.
print (re.sub('y', '!!', 'My age is 36 year and my year of Birth is
1985'))

# As count has been given value 1, the maximum time
replacement occurs is 1
print (re.sub('y', '!!', 'My age is 36 year and my year of Birth is
1985', count=1, flags = re.IGNORECASE))
```

```
# 'r' before the patter denotes RE, \s is for start and end of
a String.
print (re.sub(r'\sAND\s', ' & ', 'Book and note book',
flags=re.IGNORECASE))
```

Output:

```
M~* age is 36 ~*ear and m~* ~*ear of Birth is 1985
M!! age is 36 !!ear and m!! !!ear of Birth is 1985
M!! age is 36 year and my year of Birth is 1985
Book & note book
```

Conclusion

This chapter discussed strings and their various operations like concatenating, appending, multiplying, etc. It is concerned about the string formatting operator and its working procedure. It also discussed some built-in string functions and other operations like comparing and iterating a string, further explaining regular expression with a good example. In the next chapter, we will discuss lists as the next topic used in Python.

Points to remember

A string can be created by enclosing characters in quotes (single, double, or triple).

An immutable string means string value cannot be updated.

String uses % operator for string formatting operations.

The **ord()** function returns the value of a single Unicode character as an integer.

A regular expression is a special sequence of characters for matching purposes.

Multiple choice questions

Which of the following is False?

A string is immutable.

capitalize() function in a string is used to return a string by converting the whole string into uppercase.

lower() function in a string is used to return a string by converting the whole string into lowercase.

None of these.

Which of the following functions convert an integer to hexadecimal string in Python?

unichr(x)

ord(x)

hex(x)

oct(x)

What is the output of for x in [2, 3, 4]: print x?

x

2 3 4

Error

None of the above.

Suppose a list with name arr contains 5 elements. You can get the 2nd element from the list using:

arr[-2]

arr[2]

arr[-1]

arr[1]

Which of the following operations cannot be performed on a Python string?

Deletion of a Python string

Splitting of a Python string

Adding elements in a Python string

Reassigning of a Python string

Which string method is used to check if all the given characters in a Python string defined program are in lower case?

isnumeric()

islower()

isdigit()

islwer()

str1="8/2"

print("str1")

1

6/4

1.5

str1

str1="Information"
print(str1[2:8])

format

formatio

orma

ormat

Python has a built-in package called?

reg

regex

re

regx

Which function returns a list containing all matches?

findall

search

split

find

Which character stands for Zero or more occurrences in regex?

*

#

@

|

Answers

b

c

b

d

c

b

d

a

c

a

a

Questions

What is a string? Write a program to display a string.

What do you mean by string concatenation? Discuss all methods of string concatenation.

Discuss multiplying strings with suitable examples.

What do you mean by immutable strings? Explain with a suitable example.

Discuss string formatting operator with proper explanation.

Write any five built-in string functions with an explanation

Explain and **chr()** method with example.

Discuss all the methods for comparing and iterating the strings

What do you mean by regular expression? Explain with a suitable example.

CHAPTER 6

Lists

The list datatype in Python is the most powerful, and it can be written as a list of comma-separated values between square brackets. A list is often used to store the sequence of different types of data. The list is mutable, which means it can modify its element after it is created. This chapter covers multiple operations that can be applied on lists.

Structure

In this chapter, we will cover the following topics:

Use the concept of lists.

Use the various type of list operations

Use list functions

Use list processing

Use of different sorting in list

Objective

This chapter's goal is to introduce the lists in Python. After completing this chapter, you should be able to perform various list-related operations. Also, you will be able to write Python codes using the list. This chapter covers various types of list sorting, such as bubble sort, selection sort, insertion sort, quick sort, merge sort, and so on. You will also understand the uses of various list functions in Python.

Basics of lists

List is a very important data type in Python, and it is written as a comma-separated values enclosed with square brackets. The following is the example of lists:

```
list1 = ['Brijesh', 'Krishna', 36, 35];
list2 = [1, 2, 3, 4, 5];
list3 = ["a", "b", "c", "d"]
```

There are the following important points about lists:

List items are ordered, and that order will not change except for some list methods.

The list allows duplicate values.

The list is changeable, which means we can add and delete items in a list after it has been created.

The first item has index [0], and indexing will increase.

Let us take a basic example to understand the basic concept of lists.

Example 6.1:

```
# Basics of lists
l1 = ['Brijesh', 'Krishna', 36, 35];
l2 = [1, 2, 3, 4, 5];
l3 = ["a", "b", "c", "d"]
print (l1)
print (l2)
print (l3)

#List allow duplicates values
dl = ["Brijesh", "Krishna", "Krishna", "Krishna", "Brijesh",
"Misthi"]
print (dl)
```

Output:

```
['Brijesh', 'Krishna', 36, 35]
[1, 2, 3, 4, 5]
['a', 'b', 'c', 'd']
['Brijesh', 'Krishna', 'Krishna', 'Krishna', 'Brijesh', 'Misthi']
```

Example 6.2:

```
l1 = ['Brijesh', 'Krishna', 36, 35];

print (type(l1))

lc = list (('Brijesh', 'Krishna', 36, 35))
print (lc)
```

Output:

```
'list'>
['Brijesh', 'Krishna', 36, 35]
```

Creating lists

It can make any list by putting the series in square brackets **[]** and separating it with commas. It may contain any number of items of various kinds, such as integers, floats, strings, and so on. Let us take a basic example.

Example 6.3:

```
# empty list
l1 = []
# list of integers
l2 = [4, 5, 6, 7]
# list with mixed data types
l3 = [2, "Dr. Brijesh Bakariya", "Dr. Krishna K. Mohbey", 88.9]
print (l1)
print (l2)
print (l3)
```

Output:

```
[]
[4, 5, 6, 7]
```

[2, 'Dr. Brijesh Bakariya', 'Dr. Krishna K. Mohbey', 88.9]

Accessing values from a list

Every value in a list is indexed, and it can access them by referring to the index number. There are various ways to access a list of elements.

Using list index

It can access a list object using the index operator The indexes begin from 0. Suppose the list has seven elements, then it starts from 0 to 6. If you want to access out of index values, it will raise an One of the essential things to remember an index must be an integer value means to float, and other types are not allowed. If you put another data type in the index value, then it will raise Let us take an example to understand this concept.

Example

```
# List indexing
ListExample = ['b', 'r', 'i', 'j', 'e', 's', 'h']
print (ListExample [0])
print (ListExample [1])
print (ListExample [2])
print (ListExample [3])
print (ListExample [4])
print (ListExample [5])
print (ListExample [6])
```

Output:

```
b
r
i
j
e
s
h
```

Negative indexing

Negative indexing is also allowed for its sequences. In negative indexing, -1 refers to the last item, -2 to the second last, -3 to the third last, and so on. The following figure 6.1 shows the negative indexing in Python.

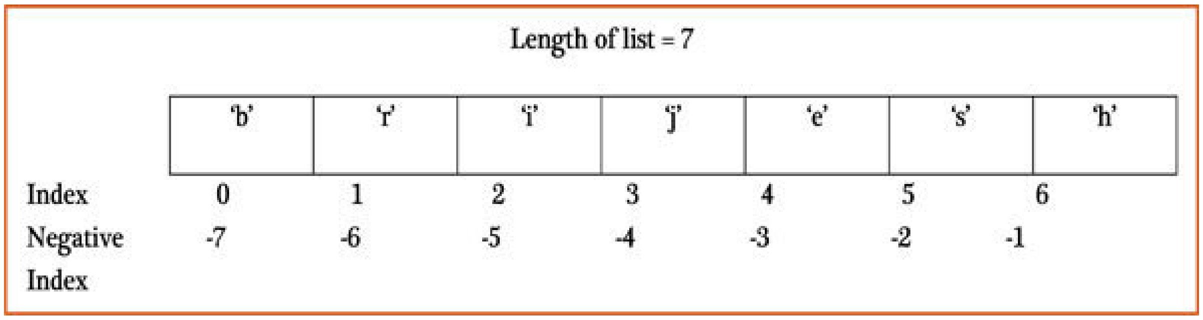


Figure 6.1: *Negative indexing in Python.*

Let us take an example to understand this concept.

Example 6.5:

```
# Example of negative list
ListExample = ['b','r','i','j','e','s','h']
```

```
print (ListExample [-1]) # Last index
print (ListExample [-2])
print (ListExample [-3])
print (ListExample [-4])
print (ListExample [-5])
print (ListExample [-6])
print (ListExample [-7]) # First Index
```

Output:

```
h
s
e
j
i
r

b
```

Example 6.6:

```
list1 = ['Brijesh', 'Krishna', 36, 35]
print (list1[-1])
print (list1[-2])
print (list1[-3])
print (list1[-4])
```

Output:

```
35
36
Krishna
Brijesh
```

To slice, delete, remove, and clear a list of elements

It can be accessed through the range of items in a list by using the slicing operator. It is represented by colon Adding and changing an element in the list is possible using the assignment operator The keyword **del** can be used to remove one or more items from a list. It can completely delete the list. The **delete()** method can be used to remove a specific object, while the **pop()** method can be used to remove an item at a specific index. To empty a list, we can use the **clear()** method. Let's look at an example to better understand the operators and methods described.

Example 6.7:

```
ml = ['b', 'r', 'i', 'j', 'e', 's', 'h']
print(ml[2:5]) #elements 3rd to 5th
print(ml[:-6]) #elements beginning to 5th
print(ml[3:]) #elements 4th to end
print(ml[:]) #elements beginning to end

ml = ['b', 'r', 'i', 'j', 'e', 's', 'h']
```

```
del ml[2] # delete one item
print(ml)
del ml[1:5] # delete multiple items
print (ml)

mlist = ['b', 'r', 'i', 'j', 'e', 's', 'h']
mlist.remove('r')
print (mlist)

print (mlist.pop(1))
print (mlist)

print (mlist.pop())
print (mlist)



mlist.clear()
print (mlist)

del ml # delete entire list
print (ml) # Error: List not defined
```

Output:

```
['i', 'j', 'e']
['b']
```

```
['j', 'e', 's', 'h']
['b', 'r', 'i', 'j', 'e', 's', 'h']
['b', 'r', 'j', 'e', 's', 'h']
['b', 'h']
['b', 'i', 'j', 'e', 's', 'h']
i
['b', 'j', 'e', 's', 'h']
h
['b', 'j', 'e', 's']
[]
Traceback (most recent call last):
File "C:\Users\brije\.spyder-py3\temp.py", line 31, in
print(ml) # Error: List not defined
NameError: name 'ml' is not defined
```

Updating values in Lists

We can update single or multiple lists. It can use assignment operator, append and extend the method for accomplishing the task. Let us take an example to understand the mentioned operators and methods

Example 6.8:

```
# Add/Change List Elements
var = [1, 2, 3, 4]
var[0] = 9 # change the 1st item
print (var)

var[1:4] = [22, 33, 44] # change 2nd to 4th items
print (var)

# Appending and Extending lists in Python using the append()
var = [11, 22, 33]
var.append(44)
print (var)
```

```
var.extend([55, 66, 77]) #Add several items using extend()
method
print (var)

print (var + [224,353,464]) # Add + operator to combine two
lists.
```

Output:

```
[9, 2, 3, 4]
[9, 22, 33, 44]
[11, 22, 33, 44]
[11, 22, 33, 44, 55, 66, 77]
[11, 22, 33, 44, 55, 66, 77, 224, 353, 464]
```

Nested list

A nested list means a list within a list. Moreover, a list contains a sublist of elements. Suppose we have a list named **lst** then it includes a list and sublist. A nested list is made up of a series of sublists separated by commas.

```
lst = ['ab', ['cd', ['efg', 'hij'], 'kl', 'mn'], 'o', 'p']
```

The following *figure 6.2* shows the hierarchical structures for understanding above example:

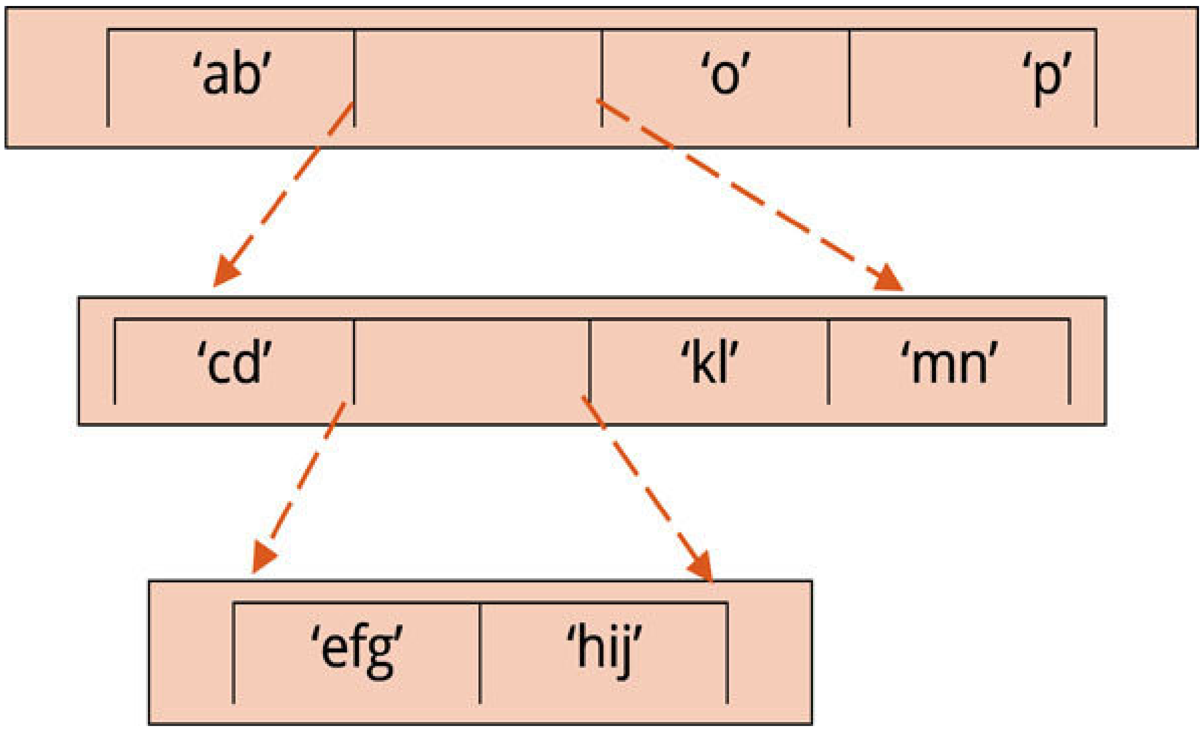


Figure 6.2: *Nested list in Python.*

Example 6.9:

```
lst = ['ab', ['cd', ['efg', 'hij'], 'kl', 'mn'], 'o', 'p']

print (L[2]) # Prints' o'

print (L[2][2]) # Prints' kl'

print (L[2][2][0]) # Prints
```

Output:

```
o
kl
k
```

It can add and remove items in a nested list by using **remove()** method. It can also find length using **len()** and iterate through a nested list by using **for** loop. Let us take an example to understand this concept.

Example 6.10:

```
lst = ['b', ['ri', 'je'], 'sh']
lst[1][1] = 0
print (lst)

lst[1].append('BB')
print (lst)

lst[1].insert(0,'PP')
print (lst)

lst[1].extend([77,88,99])
print (lst)
```

Output:

```
['b', ['ri', 0], 'sh']
['b', ['ri', 0, 'BB'], 'sh']
['b', ['PP', 'ri', 0, 'BB'], 'sh']
['b', ['PP', 'ri', 0, 'BB', 77, 88, 99], 'sh']
```

Aliasing

Variables are objects, and when we assign one to another, all variables refer to the same object. Moreover, when two identifiers refer to the same variable, then it is called aliasing. Let us take an example to understand this concept.

Example 6.11:

```
var1 = [555, 666,777]
var2 = var1 #Aliasing
print (var1 is var2)
print (var1)
print (var2)
```

Output:

```
[True
[555, 666, 777]
[555, 666, 777]
```

The same list has two different names in the above example, **var1** and however, after executing the assignment statement

var2 = you can see that **var1** and **var2** relate to the same list. This indicates that it is aliasing.

Example 6.12:

```
var1 = [555, 666,777]
var2 = [555, 666,777]
print (var1)
print (var2)
print (var1 == var2)
print (var1 is var2)
var2 = var1
print (var1 == var2)
print (var1 is var2)



var2[0] = 999
print (var1)
print (var2)
```

Output:

```
[555, 666, 777]
[555, 666, 777]
True
False
```

```
True
True
[999, 666, 777]
[999, 666, 777]
```

Cloning lists

If we want to change a list while keeping a copy of the original, we must copy the entire list, not just the reference. To prevent the ambiguity of the word clone, this method is often referred to as cloning. The quickest and simplest way to clone a list is to use the slicing process.

Example

```
list1 = [777, 888, 999]
list2 = list1[:] # make a clone using slice
print (list1 == list2)
print (list1 is list2)
list2[0] = 555
print (list1)
print (list2)
```

Output:

```
True
False
[777, 888, 999]
```

```
[555, 888, 999]
```

There are various types of ways to cloning a list.

Using slicing

It is the most straightforward method of cloning a list. When we want to change a list while keeping a copy of the original, we use this form. Let us take an example to understand this concept.

Example

```
def cloning_List(li1):
lst1 = li1[:] #Slice Operator
return lst1

lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Using the extend() method

The **extend()** function can be used to copy the lists into a new list. Let us take an example to understand this concept.

Example

```
def cloning_List(li1):
lst1 = []
lst1.extend(li1)
return lst1

lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Using the list() method

Using the built-in function list, it's also the simplest way to clone a **list** Let us take an example to understand this concept.

Example

```
def cloning_List(li1):
lst1 = list(li1)
return lst1

lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Using list comprehension

To copy all the elements from one list to another, a list comprehension method is used. Let us take an example to understand this concept.

Example

```
def cloning_List(li1):
lst1 = [i for i in li1]
return lst1

lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Using the append() method

The **append()** method can be used to append or add elements to the list or copy them to a new list. It's used to add elements to the list's last location. Let us take an example to understand this concept.

Example 6.18:

```
def cloning_List(li1):
lst1 = []
for i in li1: lst1.append(i)
return lst1

lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Using the copy() method

To copy all the elements from one list to another, use this method. Let us take an example to understand this concept.

Example

```
def cloning_List(li1):
lst1 =[]
lst1 = li1.copy()
return lst1

lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

List parameters

The **list()** constructor takes a single argument; this argument could be a sequence like string, tuples, set, dictionary, or an iterator object. It can also pass a list as an argument. Let us take an example to understand this concept.

Example

```
name_list = ['Brijesh', 'Krishna', 'Ravi', 'Ram', 'Misthi']
print (list(name_list))

def list_Fun(l):
print (l)

name_list = ['Brijesh', 'Krishna', 'Ravi', 'Ram', 'Misthi']
list_Fun(name_list)
```

Output:

```
['Brijesh', 'Krishna', 'Ravi', 'Ram', 'Misthi']
['Brijesh', 'Krishna', 'Ravi', 'Ram', 'Misthi']
```

Basic list operations

Generally, list support + and * operator for list concatenation and list repetition. But there are other list methods to do the same task. In the previous example, we have also used the slice, append, and extend method performed on a string. Let us take an example to understand this concept.

Example

```
lst1 = ['b', 'r', 'i', 'j', 'e', 's', 'h']
print (lst1 + ['p','q','r']) # A + operator to combine two lists.
print (2*lst1) # A * operator to repetition.

print (lst1[1:3]) #elements 1st to 3rd

# Appending and Extending lists in Python using the append()

lst1.append('bakariya')
print (lst1)

lst1.extend(['x','y','z']) #Add several items using extend() method
```

print (lst1)

Output:

```
['b', 'r', 'i', 'j', 'e', 's', 'h', 'p', 'q', 'r']
['b', 'r', 'i', 'j', 'e', 's', 'h', 'b', 'r', 'i', 'j', 'e', 's', 'h']
['r', 'i']
['b', 'r', 'i', 'j', 'e', 's', 'h', 'bakariya']
['b', 'r', 'i', 'j', 'e', 's', 'h', 'bakariya', 'x', 'y', 'z']
```

There are various types of built-in functions which are performed on the list. We can easily accomplish our tasks by directly using these functions.

The built-in list functions:

cmp(list1, This function compares elements of both lists.

This function returns the length of the list.

This function returns an element from the list with max value.

This function returns an element from the list with min value.

This function converts a tuple into the list.

Let us take an example to understand the above built-in list functions.

Example

```
list1 = ['Brijesh', 'Krishna']
list2 = ['Ramesh', 'Rajesh']

def cmp(p, q):
return (p > q) - (p < q)

print (cmp(list1, list2)) # List comparison

print (len(list1)) # Length of list

list3 = [53535, 68868, 67656]
print ("Maximum value element : ", max(list3)) # Getting Max
value from list

print ("Minimum value element : ", min(list3)) # Getting Min
value from list
tuple_var = ('Brijesh', 'Krishna', 123, 'xyz');
print ("Tuple elements : ", tuple_var)
list_var = list(tuple_var) # Converts a tuple into list.
```

```
print ("List elements : ", list_var)
```

Output:

```
-1
2
Maximum value element : 68868
Minimum value element : 53535

Tuple elements : ('Brijesh', 'Krishna', 123, 'xyz')
List elements : ['Brijesh', 'Krishna', 123, 'xyz']
```

List methods

Some other built-in methods are also performed on the list. We can easily accomplish our task by directly using this method.

The built-in list methods:

This method appends object **obj** to the list.

This method returns a count of how many times **obj** occurs in the list.

This method appends the contents of **seq** to the list.

This method returns the lowest index in the list that **obj** appears.

list.insert(index, This method inserts object **obj** into the list at offset index.

This method removes and returns the last object or **obj** from the list.

This method removes object **obj** from the list.

This method reverses objects of the list in place.

This method sorts objects of the list.

Let us take an example to understand the above built-in list methods.

Example 6.23:

```
lst1 = ['b', 'r', 'i', 'j', 'e', 's', 'h']

lst1.append('bakariya')

print (lst1)

lst1.extend(['x','y','z'])
print (lst1)



print ("Count : ", lst1.count('bakariya'))
print ("Index for bakariya : ", lst1.index('bakariya'))

lst1.insert(3, 100)
print (lst1)

print ("Poped List : ", lst1.pop())

lst1.remove('bakariya')
print (lst1)

print (lst1.reverse())

lst2 = [44,464,686,313,668,242,575,1212,464]

lst2.sort()
print (lst2)

print (lst2.reverse())
print (lst2)
```

Output:

```
['b', 'r', 'i', 'j', 'e', 's', 'h', 'bakariya']
['b', 'r', 'i', 'j', 'e', 's', 'h', 'bakariya', 'x', 'y', 'z']
```

```
Count : 1
Index for bakariya : 7
['b', 'r', 'i', 100, 'j', 'e', 's', 'h', 'bakariya', 'x', 'y', 'z']
Poped List : z
['b', 'r', 'i', 100, 'j', 'e', 's', 'h', 'x', 'y']
None
[44, 242, 313, 464, 464, 575, 668, 686, 1212]
None
[1212, 686, 668, 575, 464, 464, 313, 242, 44]
```

List as arrays

We know about the list; it is a collection of items containing elements of multiple data types, which may be either numeric, logical character values, etc. It is also known as an array because it is used in various programming languages like C, C++, Java, etc. An array is a collection of homogeneous (same type) elements. If the array elements belong to different data types, an exception (incompatible data types) is thrown. Let us take an example to understand this concept.

Example 6.24:

```
# Simple list
list1 = ['Dr. Brijesh Bakariya','Dr. Krishna K. Mohbey', 36, 35]
print (list)

# Using array in Python

import array

sa = array.array('i', [33, 55, 77])
```

```
# accessing elements of array
for i in sa:
print (i)
```

Output:

```
'list'>
33
55
77
```

There are a few differences between list and array in Python:

A list can consist of elements belonging to different data types, but an array only consists of elements belonging to the same data type.

A list does not need to import a module for declaration explicitly, but an array needs to import a module for declaration explicitly.

A list cannot directly handle arithmetic operations, but an array can directly manage arithmetic operations

A list can be nested to contain different elements, but an array must contain either all nested elements of the same size.

A list is preferred for a shorter sequence of data items, but an array is preferred for a longer sequence of data items.

A modification is easier in the list compared to an array.

The entire list can be printed without any explicit looping but array using a loop to be formed to print or access the components.

Looping in lists

Looping in lists means accessing all the elements in a list one by one. Moreover, it is an iteration through a list in Python. There are various ways to iterate through a list in Python.

Using for loop

A **for** loop is used for showing all the elements in a list. Let us take an example to understand this concept.

Example 6.25:

```
list = [11, 22, 33, 44, 55]

for var in list:
print (var)
```

Output:

```
11
22
33
44
55
```

Using loop and range()

Suppose we want to use the traditional **for** loop, which iterates the element till the range specified. Let us take an example to understand this concept.

Example 6.26:

```
list = [11, 22, 33, 44, 55]

length = len(list) # getting the length of a list

for var in range(length):
print (list[var])
```

Output:

```
11
22
33
44
55
```

Using while loop

It is similar to **for** loop, but its syntax is different. Let us take an example to understand this concept. The following example prints all the list values using a **while** loop.

Example

```
list = [11, 22, 33, 44, 55]

length = len(list) # getting length of list
x1 = 0
while x1 < length:
print (list[x1])
X1 += 1
```

Output:

```
11
22
33
44
55
```

Using list comprehension

It is used to iterate through a list in Python. It is another way of generating a list of elements that possess a specific property. Let us take an example to understand this concept.

Example

```
list = [11, 22, 33, 44, 55]

[print (i) for i in list] # Using list comprehension
```

Output:

```
11
22
33
44
55
```

Using enumerate()

This method can be used when we want to convert the list into an iterable list of tuples. Let us take an example to understand this concept.

Example

```
list = [11, 22, 33, 44, 55]

for i, val in enumerate(list): # Using enumerate()
print (i, "=",val)
```

Output:

```
0 = 11
1 = 22
2 = 33
3 = 44
4 = 55
```

Using Numpy

For huge n-dimensional lists, it is sometimes better to use an external library such as NumPy. Let us take an example to understand this concept.

Example

```
import numpy as np

var = np.arange(4)
var = var.reshape(2, 2)

for var1 in np.nditer(var): # iteration
print (var1)
```

Output:

```
0
1
2
3
```

Passing list to function

Passing a list to function means to pass a list variable to a function, and the function copy the value of the list variable to function parameters. So it is easy to show all the values of the list through the function parameters. Let us take an example to understand this concept.

Example

```
def faculty_name(f):
for i in f:
print (i)

faculty = ["Dr. Brijesh", "Dr. Krishna", "Dr. Amit", "Dr. Raj"]
faculty_name(faculty)
```

Output:

```
Dr. Brijesh
Dr. Krishna
Dr. Amit
Dr. Raj
```

Returning list from the function

Returning list to function means to return the value of the list and catch to the calling function. It is assigned a calling function to another variable for printing purposes. Let us take an example to understand this concept.

Example

```
def faculty_name(f):
for i in f:
return(i)

faculty = ["Dr. Brijesh", "Dr. Krishna", "Dr. Amit", "Dr. Raj"]

for i in faculty:
faculty_name(i)
print (i)
```

Output:

Dr. Brijesh

Dr. Krishna
Dr. Amit
Dr. Raj

filter() function

A **filter()** function filters an iterable using a function that checks if each element in the iterable is valid or not. A **filter()** function returns an iterator that has passed and part of the iterable through the function check.

filter(function, iterable)

There are two parameters of the **filter()** function as a function and iterable.

It returns true if elements are iterable, false otherwise.

It means that the iterable to be filtered could be anything from sets to lists to tuples to containers of any iterators.

Let us take an example to understand this concept.

Example 6.33:

```
# Program to filter the list, and return a new list with only
the values equal to or above 18:
agesPerson = [16, 17, 23, 35, 19, 38]

def my_fun(n):
if n < 18:
return False
else:
return True

adults = filter(my_fun, agesPerson)

for i in adults:
print (i)
```

Output:

```
23


35
19
38
```

Example 6.34:

```
# function that filters vowels
```

```
def fun_vowel(n):
vow = ['a', 'e', 'i', 'o', 'u']
if (n in vow):
return True
else:
return False

# sequence
seq = ['b', 'r', 'i', 'j', 'e', 's', 'h', 'b', 'a', 'k', 'a', 'r', 'i', 'y', 'a']

fil= filter(fun_vowel, seq) # using filter function
print ('The filtered letters are:')
for i in fil:
print (i)
```

Output:

```
The filtered letters are:
i
e
a
a
i
a
```

map() function

For each item in an iterable, this function executes a given function. As a parameter, the item is passed into the function. Furthermore, after applying the defined function to each item of a given iterable like a list, tuple, or other iterable, the **map()** function returns a map object of the results.

Syntax:

```
map(function, iterables)
```

The **map()** function's first parameter executes for each component, and the second parameter iterable requires a sequence, collection, or iterator object. You can submit as many iterables as you want, as long as each one has its parameter in the function. Let us take an example to understand this concept.

Example

```
def my_function(x, y):
return x + y
```

```
m = map(my_function, ('Brijesh', 'Krishna', 'Ramesh'), ('Amit',
'Sumit', 'Mohit'))
print (m)

# Add two lists using map and lambda
n1 = [11, 22, 33]
n2 = [44, 54, 66]

res = map(lambda a, b: a + b, n1, n2)
print (list(res))

# List of strings
lst = ['Brijesh', 'Krishna', 'Ramesh']

m = list(map(list, lst))
print (m)
```

Output:

```
object at 0x000001EA94A9B108>
[55, 76, 99]
[['B', 'r', 'i', 'j', 'e', 's', 'h'], ['K', 'r', 'i', 's', 'h', 'n', 'a'], ['R', 'a',
'm', 'e', 's', 'h']]
```

reduce() function

The **reduce()** function is defined in the **functools** module, and it takes two arguments: one is a function, and the other is an iterable. One important thing which you have to remember the **reduce()** function does not return another iterable. Instead, it returns a single value. The **reduce()** is useful when you need to apply a function to iterable and reduce it to a single cumulative value. Let us take an example to understand this concept.

Syntax:

```
functools.reduce(myfunction, iterable, initializer)
```

Example 6.36:

```
import functools
def multiply_fun(a,b):
print ("a=",a," b=",b)
return a*b

factorial_var = functools.reduce(multiply_fun, range(1, 6))
```

```
print ('Factorial of 5: ', factorial_var)
```

Output:

```
a= 1 b= 2
a= 2 b= 3
a= 6 b= 4
a= 24 b= 5
Factorial of 5: 120
```

The multiply function is defined to return the product of two numbers in the example above. This function is used in the **reduce()** function along with a number range of 1, 2, 3, 4, and 5 between 1 and 6. The result is a 5 factorial value.

Example 6.37:

```
import functools

lst1 = [23, 76, 34, 87, 59, 66, 324]

print ("list elements sum : ",end="")
print (functools.reduce(lambda x,y : x+y,lst1))

print ("max from list: ",end="")
print (functools.reduce(lambda x,y : x if x > y else y,lst1))
```

Output:

```
list elements sum : 669
max from list : 324
```

Searching in list

It means searching whether your element is present in the list or not. Moreover, it simply checks if a list contains a particular item. We use operator for accomplishing this task. Let us take an example to understand this concept.

Example 6.38:

```
names = ['Brijesh', 'Krishna', 'Ramesh','Amit', 'Sumit', 'Mohit']

if 'Amit' in names:
print('Amit Found')

names = ['Brijesh', 'Krishna', 'Ramesh','Amit', 'Sumit', 'Mohit']
amit_pos = names.index('Amit') #index of an item
print (amit_pos)
```

Output:

```
Amit Found
3
```

Example 6.39:

```
flowerList = ['lotus', 'rose', 'sunflower', 'lily', 'bluebell']

flowerName = input("Enter a flower name:") # input from user

if flowerName.lower() in flowerList:

print ("%s is found in the list" %(flowerName))
else:
print ("%s is not found in the list" %(flowerName))
```

Output:

```
Enter a flower name:lily
lily is found in the list
```

Sequential search

It is the process of searching an element sequentially. It is also called a linear search. There are the following processes of linear search

Begin with the list's leftmost element and compare **x** to each of the list's elements one by one.

Return **True** if **x** matches an element.

Return **False** if **x** does not match any of the elements.

Let us take an example to understand this concept.

Example

```
def lin_search(list,n):

for var1 in range(len(list)):
if list[var1] == n:
return True
return False

list1 = [54, 99, 'Brijesh', 335,'Krishna', 643]

n = 'Krishna'

if lin_search(list1, n):
print ("It is Found")
else:
print ("It is Not Found")
```

Output:

```
It is Found
```

Binary search

The binary search technique is to find search elements from a list of elements. There are the following properties of binary search.

It works only for a sorted sequence of an element.

The first searching process starts by comparing the search element with the middle element in the list.

If the search element is smaller than the middle element, it will search the left side of the list and repeat it.

If the search element is greater than the middle element, it will search the right side of the list and repeat it.

The binary search process is implemented using recursion.

If an element is search middle, left, and right part of the list, then it displays the element found.

The worst-case time complexity is log n. It is efficient than a linear search algorithm.

There are the following steps for implementing a binary search

Start

Give a search element from the user

Sort the list of elements

In the sorted list, find the middle element.

Compare the search element to the sorted list's middle element.

If both elements match, show "Element Found" and exit.

Check if the search element is smaller or larger than the middle element if they don't match.

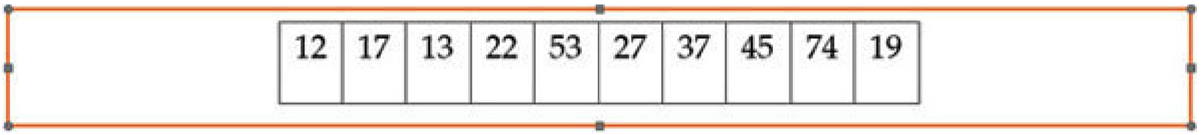
If the search element is smaller than the middle element, repeat steps 3–5 for the middle element's left sublist.

If the search element is larger than the middle element, repeat steps 3–5 for the middle element's right sublist.

Repeat the same steps until the search element is found in the list or the sublist contains only one element.

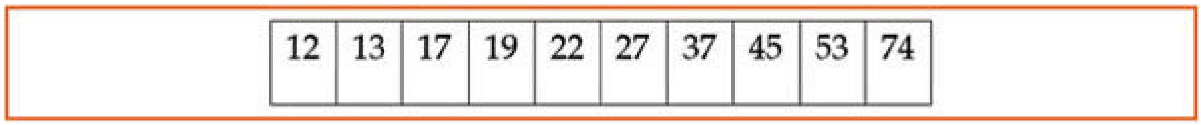
Show "Element not found" and stop if that element does not match the search element.

Let us take an example to understand the binary search process:



12	17	13	22	53	27	37	45	74	19

Figure 6.3: *Input linear List*



12	13	17	19	22	27	37	45	53	74

Figure 6.4: *Sorted linear List*

The size of the list is 10 then the middle element is 5. The element is 22. Suppose our search element is 19. Compare 19 to the middle element 22. If it is lesser, then it goes to the left part of the list. Again find the middle element from the left part of the list and follow this process until the element is searched. Let us take an example to understand this concept.

Example 6.41:

```
# Program for iterative Binary Search Function

def binary_search(list2, x):

beg = 0
end = len(list2) - 1
mid = 0
while beg <= end:
mid = (beg+ end) // 2

if list2[mid] < x:
beg = mid + 1

elif list2[mid] > x:
end = mid - 1

else:
return mid

return -1
```

```
# Test list
list1 = [2, 3, 4, 10, 40]
x = 40

# calling function
result = binary_search(list1, x)

if result != -1:
print ("Element is present at index", str(result))
else:
print ("Element is not present in list")
```

Output:

```
Element is present at index 4
```

Example 6.42:

```
# Program for recursive binary search
def binary_search(list2, low, high, x):

# Check base case
if high >= low:
```

```
mid = (high + low) // 2



if list2[mid] == x:
return mid


elif list2[mid] > x:
return binary_search(list2, low, mid - 1, x)


else:
return binary_search(list2, mid + 1, high, x)


else:
return -1 # Element is not present in the list


# Test list
list1 = [2, 3, 4, 10, 40]
x = 10


# Function call
result = binary_search(list1, 0, len(list1)-1, x)


if result != -1:
print ("Element is present at index", str(result))
else:
print ("Element is not present in list")
```

Output:

```
Element is present at index 3
```

Sorting

Sorting means arranging the data in either ascending or descending order. The primary task of sorting algorithms is to organize data elements in a particular order. If our data is properly sorted in order, then it would search the element fast. Moreover, sorting maintains the data structure. Sorting may also be used to display data in a more readable format. Sorting can be used in a variety of contexts, including the dictionary, telephone directory, and so on.

Python uses the **sort()** method for sorting elements of a given list in a specific ascending or descending order. The syntax of the **sort()** method is:

Syntax:

```
list.sort(key=..., reverse=...)
sorted(list, key=..., reverse=...)
```

An important point which we have to remember is the basic difference between **sort()** and **sorted()** is that the **sort()** method changes the list directly and doesn't return any value, while **sorted()** doesn't change the list and returns the sorted list.

In the **sort()** and **sorted()** methods, there are two parameters: **reverse** and The sorted list is reversed if the **reverse** parameter is true. A **key** parameter is used as a sort comparison key.

Let us take an example to understand this concept.

Example 6.43:

```
vowels_var = ['i', 'a', 'u', 'o', 'e'] # vowels list
vowels_var.sort() # sort the vowels
print ('Sorted list:', vowels_var) # print vowels

vowels_var = ['i', 'a', 'u', 'o', 'e'] # vowels list

vowels_var.sort(reverse=True) #Sort the list in Descending order
print ('Sorted list:', vowels_var) # print vowels

#Custom Sorting With key
list1 = ['ppp', 'aaaa', 'd', 'bb']
print (sorted(list1, key=len))

#Sort the list by the length of the values:
def my_sort_fun(n):
```

```
return len(n)

faculty = ['Brijesh', 'Krishna', 'Amit', 'Zubin']
faculty.sort(key=my_sort_fun)
```

Output:

```
Sorted list: ['a', 'e', 'i', 'o', 'u']
Sorted list: ['u', 'o', 'i', 'e', 'a']
['d', 'bb', 'ppp', 'aaaa']
```

Types of sorting

The following are the various types of sorting:

In-place In this sorting technique, the elements are sorted within a list. There is no extra space required for sorting an element.

Not-in-place In this sorting technique, the elements are sorted, but a different list is required. There is extra space needed for sorting elements.

Stable In a sorting algorithm, stable sorting occurs when the contents do not change the sequence of related content in which they appear after sorting.

Unstable After sorting, if the contents change the sequence of similar content in which they appear, it is called unstable sorting in a sorting algorithm.

Bubble sort

Bubble sort is a comparison-based algorithm that compares each pair of adjacent elements and swaps them if they are out of order. Since the average and worst-case complexity of this algorithm are O(n2), where n is the number of items, it is not suitable for large data sets. The bubble sorting procedure is as follows. To begin with, we'll make a list of an element that will hold the integer numbers.

list1 = [5, 3, 8, 6, 7, 2]

There are following iterations for sorting.

First iteration

[5, 3, 8, 6, 7, 2]

It compares the first two elements and here 5>3 then swap with each other. Now we get a new list that is [3, 5, 8, 6, 7, 2]

In the second comparison, 5 < 8, then swapping will happen [3, 5, 8, 6, 7, 2]

In the third comparison, 8>6, then swap [3, 5, 6, 8, 7, 2]

In the fourth comparison, 8>7, then swap [3, 5, 6, 7, 8, 2]

In the fifth comparison, 8>2, then swap [3, 5, 6, 7, 2, 8]

Here, the first iteration is complete and we get the largest element at the end. Now we need to the **len(list1)** –

Second iteration

[3, 5, 6, 7, 2, 8] - > [3, 5, 6, 7, 2, 8] here, 3<5 then no swap taken place.

[3, 5, 6, 7, 2, 8] - > [3, 5, 6, 7, 2, 8] here, 5<6 then no swap taken place.

[3, 5, 6, 7, 2, 8] - > [3, 5, 6, 7, 2, 8] here, 6<7 then no swap taken place.

[3, 5, 6, 7, 2, 8] - > [3, 5, 6, 2, 7, 8] here 7>2 then swap their position.

Now [3, 5, 6, 2, 7, 8] - > [3, 5, 6, 2, 7, 8] here 7<8 then no swap taken place.

Third iteration

[3, 5, 6, 2, 7, 8] - > [3, 5, 6, 7, 2, 8] here, 3<5 then no swap taken place

[3, 5, 6, 2, 7, 8] - > [3, 5, 6, 7, 2, 8] here, 5<6 then no swap taken place

[3, 5, 6, 2, 7, 8] - > [3, 5, 2, 6, 7, 8] here, 6<2 then swap their positions

[3, 5, 2, 6, 7, 8] - > [3, 5, 2, 6, 7, 8] here 6<7 then no swap taken place.

Now [3, 5, 2, 6, 7, 8] - > [3, 5, 2, 6, 7, 8] here 7<8 then swap their position.

It will iterate until the list is sorted.

Fourth iteration

[3, 5, 2, 6, 7, 8] - > [3, 5, 2, 6, 7, 8]

[3, 5, 2, 6, 7, 8] - > [3, 2, 5, 6, 7, 8]

[3, 2, 5, 6, 7, 8] - > [3, 2, 5, 6, 7, 8]

[3, 2, 5, 6, 7, 8] - > [3, 2, 5, 6, 7, 8]

[3, 2, 5, 6, 7, 8] - > [3, 2, 5, 6, 7, 8]

Fifth iteration

[3, 2, 5, 6, 7, 8] - > [2, 3, 5, 6, 7, 8]

Let us take an example to understand this concept.

Example 6.44:

```
def bubble_sort_fun(list2):

for i in range(0,len(list2)-1):
for j in range(len(list2)-1):
if(list2[j]>list2[j+1]):
```

```
temp = list2[j]
list2[j] = list2[j+1]
list2[j+1] = temp
return list1


list1 = [5, 3, 8, 6, 7, 2]
print ("The unsorted list is: ", list1)


print ("The sorted list is: ", bubble_sort_fun(list1)) # Calling
function
```

Output:

```
The unsorted list is: [5, 3, 8, 6, 7, 2]
The sorted list is: [2, 3, 5, 6, 7, 8]
```

Selection sort

It is used to sort a list of items into ascending or descending order. In selection sort, the first element in the list is chosen and compared to the rest of the list's elements repeatedly. Both elements are swapped if one is smaller than the other (in ascending order). The element in the second place in the list is then selected and compared to the remaining elements in the list. Both elements are swapped if one of them is smaller than the other. This process is repeated until all of the items in the list have been sorted. The selection sorting procedure is as follows.

Suppose we have a list of the action of the following elements with a list that contains the following elements: [3, 5, 1, 2, 4].

We'll start with the list that hasn't been sorted:

3 5 1 2 4

All of the elements can be found in the unsorted portion. We examine each item and decide that the smallest element is 1. As a result, we'll swap 1 with 3:

1 5 3 2 4 of the remaining unsorted elements, [5, 3, 2, 4], 2 is the lowest number. We now swap 2 with 5:

1 2 3 5 4

This procedure is repeated until the list has been sorted:

1 2 3 5 4

1 2 3 4 5

1 2 3 4 5

Let us take an example to understand this concept.

Example

```
def selection_sort_fun(list1):

for i in range(len(list1)-1):

idx = i
```

```
for j in range(i+1, len(list1)-1):

if list1[j] < list1[idx]:
idx = j

list1[i], list1[idx] = list1[idx], list1[i]

list = [23, 12, 17, 34, 98, 77, 58]
print (list)
selection_sort_fun(list)
print (list)
```

Output:

```
[23, 12, 17, 34, 98, 77, 58]
[12, 17, 23, 34, 77, 98, 58]
```

Insertion sort

The insertion sort algorithm places elements in a specific order in a list. In the insertion sort algorithm, each iteration transfers an element from an unsorted portion to a sorted piece until all of the elements in the list are sorted. To sort the list using insertion sort, go through the steps below:

We split a list into two parts, i.e., sorted and unsorted.

Iterate from a **list[1]** to **list[n]** over the given list.

Compare the current element to the next element.

If the current element is smaller than the next element, compared to the element before. Move to the greater elements one position up to make space for the swapped element.

Suppose we have a list of following elements action with a list that contains the following elements: [10, 4, 25, 1, 5]

The first step to add 10 to the sorted sub-list

[10, 4, 25, 1, 5]

Now we take the first element from the unsorted list, i.e., 4. We store this value in a new variable temp. We can see that the 10>4, then we move the 10 to the right and overwrite the 4 that was previously stored.

10, 25, 1, 5] (temp = 4)

Here 4 is lesser than all elements in the sorted sub-list, so we insert it at the first index position.

10, 25, 1, 5]

We have two elements in the sorted sub-list

Now check the number 25. We have saved it into the temp variable. 25> 10 and also 25> 4, then we put it in the third position and add it to the sorted sub-list.

[4, 25, 1, 5]

Again we check the number 1. We save it in temp. 1 is less than 25. It overwrites 25.

[4, 10, 25, 5] 10>1 then it overwrites again

[4, 25, 25, 5]

[25, 4, 25, 5] 4>1 now put the value of temp = 1

4, 10, 5]

Now, we have 4 elements in the sorted sub-list. 5<25, then shift 25 to the right side and pass temp = 5 to the left side.

[1, 4, 10, 25] put temp = 5

Now, we get the sorted list by simply putting the temp value.

[1, 4, 5, 10, is sorted list.

Let us take an example to understand this concept

Example

# creating a function for insertion

```
def insertion_sort_fun(list2):

for i in range(1, len(list2)):

value = list2[i]
j = i - 1
while j >= 0 and value < list2[j]:
list2[j + 1] = list2[j]
j -= 1
list2[j + 1] = value
return list2

list1 = [23, 16, 7, 88, 73]
print ("The unsorted list is:", list1)
print ("The sorted list1 is:", insertion_sort_fun(list1))
```

Output:

```
The unsorted list is: [23, 16, 7, 88, 73]
The sorted list1 is: [7, 16, 23, 73, 88]
```

Quicksort

Quicksort is an efficient sorting algorithm because it is based on a partition of a list. It uses a divide and conquer approach and divides a list into smaller parts. First of all, it picks the pivot element from a list and applies a partition technique. The result of the partition approach is to locate the pivot element into its actual position. In this technique, there are two types of partitions:

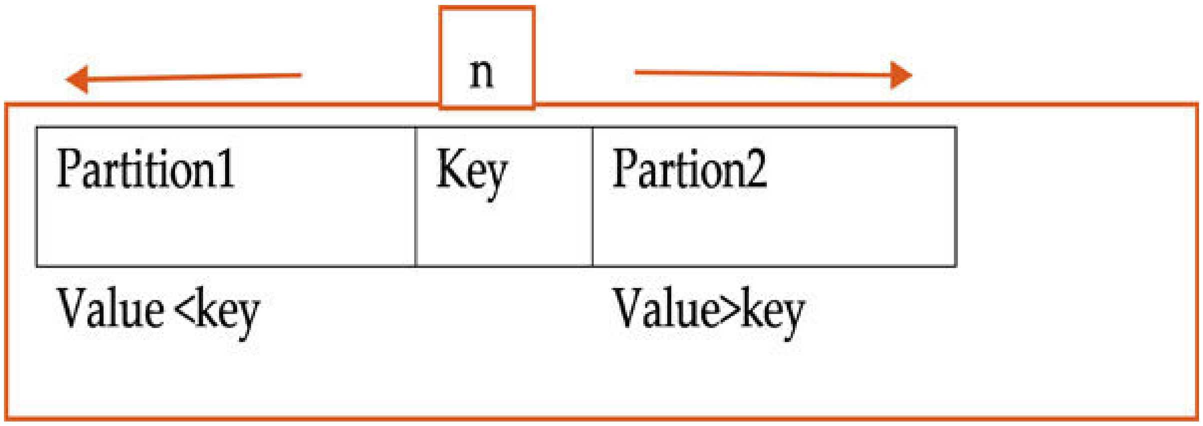


Figure 6.5: *Divide an element by quicksort*

Let us take an example of a quick sort. The key elements are:

45, 26, 177, 14, 68, 61, 97, 39, 99, 90

First of all, we chose an element as a pivot or a key value. Suppose 45 is a key value. After that, it uses two indexes, low and high. The elements in the low index are those elements whose values are less than the key value. The elements in the high index are those elements whose values are less than the key value.

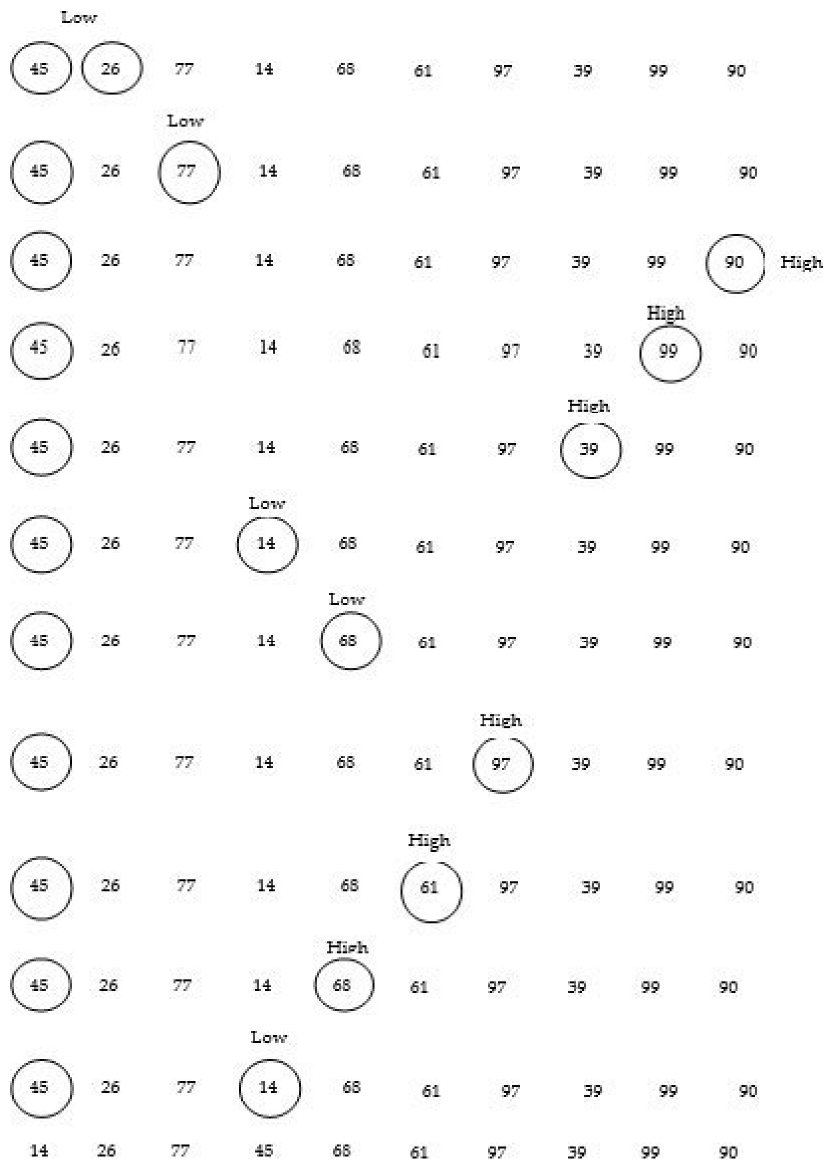


Figure 6.6: *Process of sorting by quicksort*

Let us take an example of a quick sort.

Example

```
def partitionMethod(list1,low,high): # divide function
i = (low-1)

pivot = list1[high] # pivot element
for j in range(low, high):
# If current element is smaller
if list1[j] <= pivot:
# increment
i = i+1
list1[i],list1[j] = list1[j],list1[i]
list1[i+1],list1[high] = list1[high],list1[i+1]
return (i+1)
# sort

def quickSortMethod(list2,low,high):
if low < high:
# index
pi = partitionMethod(list2,low,high)
# sort the partitions
quickSortMethod(list2, low, pi-1)
quickSortMethod(list2, pi+1, high)

list3 = [22,44,21,12,56,87,35,23,66]
n = len(list3)
quickSortMethod(list3,0,n-1)
print ("Sorted list is:")
for i in range(n):
```

```
print (list3[i],end=" ")
```

Output:

```
Sorted list is:
12 21 22 23 35 44 56 66 87
```

Merge sort

Merge sorting employs the divide and conquer strategy. Since a list with only one element is always sorted, the unsorted list is divided into n sublists, each with one element. Then it merges these sublists, again and again, to create new sorted sublists until only one sorted list remains. Merge sort has an *O(n log n)* time complexity. It is also a stable sort but not in place of the sorting approach. Let us take an example of merge sort 85, 76, 46, 92, 30, 41, 12, 19, 93, 3, 50, 11. The following *figure 6.7* and *figure 6.8* shows the divide and conquer and merging process of merge sort.

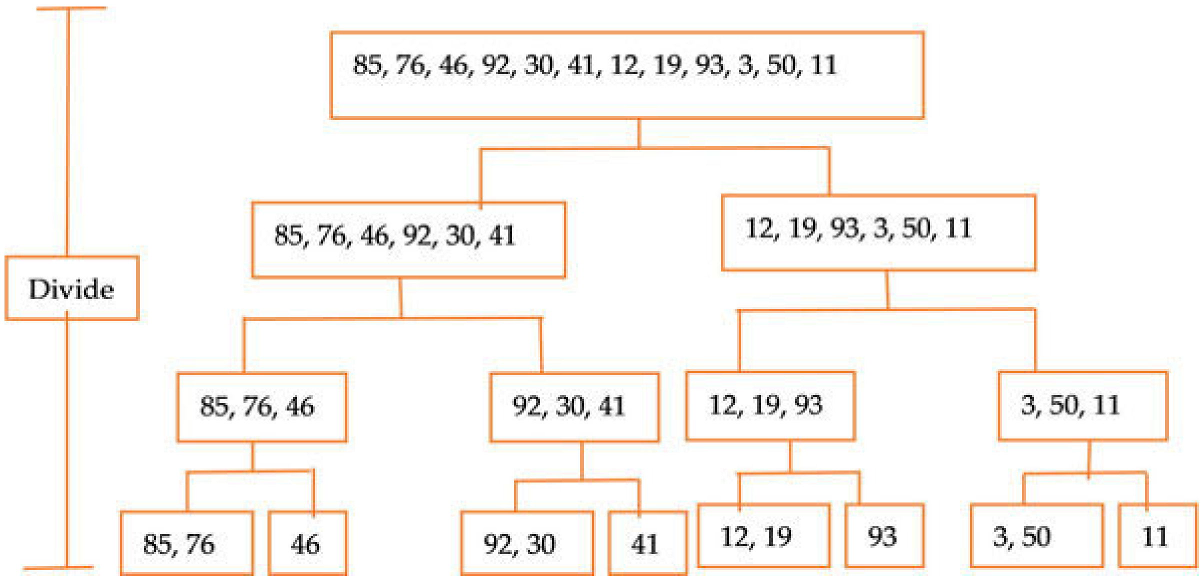


Figure 6.7: Process of divide and conquer in merge sort

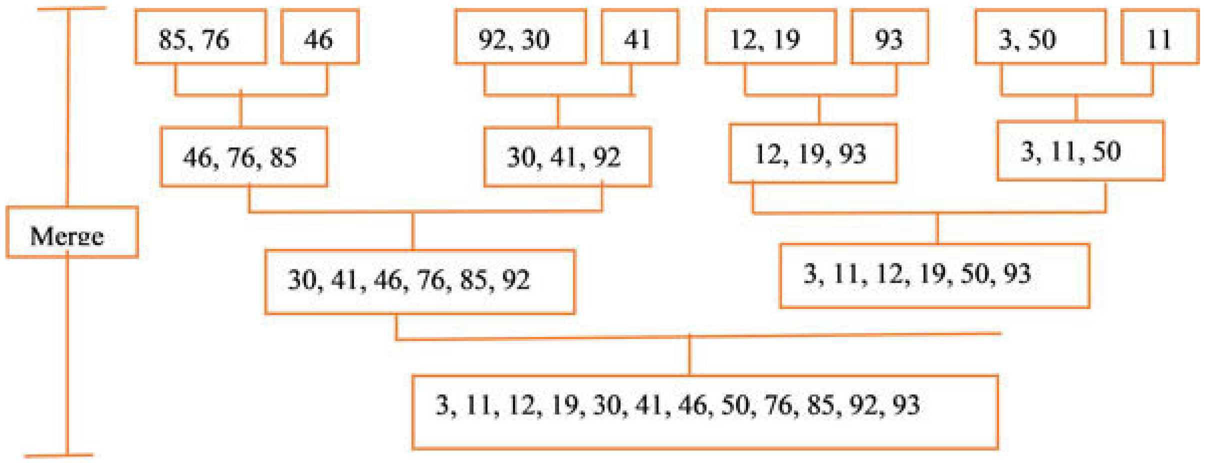


Figure 6.8: *Process of merging in merge sort*

Let us take an example for understanding this concept.

Example

```
def merge_sort_fun(list1, left_index, right_index):
if left_index >= right_index:
return

middle = (left_index + right_index)//2
merge_sort_fun(list1, left_index, middle)
merge_sort_fun(list1, middle + 1, right_index)
merge_fun(list1, left_index, right_index, middle)
```

```
def merge_fun(list1, left_index, right_index, middle):

left_sublist = list1[left_index:middle + 1]
right_sublist = list1[middle+1:right_index+1]

left_sublist_index = 0
right_sublist_index = 0
sorted_index = left_index

while left_sublist_index < len(left_sublist) and right_sublist_index
< len(right_sublist):

if left_sublist[left_sublist_index] <=
right_sublist[right_sublist_index]:
list1[sorted_index] = left_sublist[left_sublist_index]
left_sublist_index = left_sublist_index + 1

else:
list1[sorted_index] = right_sublist[right_sublist_index]
right_sublist_index = right_sublist_index + 1

sorted_index = sorted_index + 1

while left_sublist_index < len(left_sublist):
list1[sorted_index] = left_sublist[left_sublist_index]
```

```
left_sublist_index = left_sublist_index + 1
sorted_index = sorted_index + 1



while right_sublist_index < len(right_sublist):
list1[sorted_index] = right_sublist[right_sublist_index]
right_sublist_index = right_sublist_index + 1
sorted_index = sorted_index + 1


list1 = [44, 65, 2, 3, 58, 14, 57, 23, 10, 1, 7, 74, 48]
merge_sort_fun(list1, 0, len(list1) -1)
print (list1)
```

Output:

```
[1, 2, 3, 7, 10, 14, 23, 44, 48, 57, 58, 65, 74]
```

Conclusion

This chapter discussed the lists and their creation, accessing, and updating values in lists. It is concerned about nesting, aliasing, and cloning of lists. Some list parameters, operations, and methods are discussed. Also, passing and returning list from function and some other list function like and **reduce()** function also included. It also discusses searching and sorting methods in the list such as linear search, binary search, bubble, selection, insertion, quick, and merge sort. In the further chapter, we will discuss tuple as the next topic used in Python.

Points to Remember

In a list, values are enclosed with square brackets.

The first item on the list is indexed as

A slice is represented by a colon

A nested list means a list within a list.

The **list()** constructor takes a single argument.

A list is used in an operator for searching in a list.

The **sort()** method is used for sorting elements of a given list.

Multiple Choice Questions

Which of the following is True regarding lists in Python?

Lists are immutable.

The size of the lists must be specified before their initialization

Elements of lists are stored in a contagious memory location.

size(list1) command is used to find the size of lists.

What is the output of the following code:

```
list1=[1,3,5,2,4,6,2]
list1.remove(2)
print(sum(list1))
```

18

19

21

Which of the following commands will create a list?

list1 = list()

list1 = []

list1 = list([1, 2, 3])

all of the mentioned

What is the output when we execute the list("hello")?

['h', 'e', 'l', 'l', ‘o’].

[‘hello’].

[‘llo’].

[‘olleh’].

Suppose list Example is ['h',' e',' l',' l','o'], what is len(listExample)?

5

4

None

Error

Suppose list1 is [4, 2, 2, 4, 5, 2, 1, 0], which of the following is the correct syntax for slicing operation?

print(list1[0])

print(list1[:2])

print(list1[:-2])

all of the mentioned

What is the output of the following code:

```
list1 = range(100, 110)
print (list1.index(105))
```

105

5

106

104

What is the output of the code?

```
list1 = [1, 2, 3, 4, 5]
list2 = list1
list2[0] = 0;
print(list1)
```

[1, 2, 3, 4, 5, 0]

[0,1, 2, 3, 4, 5]

[0, 2, 3, 4, 5]

[1, 2, 3, 4, 0]

Suppose list1 is [2, 33, 222, 14, 25], What is list1[-1]?

Error

None

25

2

What is the best time complexity of bubble sort?

N^2

NlogN

N

N(logN)^2

What is the best case for linear search?

O(nlogn)

O(logn)

O(n)

O(1)

What is the worst case for linear search?

O(nlogn)

O(logn)

O(n)

O(1)

Which of the following is a disadvantage of linear search?

Requires more space

Greater time complexities compared to other searching algorithms

Not easy to understand

Not easy to implement

What is the worst-case complexity of selection sort?

O(nlogn)

O(logn)

O(n)

O(n2)

Answers

c

c

d

a

a

d

b

c

c

c

d

c

b

d

Questions

How to create a list? Explain with example.

How to access values from lists? Explain with example.

Explain slice, delete, remove and clear methods of a list. Give a suitable example.

How to update the values in a list? Explain with example.

What do you mean by aliasing and cloning a list? Explain with example.

Write any five built-in list functions with suitable examples.

What do you mean by the built-in list methods? Explain with example.

How to use loops in lists? Explain with example.

Explain the working and **reduce()** methods with examples.

Explain linear and binary search with proper explanation.

Explain quick sort and merge sort with example.

CHAPTER 7

Tuple

The tuple is an important data type in Python, which stores various elements as an object. It is also known as an immutable data type that means its elements are fixed and cannot be changed. In this chapter, you will see various properties and operations that can be applied to tuples. Also, various functions and operations of tuples are demonstrated with the help of examples and codes.

Structure

In this chapter, we will cover the following topics:

The concepts and properties of tuples

Tuple creation and accessing elements

Various tuple operations

Packing and unpacking of tuple elements

Using tuples with functions

Objective

The objective of this chapter is to introduce the concept of Python tuples. After completing this chapter, you should understand the various operations and properties of tuples. Also, you will be able to write various codes related to tuples.

What is a tuple?

A tuple is a group of objects that have been ordered and are immutable. Tuples are like lists and strings. Here, the meaning of immutable is that the tuple elements are fixed. Once a tuple is created, elements addition and deletions are not possible. Even we cannot shuffle the order of tuple elements. The length of the tuple is also fixed. If we want to increase or decrease a tuple, we need to create a new tuple.

The distinctions between tuples and lists are that, unlike lists, tuples cannot be altered, and tuples use parentheses, while lists use square brackets. Tuples are the type of sequence the same as string. Strings can only store characters, but tuples are able to store any kind of elements. It means that the tuple either stores a group of student's names or stores the IDs of employees. Tuples can also store elements of mixture types that mean a tuple may have elements of numbers, characters, and decimal formats together. Tuples can also store a sequence of sound files, images, etc.

Difference between tuple and list

The following table describes the differences between tuples and lists:

lists:
lists: lists: lists: lists: lists: lists: lists:
lists: lists: lists: lists: lists: lists: lists: lists:

lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists:
lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists: lists:
lists: lists: lists: lists:
lists: lists: lists:
lists: lists: lists: lists:

Table 7.1: *Differences between tuples and lists*

Creating tuple

To create a tuple in Python, all the elements are enclosed in **()** parenthesis, separated by a comma. A tuple can store heterogeneous data elements. Below are the examples of creating tuples:

Example 7.1:

```
# Create an empty tuple
Tuple1 = ()
# tuple with 5 elements
Tuple2 = (10,20,30,40,50)
print (Tuple2)
# tuple of character elements
Tuple3 = ('x', 'y', 'z')
print (Tuple3)
# tuple of strings
Tuple4 = ("Python", "Programming", "Book")
print (Tuple4)
# tuple of heterogeneous data elements
Tuple5 = (100, 20.123, "Python Programming")
print (Tuple5)
# tuple of string and list
```

```
Tuple6 = ("Python Book", [10, 20, 25])
print (Tuple6)
```

Output:

```
(10, 20, 30, 40, 50)
('x', 'y', 'z')
('Python', 'Programming', 'Book')
(100, 20.123, 'Python Programming')
('Python Book', [10, 20, 25])
```

Single element tuple

To create a tuple of single element, it should be followed by a comma. The following example creates a tuple of a single element:

```
Tup1 = (100,)
```

In the preceding case, if we do not put a comma after then Python would treat **Tup1** as an **int** rather than a tuple variable.

Example 7.2:

```
Tup1 = (100,)
type (Tup1)
Tup2=(100)
type (Tup2)
```

Output:

```
'tuple'>
'int'>
```

We have to add a comma after the element when a tuple has just one element; otherwise, Python would not consider it as a tuple.

The tuple () function

The **tuple ()** function is a Python built-in function that can be used to create a tuple. We can even create a tuple without using this function, but it provides an alternative way of creating tuples. The following example shows the use of **tuple ()** function to create a tuple:

Example 7.3:

```
# example of tuple() function
# creating an empty tuple
T1 = tuple()
print(T1)
# creating a tuple from list
T2 = tuple([1,2,3,4,5])
print (T2)
# creating a tuple from strings
T3 = tuple("Python Programming")
print (T3)
```

Output:

```
()
(1, 2, 3, 4, 5)
('P', 'y', 't', 'h', 'o', 'n', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n',
'g')
```

Accessing values in tuple

In order to access the elements of a tuple, indexes are used. Tuple indexes start at 0, like string indices, and can be sliced, concatenated, and so on. Following examples shows how we can access values in tuples:

Example 7.4:

```
# tuple of characters
tup1 = ('A', 'B', 'C', 'D')
# tuple of integers
tup2 = (10, 20, 30, 40, 50, 60, 70)
# displaying first element of tuple1
print tup1[0]
# displaying second to fifth elements of tuple 2
print tup2[1:5]
```

Output:

```
A
(20, 30, 40, 50)
```

The value of the index is always an integer. For example, print tup1[2.0] will generate type error.

```
TypeError: tuple indices must be integers, not float
```

Positive and negative indexes

In Python, the same as list and string indexes can be positive and negative. Positive indexes start from 0, while negative indexes start from -1. Positive index access elements from the beginning of the tuple, while negative indexes are used to access elements from the tuple's end. The positive and negative indexes are shown in the following figure:

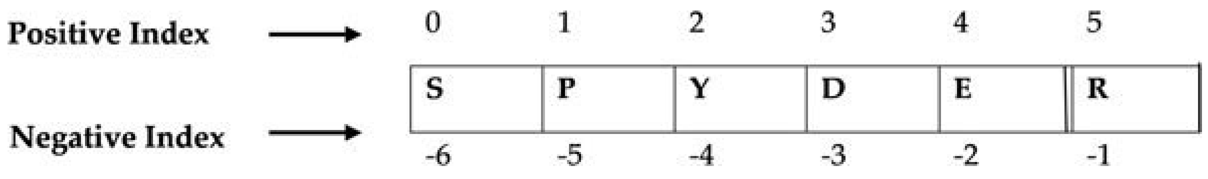


Figure 7.1: *Positive and negative indexing*

The following example demonstrates the use of a negative index:

Example 7.5:

```
Tuple1 = (10, 20, "Books", 108.92)
# Accessing last element using negative index
print(Tuple1[-1])
```

```
# Accessing third element from last
print(Tuple1[-3])
```

Output:

```
108.92
20
```

Slicing tuple

Slicing acts in the same manner as strings. We ought to have a position for the beginning and end. The outcome is a tuple between those two positions, containing every element. We must use the **[]** operator on the tuple to slice a tuple. If we have a positive integer when indexing a tuple, the index is obtained from the tuple count on the left. In case of a negative index, the index is obtained from the right tuple count. The following code is an example of slicing operations on the tuple:

Example 7.6:

```
T1 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# print elements from 3rd to 6th
print(T1[2:6])
# print elements from start to 4th position
print(T1[:4])
# print elements from 4th to last
print(T1[3:])
# print elements from 6th to second last
print(T1[5:-1])
# print entire tuple
```

```
print(T1[:])
```

Output:

```
(3, 4, 5, 6)
(1, 2, 3, 4)
(4, 5, 6, 7, 8, 9, 10)
(6, 7, 8, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Updating tuple

Tuples are immutable, which indicates that the values of tuple elements will not be changed or modified. However, we may change the elements of nested items that are mutable. In the following code, for example, we change the list element present within the tuple. Elements in the list are mutable, so it is permissible:

Example 7.7:

```
T1= (1, [2, 3, 4,5], "Python")
print(T1)
# Updation is not valid for tuples
#T1[0] = 100      # It generates error
# Changing the list element, this is valid because list is
mutable
T1[1][2] = 10
print(T1)
```

Output:

```
(1, [2, 3, 4, 5], 'Python')
```

```
(1, [2, 3, 10, 5], 'Python')
```

We can create a new tuple using a portion of an existing tuple. The example below demonstrates the creation of a new tuple:

Example 7.8:

```
T1 = (1, 2, 3, 4.5)
T2 = ('ABCD', 'XYZW')
# New tuple creation from existing
T3 = T1 + T2[0:1]
print T3
```

Output:

```
(1, 2, 3, 4.5, 'ABCD')
```

Deleting elements in tuple

We have already demonstrated that tuple elements are immutable, which implies that the elements in a tuple cannot be deleted. However, deleting the whole tuple can be possible. The **del** statement can be used to delete an entire tuple:

Example 7.9:

```
T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# printing whole tuple
print(T2)
#del T2[2]      #it will generate an error
# deleting an entire tuple
del T2
#print after deleting a tuple
print(T2)             #it will generate error
Output:
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Traceback (most recent call last):
File "main.py", line 11, in
print(T2)             #it generates error
NameError: name 'T2' is not defined
```

Basic tuple operations and functions

The different operations on tuples that we can perform are very similar to lists. In general, with tuples, we just saw the + operator. It works with tuples just like it works with a list. Tuples behave much like strings to the + and * operators; they also imply concatenation and repetition here, except that a new tuple is a result, not a string. Tuples respond the same as strings to all the general sequence operations.

Addition operators combine all the elements of two or more tuples into a new tuple.

Example 7.10:

```
T1 = (1, 2, 3)
T2=(4, 5)
T3=(10,)
T4=T1+T2+T3
print (T4)
```

Output:

```
(1, 2, 3, 4, 5, 10)
```

If a tuple is multiplied by any integer, **x** would basically create another tuple with all the elements repeated **x** times from the first tuple. **T*5** indicates, for instance, the elements of tuple **T** will be replicated 5 times:

Example 7.11:

```
#tuple with three elements
T = (1, 2, 3)
#replicate 5 times
print(T*5)
#replicate string 3 times
print("Python"*3)
```

Output:

```
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
PythonPythonPython
```

len() This function is used to get the number of elements in the tuple.

max() and min() These functions are used to get the maximum and minimum value from the tuple.

sum() This function is used to get the sum of all the elements of tuple.

in This keyword is used with tuple same as strings and lists. It is used to check whether any element is present or not in the tuple. If the element is present, it returns otherwise it returns

Example 7.12:

```
#tuple with six elements
T = (10, -2, 300, 40, 302, 900)
# length of the tuple
print ("Length:", len(T))
# Max value from tuple
print ("Max:", max(T))
# Min value from tuple
print ("Min:", min(T))
# Sum of all elements of tuple
print ("Sum:", sum(T))
# Check whether 40 is present or not in tuple
print (40 in T)
# Check whether 500 is present or not in tuple
print (500 in T)
```

Output:

```
('Length:', 6)
('Max:', 900)
('Min:', -2)
('Sum:', 1550)
True
False
```

cmp() The **cmp ()** function is used to compare two tuples. Based on whether the two tuples being compared are similar or not, it will return either 1, 0 or -1. This function takes two tuples as arguments for comparison. If **T1** and **T2** are two tuples then the following result will be produced based on the comparison.

if **T1** > then **cmp(T1, T2)** returns 1

if **T1** = then **cmp(T1, T2)** returns 0

if **T1** then **cmp(T1, T2)** returns -1

Example 7.13:

```
#first tuple
T1 = (100, 20,30,40)
#second tuple
T2=(100,2,3,4,5)
print("T1>T2:",cmp(T1,T2))
print("T1<__cmp_T2_T1__ _="" span=""/>
#third tuple
T3 = (100, 20,30,40)
print("T1=T3:",cmp(T1,T3))
```

Output:

```
('T1>T2:', 1)
('T1<__ _="" span="">-1)
('T1=T3:', 0)
```

index() index() is a built-in Python function which searches for a given element in a tuple and returns the lowest index where the element appears. The **index()** function takes maximum three arguments and follows the given syntax:

```
tuple.index(element, start, end)
```

Here **element** is the first argument that must be searched in a tuple. **Start** and **end** are an optional arguments that indicates starting and ending point index of a searching element:

Example 7.14:

```
# Tuple of four elements
T1 = (100, 20,30,40)
# tuple of characters
T2=("PYTHON")
# print the index of 100 in tuple T1
print("Index No.:",T1.index(100))
# print the index of T in tuple T2
print("Index No.:",T2.index('T'))
# search the index of 20, start from 1 to 4
print("Index No.:",T1.index(20,1,4))
```

Output:

```
('Index No.:', 0)
('Index No.:', 2)
('Index No.:', 1)
```

count() The **count()** function returns the number of occurrences in the given tuple for an element.

Example 7.15:

```
#tuple 1

T1 = (1, 1, 1, 2, 2, 2, 1, 2, 3, 1, 1)
#tuple 2
T2 = ('a', 'a', 'a', 'b', 'b', 'a', 'c', 'b', 'a','b')
#tuple 3
T3 = ['Bat', 'Cat', 'Sat', 'Bat', 'cat', 'Bat']
# Counts the number of times 1 appears in tuple 1
print("Count of 1:", T1.count(1))
# Counts the number of times 'a' appears in tuple 2
print("Count of a:", T2.count('a'))
# Counts the number of times 'Bat' appears in tuple 3
print("Count of Bat:", T3.count('Bat'))
```

Output:

```
('Count of 1:', 6)
('Count of a:', 5)
('Count of Bat:', 3)
```

Packing and unpacking a tuple/tuple assignment

Python has a very useful function known as tuple assignment, which allows values to be assigned to a tuple of variables on the left-hand side of an assignment from a tuple on the right-hand side of the assignment. It is also called **unpacking** a tuple of values into a variable. We put values into a new tuple while while we extract those values into a single variable while unpacking. This feature helps more than one variable to be assigned at a time while a sequence is on the left. In the example below, one tuple T1 assigns values to five different variables. In this assignment, the number of variables on the left-hand side must be equal to the number of tuple elements on the right-hand side. Otherwise, this assignment generates an error.

Example 7.16:

```
# tuple with five elements/Packing
T1= ('David', 'India', 1990, 1001)
print ("Tuple Elements are:", T1)
# tuple assignment to different variables/Unpacking
(EmpName, Country, BirthYear, EmpId)=T1
# printing individual variables
```

```
print("-------------------------")
print ("EmpName:",EmpName)
print ("Country:", Country)
print ("BirthYear:", BirthYear)
print ("EmpId:",EmpId)
```

Output:

```
Tuple Elements are: ('David', 'India', 1990, 1001)
-------------------------
EmpName: David

Country: India
BirthYear: 1990
EmpId: 1001
```

In the example above, we have a tuple **T1** which contains five elements and assign different values to the variables and **EmpId** in a single statement. We can also assign individual element values to variables such as:

Example 7.17:

```
# tuple with five elements
T1= ('David', 'India', 1990, 1001)
print ("Tuple Elements are:", T1)
```

```
# assigning one value at a time
EmpName=T1[0]
Country=T1[1]
BirthYear=T1[2]
EmpId=T1[3]
#printing individual values
print ('------------------')
print (EmpName)
print (Country)
print (BirthYear)
print (EmpId)
```

Output:

```
Tuple Elements are: ('David', 'India', 1990, 1001)
------------------
David
India
1990
1001
```

Receiving tuples in functions

There is a special way of receiving parameters using the * prefix for a function as a tuple. This is helpful when taking the number of arguments in the function as a function. All extra arguments passed to the function are stored in **args** as a tuple due to the * prefix on the **args** variable:

Example 7.18:

```
# This function receives a tuple as an argument
def fun1(*args):
print(type(args))   #printing the type of args
Print(args)
# creating a tuple
T1=("Python", "Book", 2020)
# passing tuple to a function
fun1(T1)
```

Output:

```
'tuple'>
(('Python', 2020),)
```

Returning multiple values as a tuple

In C and other programming languages, returning multiple values from a function is impossible, but it is quite simple to do with Python. In Python, by simply returning statements separated by commas, we can return multiple values. In the example below, a function **fun1()** is defined to return a string and a number:

Example 7.19:

```
#Returning multiple values as tuple
# This function returns a tuple
def fun1():
s1 = "Python"
x = 2020
return (s1, x)
# calling function
T1= fun1() #T1 is a tuple
print(type(T1))
print(T1)
```

Output:

```
'tuple'>
('Python',  2020)
```

Nested tuples

We may declare a tuple inside a tuple in Python, and this is called a nesting tuple. To arrange data into hierarchical structures, nesting tuples are used. If only one element is present in a tuple, then it is not called as a To show to the interpreter that it is a tuple, it should be a trailing comma. We have declared a tuple inside another tuple in the following example:

Example 7.20:

```
# Nested tuples create
Tuple1 = ("Python", [1,2,3,4], (5, 5.6, "Programming"))
# Print all elements of tuple
print ("Tuple values are:", Tuple1)
# Print first element
print ("first Element:", Tuple1[0])
#print second element or sub-tuple
print ("sub tuple:", Tuple1[1])
#print third element or sub-tuple
print ("sub tuple:", Tuple1[2])
```

Output:

```
Tuple values are: ('Python', [1, 2, 3, 4], (5, 5.6, 'Programming'))
first Element: Python
sub tuple: [1, 2, 3, 4]
sub tuple: (5, 5.6, 'Programming')
```

Creating tuple from string and list

In Python, a tuple can be created from list as well as string. We need this kind of conversion when there are different types of data available and new tuple must be created. The **tuple ()** function is used to convert string or list to a tuple. This concept is illustrated in the following example:

Example 7.21:

```
# creating list and string
L1 = ['Python',1,2,3,4]
S1 = "Programming"
# printing original list and string
print ("The original list : ", L1)
print ("The original string : ", S1)
print (type(L1))
print(type(S1))
#concert list to tuple
result1 = tuple(L1)
#concert string to tuple
result2=tuple(S1)
print("Converted List : ", result1)
print (type(result1))
```

```
print("Converted String : ", result2)
print (type(result2))
```

Output:

```
The original list : ['Python', 1, 2, 3, 4]
The original string : Programming
'list'>
'str'>
Converted List : ('Python', 1, 2, 3, 4)

'tuple'>
Converted String : ('P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g')
'tuple'>
```

The zip() function

Python **zip ()** function is mainly used to combine data of two different container elements together. This function takes iterables and aggregates their elements based on their order in the iterables. As an input it takes any number of iterables and returns an iterator of tuples. Iterables may be list, string, etc. The following example groups items of two different list based on the same index:

Example 7.22:

```
#creating two lists
List1 = ['A', 'B', 'C', 'D']
List2 = [10, 20, 30, 40]
#combine using zip
Result = tuple(zip(List1, List2))
print (Result)
print (type(Result))
```

Output:

```
(('A', 10), ('B', 20), ('C', 30), ('D', 40))
```

'tuple'>

The returned iterator takes the length of the shortest iterator passed to the function if the passed iterables are of different lengths. In the example below, we have two lists of size four and two, respectively. The result tuple has length two, based on the shortest list

Example 7.23:

```
#creating two lists
List1 = ['A', 'B', 'C', 'D']
List2 = [10, 20]
#combine using zip

Result = tuple(zip(List1, List2))
print (Result)
print (type(Result))
```

Output:

```
(('A', 10), ('B', 20))
'tuple'>
```

The **zip()** function also works for more than two iterables, such as:

Example 7.24:

```
#creating three tuples
T1 = ('Amit', 'Sumit', 'Joy', 'Dev')
T2 = (50000, 20000, 25000, 80000)
T3=('HR', 'Clerk','IT', 'Manager')
#combine using zip
Result = tuple(zip(T1, T2, T3))
print (Result)
print (type(Result))
```

Output:

```
(('Amit', 50000, 'HR'), ('Sumit', 20000, 'Clerk'), ('Joy', 25000,
'IT'), ('Dev', 80000, 'Manager'))
'tuple'>
```

The inverse zip(*) function

The * operator can be used with the **zip ()** function. It unzips the tuples into independent lists. In other words, the * operator with **zip ()** function unpacks a sequence into positional arguments. The following example shows the concept of unzipping tuples:

Example 7.25:

```
# using zip() and * operator to unpack sequences
# creating tuple
T1 = [('A', 100), ('B', 200), ('C', 300), ('D', 4000)]
# Original tuple
print ("Original tuple is : ", T1)
# using zip() and * operator to perform unzipping
T2,T3 = zip(*T1)
T4=zip(*T1)
# Printing modified tuple 1
print ("Modified Tuple1 : ", T2)
print (type(T2))
# Printing modified tuple 2
print ("Modified Tuple2 : ", T3)
print (type(T3))
```

```
# Printing modified tuple 3
print ("Modified Tuple2 : ", tuple(T4))
```

Output:

```
Original tuple is : [('A', 100), ('B', 200), ('C', 300), ('D', 4000)]
Modified Tuple1 : ('A', 'B', 'C', 'D')
'tuple'>
Modified Tuple2 : (100, 200, 300, 4000)
'tuple'>
Modified Tuple2 : (('A', 'B', 'C', 'D'), (100, 200, 300, 4000))
```

Tuple sorting

We have an in-built function named **sorted()** in python to sort the tuple elements. This function takes one argument as a parameter to sort tuples.

```
sorted(arg1)
```

By default, the tuple is sorted in ascending order using **sorted()** function. If we want to sort in descending order, we need to set the parameter reverse to True.

```
sorted(arg1, reverse=True)
```

Example

```
# Create a tuple
Tuple1 = (20, 4, 6, -1, 4, 7.8, 2.79, 5.90)
# Sorting in ascending order
TDefault=sorted(Tuple1)
# Sorting in descending order
TReverse=sorted(Tuple1, reverse=True)
# Printing original tuple
```

```
print ("Original tuple:", Tuple1)
# Printing sorted tuple-Ascending
print ("Sorted tuple:", TDefault)
# Printing sorted tuple-Descending
print ("Sorted tuple in reverse order:", TReverse)
```

Output:

```
Original tuple: (20, 4, 6, -1, 4, 7.8, 2.79, 5.9)
Sorted tuple: [-1, 2.79, 4, 4, 5.9, 6, 7.8, 20]
Sorted tuple in reverse order: [20, 7.8, 6, 5.9, 4, 4, 2.79, -1]
```

Example 7.27:

```
def TupleSort(t):
sort1 = sorted(t)

sort2= sorted(t, reverse=True)
return sort1, sort2
# Create a tuple
Tuple1 = (20, 4, 6, -1, 4, 7.8, 2.79, 5.90,900)
# passing parameter to function for sorting
TDefault, TReverse=TupleSort(Tuple1)
# Printing original tuple
print ("Original tuple:", Tuple1)
# Printing sorted tuple-Ascending
print ("Sorted tuple:", TDefault)
```

```
# Printing sorted tuple-Descending
print ("Sorted tuple in reverse order:", TReverse)
```

Output:

```
Original tuple: (20, 4, 6, -1, 4, 7.8, 2.79, 5.9, 900)
Sorted tuple: [-1, 2.79, 4, 4, 5.9, 6, 7.8, 20, 900]
Sorted tuple in reverse order: [900, 20, 7.8, 6, 5.9, 4, 4, 2.79,
-1]
```

Conclusion

In this chapter, we discussed the concepts and various properties of tuples. We also discussed how different operations can be applied on the tuples. Various examples and Python codes are given to understand tuple systematically. Based on these examples you can write your own code for a specific task. In the next chapter, we will discuss dictionary as the next data type used in Python.

Points to remember

Tuples are immutable data type.

In Python, tuples are written with round brackets.

The elements of the tuples are not changeable.

Tuples' operations are very similar to list and strings.

Multiple choice questions

Choose the correct option.

Both tuples and lists are immutable.

Tuples are immutable, while lists are mutable.

Both tuples and lists are mutable.

Tuples are mutable while lists are immutable.

Suppose T = (1, 2, 3, 4,5), which of the following is incorrect?

T[2] = 50

print(T[2])

print(max(T))

print(len(T))

What will be the output of the below code?

```
tuple1=(1,2,3)
tuple3=tuple1*2
print(tuple3)
```

(2,4,6)

(1,2,3,1,2,3)

(1,1,2,2,3,3)

Error

Which is a correctly declared tuple?

T = ("orange", "yellow", "red")

T = "orange", "yellow", "red"

T = ["orange", "yellow", "red"]

T = "orangeyellowred"

What will be the output of the following program?

```
Tuple1 = (10, 20, 40, 30)
Tuple2 = (10, 20, 30, 40)
print(Tuple1
```

Error

True

False

Unexpected

Answers

b

a

b

a

c

Questions

What is the difference between a list and a tuple?

How would you convert a list into a tuple?

Discuss built-in function supported by tuples?

How does packing and unpacking work on the tuple? Give a suitable example.

What is positive and negative index in a tuple? Write an example.

How can a tuple be passed to a function?

Write Python code to sort tuple elements.

Write Python code to copy the first and second elements of a tuple into a new tuple.

Write code to count the number of occurrences of an item from a tuple.

Write code to unpack the following tuple into five variables.

Tuple1 = (100, 200, 300, 400, 500)

CHAPTER 8

Dictionaries

Dictionary is another data type in Python, which is similar to lists and tuples. It is also known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value. In this chapter, you will see various properties and operations that can be applied to dictionaries. Besides, we have demonstrated various functions and operations of dictionaries with the help of examples and codes.

Structure

In this chapter, we will cover the following topics:

The basic concepts of the dictionary

Dictionary creation and accessing elements

Various operations related to dictionaries

The concepts of nested dictionaries

Formatting dictionary elements

Objective

The objective of this chapter is to introduce the concept of Python dictionaries. After completing this chapter, you should be able to understand the various operations and properties of the dictionaries. Also, you will be able to write various codes related to dictionaries.

Basics of dictionary

Dictionary is another important built-in data type in Python. It is used to store data as collections. Dictionaries are not commonly used, such as arrays and lists. In Python, a dictionary is a collection that keeps information or value associated with some specific keys. The combination of values along with keys is known as mapping relation. Hence Python dictionaries follow mapping relations among elements.

Dictionaries work differently as compared to lists and arrays. List stores and retrieves elements from the collection using index positions. It means that we can look up by the index number whenever we want to access an element in a list or array.

Many applications require a more flexible way of getting information from the collection, and programmers are not interested in working with indexing structures. Hence, dictionaries can provide a better solution. The concept of dictionaries can be understood by some real examples. If we want to get the phone number of an employee based on their name or want to know the city name according to the given pin code, then we have an efficient structure known as a dictionary that follows mapping between values and keys. In programming, this combination is referred to as a key-value pair. In this combination, we access any information/value (city name) associated with a particular key (pin code). The following figure shows the simple structure of the dictionary with a *key-value* example:

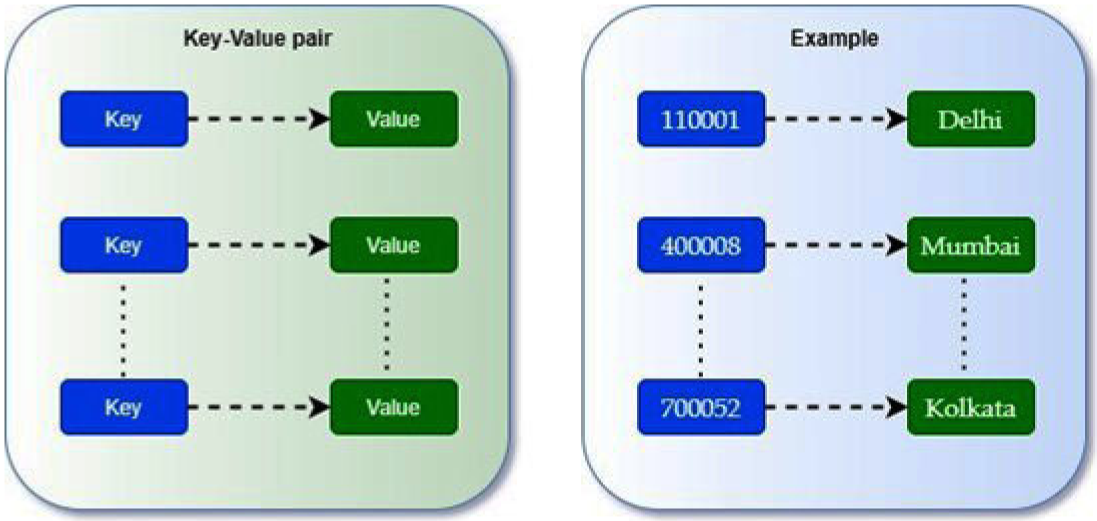


Figure 8.1: *Simple structure of a dictionary*

Hash and associative array are also another name of a dictionary used in some other programming languages. The following example shows the concept of a dictionary that stores city names associated with pin codes:

{"110001":"Delhi", "400008": "Mumbai", "700052":" Kolkata"}

In Python dictionary, all the elements are enclosed in curly brackets such as **{** and A colon is used to join the key-value pair. The following is the syntax of the dictionary:

```
dictionary = {
: ,
: ,
.
.
.
:
}
```

Creating a dictionary

A dictionary can be created in Python by placing element sequences inside the curly brackets of **{** and separated by comma There are several values in the dictionary: the key and the other corresponding part of the pair being the value. Values in a dictionary can be of any data type and can be duplicated, while keys cannot be repeated and must be immutable. Dictionary keys are case-sensitive, which means different cases represent distinct keys. There are several methods to create a dictionary.

Method 1: Create empty dictionary and then add items

The first method of dictionary creation is to create an empty dictionary and then add items to it:

Example 8.1:

```
# Creating an empty dictionary
Dict1 = {}
# Adding items to dictionary
Dict1["key1"] = [1, 2, 3]
Dict1["key2"] = ["Code", "From", "Home"]
print (Dict1)
# Checking type of Dict1
print (type(Dict1))
```

Output:

```
{'key1': [1, 2, 3], 'key2': ['Code', 'From', 'Home']}
'dict'>
```

Method 2: By passing key-value pair literals

Another way of creating a dictionary by passing key-value pairs literals:

Example 8.2:

```
# Creating a dictionary
Dict1 = {
110001 : "Delhi",
400008 : "Mumbai",
700052 : "Kolkata",
600001 : "Chennai",
}
# Printing dictionary
print (Dict1)
# Checking type of Dict1
print (type(Dict1))
```

Output:

```
{110001: 'Delhi', 400008: 'Mumbai', 700052: 'Kolkata', 600001:
'Chennai'}
```

'dict'>

Method 3: Passing key-value pairs to dict() function

We can create a dictionary by passing key-value pairs **todict()** function:

Example 8.3:

```
# Creating a dictionary
Dict2 = dict(
Name='Joy',
Age=25,
Address='Delhi',
Salary=50000)
# Printing dictionary
print(Dict2)
# Checking type of Dict1
print (type(Dict2))
```

Output:

```
{'Name': 'Joy', 'Age': 25, 'Address': 'Delhi', 'Salary': 50000}
'dict'>
```

Method 4: Passing list of tuples to dict() function

The next method of creating dictionaries is to pass a list of tuples to **dict ()** function:

Example 8.4:

```
# Passing list of tuples
Dict3= dict(
[('Name', 'John'),
('Age', 25),
('Address', 'Delhi'),
('Salary', 50000)])
# Printing dictionary
print(Dict3)
# Checking type of Dict1
print (type(Dict3))
```

Output:

```
{'Name': 'John', 'Age': 25, 'Address': 'Delhi', 'Salary': 50000}
'dict'>
```

Dictionary does not contain any duplicate keys.

Accessing dictionary values

A dictionary uses key names to access any value, while indexing is used for other data types to reach values. Key names can be used within square brackets **[]** to access associated values. The following syntax is used to retrieve any value associated with the key in the dictionary:

```
Dictionaryname[keyname]
```

Example 8.5:

```
# Creating a dictionary
Dict1= {
110001 : "Delhi",
400008 : "Mumbai",
700052 : "Kolkata",
600001 : "Chennai",
}
# Retrieving values associated with keys
print (Dict1[110001])
print (Dict1[400008])
print (Dict1[600001])
```

Output:

```
Delhi
Mumbai
Chennai
```

An alternate way is to use **get()** function with key to retrieve associated value from the dictionary. This function can be used as follows:

Example 8.6:

```
# Creating a dictionary
Dict1= {

110001 : 'Delhi',
400008 : "Mumbai",
700052 : "Kolkata",
600001 : "Chennai",
}
# Retrieving values associated with keys
print (Dict1.get(110001))
print (Dict1.get(400008))
print (Dict1.get(600001))
```

Output:

```
Delhi
Mumbai
Chennai
```

If square brackets [] are used, KeyError will be raised if a key is not present in the dictionary. The get() function, on the other hand, returns None if the key is not present.

Adding and modifying an item in a Dictionary

Dictionaries are a mutable data type that means element values can be changed. Using an assignment operator with keys, we can add new elements or modify existing elements' value. If the key is already present, then it will change the current value. A new key-value pair is added to the dictionary if the key is not present. The following syntax is used to add a new element to a dictionary.

```
Dictionaryname[keyname]=value
```

Example 8.7:

```
# Adding and modifying dictionary elements
dict1 = {'name': 'Amit', 'age': 30, 'city': 'Mumbai'}
# Print the elements
print (dict1)
# Add a new element
dict1['Post'] = 'Manager'
# After adding new element, print all values
print (dict1)
# modifiying an existing value
```

```
dict1['age'] = 32
# After modifying value, print all values
print (dict1)
```

Output:

```
{'name': 'Amit', 'age': 30, 'city': 'Mumbai'}
{'name': 'Amit', 'age': 30, 'city': 'Mumbai', 'Post': 'Manager'}
{'name': 'Amit', 'age': 32, 'city': 'Mumbai', 'Post': 'Manager'}
```

As we have seen in the above example, if a key is existing in a dictionary, then it modifies or replaces old value with the new value

Deleting elements from a dictionary

Any elements can be deleted from the dictionary using its associated keys. You may use the **del** operator to delete a component that is present in the dictionary. The downside that can be thought of using this is that an exception is raised if the key is not present. The following syntax is used to delete an element from a dictionary:

```
del Dictionaryname[keyname]
```

The following example deletes an element such as **age** with the associated value from the dictionary

Example 8.8:

```
# Adding and modifying dictionary elements
dict1 = {'name': 'Amit', 'age': 30, 'city': 'Mumbai'}
# Add a new element
dict1['Post'] = 'Manager'
# Print the elements
print ("Available Elements are:", dict1)
# After adding new element, print all values
```

```
print ("Before Deletion:",dict1)
# deleting an element age
del dict1['age']
# Values after deleted an element
print ("After Deletion:", dict1)
```

Output:

```
Available Elements are: {'name': 'Amit', 'age': 30, 'city':
'Mumbai', 'Post': 'Manager'}
Before Deletion: {'name': 'Amit', 'age': 30, 'city': 'Mumbai',
'Post': 'Manager'}

After Deletion: {'name': 'Amit', 'city': 'Mumbai', 'Post':
'Manager'}
```

The **del** operator can even completely delete the dictionary such as:

```
del dict1
```

After deleting dictionary if we try to access it will cause an error because **dict1** no longer exists.

An alternative method is to delete a key and its corresponding value from a dictionary using the **pop()** function. The advantage of using the **del** operator is that if tried to delete a non-existent element, it can print the desired value or message. Second, in addition to completing a basic delete operation, it also returns the value of a key that is being deleted:

Example 8.9:

```
# Example of deleting dictionary element using pop() function
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
# Printing dictionary before element deletion
print ("The dictionary before performing deletion is : ", D1)
# Using pop() function to delete an element such as John
del_value = D1.pop('John')
# Printing dictionary after element deletion
print ("The dictionary after deletion is : ", D1)
print ("The removed key's value is : ", del_value)
# Again deleting an element which is not present
# assigns 'Not available' to del_value
del_value1 = D1.pop('Jai', 'Not available')
# Printing dictionary after element deletion
print ("The dictionary after remove is : ", D1)

print ("The removed key's value is : ", del_value1)
```

Output:

```
The dictionary before performing deletion is : {'Amit': 20,
'Sumit': 25, 'John': 28, 'David': 29, 'Bob': 30}
The dictionary after deletion is : {'Amit': 20, 'Sumit': 25,
'David': 29, 'Bob': 30}
The removed key's value is : 28
The dictionary after remove is : {'Amit': 20, 'Sumit': 25, 'David':
29, 'Bob': 30}
The removed key's value is : Not available
```

The clear () function

Using the **clear ()** function, we can delete all elements from the dictionary at once. Following example deletes all the entries from the dictionary

Example 8.10:

```
# Example of deleting all element using clear () function
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
# Printing dictionary before element deletion
print ("The dictionary before performing deletion is : ", D1)
# Deleting all elements
D1.clear()
# Printing dictionary after element deletion
print ("The dictionary after remove is : ", D1)
```

Output:

```
The dictionary before performing deletion is : {'Amit': 20,
'Sumit': 25, 'John': 28, 'David': 29, 'Bob': 30}
```

```
The dictionary after remove is : {}
```

Sorting elements in a dictionary

A key and its corresponding value consist of a dictionary element. Sorting for a dictionary may then be carried out as a parameter by using any of the key or value components. The **sorted()** function is used to sort dictionary elements either by keys or by values.

Sorting the dictionary by key

We can sort a dictionary directly using the built-in **sorted()** function. This can be achieved by passing the dictionary itself and a function specifying the key parameter that the sorting is to be performed based on:

Example 8.11:

```
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
print (type(D1))
# Printing dictionary before sorting
print ("The original dictionary is: ", D1)
# Sorting by key
D2= dict(sorted(D1.items(), key=lambda x: x[0]))
print ("After sorting by key: ", D2)
print (type(D2))
```

Output:

```
'dict'>
```

```
The original dictionary is: {'Amit': 20, 'Sumit': 25, 'John': 28,
'David': 29, 'Bob': 30}
After sorting by key: {'Amit': 20, 'Bob': 30, 'David': 29, 'John':
28, 'Sumit': 25}
'dict'>
```

Here, the function **D1.items()** returns a list of tuples containing the keys and their respective values. For a particular item tuple, the **lambda** function returns the key (0th element of the dictionary). Once these are passed to the **sorted()** function, a sorted sequence is returned, which is then typed into a dictionary.

In versions of Python 3.6+, this function can be used as it treats dictionaries as ordered sequences. For older versions, we may substitute the lambda function from the operator module with the **itemgetter()** as follows:

Example 8.12:

```
from operator import itemgetter
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
print (type(D1))
# Printing dictionary before sorting
```

```
print ("The original dictionary is: ", D1)
# Sorting by key
D2= dict(sorted(D1.items(), key=itemgetter(0)))
print ("After sorting by key: ", D2)
print (type(D2))
```

Output:

```
'dict'>
The original dictionary is: {'Amit': 20, 'Sumit': 25, 'John': 28,
'David': 29, 'Bob': 30}
After sorting by key: {'Amit': 20, 'Bob': 30, 'David': 29, 'John':
28, 'Sumit': 25}
'dict'>
```

Sorting dictionary by value

It is like sorting a dictionary by value to sorting by key. The only distinction is that the parameter based on which the sorting will be carried out is the value component of this kind's corresponding element. The following example sorts the dictionary according to the value parameter:

Example 8.13:

```
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
print (type(D1))
# Printing dictionary before sorting
print ("The original dictionary is: ", D1)
# Sorting by value
D2= dict(sorted(D1.items(), key=lambda x: x[1]))
print ("After sorting by key: ", D2)
print (type(D2))
```

Output:

```
'dict'>
The original dictionary is: {'Amit': 20, 'Sumit': 25, 'John': 28,
'David': 29, 'Bob': 30}
After sorting by key: {'Amit': 20, 'Sumit': 25, 'John': 28, 'David':
29, 'Bob': 30}
'dict'>
```

In the preceding example, the dictionary **D1** is sorted according to the values returned by the lambda function value for **x** element). We use the following code for the older versions of Python:

Example 8.14:

```
from operator import itemgetter
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
print (type(D1))
# Printing dictionary before sorting
print ("The original dictionary is: ", D1)
# Sorting by value
D2= dict(sorted(D1.items(), key=itemgetter(1)))
print ("After sorting by key: ", D2)
print (type(D2))
```

Output:

```
'dict'>
The original dictionary is: {'Amit': 20, 'Sumit': 25, 'John': 28,
'David': 29, 'Bob': 30}
After sorting by key: {'Amit': 20, 'Sumit': 25, 'John': 28, 'David':
29, 'Bob': 30}
'dict'>
```

Sorting dictionary in reverse order

The **sorted()** function uses another argument known as reverse. This may be used to determine the order in which the sorting should be conducted. The sorting takes place in reverse order (descending) if **True** is passed. And if **False** (default) is passed, the sorting will be performed in ascending order.

Example 8.15:

```
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
# Printing dictionary before sorting
print ("The original dictionary is: ", D1)
# Sorting by key in reverse order
D2= dict(sorted(D1.items(),reverse=True, key=lambda x: x[0]))
print ("After Reverse order Sorting by key: ", D2)
# Sorting by value in reverse order
D3= dict(sorted(D1.items(),reverse=True, key=lambda x: x[1]))
print ("After Reverse order Sorting by value: ", D3)
```

Output:

```
The original dictionary is: {'Amit': 20, 'Sumit': 25, 'John': 28,
'David': 29, 'Bob': 30}
After Reverse order Sorting by key: {'Sumit': 25, 'John': 28,
'David': 29, 'Bob': 30, 'Amit': 20}
After Reverse order Sorting by value: {'Bob': 30, 'David': 29,
'John': 28, 'Sumit': 25, 'Amit': 20}
```

Iterating over a Dictionary

The dictionary itself is not an iterable object, the functions **keys()** and values return iterable view objects that can be used for dictionary iteration. To traverse all the key-value pairs, we can use **for** loop. A list of tuples is returned by the **items()** function, each tuple being a key and value pair:

Example 8.16:

```
# Creating a dictionary
Dict1= {0 : 'Zero', 1 : 'One', 2 : 'Two', 3 : 'Three', 4 :'Four', 5:
'Five'}
# Printing dictionary
print ("The original dictionary is: ", Dict1)
# Iterating with for loop
print ('Iterating with for loop:')
for t in Dict1.items():
print (t)
```

Output:

```
The original dictionary is: {0: 'Zero', 1: 'One', 2: 'Two', 3:
'Three', 4: 'Four', 5: 'Five'}
Iterating with for loop:
(0, 'Zero')
(1, 'One')
(2, 'Two')
(3, 'Three')
(4, 'Four')
(5, 'Five')
```

We can also iterate dictionary by storing key and value out of each pair in two separate variables such as:

Example 8.17:

```
# Creating a dictionary
Dict1= {0 : 'Zero', 1 : 'One', 2 : 'Two', 3 : 'Three', 4 :'Four', 5:
'Five'}
# Printing original dictionary
print ("The original dictionary is: ", Dict1)
# Iterating with for loop and storing key and value separately
print ('Iterating with storing separate values:')
for key,value in Dict1.items():
print (key,value)
```

Output:

```
The original dictionary is: {0: 'Zero', 1: 'One', 2: 'Two', 3:
'Three', 4: 'Four', 5: 'Five'}
Iterating with storing separate values:
0 Zero
1 One
2 Two
3 Three
4 Four
5 Five
```

Another way of dictionary iteration is to use the **key()** function. Using **key()** function, the associated value can be obtained as follows:

Example 8.18:

```
# Creating a dictionary
Dict1= {0 : 'Zero', 1 : 'One', 2 : 'Two', 3 : 'Three', 4 :'Four', 5:
'Five'}
# Printing original dictionary
print ("The original dictionary is: ", Dict1)
# Iterating with key() function

print ('Iterating using key() function:')
for k1 in Dict1.keys():
print (k1, Dict1.get(k1))        #or print(k1, Dict1[k1])
```

Output:

```
The original dictionary is: {0: 'Zero', 1: 'One', 2: 'Two', 3:
'Three', 4: 'Four', 5: 'Five'}
Iterating using key() function:
0 Zero
1 One
2 Two
3 Three
4 Four
5 Five
```

Nested dictionaries

As we have seen earlier, that unordered set of elements is known as a dictionary. It contains various elements in the key-value pairs embedded in **{}** curly brackets:

```
Dictionary = {'key1' : 'value1','key2': 'value2'}
```

If a dictionary is stored inside another dictionary is known as a nested dictionary. It is like nested records and nested structures used in other languages. In other words, we can say that it is a collection of dictionaries into one single dictionary. It can be represented as:

```
Nested_Dict1 = {
'Dict1': {'key1': 'value1'},
'Dict2': {'key1': 'value1'},
'Dict3': {'key1': 'value1'}
}
```

Here, the **Nested_Dict1** is a nested dictionary with the dictionary and Dictionary and **Dict3** are separate dictionaries that contain their key-value pairs. The following code shows the creation and accessing elements from nested dictionaries:

Example 8.19:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
}
# Printing complete nested dictionary
print (Emp)
# Printing values of D1 dictionary

print ('-----Details of D1 dictionary-------')
print (Emp['D1']['name'])
print (Emp['D1']['age'])
print (Emp['D1']['Post'])
# Printing values of D2 dictionary
print ('-----Details of D2 dictionary-------')
print (Emp['D2']['name'])
print (Emp['D2']['age'])
print (Emp['D2']['Post'])
# Printing values of D3 dictionary
print ('-----Details of D3 dictionary-------')
print (Emp['D3']['name'])
print (Emp['D3']['age'])
print  (Emp['D3']['Post'])
```

Output:

```
{'D1': {'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D2':
{'name': 'Joy', 'age': '25', 'Post': 'Clerk'}, 'D3': {'name': 'David',
'age': '35', 'Post': 'HR'}}
-----Details of D1 dictionary-------
Amit
30
Manager
-----Details of D2 dictionary-------
Joy
25
Clerk
-----Details of D3 dictionary-------
David
35

HR
```

Adding another dictionary to the existing nested dictionary

The following code shows an example of adding a subdictionary to the existing nested dictionary. For adding a new dictionary, we have to use element by its key and assign an associated value:

Example 8.20:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
}
# Printing dictionary elements before adding a sub dictionary
print ('Nested Dictionary before adding a sub dictionary:', Emp)
# Adding a sub dictionary
Emp['D4'] = {'name': 'Bob', 'age': '32', 'Post': 'Manager'}
# Printing dictionary elements after adding a sub dictionary
print ('Nested Dictionary after adding a sub dictionary:', Emp)
```

Output:

```
Nested Dictionary before adding a sub dictionary: {'D1':
{'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D2': {'name':
'Joy', 'age': '25', 'Post': 'Clerk'}, 'D3': {'name': 'David', 'age': '35',
'Post': 'HR'}}
Nested Dictionary after adding a sub dictionary: {'D1': {'name':
'Amit', 'age': '30', 'Post': 'Manager'}, 'D2': {'name': 'Joy', 'age':
'25', 'Post': 'Clerk'}, 'D3': {'name': 'David', 'age': '35', 'Post':
'HR'}, 'D4': {'name': 'Bob', 'age': '32', 'Post': 'Manager'}}
```

Updating nested dictionary items

Nested dictionary elements can easily be updated. Simply refer to the item by the key and assign a new value. If the key is already in the dictionary, the value of that key is replaced by a new one. Following example updates the values of **D1** dictionary inside the **Emp** dictionary:

Example 8.21:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
}
# Printing dictionary elements before update
print ('Nested Dictionary before update:', Emp)
# Updating values of D1
Emp['D1'] = {'name': 'Bob', 'age': '32', 'Post':'Director'}

# Printing dictionary elements after update
print ('Nested Dictionary after updating values:', Emp)
```

Output:

```
Nested Dictionary before update: {'D1': {'name': 'Amit', 'age':
'30', 'Post': 'Manager'}, 'D2': {'name': 'Joy', 'age': '25', 'Post':
'Clerk'}, 'D3': {'name': 'David', 'age': '35', 'Post': 'HR'}}
Nested Dictionary after updating values: {'D1': {'name': 'Bob',
'age': '32', 'Post': 'Director'}, 'D2': {'name': 'Joy', 'age': '25',
'Post': 'Clerk'}, 'D3': {'name': 'David', 'age': '35', 'Post': 'HR'}}
```

Deleting elements from a nested dictionary

We can use **del** operator to delete elements from the nested dictionary. The following example deletes the value of age according to given key from **D2** dictionary:

Example 8.22:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
}
# Printing dictionary elements before delete
print ('Nested Dictionary before delete:')
print (Emp)
# Deleting element from D2
del Emp['D2']['age']
# Printing dictionary elements after Delete
print ('Nested Dictionary after Deleting values:')
print (Emp['D2'])
```

Output:

```
Nested Dictionary before delete:
{'D1': {'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D2':
{'name': 'Joy', 'age': '25', 'Post': 'Clerk'}, 'D3': {'name': 'David',
'age': '35', 'Post': 'HR'}}
Nested Dictionary after Deleting values:
{'name': 'Joy', 'Post': 'Clerk'}
```

Deleting a dictionary from nested dictionary

We can delete a complete subdictionary from nested dictionary using **del** operator same as elements. Following example deletes **D2** sub dictionary from the nested dictionary

Example 8.23:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
}
# Printing dictionary elements before Deleting a sub dictionary
print ('Nested Dictionary before delete:')
print (Emp)
# Deleting D2
del Emp['D2']
# Printing dictionary after Deleting a sub dictionary
print ('Nested Dictionary after Deleting D2:')
print (Emp)
```

Output:

```
Nested Dictionary before delete:
{'D1': {'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D2':
{'name': 'Joy', 'age': '25', 'Post': 'Clerk'}, 'D3': {'name': 'David',
'age': '35', 'Post': 'HR'}}
Nested Dictionary after Deleting D2:
{'D1': {'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D3':
{'name': 'David', 'age': '35', 'Post': 'HR'}}
```

Iterating through a nested dictionary

We can traverse through each element in a nested dictionary using **for** loops such as:

Example 8.24:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
}
# Iterating dictionary elements using for loop
for Dict_id, value in Emp.items():
print ("\n Dict ID:", Dict_id)
for key in value:
print (key + ':', value[key])
print ("------------------")
```

Output:

```
Dict ID: D1
name: Amit
```

```
age:  30
Post:  Manager
------------------
Dict  ID:  D2
name:  Joy
age:  25
Post:  Clerk
------------------
Dict  ID:  D3
name:  David
age:  35

Post:  HR
------------------
```

In the previous example, the first loop returns all the keys in the nested dictionary It consists of the **Dict ID** of each employee. We have used these IDs to unpack the values of each employee. The second loop goes through the values of each employee. Then, it returns all the keys **Post** of each employee's dictionary. Then the keys of the employees and the associated values are printed.

Built-in dictionary functions

There are various built-in functions and methods available in Python that are used to perform various operations on dictionaries. The following are some important functions:

len () This function returns the total length of the dictionary elements. This would be equal to the number of elements in the dictionary. It can be used as follows:
len(dict)

str() This function produces a printable string representation of a given dictionary. It has the following syntax:
str(dict)

type () The function **type()** returns the type of the passed parameter. If the passed parameter is a dictionary, then it would return type as a dictionary:
type(dict)

Example 8.25:

```
# Built-in dictionary functions
# Creating a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29,
"Bob" :30}
# Finding the length of the dictionary
print ("The length of dictionary is : ", len(D1))
# String representation of dictionary
D2=str(D1)
print ("The String representation of dictionary is : ", D2)
print (type(D2))
print ('The type of D1 is:', type(D1))
```

Output:

```
The length of dictionary is : 5
The String representation of dictionary is : {'Amit': 20, 'Sumit':
25, 'John': 28, 'David': 29, 'Bob': 30}
'str'>
The type of D1 is: 'dict'>
```

cmp () The function **cmp** compares two dictionary elements based on key and values. Based on whether the two dictionaries being compared are similar or not, it will return either 1, 0, or -1. This function takes two dictionaries as arguments for comparison. If **D1** and **D2** are two dictionaries, then following result will be produced based on the comparison. In Python 3.x, this feature is not available.

if D1 = then **cmp (D1, D2)** returns **0**

if D1 < then **cmp (D1, D2)** returns **-1**

if D1 > then **cmp (D1, D2)** returns **1**

Example 8.26:

```
D1 = {'name': 'Amit', 'age': 25};
D2 = {'name': 'Amit', 'age': 25};
D3 = {'name': 'Sumit', 'age': 20};
D4 = {'name': 'Rahul', 'age': 30};
print ("Return Value : %d" % cmp(D1, D2))
print ("Return Value : %d" % cmp(D2, D3))
print ("Return Value : %d" % cmp(D1, D4))
```

Output:

```
Return Value : 0
Return Value : 1
Return Value : -1
```

any () This function checks whether any key is **True** or not. It takes an iterable parameter such as dictionary. If all keys are **False** or the dictionary is empty, it returns If at least one key is true, it returns

Example 8.27:

```
# 0 or False
D1 = {0: 'It is False'}
print (any(D1))
# 1 or True
D2 = {0: 'It is False', 1: 'It is True'}
print (any(D2))
# Both keys are false
D3 = {0: 'It is False', False: 'It is False'}
print (any(D3))
# Empty Dictionary
D4 = {}
print (any(D4))
# All are True
D5 = {True: 'It is True', 1: 'It is True'}
print (any(D5))
```

Output:

False

```
True
False
False
True
```

all () When all elements are **True** in the given dictionary, this function returns If not, **False** will return it. In other words, If all keys are **True** or the dictionary is empty, **True** is returned by **all()** function. Else, like all other cases, this returns

Example 8.28:

```
# 0 or False
D1 = {0: 'It is False'}
print (all(D1))
# 1 or True
D2 = {0: 'It is False', 1: 'It is True'}
print (all(D2))
# Both keys are false
D3 = {0: 'It is False', False: 'It is False'}
print (all(D3))
# Empty Dictionary
D4 = {}
print (all(D4))
# 1 or True
D5 = {True: 'It is True', 1: 'It is True'}
```

```
print  (all(D5))
```

Output:

```
False
False
False
True
True
```

The copy()method

The **copy()** method returns a shallow copy of the dictionary. The original dictionary does not change it. The following example illustrates the use of **copy()** method:

Example 8.29:

```
D1 = {'name': 'Python', 'year': 2021}
print ('Original Value of D1:', D1)
# creating a copy of D1
D2 = D1.copy()
print ('Copied Version :', D2)
```

Output:

```
Original Value of D1: {'name': 'Python', 'year': 2021}
Copied Version: {'name': 'Python', 'year': 2021}
```

Formatting dictionaries

We can interpolate strings using dictionaries. They have a syntax in which, between the % operator and the conversion character, we need to include the key in the parentheses. For instance, if we have an **int** stored in a **salary** key and we want to format it as **xxxx.xx** then we can put **%(salary).2fat** in the position we want it to be shown.

Example 8.30:

```
# Creating a dictionary
D1= {'name': 'Amit', 'salary': 30000, 'Post': 'Manager'}
# Printing dictionary
print (D1)
print ('--------------')
print ('----------Formatted dictionary-----------')
# Formatted printing
print ("My Name is: %(name)s" %D1)
print ("My Salary is: %(salary).2f and POST is: %(Post)s "
%D1)
```

Output:

```
{'name': 'Amit', 'salary': 30000, 'Post': 'Manager'}
--------------
----------Formatted dictionary-----------
My Name is: Amit
My Salary is: 30000.00 and POST is: Manager
```

Conclusion

In this chapter, we discussed the concepts and various functionalities of dictionaries. We also discussed how dictionaries could be created in different ways. Various examples and Python code are given for dictionary implementations. Based on these examples, you can write your own code to use dictionaries. In the next chapter, we will discuss the file handling concepts in Python.

Points to remember

Dictionaries are an unordered and changeable collection of data values that hold key-value pairs.

In Python, dictionaries are written inside curly braces

Dictionaries operations are very similar to lists and tuples.

We can shrink or grow a nested dictionary as need.

Multiple choice questions

Which of the following statements create a dictionary in Python?

D = {}

D = {'David':50, 'Alex':35}

D = {35: 'David', 50: 'Alex'}

All of the mentioned

Which one of the following is correct with respect to dictionaries?

In Python, a dictionary can have two same keys with different values.

In Python, a dictionary can have two same values with different keys

In Python, a dictionary can have two same keys or same values but cannot have two same key-value pair

In Python, a dictionary can neither have two same keys nor two same values.

If we write the below code, which of the following will give an error?

Suppose D1={"A":1,"B":2,"C":3, "D":4, “E":5}

print(len(D1))

print(D1.get("B"))

D1["A"]=5

None of these

Which one of the following is correct to empty a dictionary?

Student = {
"name": "Amit",
"age": 30,
"percent": 85

}

del Student

del Student[0:2]

Student.clear()

empty Student

What will be the output of the following Python code?

```
a={1:"A",2:"B",3:"C"}
b=a.copy()
b[2]="D"
print(a)
```

Error, copy () method does not exist for dictionaries

{1: 'A', 2: 'B', 3: 'C'}

{1: 'A', 2: 'D', 3: 'C'}

"None" is printed

Answers

d

b

d

c

b

Questions

What is the difference between a list, tuple, and a dictionary?

What are the features of python dictionaries? Discuss.

What is meant by key-value pairs in a dictionary?

How to create a dictionary in Python? Give an example.

Discuss built-in functions supported by dictionaries?

How does **del** operation work on dictionaries? Give an example.

Explain nested dictionary with suitable example.

Write a Python program to sort dictionary elements in reverse order.

Write a program to input students' names and phone numbers and store them in the dictionary as the key-value pair. Perform the following operations on the dictionary:

Display name and phone number of all the students.

Add a new key-value pair in this dictionary and display the modified dictionary.

Delete a particular student from the dictionary.

Modify the phone number of an existing student.

Check if a student is present in the dictionary or not

Display the dictionary in descending order of names

CHAPTER 9

File Handling

When data must be saved permanently in a file, file handling is essential. A file is a named location on the disc where relevant data is stored. After the program has been terminated, we can access the data that has been saved. The idea of file handling has been extended to various other languages, but the implementation is either complicated or lengthy. However, like other Python concepts, this concept is simple and straightforward. In this chapter, you will see various concepts and functions related to file handling. Besides, we have demonstrated various file operations with the help of examples and codes.

Structure

In this chapter, we will cover the following topics:

The need for file handling

Types of files

Opening and closing files

Reading and writing in files

Various methods of file handling

Objective

The objective of this chapter is to introduce the concept of file handling in Python. After completing this chapter, you should be able to perform various file-related operations. Also, you will be able to write Python codes to manage files on a disk. You will also understand the uses of different file-related methods in Python.

File handing

A file is a collection of bytes used to store information. This information is organized in a particular format, ranging from a simple text file to a complex program executable. Finally, these byte files are converted into binary 1 and 0 for faster computer processing.

Python, like many other programming languages, supports file handling and allows its users to read and write files and perform a variety of other file-related tasks. Python handles files differently depending on whether they are text or binary, which is crucial. Each line of code consists of a series of characters that together form a text file. A unique character called as the EOL or End of Line character, such as the comma, or a newline character, is used to end each line of a file. It signals to the interpreter that the current line has ended and that a new one has started. On most modern file systems, files are divided into three parts:

Details about the file's contents (file name, size, type, and so on).

The file's contents as written by the author or writer.

End of A special character that denotes the file's termination.

Need of file handling

In programming, we need a particular piece of input data to be generated multiple times. It is not always sufficient to simply show data on the console. The data to be displayed can be huge. Only a limited amount of data can be displayed on the console; additionally, since memory is volatile, it is challenging to recover programmatically generated data repeatedly. If we need to store something, we can do so on the local file system, which is volatile and accessible at any time. This necessitates the use of Python's file handling functionality. File handling in Python allows us to use our Python program to create, edit, read, and delete files on the local file system. On a file, you can perform the following operations:

Creating a new file

Opening previously saved file

Closing an opened file

Reading from a file

Writing into a file

File deletion

File path

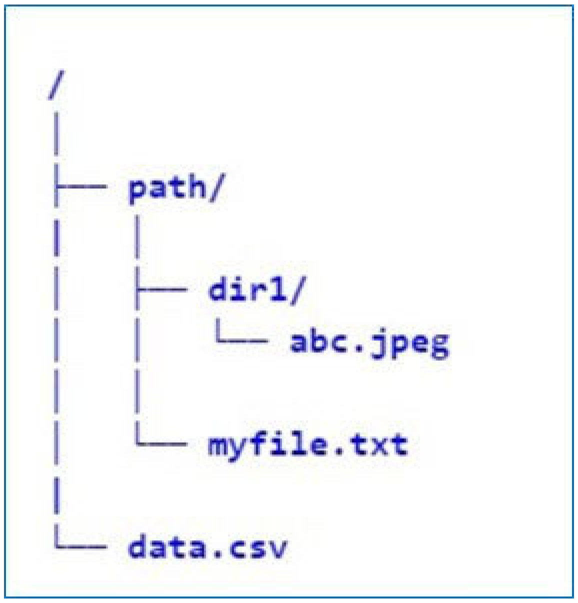
A file path is required when accessing a file on an operating system. A file path is a string that defines a file's location. It is divided into three major segments:

Directory The location of a file or folder on a file system, separated by a forward slash in Linux or Unix or a backslash in Windows.

File The file's real name.

It defines the file type.

The following figure shows a file path example. Assume you needed to access the **abc.jpeg** file, and your current location was the same as the path. To get to the file, you must first navigate to the path folder, then to the **dir1** directory, and finally to the **abc.jpeg** file. **Path/dir1/** is the folder path. **Abc** is the name of the file. **.jpeg** is the file extension. So **path/dir1/abc.jpeg** is the complete path:



```
/
|
├── path/
|   |
|   ├── dir1/
|   |   └── abc.jpeg
|   |
|   └── myfile.txt
|
└── data.csv
```

Figure 9.1: *Example of file path*

Opening a file

To perform reading and writing operations on a file, it needs to be opened first. To open a file in Python, the user needs to create a file object associated with a physical file. In addition, the **open()** function is used to open a file in Python. The **open()** function in Python takes two arguments: the file name and the access mode. The function returns a file object, which can be used for reading, writing, and other operations. The following syntax is used to open a file in Python.

Syntax:

File_object_Name = open(, ,)

Various modes, such as read, write, and append are available for accessing the files. The access mode to open a file is defined as follows:

follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows:

follows: follows:
follows: follows:

follows: follows:
follows: follows:
follows: follows:
follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

Table 9.1: *File access modes*

Example 9.1:

```
fileobj = open("file1.txt","r") #Opening file1.txt in read mode

if filobj:
print ("File opened.")
```

Output:

```
File opened.
```

We passed file name as the first argument and opened the file in read mode with **r** as the second argument in the above example. The **fileobj** holds the file object, and if the file is successfully opened, the print statement is executed.

We have stored file1.txt in the same directory.

Closing a file

We can perform any operation on the file using the file system that is currently open in Python; thus, it is best to practice closing the file once all processes have been completed. After we've completed all of the files' operations, we'll use the **close()** method to close it. When the **close()** method on a file object is called, any unwritten data is deleted.

Syntax:

```
File_Object_Name.close()
```

Example 9.2:

```
fileobj = open("file1.txt","r") #Opening file1.txt in read mode

if fileobj:
print ("File opened.")
fileobj.close()              # Closing file1.txt
```

Output:

File opened.

Writing into a file

A string is written to a file using the **write()** method. To write text to a file, we must first open it with the open method and one of the access modes mentioned below:

w: If the file already exists, it will be overwritten. The file pointer is located at the start of the file.

a: The file will be appended to the current one. The file pointer is at the file's end. If no such file exists, it creates one.

Example 9.3:

```
# opening f1.txt in write mode. If this file is not present, a
new file will be created.
Obj1= open ("f1.txt", "w")
# writing contents in this file
Obj1.write("Python programming \n")
Obj1.write("It is a programming language \n")
Obj1.write("It supports file handling \n")

# closing this file
```

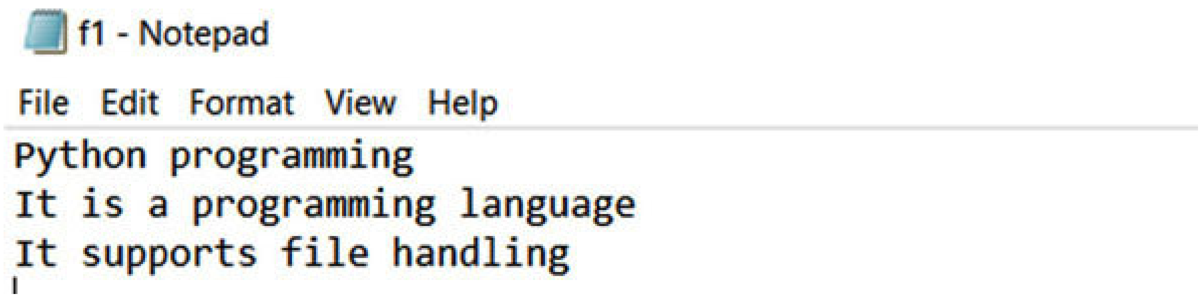
Obj1.close()

Output:

File Name: f1.txt

```
Python programming
It is a programming language
It supports file handling
```

Screenshot of f1.txt



```
Python programming
It is a programming language
It supports file handling
```

Figure 9.2: *Screenshot of f1.txt*

Example 9.4:

```
# opening f1.txt in Append mode. If this file is not present, a new file will be created.
Obj1= open ("f1.txt", "a")
# Appending contents in this file
```

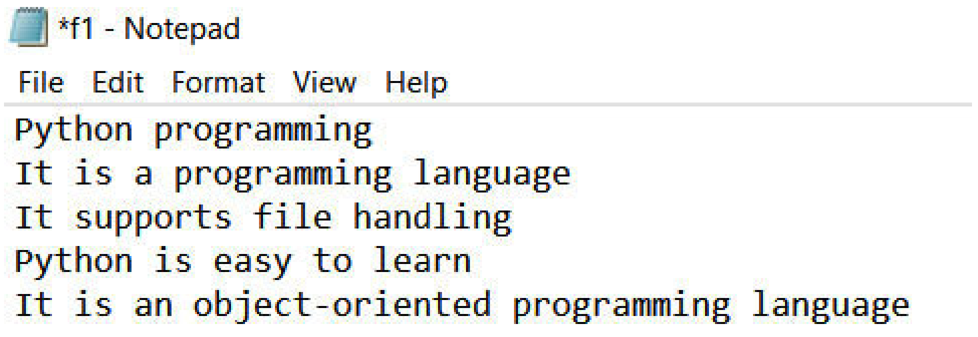
```
Obj1.write("Python is easy to learn \n")
Obj1.write("It is an object-oriented programming language")

# closing this file
Obj1.close()
```

Output:

File Name: f1.txt

```
Python programming
It is a programming language
It supports file handling
Python is easy to learn
It is an object-oriented programming language
```

Screenshot of f1.txt



```
Python programming
It is a programming language
It supports file handling
Python is easy to learn
It is an object-oriented programming language
```

Figure 9.3: Screenshot of f1.txt

writelines() method

The **writelines()** method is a Python built-in method for writing a list of strings or list items to a file.

Syntax:

```
fileobject.writelines(list_of_items)
```

Example 9.5:

```
# file creation
fobj = open("f2.txt", "w")
# writing multiple strings
fobj.writelines(["Python programming ","Programming book \n"])
# list
list1 = ["Krishna ","Brijesh ","Arvind ","Joy "]
# writing list
fobj.writelines(list1)
# closing the file
fobj.close()

# again opening file in read mode
```

```
fobj = open("f2.txt", "r")
print (fobj.read())
# closing the file
fobj.close()
```

Output:

```
Python programming book
Krishna Brijesh Arvind Joy
```

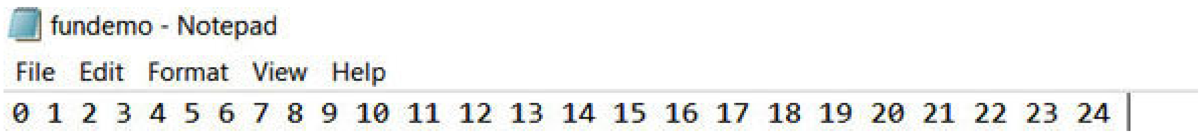
Writing numbers to a file

Python reads files as strings by default, and the **write()** function expects only to write strings. If we want to write numbers to a file, we'll need to use the **str()** function to cast them as strings.

Example 9.6:

```
n= range(0, 25)
fn = "fundemo.txt"
# open file to write
obj= open(fn, 'w')
for i in n:
obj.write(str(i) + " ")
obj.close() #Close the file
```

Output:



```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

***Figure 9.4:** fundemo output*

Reading a file

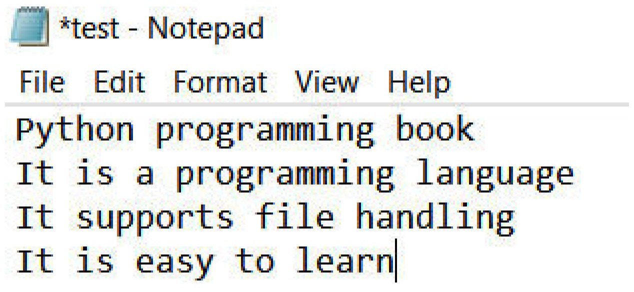
In Python, there are several ways to read a text file. The **read()** method can be used to retrieve a string that includes all the characters in a file.

Syntax:

File_Object.read()

Before reading a file, it is required to open that file in (read) mode.

Example The contents of the **test.txt** file are shown as follows:



```
Python programming book
It is a programming language
It supports file handling
It is easy to learn
```

Figure 9.5: *test.txt file*

```
# opening test.txt file in read mode.
Obj1= open ("test.txt", "r")
# reading whole content
Data=Obj1.read()
print (Data)
# closing this file
Obj1.close()
```

Output:

```
Python programming book.
It is a programming language.

It supports file handling.
It is easy to learn.
```

Using a certain number of characters is another way to read a text. The interpreter will, for example, read the first twenty characters of stored data and return it as a string using the following code:

Example 9.8:

```
# opening test.txt file in read mode.
Obj1= open ("test.txt", "r")
```

```
# reading whole content
Data=Obj1.read(20)
print (Data)
# closing this file
Obj1.close()
```

Output:

```
Python programming b
```

readline() method

The **readline()** function in Python makes it easy to read a file line by line. The **readline()** method reads the file's lines starting at the beginning, so if we call it three times, we'll get the first three lines of the file. This method will return a string of characters containing a single line of data from the file each time it is called:

```
Obj1= open ("test.txt", "r")
print Obj1.readline()
```

This will return the file's first line, as follows:

Python programming book.

If we just wanted to return the file's fourth line, we'd do it like this:

```
Obj1= open ("test.txt", "r")
print Obj1.readline(4)
```

It is easy to learn.

Example 9.9:

```
# opening test.txt file in read mode.
Obj1= open ("test.txt", "r")
# reading first line
Line1=Obj1.readline()
# reading second line
Line2=Obj1.readline()

# printing lines
print (Line1)
print (Line2)
# closing this file
Obj1.close()
```

Output:

```
Python programming book.
It is a programming language.
```

readlines() method

If we wanted to get back every line in the file, separated properly. The same function will be used but in a different format. The **FileObject.readlines()** method is used to accomplish this. It returns a list of lines up to the **end of the file**

Example 9.10:

```
# opening test.txt file in read mode.
Obj1= open ("test.txt", "r")
# reading whole content
Data=Obj1.readlines()
print (Data)
# closing this file
Obj1.close()
```

Output:

```
['Python programming book.', 'It is a programming language.',
'It supports file handling.', 'It is easy to learn.']
```

Reading file through a loop

The loop-over method can be used to read all the lines from a file in a more memory accessible and fast manner. The corresponding code is simple and easy to read, which is a benefit of using this method.

Example 9.11:

```
# opening test.txt file in read mode.
Obj1= open ("test.txt", "r")
# reading content through loop
for x in Obj1:
print (x)
# closing this file
Obj1.close()
```

Output:

```
Python programming book.
It is a programming language.
It supports file handling.
It is easy to learn.
```

Reading numbers from a file

To read numbers from a file, **read()** method is used. Firstly, we have to open a file in reading mode then; the numbers can be easily read. The following example reads stored numbers from the **fundemo.txt** file.

Example 9.12:

```
fn = "fundemo.txt"
# open file to write
obj= open(fn, 'r')
data=obj.read()
print (data)
obj.close() #Close the file
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24
```

Creating a new file

With the help of the **open()** function, you can create a new file by using one of the following access modes.

x: it creates a new file with the name you specify. It throws an error if a file with the same name already exists.

a: If no such file exists, it creates a new one with the specified name. If the file with the given name already exists, it appends the content to it.

w: If no such file exists, it creates a new one with the specified name. If a file exists, it overwrites.

Example 9.13:

```
#creating a new empty file
fileobj= open("myfile1.txt", "x")
#close this file
fileobj.close()
```

The preceding code will create a new file in the current working directory. If you want your file to be created at an absolute path on the disc, you can also give it a complete path.

Example 9.14: Creating a new file with an existing name

```
#creating a new file, but myfile1.txt is already exist
fileobj= open("myfile1.txt", "x")
```

Output:

```
Traceback (most recent call last):
File "main.py", line 2, in
fileobj= open("myfile1.txt", "x")
FileExistsError: [Errno 17] File exists: 'myfile1.txt'
```

In the previous example, we tried to create a new file However, this file already exists in the same directory. We get

The file object attributes

You can get different file-related details once a file is opened and you have one file object. The following is a list of all file object attributes:

It is a **bool** value that indicates the current state of the file object.

The encoding details of the file.

It returns the file mode of the file in which the file is opened.

It returns the file name.

Example 9.15:

```
# Opening a file in read mode
Fileobj = open("fundemo.txt", "r")
print ("File Name:", Fileobj.name)
print ("File closed or not : ", Fileobj.closed)
print ("File Mode : ", Fileobj.mode)
print ("File Encoding", Fileobj.encoding)
```

Output:

```
File Name: fundemo.txt
File closed or not : False
File Mode : r
File Encoding UTF-8
```

File positions

When we use Python to open a file for reading, we get a file handle those points to the file's beginning. The pointer always points to the end of the reading as we read from the file, and the next read will begin there. The following methods are used to get file position and change position, respectively:

The **tell()** method returns the current file position; in other words, the next read or write will occur at that many bytes from the file's beginning.

The **seek(offset[, from])** method is used to adjust the file's current location. The number of bytes to be moved is defined by the offset statement. The **from** statement determines the starting point for the bytes to be transferred from.

If from is set to 0 in the **seek()** function, the file's beginning will be used as the reference position, 1 the current position will be used as the reference position, and 2 the file's end will be used as reference position.

Example 9.16:

```
# Creating a file`
Obj1= open ("testfile.txt", "w")
# writing contents in this file
Obj1.write("Python programming \n")
Obj1.write("It is a programming language \n")
Obj1.write("It supports file handling \n")
Obj1.write("I like Python.")

# closing this file
Obj1.close()

# Opening a file in read mode

Fileobj = open("testfile.txt", "r")

#initially the filepointer is at 0
print ("Pointer Position: ",Fileobj.tell())      #0

#reading the content of the file

Line1= Fileobj.readline()
print (Line1)

print ("Pointer Position:", Fileobj.tell())          # 20
Line2= Fileobj.readline()
```

```
print (Line2)

print ("Pointer Position:", Fileobj.tell())         # 50
Line3= Fileobj.readline()
print (Line3)

print ("Pointer Position:", Fileobj.tell())         # 77
Line4= Fileobj.readline()
print (Line4)
print ("Pointer Position:", Fileobj.tell())         # 91
```

Output:

```
Pointer Position: 0
Python programming
Pointer Position: 20
It is a programming language

Pointer Position: 50
It supports file handling
Pointer Position: 77
I like Python.
Pointer Position: 91
```

Example 9.17:

```
# Creating a file`
Obj1= open ("testfile1.txt", "w")

# writing contents in this file
Obj1.write("Python programming \n")
Obj1.write("It is a programming language \n")
Obj1.write("It supports file handling \n")
Obj1.write("I like Python.")

# closing this file
Obj1.close()

# Opening a file in read mode
Fileobj = open("testfile1.txt", "r")
#initially the filepointer is at 0
print ("Pointer Position: ",Fileobj.tell())

#reading the content of the file

Line1= Fileobj.readline()
print (Line1)

print ("Pointer Position:", Fileobj.tell())
Line2= Fileobj.readline()
print (Line2)
#changing the file pointer location to 10.
```

```
Fileobj.seek(10);

print ("Pointer Position:", Fileobj.tell())
Line3= Fileobj.readline()
print (Line3)

print ("Pointer Position:", Fileobj.tell())

Line4= Fileobj.readline()
print (Line4)
print ("Pointer Position:", Fileobj.tell())
#changing the file pointer location to 10.
Fileobj.seek(20);
print ("Pointer Position:", Fileobj.tell())
```

Output:

```
Pointer Position: 0
Python programming
Pointer Position: 20
It is a programming language
Pointer Position: 10
gramming
Pointer Position: 20
It is a programming language
```

```
Pointer Position: 50
Pointer Position: 20
```

Python directory operations

A directory is a structure that holds all of the documents, files, and directories associated with it. In Python, the OS module offers functions for communicating with the operating system. This module provides access to various operating-system-specific functions for manipulating processes, files, file descriptors, folders, and other low-level OS features.

Current working directory

getcwd() returns the current working directory's path. This is the location where the operating system converts a relative file name to an absolute file name.

Example 9.18 :

```
#importing os module
import os
#print the current working directory
print (os.getcwd())
```

Output:

```
/home/Krishna
```

List of directories

You can get a directory list from a specific location. You must use the **listdir(location)** function to accomplish this. The function will return a list of strings containing the names of the directories in the specified location if you pass it in the location. The simple **listdir()** function will return the contents of the current directory.

Example 9.19:

```
import os
data = os.listdir() #list of contents from current directory
print (data)

print (os.listdir('/usr')) # list of contents from usr directory
```

Output:

```
['main.py']
['games', 'local', 'include', 'share', 'lib', 'bin', 'src', 'sbin']
```

Creating a directory

The **os.mkdir()** method is used to create a directory. Let's make a new directory called The **os.listdir()** method will then print a list of directories along the path.

Example 9.20:

```
import os
#create directory
os.mkdir("mydir")
print (os.listdir()) # list of contents
```

Output:

```
['mydir', 'main.py']
```

Change directory

To change the directory, we must first import the **os** module and then use the **os.chdir()** method to change our program's base path.

Example 9.21:

```
import os
#change directory
os.chdir('/Users/Krishna/')
#print current working directory
print (os.getcwd())
```

Output:

```
/Users/Krishna
```

Renaming a directory

The **os.rename()** method allows you to rename a folder from one name to another.

Example 9.22:

```
import os
#rename directory
os.mkdir('mydir1')
print (os.listdir(os.getcwd()))

#rename
os.rename("mydir1","mydir2")
print (os.listdir(os.getcwd()))
```

Output:

```
['mydir1', 'main.py']
['mydir2', 'main.py']
```

Delete a directory

The **rmdir()** function is used to remove an already empty directory. The directory will not be removed if it is not empty.

Example 9.23:

```
import os
os.mkdir('mydir1')
print ('Before deletion:',os.listdir(os.getcwd()))

#delete directory
os.rmdir("mydir1")
print ('After deletion:',os.listdir(os.getcwd()))
```

Output:

```
Before deletion: ['mydir1', 'main.py']
After deletion: ['main.py']
```

Renaming and deleting file

The **os** module in Python provides methods for performing file-processing operations, including renaming and removing files. To use this module, you must first import it before calling any related functions. The renaming and deletion of files are the same as the directories. The **os.rename()** function is used to rename a file with a new name and **os.remove()** function can delete a specific file.

Syntax for rename:

```
os.rename(Previous_file_name, new_file_name)
```

Syntax for delete:

```
os.remove(file_name)
```

Example 9.24:

```
# File rename example
import os
```

```
print ('Before rename:',os.listdir(os.getcwd()))

# Rename a file from myfile1.txt to newfile1.txt
os.rename("myfile1.txt", "newfile1.txt")
print ('After rename:',os.listdir(os.getcwd()))
```

Output:

```
Before rename: ['f1.txt', 'myfile1.txt', 'main.py', 'myfile2.txt']
After rename: ['f1.txt', 'newfile1.txt', 'main.py', 'myfile2.txt']
```

Example 9.25:

```
#file delete example
import os

print ('Before Delete:', os.listdir(os.getcwd()))

# Delete a file newfile1.txt
os.remove("newfile1.txt")
print ('After Delete:', os.listdir(os.getcwd()))
```

Output:

```
Before Delete: ['f1.txt', 'myfile1.txt', 'newfile1.txt', 'main.py',
'myfile2.txt']
After Delete: ['f1.txt', 'myfile1.txt', 'main.py', 'myfile2.txt']
```

Binary files

In Python, there are two kinds of files: text and binary. Binary files make up the majority of the files we see on our computers. Working with binary files is easy with Python's tools. Strings of form bytes are used in binary files. It means that you'll get a byte's object back when reading binary data from a file. We must use the same modes (as before) with the letter **b** at the end when working with binary files. Such that Python recognizes that we're dealing with binary files. The **wb** mode is used to write the binary file. Similarly, **rb** and **ab** modes are used to read and append data in binary files, respectively.

Example 9.26: Writing into binary files

```
data = [12,34,100,240,255]
#printing data
print (data)
#converts to bytes
bdata = bytes(data)
print (bdata)
fileobj = open("myfile1.bin", "wb")
fileobj.write(bdata)
fileobj.close()
```

Output:

```
[12, 34, 100, 240, 255]
b'\x0c"d\xf0\xff'
```

Example 9.27: Reading from binary file

```
# Open binary file
fobj = open('myfile1.bin', 'rb')

# read string from binary file
result = fobj.read()

print (result)
```

Output:

```
b'\x0c"d\xf0\xff'
```

Conclusion

In this chapter, we discussed the file handling concepts in Python. We also discussed various file modes in which it can be opened for processing. Multiple examples and Python code are given for file processing, such as read, write, delete, rename, etc. Based on these examples, you can write your code for file processing. In the next chapter, we will discuss the exception handling concept in Python with various examples.

Points to remember

Primary modes of files are read, write, and append.

The **os** module can be used to process various files and directories.

Binary files can be processed, including **b** at the end of file modes.

The **read()** method returns all the data as one string.

Multiple choice questions

To read three characters from a file object Obj, we use ____________

Obj.read(3)

Obj.read()

Obj.readline()

Obj.readlines()

To read the next line of the file from a file object Obj, we use ____________

Obj.read(3)

Obj.read()

Obj.readline()

Obj.readlines()

Assuming file1.txt contains:

This is line 1

This is line 2

This is line 3

This is line 4

Code snippet:

```
f1= open("file1.txt",'a')
print (f1.read())
f1.close()
```

What is the output of the program snippet?

This is line 1

This is line 4

Entire file contents

None, IOError

Which one of the following is not an attribute of the file?

closed

encoding

rename

mode

What is the use of the tell() method in Python?

tells you the current position within the file

tells you the end position within the file

tells you the file is opened or not

none of the mentioned

Answers

a

c

d

c

a

Questions

What do you mean by file handling?

Explain the **open()** function with its syntax in detail.

List out the basic file modes available in Python.

Discuss various file object attributes.

What are the various directory operations available in Python? Explain.

What is a binary file? Write code to read data from a binary file.

What are the uses of **tell()** and **seek()** functions?

How can a file be deleted using a Python code? Discuss.

Write a function in Python to read the content from a text file line by line and display it on the screen.

Write a Python program to count the number of lines in a text file.

CHAPTER 10

Exception Handling, Modules, and Packages

Exception handling enhances your code and helps you avoid potential errors that would cause your application to fail unexpectedly. In this chapter, we will look at how Python handles exceptions. This chapter focuses on Python classes that are highly well-built, as well as the **try-finally** clause. Besides, we have demonstrated various error handling codes with the help of examples. Modules and packages are discussed in this chapter, as well.

Structure

In this chapter, we will cover the following topics:

The basic concept of errors and exceptions

Python exception and its hierarchy

Exception handling

Built-in and user-defined exceptions

Modules

Packages

Objective

The objective of this chapter is to introduce exception handling in Python. After completing this chapter, you should be able to handle exceptions using built-in Python functions. Also, you will be able to write Python codes for handling exceptions. You will also be able to understand the uses of modules and packages in Python.

Errors and exceptions

Error handling improves the robustness of your code, protecting it from possible errors that might cause your application to exit unexpectedly. Errors cannot be dealt with, but Python exceptions can. An error can be a syntax error, but several different types of exceptions can occur during execution, which aren't always inoperable. An Error may indicate crucial issues that a reasonable application should avoid, while an Exception may indicate circumstances that an appropriate application should try to capture. Errors are a type of uncontrolled exception that is unrecoverable, such as an which a programmer should avoid handling. Consider what would happen if you wrote code deployed in production but still terminated due to an exception. The client would be unhappy, so it's best to treat the exception ahead of time and prevent the uncertainty. Syntax errors and exceptions are two distinct types of errors.

Syntax error

Syntax errors, often known as parsing errors, occur when the parser detects a syntactic error in your code. To better understand it, let's look at an example.

Example

```
#Syntax error example
x = 8
y = 10
z = x y
```

Output:

```
File "", line 3
z = x y
^

SyntaxError: invalid syntax
```

When the parser encounters an error while executing the code, the arrow above (in output) indicates it. The failure is caused by the token preceding the arrow. Python will do much of the work for you in resolving such basic errors since it will print the file name and line number where the error occurred.

Exception

An exception is an error that occurs when a program is being executed. Non-programmers understand exceptions as instances that do not follow a general rule. Even if a sentence or expression's syntax is correct, it can still produce an error when executed. Exceptions in Python are errors that are observed during execution but are not always serious. When a Python code throws an error, an exception object is formed. The program will be forced to end suddenly if the code does not explicitly handle the exception. Exceptions are usually ignored by programs, resulting in error messages like this:

Example 10.2:

```
# Example of type error
name= 'amit'
age=30
age+name
```

Output:

Traceback (most recent call last):

```
File "", line 3, in
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Example 10.3:

```
#Example of division by zero exception
100 * (10/0)
```

Output:

```
Traceback (most recent call last):
File "", line 1, in
ZeroDivisionError: division by zero
>
```

Python exceptions come in various forms, and the form is written alongside the message: in the above two cases, the types are **TypeError** and The name of the Python built-in exception is printed in both error strings as the exception type. Based on the type of exception, the remaining portion of the error line contains information about what caused the error.

Handling Exceptions

Python's exception handling is close to Java's. A try block contains the code, which can throw an exception. The **catch** clauses catch exceptions in Java, but statements added by the **except** keyword **catch** exceptions in Python. It is possible to make customized exceptions. It's possible to force an exception to occur using the **raise** statement. If you have some suspicious code that might throw an exception, you can protect your program by putting it in a block. Have an except declaration after the **try:** block, followed by a block of code that as elegantly as possible solves the problem. The syntax for **try....except...else** blocks is as follows.

Syntax:

```
try:
# Our code
except Exception_1:
#If Exception_1 occurs, this block will be executed.
except Exception_2:
# If Exception_2 occurs, this block will be executed.
else:
# If No exception occurs, this block will be executed.
```

The main components of exception handling are as follows:

It will execute the code block in which you expect an error.

It catches the exception, which occurs in a **try** block.

This block of code will be executed if there are no exceptions.

This block of code will always be executed, regardless of whether there is an exception.

There can be several except statements in a single, **try** block.

A generic except clause may also be used to handle any exception.

You may add an **else** clause after the **except** clause(s). This else block runs when no exception occurs in the **try** block.

Example 10.4:

```
#Example of ValueError exception
try:
```

```
my_var = input ("Enter a Number: ")
my_var = int(my_var)
print(my_var)
except ValueError:
print("Exception occurs")
else:
print('No exception occurred')
```

Output:

```
---------------------Run program with following input--------------------------
Enter a Number: 5
5
No exception occurred
---------------------Run program with following input--------------------------
Enter a Number: amit
Exception occurs
```

Example 10.5:

```
#Example of ZeroDivisionError exception
try:
My_var1=int (input ("Enter Value for 1st variable:"))
My_var2=int (input ("Enter Value for 2nd variable:"))
print (My_var1/My_var2)    # It will run if My_var2 is not zero
except ZeroDivisionError:
```

```
# It will execute if My_var2=0
Print ("Zero Division Exception Occurred")
else:
print ('No exception occurred')
```

Output:

```
---------------------Run program with following input--------------------------
Enter value for 1st variable: 12
Enter value for 2nd variable: 2
6.0
No exception occurred
---------------------Run program with following input--------------------------
Enter value for 1st variable:123
Enter value for 2nd variable: 0
Zero Division Exception Occurred
```

Example 10.6:

```
#Example of LookupError exception
try:
x = ['a', 'b', 'c', 'd']
print (x[5])
except LookupError:
print ("Index Error Exception Occurred")

else:
```

```
print ('No exception occurred')
```

Output:

```
Index Error Exception Occurred
```

Single except block

A single **except** statement can be used with a **try** block. This block can catch all types of exceptions. In this type of exception handling, the programmer cannot find the actual reason for the problems.

Example 10.7:

```
try:
var1 = int (input("Input 1st number:"))
var2 = int (input("Input 2nd number:"))
var3 = var1/var2
print ("var1 divided var2 =", var3)
except:
print ("Any Exception.")
else:
print ("Else block.")
```

Output:

```
---------------------Run program with following input--------------------------
Input 1st number:5
Input 2nd number:2
x divided y = 2.5
Else block.
---------------------Run program with following input-------------------------
Input 1st number:5
Input 2nd number:0
Any Exception.
```

Multiple except blocks

For a single **try** block, several **except** blocks are possible. However, only one of the exception clauses will be carried out. Let's look at the Python example of having multiple **except** blocks for a single **try** block. When the interpreter detects an exception, it seems to the **except** blocks that are related to the **try** block. The types of exceptions that these except blocks manage can be defined. The interpreter executes the **except** block when it detects a corresponding exception.

Example 10.8:

```
X1=int (input ("Input 1st value:")
Y1=int (input ("Input 2nd value: "))
try:
print (X1/Y1)     # It will run if Y1 is not zero

print ('10'+10) # It will not print
except TypeError:
print ("Type Exception")
except ZeroDivisionError:
print ("You divided by 0")
```

Output:

```
---------------------Run program with following input--------------------------
Input 1st value: 5
Input 2nd value: 0
You divided by 0
---------------------Run program with following input--------------------------
Input 1st value: 100
Input 2nd value: 25

4.0
Type Exception
```

Multiple exceptions in one except

We can also handle multiple exceptions with a single **except** block using parentheses to do this. The interpreter will return a syntax error if this is not done.

Example 10.9:

```
#Multiple exceptions in one except
X1=int (input ("Input 1st value:"))
Y1=int (input ("Input 2nd value: "))
try:
print (X1/Y1)    # It will run if Y1 is not zero

print ('10'+10) # It will not print
except (TypeError,ZeroDivisionError):
print ("Either Type or Division by zero exception")
```

Output:

```
---------------------Run program with following input-------------------------
Input 1st value: 1
Input 2nd value:0
```

```
Either Type or Division by zero exception
---------------------Run program with following input--------------------------
Input 1st value: 55
Input 2nd value: 11
5.0
Either Type or Division by zero exception
```

else and finally

The keywords **else** and **finally** can be used with the **try** and **except** clauses in Python. During program execution, if an exception occurs in the **try** block, then except block is executed. But if there is no exception found in the **try** block, then **else** block is executed. Another statement is **finally** used in exception handling. All the codes written inside the **finally** block are always executed either when the exception occurs or not. As a result, the error-free **try** block skips the **except** clause and reaches the **finally** block before executing the rest of the code.

Syntax:

```
try:
#Code which we want to run
except:
#This code will run if an exception occurs
else:
#This code will run if no exception occurs
finally:
#This code will always run
```

Example 10.10:

```
#Exception handling with else and finally
a=int (input ('Enter 1st value: '))
b=int (input ('Enter 2nd value: '))
try:
print ('Code in try block')

c=a/b
except ZeroDivisionError:
print ("Except block.")
print ("Division by zero error.")

else:
print ("else block.")
print ("a/b= ", c)
finally:
print ("finally block. Always run.")
```

Output:

```
---------------------Run program with following input--------------------------
Enter 1st value: 200
Enter 2nd value: 13
Code in try block
```

```
else block.
a/b= 15.384615384615385
finally block. Always run.
---------------------Run program with following input--------------------------
Enter 1st value: 200
Enter 2nd value: 0
Code in try block
Except block.
Division by zero error.
finally block. Always run.
```

Raising exceptions

In the sense of exception handling, Python also includes the **raise** keyword. It results in an explicit exception being thrown. Built-in errors are raised implicitly. During execution, however, you can force a built-in or custom exception. For instance, if a program needs 1GB of memory to run, we may raise an exception to prevent the program from running. The following is the syntax for using the **raise** statement.

Syntax:

```
raise Exception_class ()
```

Example 10.11:

```
#raise exception example
try:
val=int(input('Enter a number between 1 to 50 :'))
if val > 50:
raise ValueError(val)
except ValueError:
print (val, "Number is not between 1 to 50")
```

```
else:
print (val, "Number is between 1 to 50")
```

Output:

```
---------------------Run program with following input-------------------------
Enter a number between 1 to 50 :56
56 Number is not between 1 to 50
---------------------Run program with following input-------------------------
Enter a number between 1 to 50 :20
20 Number is between 1 to 50
```

Example 10.12:

```
#raise exception with message
try:
val = int(input("Please enter a positive number: "))
if(val<= 0):
# raise statement with message
raise ValueError("Negative number.")
except ValueError as e:
print (e)
```

Output:

```
Please enter a positive number: -52
Negative number.
```

Example 10.13:

```
#raise exception
try:
x = int (input ("Enter first number:"))
y= int (input ("Enter second number:"))
if y == 0:
raise ArithmeticError(y)
else:
print ("x/y= = ",x/y)
except ArithmeticError as e:
print ("value of y is=",e)
```

Output:

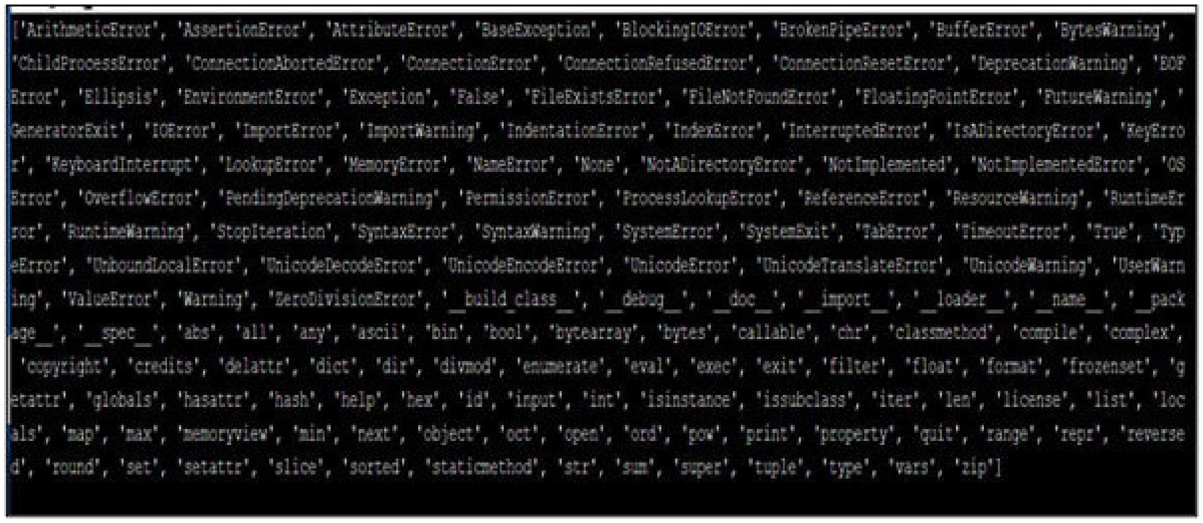
```
---------------------Run program with following input--------------------------
Enter first number:5
Enter second number:2
x/y= = 2.5

---------------------Run program with following input--------------------------
Enter first number:12
Enter second number:0
value of y is= 0
```

Built-in exceptions

When errors occur in Python, a number of built-in exceptions are raised. The **local()** built-in functions can be used to display these built-in exceptions as follows:

```
print (dir(locals()['__builtins__']))
```

Output:



```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOF
Error', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', '
GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyErro
r', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OS
Error', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'ReferenceError', 'ResourceWarning', 'RuntimeEr
ror', 'RuntimeWarning', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'Typ
eError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarn
ing', 'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__pack
age__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'g
etattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'loc
als', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reverse
d', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Figure 10.1: *Built-in exceptions*

Some important built-in exceptions are described below:

If an assert argument fails, this exception is raised.

When an attribute assignment or relation fails, this exception is raised.

When the **input()** function encounters an end-of-file state, this exception is raised.

When a floating-point operation fails, this exception is raised.

It is raised when the **close()** method of a generator is called.

When the imported module cannot be found, this exception is raised.

When the index of a series is out of control, it is raised.

When a key isn't found in a dictionary, this exception is raised.

When the user presses the interrupt key + *C* or this exception is raised.

When an action runs out of memory, this exception is raised.

When a variable isn't found in either local or global scope, this exception is raised.

Abstract techniques were used to raise it.

When a system operation results in a system-related mistake, this exception is raised.

When the product of an arithmetic operation is too high to be expressed, this exception is raised.

When a weak reference proxy is used to access a garbage collection, this exception is raised.

If an error does not fit into any of the other categories, it is raised.

The **next()** function raises this exception to mean that the iterator has no more items to return.

When a parser encounters a syntax error, it raises this exception.

If the indentation is wrong, it is raised.

When the indentation consists of a mix of tabs and spaces, the indentation is raised.

When the interpreter senses an internal error, this exception is raised.

The **sys.exit()** function raised this exception.

When a function or procedure is applied to an object of the wrong kind, this exception is raised.

When a reference to a local variable is made in a function or procedure, but no value has been bound to that variable, this exception is raised.

When a Unicode-related encoding or decoding error occurs, this exception is raised.

When a Unicode-related error occurs during encoding, this exception is raised.

When a Unicode-related error occurs, this exception is raised.

When a Unicode-related mistake occurs during translation, this exception is raised.

It occurs when a function receives an argument with the correct type but the wrong value.

When the second operand of a division or modulo operation is zero, this exception is raised.

User-defined exceptions

You can also build your own exceptions by deriving classes from the built-in exceptions in Python. By creating a new exception class, programmers can call their exceptions. Exceptions must be either directly or indirectly inherited from the **Exception** class. The below code shows an example of a user-defined exception.

Example 10.14:

```
#User-defined exception example
class MyException(Exception):
#constructor
def __init__(self, a):
self.a = a
# display function
def __str__(self):
return (repr(self.a))
try:
raise(MyException("My Exception"))
# exception value stored in e
except MyException as e:
print ('New Exception occurred:',e.a)
```

Output:

```
New Exception occurred: My Exception
```

In the preceding example, we have created a new exception class, Exceptions must be either directly or indirectly derived from the built-in **Exception** class.

Example 10.15:

```
#User-defined exception example
class MyClass(RuntimeError):

def __init__(self, a):
self.a = a
try:
raise MyClass("It is a user defined exception")
except MyClass as e:
print (e)
```

Output:

```
It is a user defined exception
```

A runtime error is a built-in class that is raised when a generated error does not fall into one of the categories listed above. The preceding code demonstrates how to use runtime error as the base class and user-defined error as the derived class.

Pre-defined clean–up action

There are various situations in which we want our program to perform a particular job, regardless of whether it runs perfectly or throws an error. We use the **try** and **except** block to catch any errors or exceptions. The **try** statement provides a beneficial optional clause that is intended for specifying clean-up actions that must be executed in any circumstances. The **finally** clause would be executed no matter what; however, the **else** clause runs only if an exception was not raised.

Example 10.16:

```
# clean up actions example
def div(x, y):
try:
z = x // y
except ZeroDivisionError:
print ("Divide by zero. ")
else:
print ("Division value is :", z)
finally:
print (" It is Finally clause, always run. ")
```

```
# passing parameters to function
div(50, 4)
div(30, 0)
```

Output:

```
Division value is : 12
It is Finally clause, always run.
Divide by zero.
It is Finally clause, always run.
```

Modules

In modular programming, a complex and unmanageable program is divided into multiple subprograms known as a module. Each of the modules can be used to perform some specific task. In Python, modules can be created by using Python files that may contain definitions and various statements. Functions, classes, and variables can all be defined in a module. Runnable code can also be used in a module. The code is easier to understand and use when it is structured into modules. It also organizes the code logically.

Our Python code file with the extension is treated as the module. The Python module can contain executable code. We must import the particular module to use the functionality of one module in another. Assume you've created a file called **mymodule.py** that includes the following code:

mymodule.py

```
# Example of a module (mymodule.py)
def fun_add (x, y):
return (x+y)
def fun_sub (x, y):
return (x-y)
```

To call the functions and **fun_sub()** specified in the module named file we must include this module in our main module.

Loading module

To use the module's functionality, we must first load it into our Python code. Python has two types of statements, which are listed below.

The **import** statement

The **from-import** statement

The import statement

A module can be linked with our Python program using an import statement. With a single **import** statement, we can import several modules, but a module is only loaded once, regardless of how many times it has been imported into our register. The **import** statement's syntax is mentioned as follows:

Syntax:

```
import module1, module2, module3, ........ module n
```

If an **import** statement is identified, the interpreter imports the module if it is included in the search path. When importing a module, the interpreter searches all the directories in the search path. For example, to import the module add the following command to the program's top.

Example

```
# importing module mymodule.py
import mymodule
print (fun_add (10, 20))
print (fun_sub (100, 50))
```

Output:

```
30
50
```

The from...import statement

Python allows you to import only the specific attributes of a module, rather than the whole module, into the namespace. The from import statement can be used for this. The **from... import** expression is used in the following syntax.

Syntax:

```
from import , ....
```

Consider the following module, which includes the functions **fun_add()** and **fun_sub().**

Example

File mymodule.py

```
# mymodule.py
def fun_add(x, y):
return (x+y)
def fun_sub(x, y):
```

```
return (x-y)
```

If we want to import only **fun_add()** function from this module, then following code will be used.

File main.py

```
# importing fun_add() from mymodule.py
from mymodule import fun_add
print (fun_add(10, 20))
```

Output:

```
30
```

Example 10.19:

```
# importing pi from math module
from math import pi
print ("pi value=", pi)
```

Output:

```
pi value= 3.141592653589793
```

When we know the attributes to be imported from the module in advance, we can use the **from...import** statement. Our code will not get heavier as a result of this. The * may also be used to import all of a module's attributes. Consider the following example, in which all the functions of **mymodule** are imported.

Example

File name: main.py

```
# importing all attributes from mymodule.py
from mymodule import *
print (sub(100, 20))
print (add(100, 20))
```

Output:

```
80
120
```

Renaming a module

Python allows us to import a module with a particular name and then use that name to reference that module in our Python source code. The following is the syntax for renaming a module.

Syntax:

```
import as
```

Example 10.21:

```
# importing from mymodule.py
import mymodule as mm # rename mymodule to mm
print (mm.sub(20, 10))
print (mm.add(100, 20))
```

Output:

```
10
120
```

The dir() built-in function

The **dir()** function can be used to find names specified within a module. In the module for example, we've specified two functions: **add()** and In the **mymodule** module, we can use **dir()** as follows:

```
>>> dir(mymodule)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add',
'sub']
```

We can see a sorted list of names here (along with **add** and All other names that begin with an underscore are the module's default Python attributes. The **dir()** function can be used without any arguments to find all the names specified in our current namespace.

Example 10.22:

```
> a=10
> b=30
> import math
> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'a', 'b', 'math']
```

The reload() function

As previously mentioned, regardless of how many times a module is imported into the Python source file, it is only loaded once. However, Python provides us with the **reload()** function to reload an already imported module and re-run the top-level code. The **reload()** function is used in the following syntax.

Syntax:

```
reload()
```

Example

```
reload(mymodule)
```

Python built-in modules

Many pre-defined functions are also available as a part of libraries bundled with Python distributions, in addition to built-in functions. Built-in modules describe the functions that are specified in modules. The Python shell is integrated with built-in modules, which are written in C. Each built-in module includes resources for system-specific functions like OS management, disc IO, and so on. Many Python scripts (with extension) containing useful utilities can also be found in the standard library. Use the following command in the Python console to see a list of all the available modules:

```
help('modules')
```

Following are some examples of built-in modules:

Example 10.24:

```
# Example of built-in module math
import math as m
# using square (sqrt) function
print (m.sqrt(100))
```

```
# using pi function
print (m.pi)
# degree
print (m.degrees(3))
# radians
print (m.radians(60))

# Sin
print (m.sin(60))

# Cosine
print (m.cos(30))

# Tangent

print (m.tan(90))
# factorial
print (m.factorial(6))
```

Output:

```
10.0
3.141592653589793
171.88733853924697
1.0471975511965976
-0.3048106211022167
```

```
0.15425144988758405
-1.995200412208242
720
```

Example 10.25:

```
# datetime built-in module
from datetime import time
from datetime import date
d = date(2000, 1, 1)
print ("Given date is:", d)

print (date.today()) #It will show current date

Time = time(5, 30, 10)

print ("hour =", Time.hour)
print ("minute =", Time.minute)
print ("second =", Time.second)
print ("microsecond =", Time.microsecond)
```

Output:

```
Given date is: 2000-01-01
2021-04-21
hour = 5
```

minute = 30

second = 10
microsecond = 0

Example 10.26:

```
# built-in module random
import random

# random integer between 0 and 10
print ('Random number between 0 and 10:',random.randint(0,
5))

# random float between 0 and 1
print ('Random float between 0 and 1:',random.random())

# random number between 0 and 100
print ('Random number between 0 and 100:',random.random()
* 100)

List = [2, 4, True, 80, "Python", 20, "Book", 45]

# random element from a list
print ('Random element from list',random.choice(List))
```

Output:

```
Random number between 0 and 10: 3
Random float between 0 and 1: 0.7476231433619301
Random number between 0 and 100: 46.881247400915406
Random element from list 80
```

Packages in Python

We organize a large number of files into various folders and subfolders based on specific parameters so that they are easier to locate and handle. On the other hand, a module in Python brings the principle of modularity to the next logical level. As you might know, a module can contain a variety of objects such as classes, functions, and so on. One or more related modules can be included in a package. A package is a folder that contains one or more module files on a physical level. Packages help us store other sub-packages and modules to be used by the user when needed, much as drives and directories allow us to store files in an operating system.

Creating packages

We build a file called **__init__.py** within a directory to inform Python that it is a package, and then it is considered a package, and we can create other modules and subpackages inside it. This **__init__.py** file may be left blank or filled in with the package's initialization code.

We can follow these three basic steps to build a package in Python:

First, we create a directory and name it a package, preferably the one related to its function.

After that, we add the required classes and functions.

Finally, inside the directory, we construct an **__init__.py** file to inform Python that it is a package.

Example 10.27:

First, we'll make a directory called After that, we must build modules. To do so, create a file called **Oppo.py** and write the below code.

File name: Oppo.py

```
# Creating Modules
class Oppo:
# constructor
def __init__(self):
self.phone = ['Android', '8GB', '32MP']

# function
def show(self):
print ('Features of Oppo')
for s in self.phone:
print (s)
```

Then we make a new file called **Vivo.py** and put the same code into it, but with different members.

File name: Vivo.py

```
# Creating Modules
class Vivo:
# constructor
def __init__(self):
self.phone = ['Android ', '16GB', '64MP']
```

```
# function
def show(self):
print ('Features of Vivo')
for s in self.phone:
print (s)
```

Then we make a new file called **iphone.py** and copy the same code into it, but with different members.

File name: iPhone.py

```
# Creating Modules
class iPhone:
# constructor
def __init__(self):
self.phone = ['IOS', '128GB', '12MP']
# function
def show(self):
print ('Features of iPhone')
for s in self.phone:
print (s)
```

The **__init__.py** file is finally created. This file will be located in the **Mobiles** directory and can either be left blank or filled with the initialization code.

File name: __init__.py

```
from Oppo import Oppo
from Vivo import Vivo
from iPhone import iPhone
```

Now it's time to put our new package to work. To do so, create a **Test.py** file in the same directory as the **Mobiles** package and write the following code:

File name: Test.py

```
# Import classes from Mobiles package
from Mobiles import Oppo
from Mobiles import Vivo
from Mobiles import iPhone

# Create an object of Oppo class and call its functions
obj1 = Oppo()
obj1.show()

# Create an object of Vivo class and call its functions
obj2 = Vivo()
obj2.show()

# Create an object of iPhone class and call its function
obj3 = iPhone()
obj3.show()
```

Output:

```
Features of Oppo
Android
8GB
32MP
Features of Vivo
Android
16GB
64MP

Features of iPhone
IOS
128GB
12MP
```

The following *figure 10.2* shows the folder structure of the **Mobiles** package:

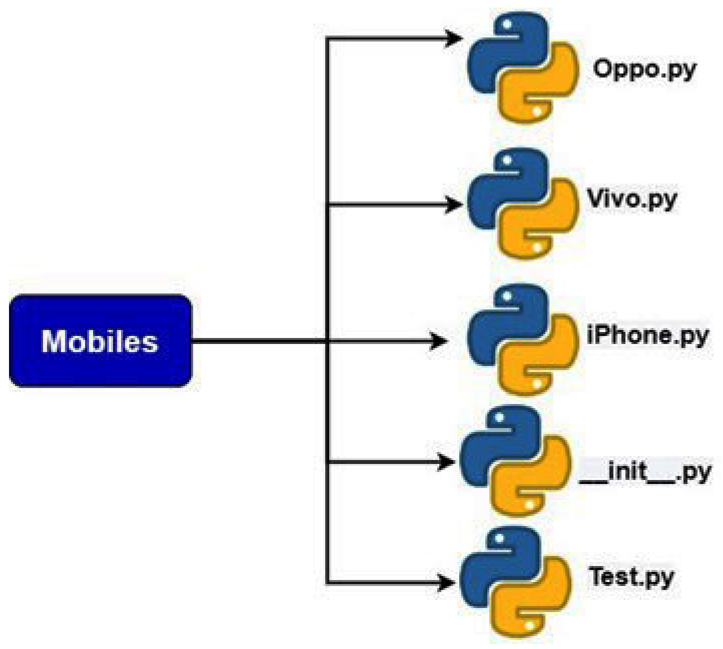


Figure 10.2: *Mobiles package folder structure*

__init__.py

The content of the package is stored in a special file named which is found in the package folder. It has two functions:

If a folder contains the **__init__.py** file, the Python interpreter recognizes it as a package.

__init__.py makes resources from its modules available for import.

When this package is imported, an empty **__init__.py** file makes all modules' functions accessible. It's worth noting that the folder's **__init__.py** file is needed for Python to recognize it as a package.

Subpackages

Packages may have any number of nested sub-packages. *The figure 10.3* shows an example of a subpackage inside a **Mobiles** package:

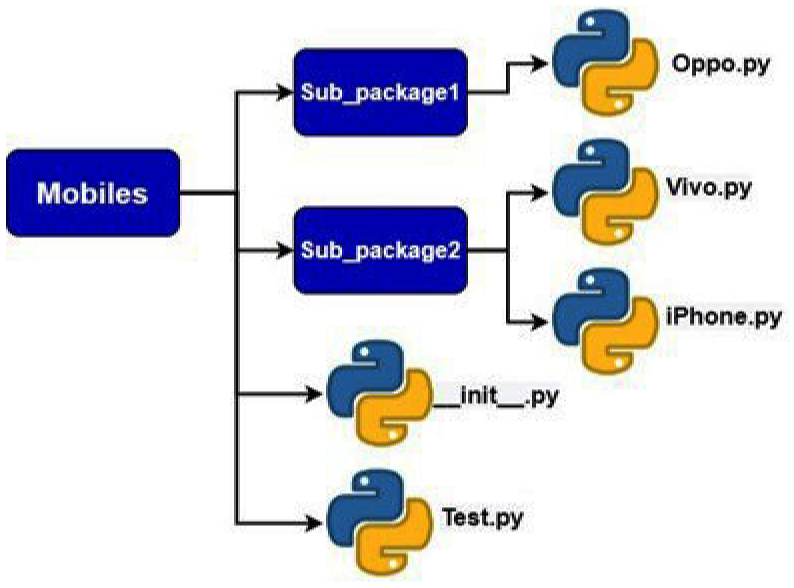


Figure 10.3: *Mobiles package folder structure with subpackages*

Like earlier, the three modules and are identified. They are now separated into two subpackage folders, **Sub_package1** and rather than being grouped in the **Mobiles** directory. Importing continues to operate in the same manner as before. The syntax is similar, but a dot separates the package name and subpackage name.

```
import Mobiles.Sub_package1.Oppo
import Mobiles.Sub_package2.Vivo
import Mobiles.Sub_package2.iPhone
```

Conclusion

In this chapter, we discussed exception handling, modules, and packages in Python. We also discussed the uses of and **finally** blocks for managing exceptions. Module implementations and their importing are also presented. Besides, package creations and services are also discussed with Python codes. Based on these examples, you can write your code. In the next chapter, we will discuss the concepts of object-oriented programming in Python.

Points to remember

An exception is a type of error that occurs during the execution of a program.

To handle exceptions, four keywords are used: and

To raise an exception, use the **raise** keyword.

A module is a smaller part of a larger program.

__init__.py file indicates that a particular folder is a package.

Multiple choice questions

The import statement is used for?

Creating package

Handling Exception

Linking module

None of the above

How many except statements can a try-except block have?

0

1

More than one

More than zero

What will be the output for the following code?

```
x = ""hello""
if not type(x) is int:
raise TypeError(""Only integers are allowed"")
```

hello

garbage value

Only integers are allowed

Error

When is the finally block executed?

When there is no exception

When there is an exception

Only when some condition that has been specified is satisfied

Always

What will be the output of the following Python code?

```
from math import factorial
print(math.factorial(5))
```

120

Nothing is printed

Error, method factorial doesn't exist in the math module

Error, the statement should be: print(factorial(5))

To obtain a list of all the functions defined under sys module, which of the following functions can be used?

print(dir(sys))

print(sys)

print(dir.sys)

```
print(dir[sys])
```

What is the output of the code shown below?

```
import random
random.choice(2,3,4)
```

An integer other than 2, 3 and 4

Either 2, 3 or 4

Error

2 only

Answers

c

d

c

d

d

a

c

Questions

What do you mean by exception handling?

How exceptions can be handled. Discuss?

What is a package?

Create a package and import it into a Python file. Also, write steps to create it.

What is the importance of **finally** block in exception handling?

What is a module? How can we create a module?

Write a Python program to demonstrate the example of **raise** keywords.

Discuss the importance of some built-in modules.

Write a Python program to handle arithmetic exceptions.

Discuss the uses of multiple **except** blocks with a single

CHAPTER 11

Object-oriented Programming

Object-oriented programming as a discipline has gained widespread acceptance among programmers. Python, a popular programming language, adheres to the object-oriented programming style as well. It is concerned with declaring Python classes and properties, which is the core of OOPs concepts. In this chapter, you will see various concepts of object-oriented programming about Python. Besides, we have demonstrated different OOPs concepts with the help of examples and codes.

Structure

In this chapter, we will cover the following topics:

The basic concepts of object-oriented programming

Objects and classes creation

Concepts and Types of constructors

Concepts of inheritance and polymorphism

Various Python methods to achieve OOPs

Objective

The objective of this chapter is to introduce the concept of object-oriented programming in Python. After completing this chapter, you should be able to understand classes, objects, constructors, inheritance, and polymorphism. Also, you will be able to write Python codes related to OOPs.

Object-Oriented Programming

Object-oriented programming is a way of structuring a program by grouping associated properties and functionalities into separate objects. Object-oriented programming is based on classes and objects. Class is the blueprint, while objects are real entities and are able to perform various tasks. Data elements, also known as properties, and behavior, such as actions or functions, make up an object. Another popular programming paradigm is procedural programming, which structures a program like a recipe by providing a series of steps, such as functions and code blocks that flow sequentially to complete a task.

Python, like other general-purpose programming languages, has been an object-oriented language. It enables us to build applications in an object-oriented manner. We can easily develop and use classes and objects in Python. Using classes and objects to construct the software is an object-oriented paradigm. The object is linked to real-world objects such as a computer, a home, a mobile, etc. The OOPS definition emphasizes the development of reusable code. It is a common method of resolving a problem by creating objects. The following are the key concepts of an object-oriented programming paradigm:

Class

Object

Method

Data Abstraction and Encapsulation

Inheritance

Polymorphism

Introduction to classes

A set of objects can be described as a class. It's a logical entity with some unique properties and methods. For instance, if you have a student class, it should have an attribute and a method, such as name, age, address, and course.

Object

The object is a self-contained entity with state and actions. It could be a laptop, a purse, a phone, a table, a pencil, or something else. In Python, everything is an object, with attributes and methods on almost everything. A class must construct an object to assign memory when it is defined.

Method

In Python, a method is like a function, except that it is linked to objects and classes. Except for two big variations, Python methods and functions are similar.

For the object for which it is named, the method is used implicitly.

Data in the class is accessible to the method.

Data abstraction and encapsulation

In object-oriented programming, encapsulation is also essential. It's used to limit access to variables and methods. Encapsulation protects code and data from accidental modification by wrapping them together in a single device.

Abstraction is a technique for hiding internal information and displaying only the functionalities. The term *abstract* refers to the process of giving items names that capture the essence of what a function or a program does. Both data abstraction and encapsulation are often used interchangeably. Since data abstraction is accomplished by encapsulation, the terms are nearly interchangeable.

Inheritance

The most fundamental component of object-oriented programming is inheritance, which simulates the process of inheritance in real life. It states that the child object inherits all of the parent object's properties and behaviors. We may construct a class that inherits all the properties and behaviors of another class by using inheritance. The new class is a derived class or a child class, whereas the base class or parent class is known as the one whose properties are acquired. It ensures that the code can be reused.

Polymorphism

Polymorphism is made up of the words *poly* and Poly stands for and morph stands for We consider polymorphism as the ability to execute a task in many forms. It uses a single type of entity (method, operator, or object) to represent multiple types in various scenarios. For instance, we have one addition operator that can add different kinds of values.

Object-oriented versus procedure-oriented programming languages

The following is a description of the differences between object-oriented and procedural programming:

programming: programming:
programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming:
programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming:
programming: programming:

programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming: programming:
programming: programming: programming: programming: programming: programming: programming: programming: programming:

Table 11.1: *Differences between object-oriented and procedural programming*

Python class and objects

A class is a virtual entity that serves as an object's blueprint. After creating an object, it occupies memory and came into existence. For example, assume a class is a model for a building structure. A building includes all the information about the rooms, gardens, area, and balcony. Based on these data items, we can construct as many houses as we want. As a result, the structure can be thought of as a class or a blueprint of the building, and we can make as many objects (houses) as we want.

The object is known as the instance of a class. The process of constructing an object is referred to as instantiation. In the above example, all the individual houses will be the objects of the class, i.e., the building.

Defining a class in Python

The keyword followed by the class name, is used to build a class in Python. The following is the syntax for creating a class.

```
class
# data members
# member functions
```

A class declares all its attributes in a new local namespace. Data members or functions may be included as attributes.

It also contains unique attributes that start with double underscores. For example, **__doc__** returns the class's docstring. The following statement can access it:
. __doc__.

When we define a class, a new class object with the same name is generated. We may use this class object to access the various attributes and create new objects of that class.

Example 11.1:

```
# Creating a class
class Student:
"This is an example of student class"
age = 25
def fun(self):
print('Python')
# display age
print (Student.age)
#call function
print (Student.fun)
# display docstring
print (Student.__doc__)
```

Output:

```
25
Student.fun at 0x7f4fbdb88e50>
This is an example of student class
```

Example 11.2:

```
# Creating a student class
```

```
class Student:
stuid = 1001
stuname = "Amit"
def show (self):
print (self.stuid,self.stuname)
# Creating an object
obj=Student()
# Calling a member function
obj.show()
```

Output:

```
1001 Amit
```

In the above example, a **Student** class is created with two fields: **stuid** and This class also contains a function **show** which can show the student information.

The is used here as a reference variable to the current class object. In the function description, it is always the first argument. In the function call, however, using self is optional.

The accesses the class variables and corresponds to the current instance of the class. Instead of self, we can use anything, but it must be the first parameter of any class function.

Creating an object

If we want to use a class's attributes in another class or method, we must first instantiate it. A class can be instantiated by calling its name. The syntax for creating a class instance (object) is mentioned below:

```
= ()
```

The following code shows an example of creating objects of the **book** class.

Example 11.3:

```
# Creating a class book
class Book:
# defining attribute
name= "Python Programming"
author = "Krishna"
year=2021

# creating function
def show(self):
```

```
print ("Book Name:", self.name)
print ("Author:", self.author)
print ("Year:", self.year)
# Object creation
B1 = Book()
B2 = Book()
# Accessing class attribute and method through objects
print (B1.name)
print ('----------')
B2.show()
```

Output:

```
Python Programming

----------
Book Name: Python Programming
Author: Krishna
Year: 2021
```

Deleting properties or object

Using the **del** keyword, we can delete the object's properties or the object itself. The following examples demonstrate the deletion of some properties and objects, respectively.

Example 11.4:

```
# Deleting properties of a class
class Book:
# defining attribute
name= "Python Programming"
author = "Krishna"
year=2021

# creating function
def show(self):
print ("Book Name:", self.name)
print ("Author:", self.author)
print ("Year:", self.year)
# Object creation
B1 = Book()
# Accessing class attribute and method through objects
```

```
print ('-----Before Deletion-----')
B1.show()
# deleting a property (name)
del B1.name
print ('-----After Deletion-----')
B1.show()
```

Output:

```
-----Before Deletion-----
Book Name: Python Programming
Author: Krishna

Year: 2021
Traceback (most recent call last):
File "", line 19, in
AttributeError: name
```

Since we deleted the name property, it will throw an Attribute error: name.

Example 11.5:

```
# Example of object deletion
# Creating a class book
class Book:
```

```
# defining attribute
name= "Python Programming"
author = "Krishna"
year=2021
# creating function
def show(self):
print ("Book Name:", self.name)
print ("Author:", self.author)
print ("Year:", self.year)
# Object creation
B1 = Book()
# Accessing class attribute and method through objects
print ('-----Before Deletion of Object-----')
B1.show()
# deleting object B1
del B1
print ('-----After Deletion of Object-----')
B1.show()
```

Output:

```
-----Before Deletion of Object-----
Book Name: Python Programming
Author: Krishna
Year: 2021
-----After Deletion of Object-----
```

```
Traceback (most recent call last):
File "", line 21, in
NameError: name 'B1' is not defined
```

Since we deleted the B1 object, it will throw a NameError.

Data abstraction

In object-oriented programming, data abstraction is an important feature. We can also hide Python data by prefixing the attribute to be hidden with a double underscore The attribute will no longer be available outside the class via the object after performing this. The following Python code shows the concept of data abstraction.

Example 11.6:

```
class Student:
__n = 0;
def __init__(self):
Student.__n+=1
def show(self):
print ("Total Students:",Student.__n)
s1 = Student()
s2 = Student()
s3 = Student()
s1.show()
s2.show()
print (s1.__n) # It will generate error because n is not
available to access outside class
```

Output:

```
Total Students: 3
Total Students: 3
Traceback (most recent call last):
File "", line 12, in
AttributeError: 'Student' object has no attribute '__n'
```

Constructors in Python

A constructor is a particular type of function that is used to initialize the class's instance members. The constructor in C++ or Java has the same name as the class, but Python handles constructors differently. When we create an object of a class, the constructor function executes it. There are two kinds of constructors such as parameterized constructors and non-parameterized constructors. In Python, the constructor is defined using the **__init__()** method. It takes the **self** keyword as a first argument, allowing access to the class's attributes and methods. Depending on the **__init__()** definition, we can pass any number of arguments when constructing the class object. It's mainly used to set up the class attributes. Every class must have a default constructor if we have not defined any other constructor.

Example 11.7:

```
# Creating a class book
class Book:
#defining constructor
def __init__(self, n, a, y):
print ('Constructor example')
```

```
self.name = n
self.author = a
self.year=y

# creating function
def show(self):
print ("Book Name:", self.name)
print ("Author:", self.author)
print ("Year:", self.year)

# Object creation and passing arguments to constructor
B1 = Book("Python Programming", "John",2021)
B2 = Book("Java Programming", "David",2020)
# Accessing class attribute and method through objects
print ('-----Values are-----')
B1.show()
B2.show()
```

Output:

```
Constructor example
Constructor example
-----Values are-----
Book Name: Python Programming
Author: John
Year: 2021
Book Name: Java Programming
```

Author: David
Year: 2020

Example 11.8:

```
#Program for counting number of objects in a class
class Test:
#Defining data member
count = 0
#Defining constructor
def __init__(self):
Test.count += 1
# Creating objects
T1=Test()
T2=Test()
T3=Test()

T4=Test()
T5=Test()
print ("The number of objects:",Test.count)
```

Output:

```
The number of objects: 5
```

Non-parameterized constructor

When we don't want to update the value or just have self as an argument, we use the non-parameterized constructor. The following code shows an example of the non-parameterized constructor.

Example 11.9:

```
# Class creation
class Test:
# Defining non-parameterized constructor
def __init__(self):
print ("Non parameterized constructor")
def display(self,s):
print ("Python",s)
# Creating objects
T1=Test()
# calling function
T1.display("Programming")
```

Output:

Non parameterized constructor

Python Programming

Parameterized constructor

Along with the self, the parameterized constructor has several parameters. The following Python code demonstrates parameterized constructor.

Example 11.10:

```
# Class creation
class Test:
# Defining Parameterized constructor
def __init__(self,arg1):
print ("Parameterized constructor")
self.arg1=arg1

def display(self,s1):
print ("Python",self.arg1,s1)
# Creating objects and passing a parameter
T1=Test("Programming")
# calling function with one argument
T1.display("Book")
```

Output:

Parameterized constructor

Python Programming Book

Default constructor

The default constructor is used when no constructor is included in the class, or when it is not declared. It defines the objects' initial values. The following example demonstrates the concept of a default constructor.

Example 11.11:

```
# Default constructor example
#Class Declaration
class Employee:
Emp_id = 5001
Emp_name = "David"
# Function definition
def show(self):
print(self.Emp_id,self.Emp_name)
# Object creation
obj= Employee()
# Function calling
obj.show()
```

Output:

5001 David

Multiple constructors in a class

We can include two or more similar constructors in a class. In the following example, we have included two constructors of the same type. The object **T1** is called the second constructor in the following code, even though both have the same configuration. The **T1** object does not have access to the first constructor. If the class has many constructors, the object of the class will always call the last constructor.

Example 11.12:

```
# Multiple constructor example
#Class Declaration
class Test:
# defining first constructor
def __init__(self):
print ("First Constructor")
# defining second constructor
def __init__(self):
print ("Second Constructor")
# defining third constructor
def __init__(self):
print ("Third Constructor")
```

```
# Creating an object
T1 = Test()
```

Output:

```
Third Constructor
```

Constructor overloading is not permitted in Python.

Python inheritance

The object-oriented framework includes inheritance as a key component. Since we can use an existing class to build a new class instead of starting from scratch, inheritance allows us to reuse the program's code. The child class inherits the parent class's properties and has access to all of the data members and functions specified in the parent class. A child class may also have its implementation for the parent class's functions. A derived class can inherit a base class in Python by simply mentioning the base class after the derived class name in brackets. To inherit a base class into a derived class, use the syntax as shown below:

```
class (base-class_Name):
# derived class body
```

A derived class can inherit properties of multiple base classes. This kind of inheritance can be achieved by mentioning all base classes inside the brackets. The following syntax is used to inherit properties of multiple classes.

Syntax:

```
class (1>, 2>, ..... n>):
# derived class body
```

Example 11.13:

```
# Inheritance example
#Creating a base class
class Base():
# defining function inside base class
def fun1(self):
print ('Base class function')

# Creating a derived class
class Derived(Base):
# defining function inside derived class
def fun2(self):
print ('Derived class function')
# Creating object of derived class
obj = Derived()
# accessing functions of both base and derived classes
obj.fun1()
obj.fun2()
```

Output:

```
Base class function
Derived class function
```

Types of inheritance

There are four types of inheritances in Python, depending upon the number of child and parent groups involved. These are as follows:

Single inheritance

Multi-level inheritance

Multiple inheritance

Hierarchical inheritance

Single inheritance

A derived class can inherit properties from a single base class, allowing for code reuse and new features to existing code. The following *figure 11.1* shows the concept of single inheritance.

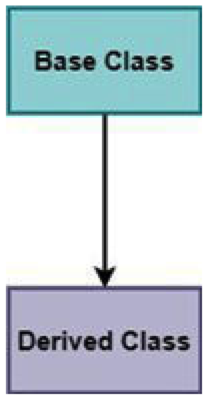


Figure 11.1: *Single inheritance*

Example 11.14:

```
# Single inheritance example
#Creating a base class
class Base():
# defining function inside base class
def fun1(self,a):
print ('Base class function')
```

```
self.a=a
print ('Base class:', self.a)
# Creating a derived class
class Derived(Base):
# defining function inside derived class
def fun2(self,b):

print ('Derived class function')
self.b=b
print ('Derived class:',self.b)
# Creating object of derived class
obj = Derived()
# accessing functions of both base and derived classes
obj.fun1(10)
obj.fun2(20)
```

Output:

```
Base class function
Base class: 10
Derived class function
Derived class: 20
```

## *Multilevel inheritance*

Features from the base and derived classes are passed down to the new derived class in multi-level inheritance. In Python, there is no limit to the number of levels to which multi-level inheritance can be archived. The following *figure 11.2* represents the concept of multi-level inheritance:

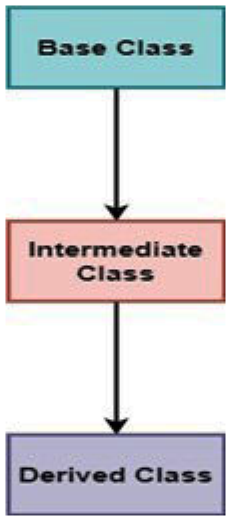


Figure 11.2: *Multilevel inheritance*

Example 11.15:

# Multilevel inheritance example

```
#Creating a base class
class Base():
def __init__(self):
print ('Base constructor')
# defining function inside base class
def funB(self,a):

print ('Base class function')
self.a=a
print ('Base class:', self.a)


class Intermediate (Base):
# defining function inside intermediate class
def funI(self,b):
print ('Intermediate class function')
self.b=b
print ('Intermediate class:', self.b)
# Creating a derived class
class Derived(Intermediate):
# defining function inside derived class
def funD(self,c):
print ('Derived class function')
self.c=c
print ('Derived class:',self.c)
# Creating object of derived class
obj = Derived()
# accessing functions of all three classes
obj.funB(10)
obj.funI(20)
obj.funD(30)
```

Output:

```
Base constructor
Base class function
Base class: 10
Intermediate class function
Intermediate class: 20
Derived class function

Derived class: 30
```

Multiple inheritance

Multiple inheritance is a type of inheritance in which a class may be derived from multiple base classes. In multiple inheritance, the derived class inherits all of the properties of the base classes. An example of multiple inheritances is shown in *figure* in which a derived class inherits properties of two base classes.

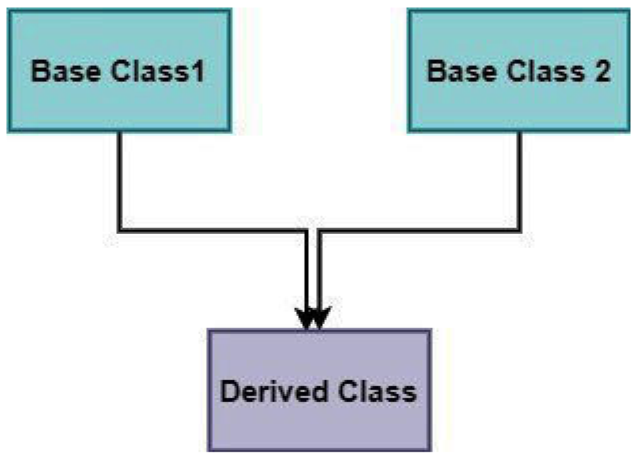


Figure 11.3: *Multiple inheritance*

Example 11.16:

```
# Multiple inheritance example
#Creating first base class
```

```
class Base1():
# defining function inside base class
def funB1(self,a):
print('Base class 1 function')
self.a=a
print('Base class 1 value:', self.a)

#Creating second base class
class Base2():
# defining function inside base class
def funB2(self,a):
print('Base class 2 function')
self.a=a
print('Base class 2 value:', self.a)

# Creating a derived class
class Derived(Base1, Base2):
# defining function inside derived class
def funD(self,c):
print('Derived class function')
self.c=c
print('Derived class value:',self.c)
# Creating object of derived class
obj = Derived()
# accessing functions of both base and derived classes
obj.funB1(10)
obj.funB2(20)
obj.funD(30)
```

Output:

```
Base class 1 function
Base class 1 value: 10
Base class 2 function
Base class 2 value: 20
Derived class function
Derived class value: 30
```

Hierarchical inheritance

Hierarchical inheritance is a form of inheritance in which multiple derived classes are inherited from a single base class. The following *figure 11.4* shows an example of hierarchical inheritance in which three derived classes are inherited from one base class:

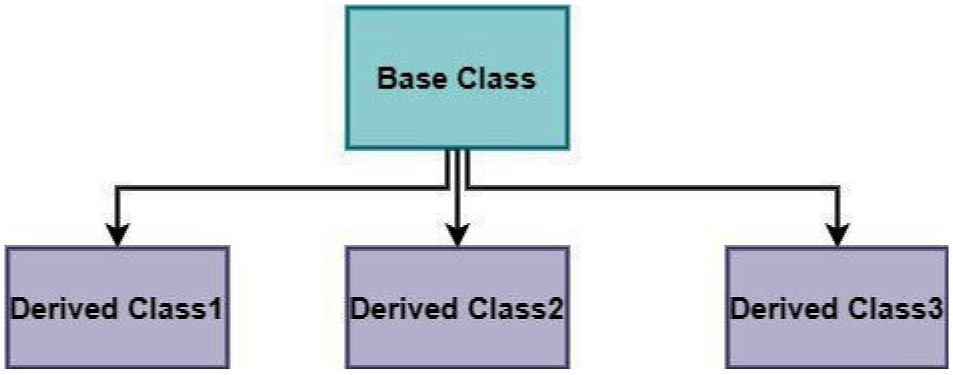


Figure 11.4: *Hierarchical inheritance*

Example 11.17:

```
# Hierarchical inheritance example
#Creating base class
class Base():
# defining funtion inside base class
```

```
def funB(self,a):
print ('Base class 1 function')
self.a=a
print ('Base class 1 value:', self.a)
# Creating first derived class
class Derived1(Base):
# defining function inside derived class
def funD1(self,c):
print ('Derived class 1 function')
self.c=c

print ('Derived class 1 value:',self.c)

# Creating second derived class
class Derived2(Base):
# defining function inside derived class
def funD2(self,c):
print ('Derived class 2 function')
self.c=c
print ('Derived class 2 value:',self.c)

# Creating third derived class
class Derived3(Base):
# defining function inside derived class
def funD3(self,c):
print ('Derived class 3 function')
self.c=c
```

```
print ('Derived class 3 value:',self.c)

# Creating object of first derived class
obj1 = Derived1()
# accessing functions from derived classes
print (' --------First derived class-----')
obj1.funB(10)
obj1.funD1(20)
# Creating object of second derived class
obj2 = Derived2()
# accessing functions from derived classes
print (' --------Second derived class-----')
obj2.funB(100)
obj2.funD2(200)
# Creating object of third derived class
obj3 = Derived3()

# accessing functions from derived classes
print (' --------Third derived class-----')
obj3.funB(1000)
obj3.funD3(2000)
```

Output:

```
--------First derived class-----
Base class 1 function
Base class 1 value: 10
```

```
Derived class 1 function
Derived class 1 value: 20
--------Second derived class-----
Base class 1 function
Base class 1 value: 100
Derived class 2 function
Derived class 2 value: 200
--------Third derived class-----
Base class 1 function
Base class 1 value: 1000
Derived class 3 function
Derived class 3 value: 2000
```

The issubclass(sub,sup) method

To check the relationships between the defined classes, use the **issubclass(sub, sup)** method. If the first class is a subclass of the second, it returns true; otherwise, it returns false.

Example 11.18:

```
class A:
def fun1(self):
print ('Base class 1')
class B:
def fun2(self):
print ('Base class 2')
class D(A,B):
def fun3(self):
print ('Derived class')
obj = D()
print ('D is sub class of B:',issubclass(D,B))
print ('A is sub class of B', issubclass(A,B))
```

Output:

```
D is sub class of B: True
A is sub class of B False
```

The isinstance (obj, class) method

To check the relationship between objects and classes, use the **isinstance()** method. If the first parameter, is an instance of the second parameter, class, it returns true.

Example 11.19:

```
class A:
def fun1(self):
print ('Base class 1')
class B:
def fun2(self):
print ('Base class 2')
class D:
def fun3(self):
print ('Derived class')
obj = D()
print ('obj is instance of A:',isinstance(obj,A))
print ('obj is instance of B', isinstance(obj,B))
print ('obj is instance of D', isinstance(obj,D))
```

Output:

```
obj is instance of A: False
obj is instance of B False
obj is instance of D True
```

Polymorphism in Python

Polymorphism is an object-oriented programming term that refers to the concept of having many forms. Polymorphism allows you to use a single interface for inputs of various data types, classes, or even for a different number of inputs. Polymorphism can be defined in a variety of ways in Python.

Polymorphism in operators

In Python programs, we know that the + operator is frequently used. It does not, however, have a single application. The + operator is used to perform arithmetic addition on integer data types. Similarly, the + operator is used to concatenate string data.

Example 11.20:

```
x = 11
y = 22
print (x+y)
A = "Python"
B = "Programming"
C= "Book"
print (A+" "+B+ ' '+C)
```

Output:

```
33
Python Programming Book
```

Function Polymorphism

Several Python functions can work for a variety of data types. The **len()** function is an example of such a function. Python allows it to work with a variety of data types. The **len()** function will operate with various data types, including string, list, tuple, set, and dictionary. It does, however, return specific information about specific data types.

Example 11.21:

```
print (len("Python Programming"))
print (len(["Amit", "Sumit", "Rohit"]))
print (len({"Id": 10001, "Address": "Mumbai"}))
print (len('1234'))
```

Output:

```
18
3
2
4
```

Polymorphism in Class methods

Since Python allows various classes to have the same name methods, we may use the principle of polymorphism when constructing class methods. We may construct a **for** loop that iterates over a tuple of objects in this case. After that, we call the methods without regard to the class type of each object. We presume that each class has these methods.

Example 11.22:

```
# Polymorphism in classes
class A():
def fun1(self):
print ("function 1 of A")

def fun2(self):
print ("function 2 of A")

class B():
def fun1(self):
print ("function 1 of B")
```

```
def fun2(self):
print ("function 2 of B")

obj1 = A()
obj2 = B()
for i in (obj1, obj2):
i.fun1()
i.fun2()
```

Output:

```
function 1 of A
function 2 of A
function 1 of B
function 2 of B
```

Method overriding (polymorphism in inheritance)

Polymorphism in Python allows one to identify methods in the child class with the same name as the parent class's methods. In inheritance, the methods of the parent class are passed on to the child class. However, a child class method that has inherited from the parent class may be modified. This is especially useful when the parent class's inherited method doesn't quite suit the child class. In such instances, the method is re-implemented in the child class. Method overriding is the process of re-implementing a method in a child's class.

Example 11.23:

```
# Example of method overriding
class Emp:
def b1(self):
print ('It is Base Class')
def fun1(self):
print ('Employee Class')

class Dept(Emp):
def fun1(self):
```

```
print ('Department class')

class Sale(Emp):
def fun1(self):
print ('Sales class')

e1 = Emp()
e1.b1()
e1.fun1()

print('------------')
d1 = Dept()
d1.b1()

d1.fun1()
print ('------------')
sl = Sale()
d1.b1()
sl.fun1()
```

Output:

```
It is Base Class
Employee Class
------------
It is Base Class
```

```
Department class
------------
It is Base Class
Sales class
```

Python built-in class functions

The following are a class's built-in functions:

getattr(obj, name, It's used to get the object's attribute.

setattr(obj, name, It's used to give a specific value to an object's attribute.

delattr(obj, It deletes a specific attribute.

hasattr(obj, If the object has a particular attribute, it returns true.

Example 11.24:

```
class Emp:
def __init__(self, Emp_name, Emp_id, Emp_age):
self.Emp_name = Emp_name
self.Emp_id = Emp_id
self.Emp_age = Emp_age
```

```
# object of the class Emp
E= Emp("Amit", 1001, 30)

# printing attribute name of the object E
print (getattr(E, 'Emp_name'))

# Changing the value of attribute Emp_age to 40
setattr(E, "Emp_age", 40)

# Display modified value of Emp_age
print (getattr(E, 'Emp_age'))

# prints true if the Emp contains the attribute with Emp_id
print (hasattr(E, 'Emp_id'))
# deleting one attribute
delattr(E, 'Emp_id')
# It will show an error since the attribute Emp_id has been
deleted

print (E.Emp_id)
```

Output:

```
Amit
40
True
```

```
Traceback (most recent call last):
File "", line 26, in
AttributeError: 'Emp' object has no attribute 'Emp_id'
```

Python built-in class attributes

In addition to the other attributes, a Python class has several built-in class attributes that provide information about the class. The built-in class attributes are below:

It provides a dictionary with class namespace details.

It has a string with the documentation for the class.

It's used to get the name of the class.

It allows you to get to the module where this class is defined.

It has a tuple that includes all base classes.

Example

```
class Emp:
def __init__(self, Emp_name, Emp_id, Emp_age):
self.Emp_name = Emp_name
self.Emp_id = Emp_id
self.Emp_age = Emp_age

def show(self):
print (self.Emp_name)
print (self.Emp_age)
obj = Emp("Joy",1001,30)
print (obj.__doc__)
print (obj.__dict__)
print (obj.__module__)
print (show.__name__)
print (Emp.__bases__)
```

Output:

```
None
{'Emp_name': 'Joy', 'Emp_id': 1001, 'Emp_age': 30}
__main__
show
('object'>,)
```

Class or static variables

All objects share the class or static variables. For different objects, non-static variables or instance variables vary. Class variables have a value assigned to them in the class declaration, while instance variables or non-static are variables that have values assigned to them within methods.

Example 11.26:

```
# Class declaration
class Student:
fees = 50000                  # Class or static variable
def __init__(self,id,name):
self.id = id              # Instance or non-static Variable
self.name = name

# Objects creation
s1 = Student(101,'Amit')
s2= Student(102,'Sumit')

print (s1.fees)   # prints 50000
print (s2.fees)   # prints 50000
print (s1.id)      # prints 101
print (s2.id)      # prints 102
print (s1.name)    # prints Amit
print (s2.name)    # prints Sumit

# Class variables can be accessed using class name also
print (Student.fees) # prints 50000

# we change the fees for just s1 it won't be changed for s2
s1.fees = 25000
print (s1.fees) # prints 25000
print (s2.fees) # prints 50000



# To change the fees for all we can change them directly from
the class
Student.fees = 40000
print (s2.fees) # prints 40000
```

Output:

```
50000
50000
101
102
Amit
```

Sumit

50000

25000

50000

40000

Class method vs. static method

The **@classmethod** decorator is a built-in function decorator that evaluates after your function is specified. The outcome of that analysis casts a shadow over your function definition. The class is implicitly passed as the first argument to a class method, just as the instance is implicitly passed to an instance method.

Syntax:

```
class :
@classmethod
def (cls, arg1, afr2,...):
#Function Body
return value
```

Class method properties are described below:

A class method is bound to the class rather than the class's object.

Since it takes a class parameter those points to the class rather than the object instance, they have access to the class's state.

It can change the state of a class that affects all of its instances. It can, for instance, change a **class** variable that affects all instances.

An implicit first argument is not passed to a static method. The **@staticmethod** decorator is used for static methods.

Syntax:

```
class :
@staticmethod
def (cls, arg1, afr2,...):
#Function Body

return value
```

The following are static method properties:

A static method is one that is bound to the class rather than the class's object.

A static method can't access or change the state of the class.

It is present in a class since the method makes sense to be present in a class.

Example 11.27:

```
# Example of class method and static method.
from datetime import date
class Student:
def __init__(self, name, age):
self.name = name
self.age = age

# a class method to create a Student object by birth year.
@classmethod
def fromBOD(cls, name, year):
return cls(name, date.today().year - year)

# a static method to check if a Student is adult or not.
@staticmethod
def isAdult(age):
return age > 18

s1 = Student('Ankit', 25)
s2 = Student.fromBOD('Ankit', 1996)
```

```
print (s1.age)
print (s2.age)

# print the result
print (s2.isAdult(25))
```

Output:

```
25
25
True
```

Class decorator

Decorators are a potent and valuable Python tool because they enable programmers to change a function or class's behavior. Decorators allow us to wrap another feature to expand its behavior without changing it permanently. We may describe a decorator as a class by using the **__call__** method. When a user wants to construct an object that acts like a function, the function decorator must return an object that acts like a function, which is where **__call__** comes into use.

Example 11.28:

```
class DecoratorExample:
def __init__(self, fun):
print ('This is constructor')
self.fun = fun

def __call__(self):
# Before function call code.
print ('Before function call')

self.fun()
```

```
# After function calls code.
print ('After function call')

# adding class decorator to the fun
@DecoratorExample
def fun():
print ("Class decorator")
# function call
fun()
```

Output:

```
This is constructor
Before function call

Class decorator
After function call
```

Recursive calls to functions

A method of programming or coding a problem in which a function calls itself one or more times in its body is known as Typically, it returns the function call's return value. We call a function recursive if its description meets the recursion condition. In Python, we can call a function inside another function. In the same way, we can call a function recursively inside its body.

Example 11.29:

```
# Recursive function example
# Function to compute factorial of a given number
def fact(n):
if n == 1:
return 1
else:
return (n * fact(n-1))

x = 6
print ("The factorial value of", x, "is", fact(x))
```

Output:

```
The factorial value of 6 is 720
```

Conclusion

In this chapter, we discussed the concepts of object-oriented programming and their implementation in Python. We discussed constructor, inheritance, polymorphism, etc. Various built-in class functions and attributes are described with the help of examples. Based on these examples, you can write your code for object-oriented programming. In the next chapter, we will discuss machine learning concepts and their implementations in Python.

Points to remember

Python is object-oriented programming.

In Python, almost everything is an object with its own set of properties and methods.

Decorators are a powerful and valuable Python tool. They allow programmers to modify the behavior of a function or a class,

We can implement hybrid inheritance with the help of other inheritances.

Multiple choice questions

Which of the following is not an OOPs concept?

Encapsulation

Polymorphism

Exception

Abstraction

What type of inheritance is illustrated in the following Python code?

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

Multi-level inheritance

Multiple inheritance

Hierarchical inheritance

Single-level inheritance

Which of the following is not a type of inheritance?

Double level

Multi-level

Single level

Multiple

The _________ keyword defines a template indicating what data and code will be contained in each object of type.

class

object

Class

Instance

_____ is used to create an object.

class

constructor

User-defined functions

In-built functions

Answers

c

b

a

a

b

Questions

What is object-oriented programming? List some of its advantages.

What are the main features of OOPs?

Differentiate between an object and a class.

Define constructor in Python? Also, give an example.

What is the difference between multiple and multi-level inheritance?

What is the concept of the overriding method? Give an example for the same.

What are Python decorators?

Write the difference between a class method and a static method.

Describe polymorphism with a suitable example.

Write code to use the **__init__()** method to assign values to data attributes in Python?

CHAPTER 12

Machine Learning with Python

Machine learning is a branch of computer science that allows computers to use data in the same way that humans do. It is a form of artificial intelligence that uses an algorithm or a method to extract patterns from raw data. The goal of machine learning is to encourage computers to learn from their experiences without being specifically programmed or requiring human interaction. In this chapter, you will be familiar with various machine learning approaches and applications. Besides, we have demonstrated various machine learning algorithms with the help of examples and codes.

Structure

In this chapter, we will cover the following topics:

The basic concepts of machine learning

Applications of machine learning

Types of machine learning

Various Python libraries for machine learning

Python codes to implement machine learning approaches

Objective

The objective of this chapter is to introduce the concept of machine learning. After completing this chapter, you will be able to implement and analyze the existing learning algorithms, including well-studied methods for classification, regression, prediction, and clustering learning. Also, you will be able to write Python codes for machine learning approaches.

Introduction to machine learning

Machine learning is the science (and art) of programming computers so they can learn from data. It is the field of study that gives computers the ability to learn without being explicitly programmed. A computer program is said to learn from experience *E* concerning some task *T* and some performance measure P, if its performance on as measured by improves with experience For example, through experience gained by playing games against itself, a computer program that learns to play checkers may improve its performance, measured by its ability to win in the class of tasks involving playing checkers games.

The concept of self-learning is the key attribute of machine learning. This refers to implementing statistical modeling, all without direct programming, instructions, identifying correlations, and enhancing performance based on data and analytical knowledge. This is what is described as the ability to learn without being explicitly programmed. The machine can perform a task using input data rather than relying on a direct input command. To construct a decision model, machine learning uses data as feedback. Decisions are generated by deciphering relationships and patterns in the data using probabilistic reasoning, trial and error, and other computation-intensive techniques. It means that the decision model's output is decided not by any pre-set rules specified by a human programmer but rather by the input data's quality. To minimize prediction errors, the human programmer always responds to feeding the data into the model, choosing an appropriate algorithm, and tweaking its settings (called hyper-parameters) and standard programming, the computer and the developer work a layer apart.

Input data is usually split into training data and test data in machine learning. The first split of data is the training data, which is the initial reserve of data used to develop the model. After developing the model, we can test the remaining data model known as the test data by using patterns extracted from the training data, satisfied with its prediction accuracy. If the model's performance is satisfactory by using the test data then the model is ready to use with the applications.

Uses of machine learning

Problems for which existing solutions require many fine-tuning or long lists of rules: one machine learning algorithm can often simplify code and perform better than the traditional approach.

Complex problems using a traditional approach yields no right solution: the best machine learning techniques can perhaps find a solution.

Fluctuating environments: a machine learning system can adapt to new data.

We get insights into complex problems and large amounts of data.

Traditional programming v/s machine learning

As a unique way to program machines, one genuine way to think about machine learning is that most programming that is not machine learning is procedural and is a set of rules defined by humans. This ruleset is called as an algorithm. The following figure shows the concepts of the traditional programming approach:

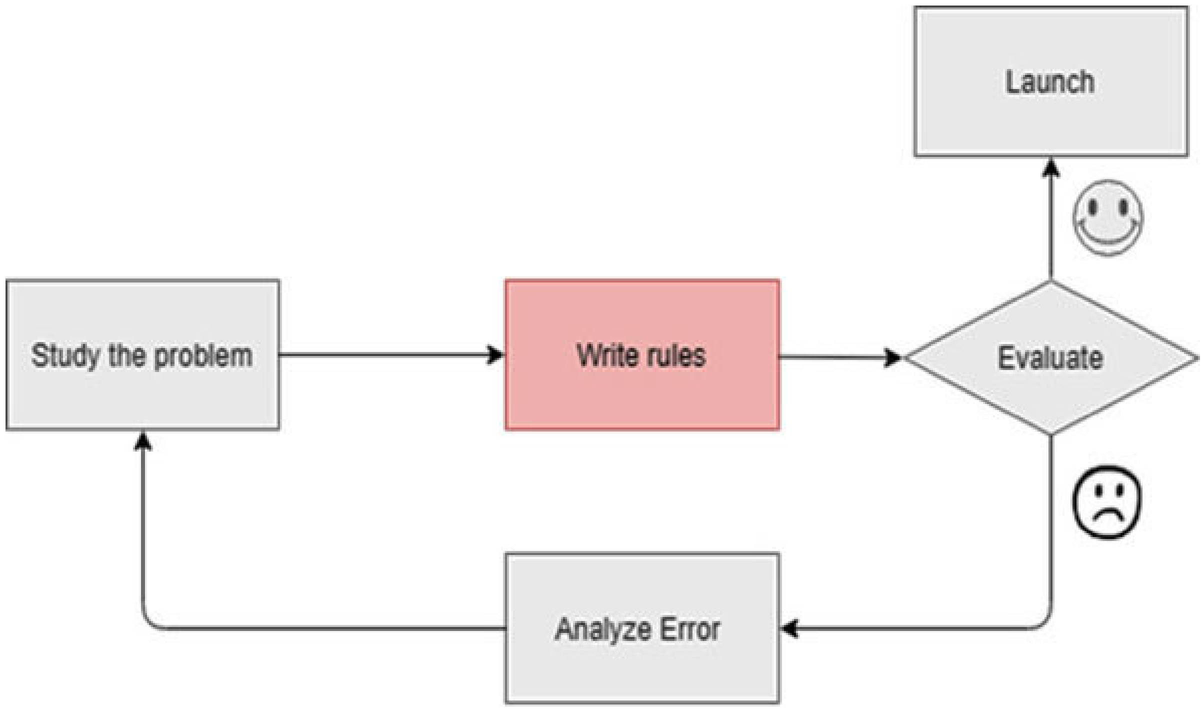


Figure 12.1: *Traditional programming approach*

In machine learning, a person chooses the underlying algorithm or designs it. However, the algorithms learn about the parameters that shape a mathematical model for making predictions from knowledge rather than direct human interference. Humans do not know certain conditions or set them-the computer does. Put another way, train a mathematical model, and use a data set when it sees something similar. The concepts of machine learning approaches are shown in figure

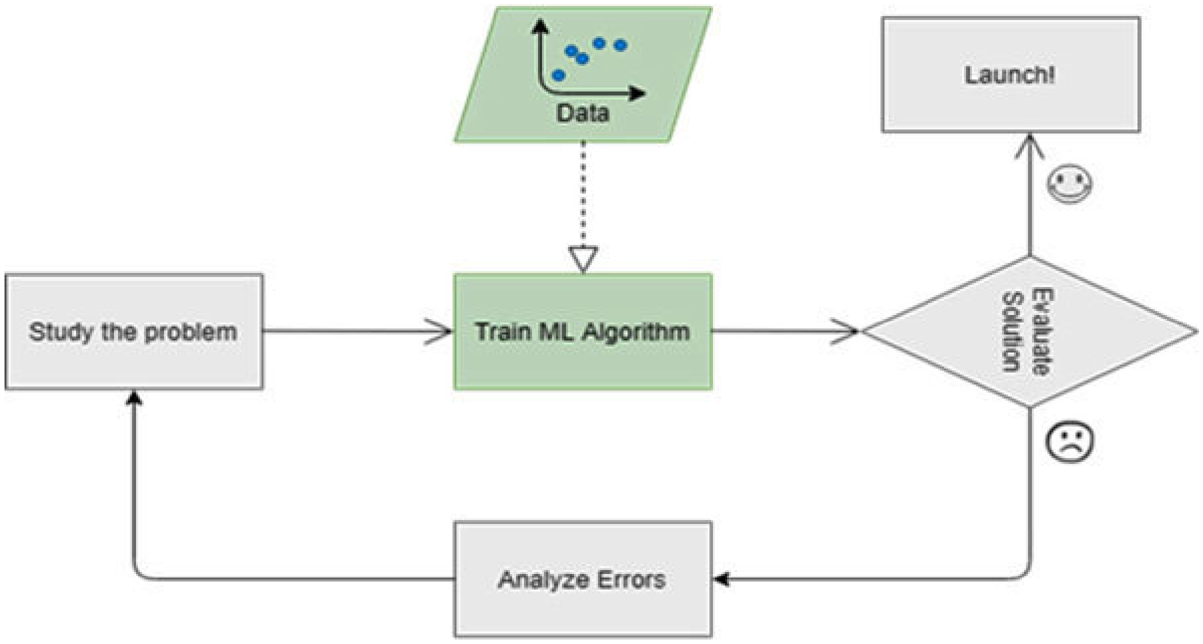


Figure 12.2: *A machine learning approach*

The following figure illustrates how machine learning integrates into data science and computer science's broader landscape:

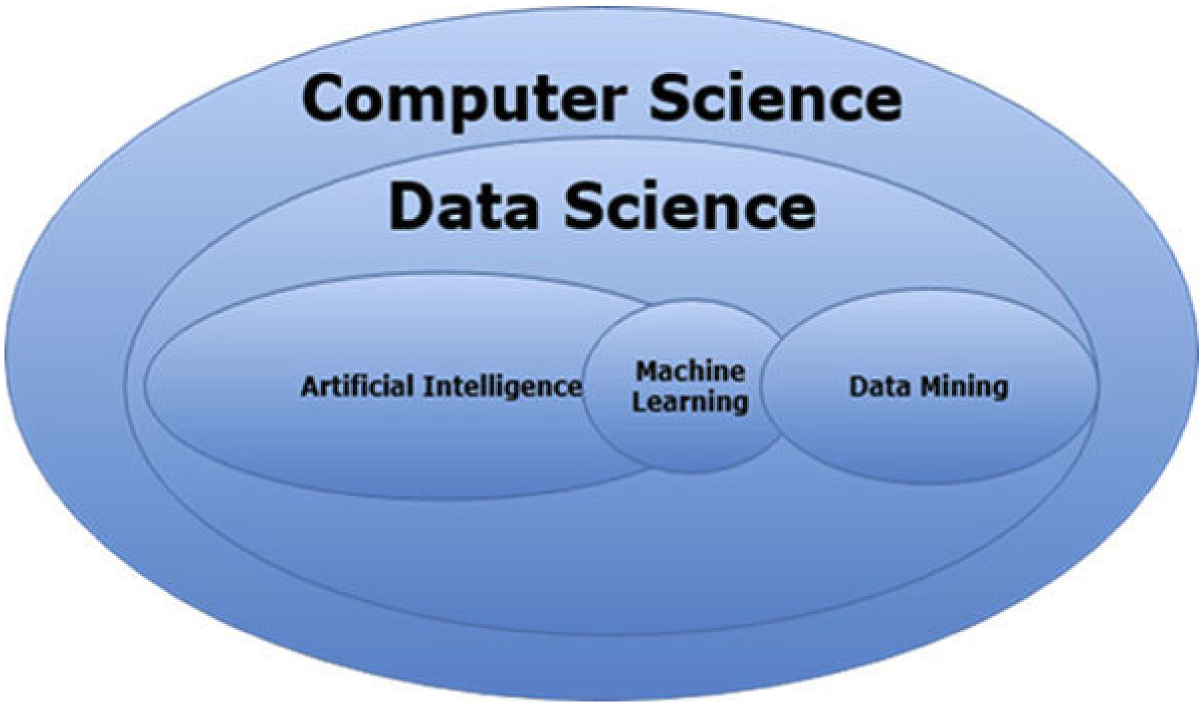


Figure 12.3: *Integration of machine learning into a data science and computer science*

Applications of machine learning

There are different areas where machine learning is commonly used. Some of the machine learning applications and their techniques are described as follows:

Image Before performing any tasks related to images, it is almost necessary to process the images to make them more suitable as input data. It is the conversion of JPEG or PNG file format images to usable data for neural networks. To process the image into data, we may use the TensorFlow library in the Python programming language as it provides a variety of tools to retrieve the data from an image file. We can resize the image file or even act on a large set of images all at once.

Speech Speech recognition saves us time by speaking instead of typing. It enables us to communicate with machines without even writing the programming code. Speech recognition is a great instance of using machine learning in real life. A suitable example is the Google Meet web application. CMU Sphinx, Kaldi, SpeechRecognition, and Wav2letter++ are the speech recognition libraries that we can use to process the audio file.

Traffic RNN Neural deep learning may be used to find patterns from traffic datasets to predict if severe traffic jams will happen shortly. Using the Geopandas package of Python, we can differentiate the traffic event of a given huge traffic dataset. Three forms of traffic incidents are seen, including road delays, collisions, and "stopped cars on the shoulder." We require GPU or **TPU Processing** and Google Colaboratory service offers this free for deep learning. On Google Chrome, we may install a plugin called Collaboratory.

Self-driving Train an end-to-end deep learning model in a driving simulator that would make a car drive around the track by itself. The regression problem between the vehicle steering angles and the road images from a car's cameras is tracked in real-time. The driving simulator saves frames from three different front-facing cameras, records data from the vehicle's point of view, and different driving data like speed, throttle, reverse, and steering angle. We may use matplotlib, Keras, NumPy, pandas, or scikit-learn environment and tools.

Email spam and malware An email is one of the most useful networking methods, and one of the essential features is efficient communication, Spam, and malware filtering. The outline for a spam filtering system with scratch is 1.) EDA

(Exploratory Data Analysis), 2.) Data Processing, 3.) Feature Extraction, 4.) Scoring & Metrics, 5.) Improvement by using Embedding and Neural Network, 6.) Comparison of ML & Deep Learning algorithms. We can avail the data set from UC Irvine Machine Learning Repository, Kaggle Datasets, AWS Datasets, or from Spam Assassin. Two types of data present in the repository are Ham (not Spam) and Spam data. The Ham data is complicated or straightforward, meaning that specific non-spam data have a strong resemblance to spam data, which could cause some trouble deciding for the system.

Detecting tumors in brain Basically, a tumor is an excessive proliferation of cells in some portion of the body, and a brain tumor is a collection or mass accumulated in the brain of these irregular cells. It is classified as primary in two categories, arising in the brain and secondary brain tumor that develops in the brain but arises from the spread of a malignant tumor anywhere in the body.

Magnetic resonance imaging is a widely used medical technology for diagnosing various tissue anomalies, including brain tumor detection and diagnosis. The active growth in computerized image segmentation allows doctors to recognize and imagine abnormalities to take the necessary steps to optimize fast decision-making therapy. The main challenges associated with medical imaging training machine learning models are the high cost of acquiring each dataset, receiving patient consent, and an expert's cost analysis of the image.

Creating a chatbot or a personal Chat-bot is software for computers that helps to communicate via messages. To imitate human behavior, they are formulated. ALICE is one of the most popular chatbots that work on pattern matching strategy. Pattern matching strategy is the **Artificial Linguistic Internet Computer Entity** ALICE's AIML files are accessible online and include categories such as music, art, philosophy, meetings, etc. for the pattern matching system of chat-bot, AIML files are used.

Fraud Bank purchases via credit or debit cards have risen dramatically in recent years, and an increase has also followed this in identity fraud. Financial institutions need to develop efficient and proactive fraud detection systems to manage this problem. To deal with this problem, machine learning is a promising solution. In the issue of fraud detection, transaction data is categorized and evaluated as legitimate or fraudulent. The fraud detection method is split into two general categories: fraud analysis (detection of misuse) and user behavior analysis (Anomaly detection).

Stock market Python can be used to illustrate the idea that stock prices obey a *random* The random walk does not eliminate the likelihood of beating the market, nor does it support tossing investment models, but it suggests that these models should be constructed and continuously re-evaluated with extreme care and diligence. The random-walk principle notes that it is not possible to forecast stock prices because stock price movements are random.

Medical Machine learning diagnosis works when the disorder can be reduced to a physiological data classification task in places where we currently rely on the clinician to be able to visually distinguish patterns that suggest the condition's presence or form.

Automatic language The use of software to translate text or speech from one language to another is investigated by automatic language translation or automatic translation abbreviated by the MT. For MT, we can use a deep neural network. We can figure out how to build a translation model for neural machines to translate one language into another (For example, English to French). The model accepts English text as input and returns the other translation of the language. To run the code that has access to GPU instances, we can use the AWS EC2 case.

Automatically classifying news We need to know what the news stories are about - we need to identify them to link people with the right content. The articles can be classified into over 80 categories, such as music, culture, architecture, opinion, food, etc. When an article is written, the first step in the pipeline is to remove all the article elements, such as text, title, images, writers, and URLs, and what the article is all about is the classification part.

Forecasting your company's revenue next year, based on many performance The SpringML app simplifies forecasting by running ML models that automatically run and include a monthly or quarterly revenue metric forecast for a consumer (Example, Revenue, ACV, and Quantity). Sales leaders will use these historical data-consuming models to gauge pattern and seasonality and current opportunities pipeline for the next 6 to 12 months to forecast. A precise forecast helps companies to make informed business choices. Some of the forecast methods are given below: 1.) Forecasting time series using Bayesian models. 2.) Provide the time series predictors. 3.) Evaluate current pipeline data on open opportunities by running classification algorithms. 4.) Before finalizing the best set of models to use, evaluate the ensemble over the previous few months.

Segmenting clients based on their purchases so that you can design a different marketing strategy for each Based on specific essential attributes that may help a business sell more goods at lower marketing prices, market segmentation distinguishes the target market or consumers. We can use ML algorithms based on multiple trees for segmentation. *Multiple Additive Regression Random* and *Boosting Stochastic Gradient* are techniques that render predictions using a multitude of trees and an ensemble of the same.

A client may be interested in recommending a product based on past Using various algorithms, a recommendation engine filters the data and recommends to users the most important things. It first captures a customer's past actions and based on that, suggests items that users would be willing to purchase. We may suggest products to a customer who is the most common among all users. Based on their interests, we can break the users into different segments and suggest products based on the segment to which they belong.

Building an intelligent bot for a We can use Reinforcement Learning basics, or we can assume that the game Snake applies Deep Reinforcement Learning (Neural Network + Q-Learning). We can create an AI Agent capable of learning how to play the iconic Snake from Scratch game, and we use Keras on top of Tensorflow to implement a Deep Reinforcement Learning Algorithm. Reinforcement learning is an approach to making decisions based on the Markov Decision Process. Instead of a conventional supervised ML approach, we could use Deep-Q-Learning. We have two main components in Reinforcement Learning: the environment (game) and the agent

(Snake). The environment rewards the agent each time the agent acts, which can be positive or negative based on how successful the state's action was.

Types of machine learning

Machine learning systems are mainly categorized into two types based on the procedure of their learning. If the machine is trained under human supervision with label data, it is classified as a supervised learning system. If the machine is trained without human or any label data, the system classifies the data based on similarity or a pattern, it is called an unsupervised learning system. Supervised learning is the most exploited form of machine learning system and requires extensive data but is the easiest form of machine learning.

Supervised learning

In supervised learning, machines learn under the supervision of label data in which the data set comprises input variables that correspond to the output variables on which the machine learns. There is training data on the which machine is trained by entering some data as an input and giving output according to its training, for example, credit risk, the machine gives outcomes based on the credit history as an input and gives a result corresponding to the credit risk. The machine learns to construct a model by providing consistent input-output samples and is trained on it. It is the type of machine learning in which we have an input corresponds to output variables and the model is supervised to learn the correlation of the variables. Supervised learning is also classified as:

A machine learning model helps us differentiate the data into separate categories and classify the data as input into specific categories. For example, in medical cases, if a patient is diagnosed with cancer, the machine gives outcomes based on symptoms as its input and gives output positive or negative. Another example is to categorize the pictures of animals into a cat group and a dog group.

Regression is like classification, but the only difference between them is that classification is done only on the discrete values, but in regression, we use continuous values to categorize the data. For example, a model predicts how many years the person will be sick or healthy by giving a radiological image as an input.

Unsupervised learning

In unsupervised learning, the output variables corresponding to input variables are not given. The variables are unlabeled. In unsupervised learning, the model tries to identify the pattern in the given variables and find the hidden pattern that can help create new labels regarding possible outputs. In supervised learning, we train the machine by using input data and no association with output variables. The motive of using unsupervised learning is to find the hidden pattern of the data. This is called an unsupervised learning because there is no labeled data, and the machine learns without the human's supervision. The machine automatically finds the pattern between the data, and it is classified into two groups clustering and association problems.

A clustering issue is where you want to discover the inherent groupings in the data, such as by buying activity and grouping customers.

The learning issue of an association rule is that you want to find rules that explain large portions of your data, such as individuals who buy *X* also tend to buy

Python libraries for machine learning

Python is now one of the most popular programming languages for machine learning tasks, and it has largely replaced many other languages in the field, due to its extensive library of functions. Machine Learning Python libraries include:

Numpy

It is one of the most popular libraries in Python for machine learning. It is used to process matrix and multi-dimensional arrays. In this mathematics, functions are present in large quantities that are very useful for the scientific computation of scientific data in machine learning.

Example 12.1:

```
import numpy as np
# Creating two arrays of rank 2
p = np.array([[1, 2], [3, 4]])
q = np.array([[5, 6], [7, 8]])
print (p)
print (q)
```

Output:

```
[[1 2]
[3 4]]
[[5 6]
[7 8]]
```

Pandas

It is a data analysis library in Python that is mainly used to manipulate and analyze data. It comes into play before the dataset is prepared for training. It is a Python library that gives us a set of tools to analyze data. In Pandas, we can load, prepare, manipulate, model, analyze data, join data, merge data, reshape data, and take data from different databases, put it together, and analyze it.

Example 12.2:

```
# importing pandas as pd
import pandas as pd
data = {"state": ["Delhi", "Rajasthan", "Maharashtra", "Gujarat",
"Punjab"],
"capital": ["New Delhi", "Jaipur", "Mumbai", "Gandhinagar",
"Chandigarh"]
}
data_table = pd.DataFrame(data)
print (data_table)
```

Output:

```
state          capital
0         Delhi        New  Delhi
1       Rajasthan          Jaipur
2   Maharashtra      Mumbai
3         Gujarat    Gandhinagar
4          Punjab        Chandigarh
```

Scikit-learn

With the help of the **scikit** library in Python, we can use various machine learning algorithms like classification, regression, clustering, reduction of dimensionality, selection of the model and preprocessing, etc. It is one of the most concerning machine learning libraries in Python. It can easily collaborate with different libraries of machine learning programming, for example, Numpy, and Pandas.

Example

```
# Sample Decision Tree Classifier
from sklearn import datasets
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
# load the iris datasets
dataset = datasets.load_iris()

# fit a CART model to the data
model = DecisionTreeClassifier()
model.fit(dataset.data, dataset.target)
print (model)
```

```
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)

# summarize the fit of the model
print (metrics.classification_report(expected, predicted))
print (metrics.confusion_matrix(expected, predicted))
```

```
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')
precision     recall   f1-score    support
0          1.00        1.00        1.00          50
1          1.00        1.00        1.00          50
2          1.00        1.00        1.00          50
avg / total          1.00        1.00        1.00          150
[[50  0  0]
[0 50  0]
[0  0 50]]
```

Matplotlib

It is a 2D plotting library for Python programming that is used for data visualization to generalize various figures and high-quality image plots; with the help of the Matplot library, we can easily generalize the various diagrams like histogram, scatter plots, plot error charts, bar charts with just only a few lines of code.

Example 12.4:

```
import matplotlib.pyplot as plt
import numpy as np
# Prepare the data
x = np.linspace(0, 20, 200)

# Plot the data
plt.plot(x, x, label ='linear')

# Add a legend
plt.legend()

# Show the plot
```

```
plt.show()
```

Output:

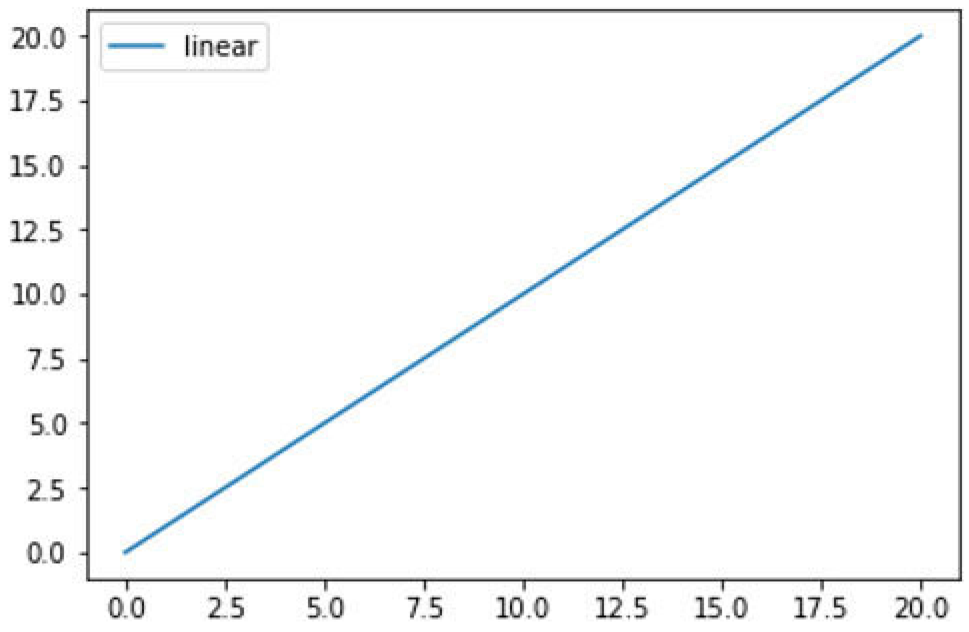


Figure 12.4: *Example of a plot*

Tensorflow

It was initially developed for internal use by the Google brain team. And came into existence in November 2015 under Apache licenses 2D. It helps to produce a machine learning models framework. It supports different types of toolkits for constructing machine learning models that provide different levels of abstraction.

Example

```
# import `tensorflow`
import tensorflow as tf
with tf.compat.v1.Session() as sess:
x = tf.constant(10.0)
y = tf.constant(8.0)
z = tf.multiply(x, y)
result = sess.run(z)
print (result)
```

Output:

```
80.0
```

Keras

Keras is widely used in Python programming. It provides us with a high-level neural network API capable of running on top of CNTK, TensorFlow, or Theano.

Example

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Pytorch

Python has a large number of tools and libraries that helps us with working computer vision, machine learning, and processing of natural language. It is one of the open-source libraries which can easily be used and learned quickly. Python can easily combine with the Python data science stack, even with the Numpy. There is a little bit of difference between the functioning of Numpy and Pytorch. It can also perform computation on tensors.

If Pytorch is not installed earlier, then it can be installed by the following command: > pip install torch

Example 12.7:

```
import torch
import math
dtype = torch.float
device = torch.device("cpu")
# Create random input and output data
x = torch.linspace(-math.pi, math.pi, 3000, device=device,
dtype=dtype)
```

```
y = torch.sin(x)
print (x)
print (y)
p=torch.linspace(3, 10, steps=5)
print (p)
q=torch.linspace(-10, 10, steps=5)
print (q)
```

Output:

```
tensor([-3.1416, -3.1395, -3.1374,   ...,   3.1374,   3.1395,   3.1416])

tensor([8.7423e-08, -2.0949e-03, -4.1901e-03,   ...,   4.1901e-03,
2.0949e-03, -8.7423e-08])
tensor([3.0000,   4.7500,   6.5000,   8.2500, 10.0000])
tensor([-10.,   -5.,     0.,     5.,   10.])
```

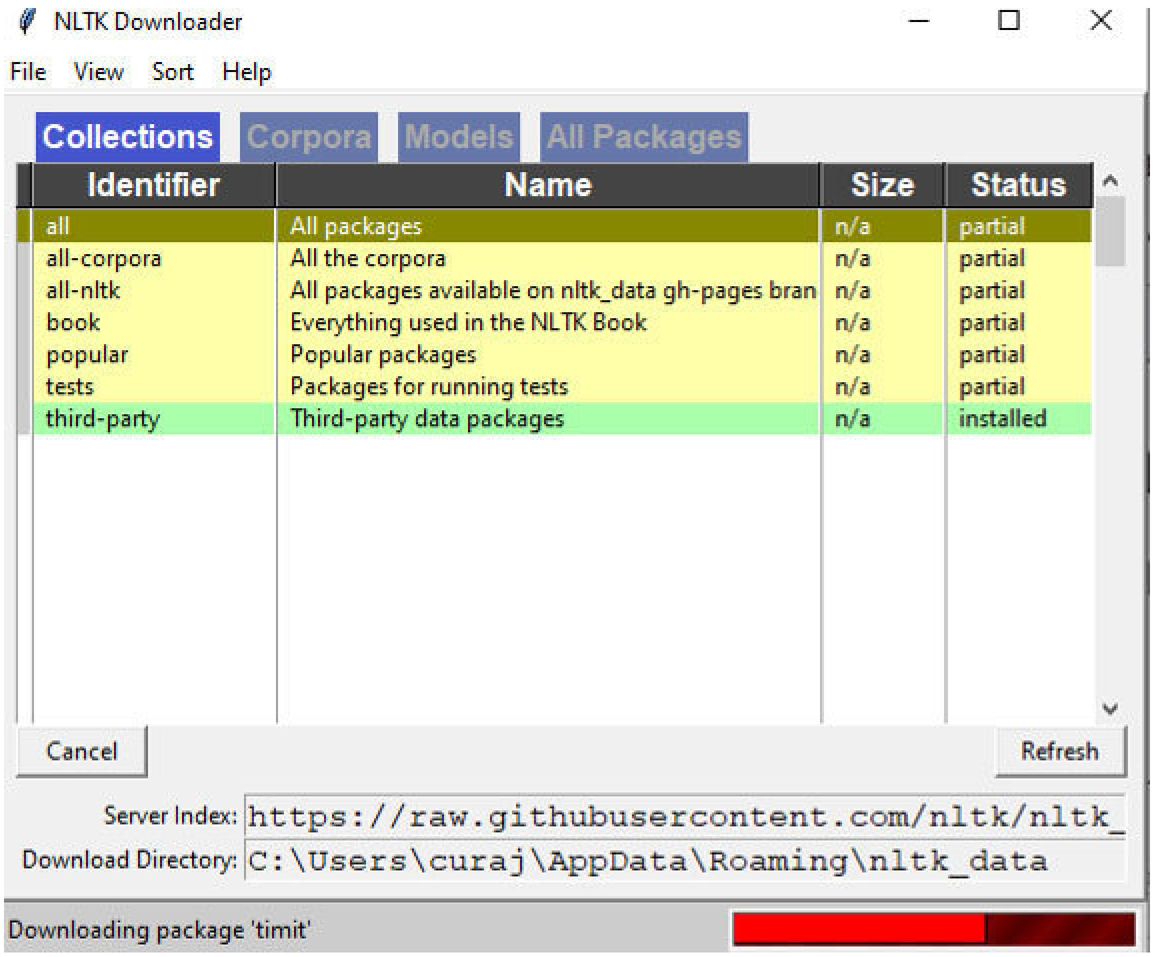
NLTK

NLTK stands for natural language toolkit and a library used in Python that helps us process natural language. NLTK is the most popular library for Python programming that helps in working with human languages. It offers to use the frame net, Wordvec, WordNet, and some others to the programmer, and it provides us with the interface and some lexical resources. Some of the primary functions of the NLTK is to find the keywords in documents, voice recognition, stemming of words, and lemmatizing and handwriting.

You can install NLTK 3.0, downloadable free of charge from before going on. To download the version needed for your platform, follow the instructions there.

Start the Python interpreter as before, after you have installed NLTK, and install the necessary data for the book by typing the following two commands at the Python prompt, then selecting the book set as shown in *figure*

```
>>> import nltk
>>> nltk.download()
```



Identifier	Name	Size	Status
all	All packages	n/a	partial
all-corpora	All the corpora	n/a	partial
all-nltk	All packages available on nltk_data gh-pages bran	n/a	partial
book	Everything used in the NLTK Book	n/a	partial
popular	Popular packages	n/a	partial
tests	Packages for running tests	n/a	partial
third-party	Third-party data packages	n/a	installed

Server Index: https://raw.githubusercontent.com/nltk/nltk_

Download Directory: C:\Users\curaj\AppData\Roaming\nltk_data

Downloading package 'timit'

Figure 12.5: *Downloading the NLTK Book Collection using nltk.download().*

Example 12.8:

```
import nltk
sentence = """At Five o'clock on Monday morning
... John didn't feel very good and want to do rest."""
```

```
tokens = nltk.word_tokenize(sentence)
print (tokens)
```

Output:

```
['At', 'Five', "o'clock", 'on', 'Monday', 'morning', '...', 'John', 'did',
"n't", 'feel', 'very', 'good', 'and', 'want', 'to', 'do', 'rest', '.']
```

Example: Loading Text

```
from nltk.book import *
```

Output:

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
```

text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908

Example 12.9: Searching Text

```
from nltk.book import *
text1.concordance("women")
```

Output:

```
Displaying 11 of 11 matches:
At creation's final day. And the women of New Bedford, they bloom like the
here these silent islands of men and women sat steadfastly eyeing several marble
Fishery, and so plainly did several women present wear the
countenance if not, and a still slighter shuffling of women' s shoes, and all was quiet again
the bitterest threat of your night - women, that beat head - winds round crone
soon. Now would all the waves were women, then I'd go drown, and chassee

by small tame cows and calves; the women and children of this routed host. N
```

the bowels, I suppose, as the old women talk Surgeon's Astronomy in the ba
up the live bodies of all the men, women, and children who were alive sevents. Lord! What an affection all old women have for tinkers. I know an old wom
ever would work for lonely widow old women ashore, when I kept my job - shop i

Understanding regression

Regression finds the relationships between the variables. For example, if we want to determine the dependency of the salaries on the other features like experience, education level, their role, and the city in which they work, etc., we can use regression. In regression, we relate the data of each employee. The data of each employee denotes the one observation in regression. In this, we assume the experience, city, education level, and role as independent features.

Linear regression

Our goal is to find the relationship between one or more. Our main motive is to find the relationship between one or more independent features and dependent features that are the continuous target in the linear regression model. If there is one feature, it is known as univariate linear regression, and if there is more than one feature, it is known as multiple linear regressions.

Example

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
# generate random data-set
np.random.seed(0)
x = np.random.rand(50, 1)
y = 2 + 3 * x + np.random.rand(50, 1)

# sckit-learn implementation
# Model initialization
```

```
regression_model = LinearRegression()
# Fit the data(train the model)
regression_model.fit(x, y)
# Predict
y_predicted = regression_model.predict(x)
# model evaluation
rmse = mean_squared_error(y, y_predicted)
r2 = r2_score(y, y_predicted)
# printing values
print ('Slope:', regression_model.coef_)

print ('Intercept:', regression_model.intercept_)
print ('Root mean squared error: ', rmse)
print ('R2 score: ', r2)
# plotting values
# data points
plt.scatter(x, y, s=20)
plt.xlabel('x')
plt.ylabel('y')
# predicted values
plt.plot(x, y_predicted, color='b')
plt.show()
```

Output:

```
Slope: [[2.90625874]]
Intercept: [2.45805209]
```

```
Root mean squared error: 0.08296096126059752
R2 score : 0.883001472727436024
```

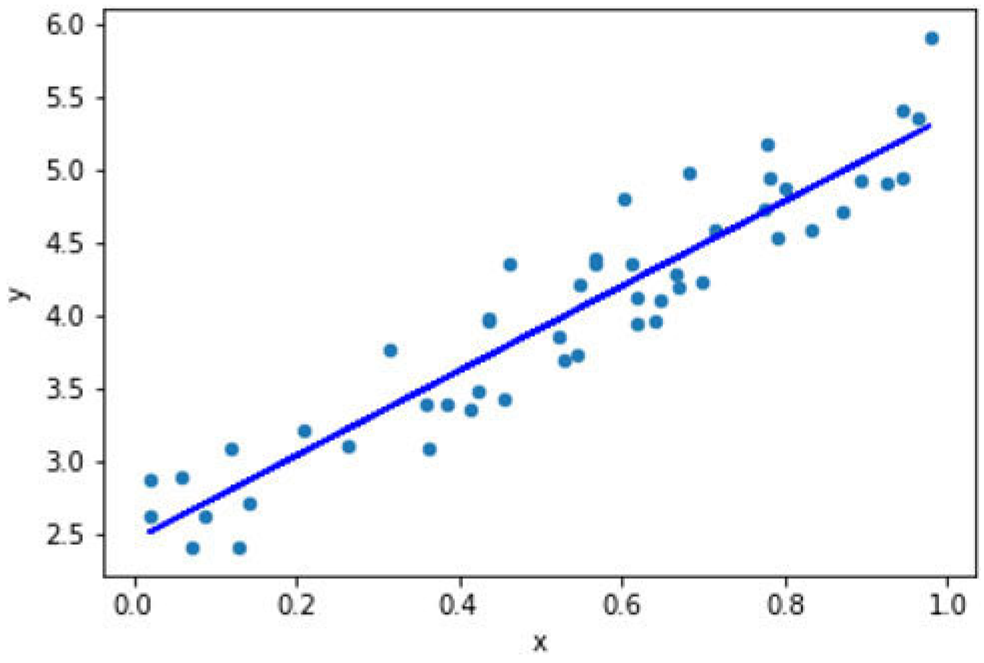


Figure 12.6: *Regression plot*

Non-linear regression

Non-linear regression is a type of polynomial regression. It is a method to create a mode that finds a non-linear relationship between the dependent and independent variables. We used non-linear regression when the data had a curving trend, and it produced more accurate findings than linear regression. This is because we already assume that the data is linear.

Example

```
import numpy, scipy, matplotlib
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from scipy.optimize import differential_evolution
import warnings
xData = numpy.array([20.1647, 19.019, 16.9580, 15.7683, 14.7044,
13.6269, 12.6040, 11.4309, 10.2987, 9.23465, 8.18440, 7.89789,
7.62498, 7.36571, 7.01106, 6.71094, 6.46548, 6.27436, 6.16543,
6.05569, 5.91904, 5.78247, 5.53661, 4.85425, 4.29468, 3.74888,
3.16206, 2.58882, 1.93371, 1.52426, 1.14211, 0.719035, 0.377708,
0.0226971, -0.223181, -0.537231, -0.878491, -1.27484, -1.45266,
-1.57683, -1.61717])
```

```
yData = numpy.array([0.644557, 0.641059, 0.637555, 0.634059,
0.634135, 0.631825, 0.631899, 0.627209, 0.622516, 0.617818,
0.616103, 0.613736, 0.610175, 0.606613, 0.605445, 0.603676,
0.604887, 0.600127, 0.604909, 0.588207, 0.581056, 0.576292,
0.566761, 0.555472, 0.545367, 0.538842, 0.529336, 0.518635,
0.506747, 0.499018, 0.491885, 0.484754, 0.475230, 0.464514,
0.454387, 0.444861, 0.437128, 0.415076, 0.401363, 0.390034,
0.378698])
def func(x, a, b, Offset):

return 1.0 / (1.0 + numpy.exp(-a * (x-b))) + Offset
# function for genetic algorithm to minimize (sum of squared
error)
def sumOfSquaredError(parameterTuple):
# do not print warnings by genetic algorithm
warnings.filterwarnings("ignore")
val = func(xData, *parameterTuple)
return numpy.sum((yData - val) ** 2.0)
def generate_Initial_Parameters():
# min and max used for bounds
maxX = max(xData)
minX = min(xData)
maxY = max(yData)
minY = min(yData)

parameterBounds = []
parameterBounds.append([minX, maxX]) # search bounds for a
parameterBounds.append([minX, maxX]) # search bounds for b
parameterBounds.append([0.0, maxY]) # search bounds for
Offset
# "seed" the numpy random number generator for repeatable
results
result = differential_evolution(sumOfSquaredError,
parameterBounds, seed=5)
return result.x

# generate initial parameter values
geneticParameters = generate_Initial_Parameters()
# curve fit the test data

fittedParameters, pcov = curve_fit(func, xData, yData,
geneticParameters)
print ('Parameter are:', fittedParameters)
modelPredictions = func(xData, *fittedParameters)
absError = modelPredictions - yData
SE = numpy.square(absError) # squared errors
MSE = numpy.mean(SE) # mean squared errors
RMSE = numpy.sqrt(MSE) # Root Mean Squared Error, RMSE
Rsquared = 1.0 - (numpy.var(absError) / numpy.var(yData))
print ('RMSE value is:', RMSE)
print ('R-squared is:', Rsquared)

# graph
def ModelAndScatterPlot(graphWidth, graphHeight):
```

```
f = plt.figure(figsize=(graphWidth/100.0, graphHeight/100.0),
dpi=100)
axes = f.add_subplot(111)
# first the raw data as a scatter plot
axes.plot(xData, yData, 'D')
# create data for the fitted equation plot
xModel = numpy.linspace(min(xData), max(xData))
yModel = func(xModel, *fittedParameters)
# now the model as a line plot
axes.plot(xModel, yModel)
axes.set_xlabel('X Data') # X axis data label
axes.set_ylabel('Y Data') # Y axis data label
plt.show()
plt.close('all')
w = 500
h = 500

ModelAndScatterPlot(w, h)
```

Output:

```
Parameter are: [0.21593498 -6.66784414 -0.35271173]
RMSE value is: 0.008432181989377313
R-squared is: 0.9886129643523501
```

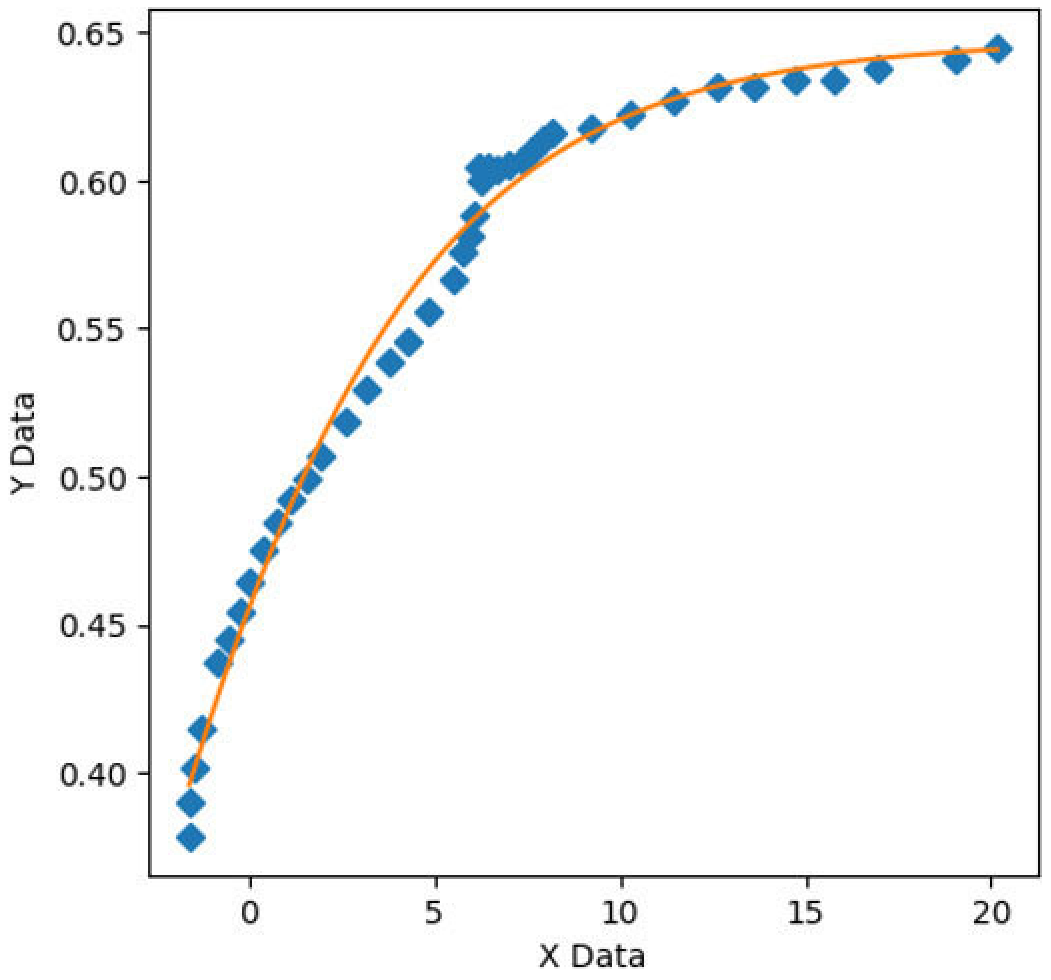


Figure 12.7: *Non-linear regression plot*

Introduction to classification

Classification refers to predictive models' problems that classify the input data into a class label. The task of a predictive model is mapping function from the input to the output Classification is a type of supervised learning where we have the output variables corresponding to the input variables. Classification is used in various domains such as in medical diagnosis, credit approval, target marketing, etc. For example, In Email, the message's detection as spam or non-spam is a classification problem. In this problem, we use binary classification that classifies messages as spam or non-spam in only two classes. In this classifier, we use the input variable as training data. So, in this case, the spam and non-spam known emails are used as the training data, and after the training, it can be used to detect the unknown emails by giving it as an input. The different types of classification predictive modeling problems are:

Binary classification

Multi-class classification

Multi-label classification

Binary classification

In binary classification, we classify the data into two class labels, for example:

Classification of emails as Spam or not.

Prediction of conversion buys or not.

Prediction of churn or not.

Some popular algorithms which are used for binary classification are:

Logistic regression

K-Nearest neighbors

Decision trees

Support vector machine

Naïve Bayes

Multi-class classification

Multi-class classification, unless binary classification, classifies the data into more than two classes, for example:

Classification of face

Classification of plant species

Recognition of optical character

The most widely used algorithm for machine learning algorithms are:

K-Nearest Neighbors

Decision Trees

Naïve Bayes

Random forest

Gradient Boosting

A binary classification algorithm can be used for multi-class problems. This can be done by using the various strategies of fitting multiple binary classification modes.

That fits for binary classification model for each class vs. all other classes.

That fits for binary classification model for every pair of classes.

We can also use these strategies of binary classification for multi-class classification includes:

Logistic regression

Support vector machine

Multi-label classification

In this type of classification, there are two or more class labels, where one or more class label is used to predict the examples. If we consider the photo classification, in which we have multiple objects in the photo, the model will predict the presence of various objects like bicycle, person, apple, etc. Multi-class classification is unlike binary classification, in which a single label of the class is predicted for any example. Multi-label classification is commonly used in predicting the multiple outputs in which every output is predicted by using Bernoulli probability distribution. For multi-label classification, we cannot directly use the binary classification algorithm or multi-class classification. We used a specialized version of the multi-label classification are:

Multi-label Decision tree

Multi-label Random forests

Multi-label Gradient boosting

Imbalanced classification

An imbalanced classification task is somewhat similar to a binary classification task in which the majority of examples in the training dataset belongs to the normal class and the minority belongs to the abnormal class. For example:

Fraud detection

Detection of outlier

Diagnosis tests in medical

The binary classification solves this problem; even some may require specialized techniques. From changing the composition of samples in the binary data set using the specialized technique to under-sampling the majority class or oversampling the minority class. For example:

Random sampling

Smote oversampling

A minority class was used to fit the model on the training dataset. Specialized modeling algorithm is used to train the dataset with minority classes such as cost-sensitive machine algorithm. For examples:

Cost-sensitive logistic Regression

Cost-sensitive Decision trees

Cost-sensitive support vector machine

Classification methods

K-Nearest Neighbors

One of the simplest classification algorithms is KNN that stands for the K-Nearest Neighbor, which classifies the data based on its similarities. This algorithm assigns the objects to the nearest class. In this, K represents the number of neighboring objects in the space of feature compared to classification of the object. For the classification of inputs, we need to perform some actions that are:

Measurement of the distance between the objects in the training samples.

Select the numbers of objects K that are classified in the class, which frequently occurs among the K-nearest neighbor.

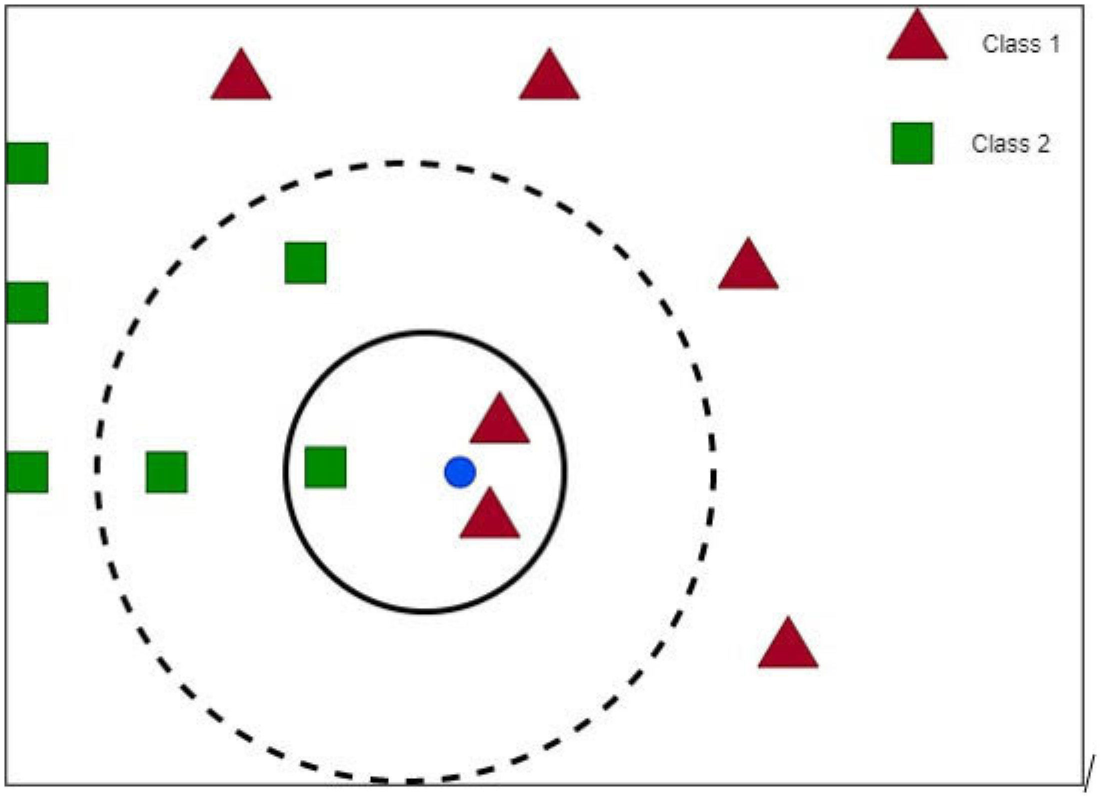


***Figure 12.8:** KNN classification*

The *figure 12.8* shows an example of KNN classification. In this figure, the test sample (blue circle) should be classified as either class 1 or class 2. If *K*=3 (single line), the test sample was assigned to class 1 because there are two class elements of class 1 and only one element of class 2.

If *K*=5 (dotted line), the test sample is assigned to class 2 because it has three elements and class 1 has two elements.

Implementation of this algorithm is very simple, robust for noisy training data, and if the training data is large, it is very effective.

The accuracy of the model depends on K's value, and its implementation cost is high as it needs to compute the distance between the objects to all the training samples.

Example

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn import datasets
dataset = pd.read_csv('Iris.csv')
#Summarize the Dataset
print ('dataset shape is :', dataset.shape)
print ('dataset head :', dataset.head(5))
print ('dataset description :', dataset.describe())
dataset.groupby('Species').size()
#Dividing data into features and labels
```

```
feature_columns = ['SepalLengthCm', 'SepalWidthCm',
'PetalLengthCm','PetalWidthCm']
X = dataset[feature_columns].values
y = dataset['Species'].values
#Spliting dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.2, random_state = 0) yu

# Instantiate learning model (k = 3)
classifier = KNeighborsClassifier(n_neighbors=5)

# Fitting the model
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print ('confusion matrix:\n',cm)
accuracy = accuracy_score(y_test, y_pred)*100
print ('Accuracy of our model is equal ' + str(round(accuracy,
2)) + ' %.')
```

dataset shape is : (150, 6)

```
dataset head :     Id   SepalLengthCm       ...
PetalWidthCm        Species
0    1              5.1       ...                  0.2   Iris-setosa
1    2              4.9       ...                  0.2   Iris-setosa
2    3              4.7       ...                  0.2   Iris-setosa
3    4              4.6       ...                  0.2   Iris-setosa


4    5              5.0       ...                  0.2   Iris-setosa
[5 rows x 6 columns]
dataset description
:                   Id   SepalLengthCm         ...
PetalLengthCm   PetalWidthCm
count   150.000000
150.000000         ...             150.000000      150.000000
mean       75.500000
5.843333         ...              3.758667         1.198667
std        43.445368
0.828066         ...               1.764420         0.763161
min         1.000000
4.300000         ...               1.000000         0.100000
25%        38.250000
5.100000         ...               1.600000         0.300000
50%        75.500000
5.800000         ...               4.350000         1.300000
75%       112.750000
6.400000         ...               5.100000         1.800000
max       150.000000
7.900000         ...               6.900000         2.500000
```

```
[8 rows x 5 columns]
confusion matrix:
[[11  0  0]
[0 12  1]
[0  0  6]]
The accuracy of our model is equal to 96.67 %.
```

Copy Iris.csv file into the current working directory before executing this program. You can get this file from a web search if not available.

Understanding decision trees

Decision trees help to build a classification model of machine learning in the form of tress structure. The decision tree splits the data into subsets in a hierarchal structure, and at the same time associated decision tree is incrementally developed. In the decision tree, the final output is decision nodes and leaf nodes. The decision tree has two or more branches and in leaf node gives the final result or classification of data into classes. The topmost decision mode in a decision tree is a root node. We can use both numerical and categorical data in the decision tree.

Assumptions while creating a decision tree:

We consider the whole training data set as the root in the beginning.

Mostly preferred feature values are categorical.

Before building the model, continuous values are discretized.

Based on attribute values, records are distributed recursively.

Ordering of attributes as root or internal nodes is done by using a statistical approach.

How do the decision trees work?

The accuracy of a decision tree depends on the strategy which splits the dataset. For both classification and regression, the criteria decision is different. In decision trees, we can use multiple algorithms to split the data from a node into two or more sub-nodes. The homogeneity increases between the sub-node by following splitting criteria, or we can say that the efficiency of the node increases concerning the target variable. In the decision, the tree splits the node based on given variables and selects the split, which gives us the most homogenous sub-nodes.

The selection of the algorithm is typically based on the type of target variables. Let us see some algorithms used in the decision tree:

ID3-which is an extended form of D3

C4.5-it is a successor of D3

CART-used for the classification and Regression Trees

MARS-multivariable adaptive regression splines

The top-down greedy search approach is used for the building of the ID3 algorithm for the possible branches and without backtracking. We choose in case of a greedy algorithm which is the best at that moment.

Steps in id3 algorithm

It starts with the set S as the root node.

With every iteration of the algorithm. We calculate the Entropy and information gain of every attribute.

Select those attributes which have the largest information gain or smallest entropy.

Then the selected attribute is used to split the set *S* to produce a subset of the data.

The algorithm is continuous to recur on the unselected subset.

Attribute selection measures:

If the dataset consists of many attributes, then to decide which attribute is used to split the data and place it at the root or the internal node at a different level is a complex task, and it cannot be solved by choosing any node randomly as the root node. It will give us a bad result with low accuracy. For this problem, the researcher finds the following solutions:

Entropy

Information gain

Gini index

Gain Ratio

Reduction in variance

Chi-square

Issues in decision tree

The following are some important issues in the decision tree:

Working with continuous attributes.

Avoiding overfitting

Super Attributes

Working with missing values.

Example 12.13:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from matplotlib import pyplot as plt
from sklearn import datasets
from sklearn import tree
import sklearn.metrics as metrics
#Loading datasets
iris_data = load_iris()
iris=pd.DataFrame(iris_data.data)
#priting features name of iris data
print ("Name of features : ", iris_data.feature_names)
#shape of datasets
print ("Dataset Shape: ", iris.shape)
#first six sample
print ("Dataset: \n",iris.head())
#priting samples and target
X = iris.values[:, 0:5]

Y = iris_data.target
#print (X)
#print (Y)
# Splitting the dataset into train and test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size =
0.3, random_state = 100)
# Decision tree classifier
clf= DecisionTreeClassifier(class_weight=None, criterion='gini',
random_state = 100)
#fitting the training data
clf.fit(X_train, y_train)
# prediction on random data
X=[[6.4,1.8,6.6,2.1]]
```

```
Y_pred=clf.predict(X)
print (Y_pred)
# prediction on X_test (testing data)
Y_pred=clf.predict(X_test)
print (Y_pred)
#Accuray of the model
print ('Accuracy:',metrics.accuracy_score(y_test, Y_pred))
#Decision making in decision tree
text_representation = tree.export_text(clf)
print (text_representation)
tree.plot_tree(clf)
```

Output:

```
Name of features :  ['sepal length (cm)', 'sepal width (cm)',
'petal length (cm)', 'petal width (cm)']
Dataset Shape:  (150, 4)
Dataset:

    0    1    2    3
0  5.1  3.5  1.4  0.2
1  4.9  3.0  1.4  0.2
2  4.7  3.2  1.3  0.2
3  4.6  3.1  1.5  0.2
4  5.0  3.6  1.4  0.2
[2]
```

```
[2 0 2 0 2 2 0 0 2 0 0 2 0 0 2 1 1 2 2 2 2 0 2 0 1 2 1 0
1 2 1 1 1 0 0 1 0
1 2 2 0 1 2 2 0]
Accuracy: 0.9555555555555556
|--- feature_2 <= 2.45
|   |--- class: 0
|--- feature_2 >  2.45
|   |--- feature_3 <= 1.65
|   |   |--- feature_2 <= 5.00
|   |   |   |--- class: 1
|   |   |--- feature_2 >  5.00
|   |   |   |--- feature_0 <= 6.05
|   |   |   |   |--- class: 1
|   |   |   |--- feature_0 >  6.05
|   |   |   |   |--- class: 2
|   |--- feature_3 >  1.65
|   |   |--- feature_2 <= 4.85
|   |   |   |--- feature_1 <= 3.10
|   |   |   |   |--- class: 2
|   |   |   |--- feature_1 >  3.10
|   |   |   |   |--- class: 1
|   |   |--- feature_2 >  4.85

|   |   |   |--- class: 2
```

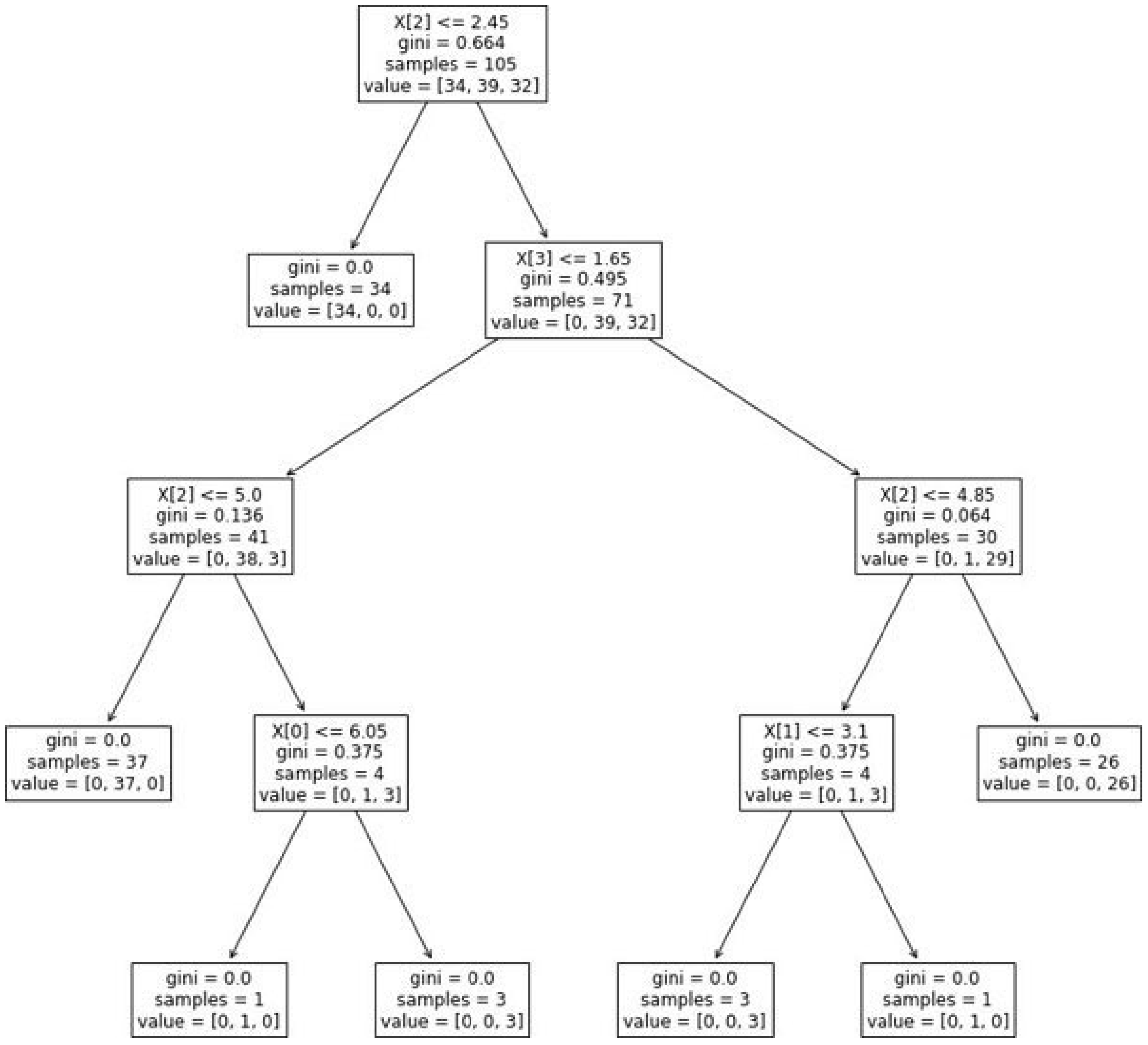
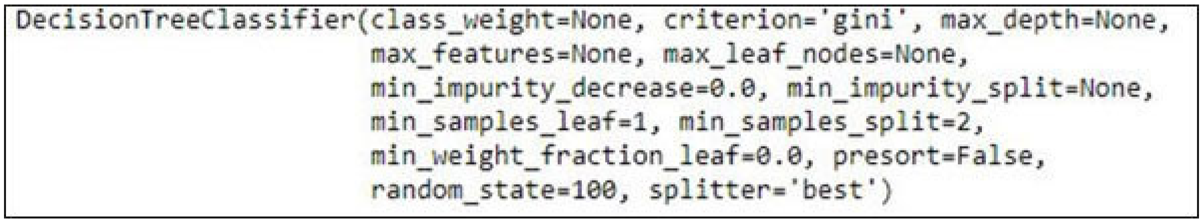


Figure 12.9: *Example of a decision tree*

If the following error occurs in plotting treeè AttributeError: module 'sklearn.tree' has no attribute then update scikit-learn by the following command.

```
>pip install -U scikit-learn
```

The following parameters can be used in the decision tree classifier:



```
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=100, splitter='best')
```

Logistic regression

It is a classification algorithm used to classify discrete data, for example, for the classification of email as spam or non-spam, fraud detection in online transactions, and classifying malignant tumor or Benign tumor. Logistic regression converts its output by using a logistic sigmoid function that returns a probability value.

A logistic regression algorithm is used for the classification of problems in machine learning. It is a predictive analysis algorithm by bussing the concept of probability. We can also call it a linear regression model, but we can use a more complex cost function in logistic regression. This function is called a sigmoid function or logistic function instead of a linear function. The limit of the cost function of the hypothesis of logistic regression is between 0 and 1. Therefore, if the linear function fails to represent it as the value greater than 1 or less than 0 that is not possible according to the hypothesis of linear regression, logistic regression hypothesis expectation can be represented as:

$$0 < h\theta \ (x) < 1$$

Sigmoid function

The sigmoid function is used to map the predicted values to probabilities—this function helps to map the real value to other values between 0 and 1. In machine learning, for the mapping of predicted values to the probabilities, we use the sigmoid function.

$$f(x) = \frac{1}{1 + e^{-(x)}}$$

Hypothesis representation

We used the hypothesis's formula by using linear regression; that is, we modify it for logistic regression, which means we expect the hypothesis value between 0 and 1. When using linear regression, we used a formula of the hypothesis, that is:

$$h\Theta(x) = \beta_0 + \beta_1 X$$

For logistic regression, we are going to modify it a little bit i.e.

$$\sigma\ (Z) = \sigma\ (\beta_0 + \beta_1 X)$$

We have expected that our hypothesis will give values between 0 and 1.

$$Z = \beta_0 + \beta_1 X$$

$$h\Theta\ (x) = sigmoid(Z)$$

$$\text{i.e. } h\Theta\ (x) = 1/(1 + e\wedge - (\beta_0 + \beta_1 X)$$

The Hypothesis of logistic regression can be represented as:

$$h\Theta\ (x) = \frac{1}{1 + e^{-(\beta 0 + \beta 1X)}}$$

Example 12.14:

```
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
#Load the data set
data = sns.load_dataset("iris")
data.head()
#Prepare the training set
# X = feature values, all the columns except the last column
X = data.iloc[:, :-1]
# y = target values, last column of the data frame

y = data.iloc[:, -1]
# Plot the relation of each feature with each species
#Split the data into 80% training and 20% testing
x_train, x_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
#Train the model
model = LogisticRegression()
```

```
model.fit(x_train, y_train) #Training the model
#Test the model
predictions = model.predict(x_test)
print (predictions)# printing predictions
#Check precision, recall, f1-score
print (classification_report(y_test, predictions))
print (accuracy_score(y_test, predictions))
```

Output:

```
['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor' 'setosa'
'versicolor' 'virginica' 'versicolor' 'versicolor' 'virginica' 'setosa'
'setosa' 'setosa' 'setosa' 'versicolor' 'virginica' 'versicolor'
'versicolor' 'virginica' 'setosa' 'virginica' 'setosa' 'virginica'
'virginica' 'virginica' 'virginica' 'virginica' 'setosa' 'setosa'
'setosa' 'setosa' 'versicolor' 'setosa' 'setosa' 'virginica'
'versicolor'
'setosa' 'setosa' 'setosa' 'virginica' 'versicolor' 'versicolor'
'setosa'
'setosa']
      precision     recall   f1-score    support
   setosa          1.00         1.00        1.00          19
versicolor         1.00        1.00        1.00          13
virginica        1.00        1.00        1.00          13

accuracy                             1.00          45
macro avg          1.00         1.00       1.00          45
```

```
weighted  avg       1.00            1.00      1.00          45
1.0
```

Support vector machine

The support vector machine is a machine learning model that is very powerful and versatile, capable of performing linear and non-linear data classification. SVM is the popular and most widely used model in machine learning, and if anyone is interested in machine learning, they should have this model in their toolbox. Support vector machine is well suited for the classification of complex medium-sized and small datasets. The *figure 12.10* demonstrates SVM classification:

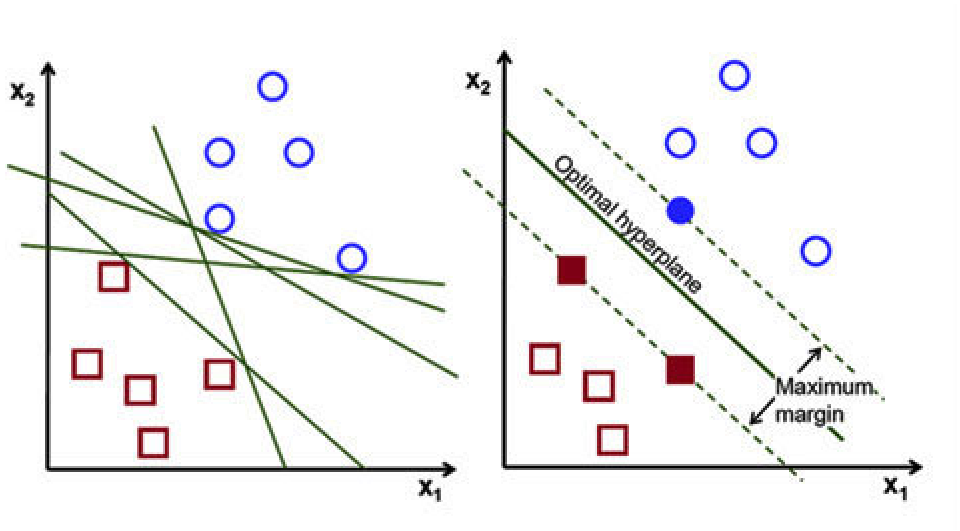


Figure 12.10: *Support vector machine classification*

To find a hyperplane in an N-dimensional space is the SVM algorithm's primary objective, which classifies the data. There may be more than one hyperplane chosen to separate the two classes of the data points and our objectives are to find a hyperplane that has the maximum margin, which means there should be maximum distance that helps for the future data point's classification by providing some reinforcement.

HYPERPLANES AND SUPPORT VECTORS

Hyperplanes are decision boundaries that classify the data. Data points that fall under either side of the plane can be attributed to different classes and the plane's dimension depending on the number of features. The hyperplane is a simple line in case the number of input features is 2 and a hyperplane is a two-dimensional plane if the number of input features is 3. If the number of features exceeds 3, it becomes difficult to imagine the hyperplane. Support vectors are data points close to the plane and impacts the hyperplane's positions and orientation. We can maximize the margin of the hyperplane by using support vectors. By deleting the support vectors the position of the hyperplanes will be changed. These points help us in the building of SVM. In SVM, we take the output of a linear function, and if the output is less than -1, we classify it as another class. That means the range of threshold values is 1 and -1 in SVM. We can obtain the reinforcement range of value ([1, -1]) as a margin. In the SVM algorithm, we try to maximize the margin between the data points and the plane.

Example 12.15:

```
#Import scikit-learn dataset library
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn import metrics
from sklearn.metrics import confusion_matrix
#Load dataset
cancer = datasets.load_breast_cancer()
# print the names of features
print ("Features: ", cancer.feature_names)
# print the label type of cancer('malignant' 'benign')

print ("Labels: ", cancer.target_names)
# print data(feature)shape
cancer.data.shape
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(cancer.data,
cancer.target, test_size=0.3,random_state=109) # 70% training
and 30% test
clf = svm.SVC(kernel='linear') # Linear Kernel
#Train the model using the training sets
```

```
clf.fit(X_train, y_train)
#Predict the response for test dataset
y_pred = clf.predict(X_test)
# Print Model Accuracy
print ("Confusion matrix:\n ",confusion_matrix(y_test, y_pred))
print ("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Output:

```
Features: ['mean radius' 'mean texture' 'mean perimeter' 'mean
area'
'mean smoothness' 'mean compactness' 'mean concavity'
'mean concave points' 'mean symmetry' 'mean fractal
dimension'
'radius error' 'texture error' 'perimeter error' 'area error'
'smoothness error' 'compactness error' 'concavity error'
'concave points error' 'symmetry error' 'fractal dimension error'
'worst radius' 'worst texture' 'worst perimeter' 'worst area'
'worst smoothness' 'worst compactness' 'worst concavity'
'worst concave points' 'worst symmetry' 'worst fractal
dimension']
Labels: ['malignant' 'benign']

Confusion matrix:
[[61   2]
[  4 104]]
Accuracy: 0.9649122807017544
```

Model evaluation methods

To measure model efficiency, model evaluation metrics are required. The choice of measurement criteria depends on the role of machine learning (such as classification, regression, ranking, clustering, topic modeling, and others). For several tasks, specific metrics are beneficial, such as precision-recall. The bulk of machine learning programs are supervised learning functions, such as classification and regression. We concentrate on metrics for these two supervised models of learning. The following are some of the metrics used in classification problems.

Confusion matrix

Accuracy

Precision

Sensitivity/ Recall

Specificity

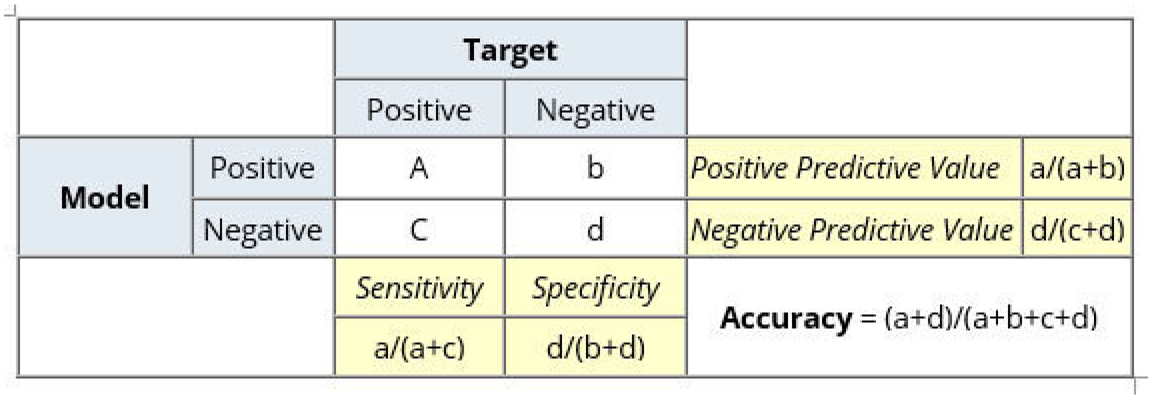
F1 score

ROC (Receiver operating characteristics) curve

AUC (Area under ROC curve)

Confusion matrix

Compared to the actual results (target value) in the data, a confusion matrix shows the number of right and wrong predictions made by the classification model. The matrix is NxN, where the number of destination values is N. (classes). Using the data in the matrix, the output of such models is generally evaluated. A 2x2 confusion matrix for two groups (positive and negative) is shown in the following table:



		Target			
		Positive	Negative		
Model	Positive	A	b	*Positive Predictive Value*	a/(a+b)
	Negative	C	d	*Negative Predictive Value*	d/(c+d)
		Sensitivity	*Specificity*	**Accuracy** = (a+d)/(a+b+c+d)	
		a/(a+c)	d/(b+d)		

Table 12.1: *Confusion matrix*

Accuracy

The percentage of the total number of predictions was correct. For classification issues, accuracy is a standard assessment metric. It's the number of accurate forecasts made as a percentage of all forecasts made.

Accuracy = (TP+TN)/total

Misclassification: (FP+FN)/total or (1-Accuracy)

Precision

What is the percentage of real true, out of the total expected true, as in how much the model correctly predicts? Accuracy can become an inaccurate criterion for assessing our success when we have a class imbalance.

$$Precision=TP/(TP+FP)$$

Sensitivity/recall

Recall calculates how many of the actual positives our model capture through labeling it as True Positive. Regarding all the examples that belong in the class, recall is defined as the fraction of examples predicted to belong to a class.

Recall: TP/(TP+FN)

Specificity

It is the percentage out of the overall real negative instances of negative instances. Therefore, here is the actual number of negative instances present in the dataset denominator (TN + FP). It is equivalent to recall, but the change is on the negative side.

Specificity: FP/(FP+TN)

F1 score

It is the harmonic means of accuracy and recall. It takes both of them into account, so the higher the F1 score, the better. See that if one goes low, the final *F1* score drops considerably because of the numerator item. So in the *F1* score, a model does well if the positive predicted is positive (precision) and does not skip positive and predict negative ones (recall).

*F1: (2*Recall*Precision)/(Recall+Precision)*

ROC (Receiver operating characteristics) curve

The ROC chart is similar to the charts of gain or rise. They provide a means of comparison between models of classification. The ROC chart displays the false positive rate (1-specificity) on the X-axis, the probability of target=1 when the true value is 0, against the true positive rate (sensitivity) on the Y-axis, the probability of target=1 when the true value is 1. Ideally, the curve would ascend rapidly to the top-left, meaning the model correctly predicts the cases. For a random construct, the diagonal red line is an example of the ROC graph is shown as follows:

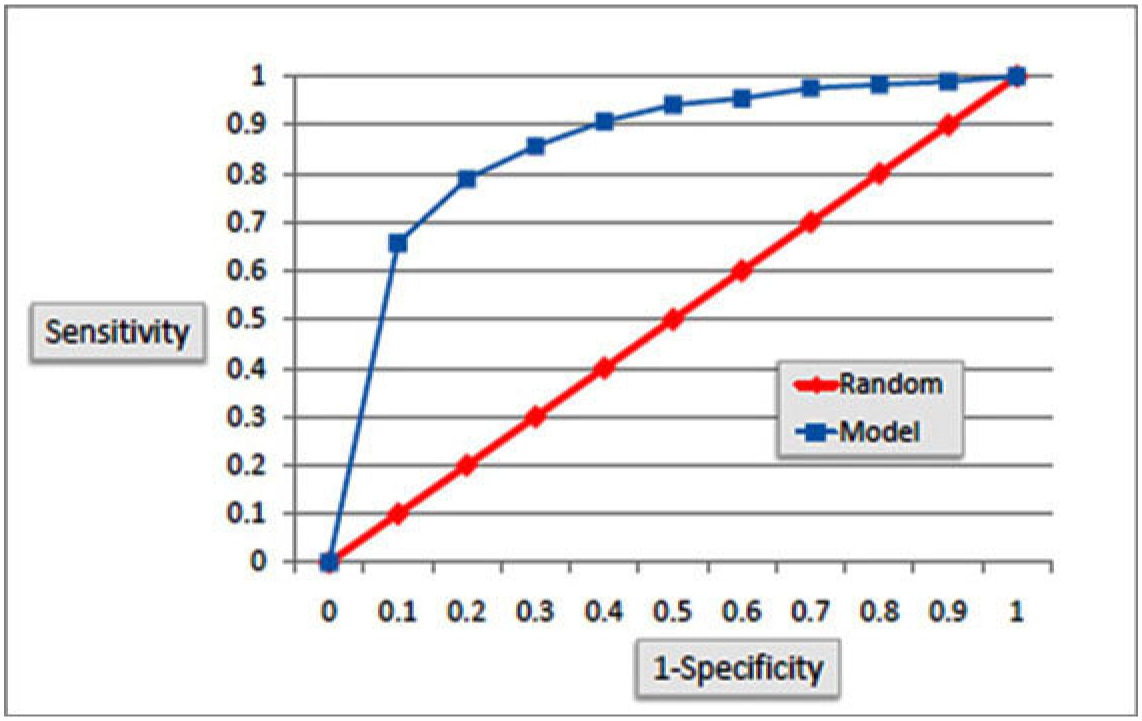


Figure 12.11: *ROC Curve*

AUC (Area under ROC curve)

The region under the ROC curve is also used as a measure of the classification models' consistency. There is a region under the curve of 0.5 for a random classifier, while AUC equals 1 for a perfect classifier. Most of the classification models have an AUC between 0.5 and 1 in action. An example of the AUC graph is shown as follows:

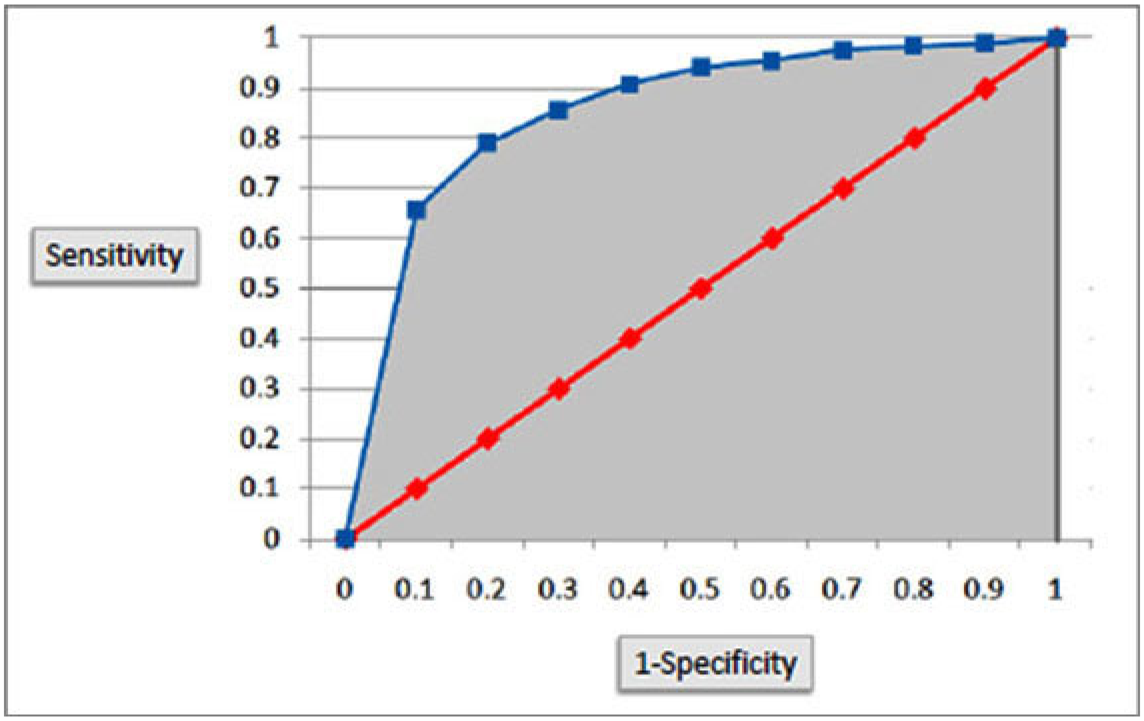


Figure 12.12: *Area under ROC Curve*

Conclusion

In this chapter, we discussed the concepts, applications, and methods of machine learning. Various examples and Python code are given for implementing machine learning approaches. Based on these examples, you can write your code for machine learning methods by setting different parameters and different datasets. In the next chapter, we will discuss clustering concepts in Python.

Points to remember

Machine learning just informs how computers do tasks on their own.

Machine learning systems are mainly categorized as supervised learning and unsupervised learning.

Regression is like classification, which uses the continuous value to categorize the data.

The number of accurate and inaccurate assumptions made by the classification model is seen in the confusion matrix.

Multiple choice questions

What is machine learning?

Machine Learning is a field of computer science.

Machine Learning is a type of artificial intelligence that extracts patterns out of raw data by using an algorithm.

Machine Learning focuses on allowing computer systems to learn from experience without being explicitly programmed or human intervention.

All the above

Which of the following is NOT supervised learning?

PCA

Decision Tree

Linear Regression

Naive Bayes

Which one of the following is used to build a model?

Validation data

Training data

Test data

Hidden data

The most widely used metrics and tools to assess a classification model are:

Confusion matrix

AUC

ROC

All the above

Which of the following Python libraries are used for data analysis and scientific computations?

Numpy

Scipy

Pandas

All the above

Answers

d

a

b

d

d

Questions

What are the popular algorithms uses in machine learning?

What is the difference between supervised and unsupervised machine learning?

Define precision and recall.

What is the ROC curve, and what does it represent?

Write some advantages and disadvantages of decision trees?

What is a confusion matrix, and why do you need it?

What is the difference between classification and regression?

What is meant by 'Training set' and 'Test Set'?

Discuss various Python libraries for machine learning.

Write a basic machine learning program to check a model's accuracy by importing any dataset using any classifier?

CHAPTER 13

Clustering with Python

Cluster analysis, also known as clustering, is an unsupervised machine learning approach for grouping unlabeled datasets. It aims to build clusters or groups from data points in a dataset with high intra-cluster similarity and low inter-cluster similarity. In this chapter, you will be familiar with various clustering methods and applications. Besides, we have demonstrated various clustering algorithms with the help of examples and codes.

Structure

In this chapter, we will cover the following topics:

The basic concepts of unsupervised learning

Various issues and applications of unsupervised learning

Types of clustering methods

Python codes to implement clustering methods

Objective

You can perform clustering using different approaches. This chapter demonstrates various clustering approaches with suitable examples. Also, you will be able to write Python codes for unsupervised machine learning approaches using different datasets and parameters.

Introduction to unsupervised learning

Unsupervised machine learning algorithms, without reference to known or labeled results, infer patterns from a dataset. With the exception of supervised machine learning, unsupervised machine learning techniques should not be best applied to a problem of regression or classification since we have no idea what the output data values may be. This makes it difficult for us to train the algorithm the way we would usually do. Instead, unsupervised learning can be used to find the data's inner principle. Such applications of techniques for unsupervised machine learning include:

Clustering helps us to divide the dataset into different groups automatically per the resemblance. Cluster analysis sometimes overestimates the correlation between groups and does not treat data sets as persons. For this purpose, cluster analysis is a bad option for client segmentation and targeting applications.

In the dataset, **anomaly detection** will automatically determine irregular data points. It helps detect fraudulent purchases, finds defective hardware pieces, or identifies an outlier during data entry triggered by a user mistake.

Sets of things that often occur together in your dataset are found through **Association** It is also used by retailers for basket research, as it enables analysts to discover items often bought at the same time and create more successful sales and marketing strategies.

For data pre-processing, **latent variable models** are frequently used, such as minimizing the cost function in a dataset (decreasing dimensional space) or decaying the dataset into different pieces.

Issues with unsupervised learning

In contrast to supervised learning assignments, unsupervised learning is complex.

How would we realize if the findings are relevant when there are no response labels available?

Let the specialist look at the outcomes (external evaluation).

Describing an objective clustering feature (internal evaluation)

Need for unsupervised learning

All sorts of unexplained trends in knowledge are identified through unsupervised machine learning.

Unsupervised approaches help you identify characteristics that can be beneficial for classification.

It occurs in real-time to evaluate and mark all the input data among students.

Unlabeled data from a computer is better to get than labeled data, which requires manual involvement.

Clustering

Cluster analysis, used for inferential statistical analysis to identify hidden trends or data grouping, is the most popular unmonitored form of learning. The clusters are simulated using a measure of similarity identified by metrics such as Euclidean or probabilistic distance. Clustering is the most critical unsupervised learning problem, and it deals with finding a structure in a set of unlabeled data, just like any other problem of this kind. *The method of arranging objects into groups whose members are similar in some way,* according to a broad concept of clustering. A cluster is thus a group of objects that are *similar* to one another but *dissimilar* to objects from other clusters. The *figure 13.1* shows the concept of the clustering process:

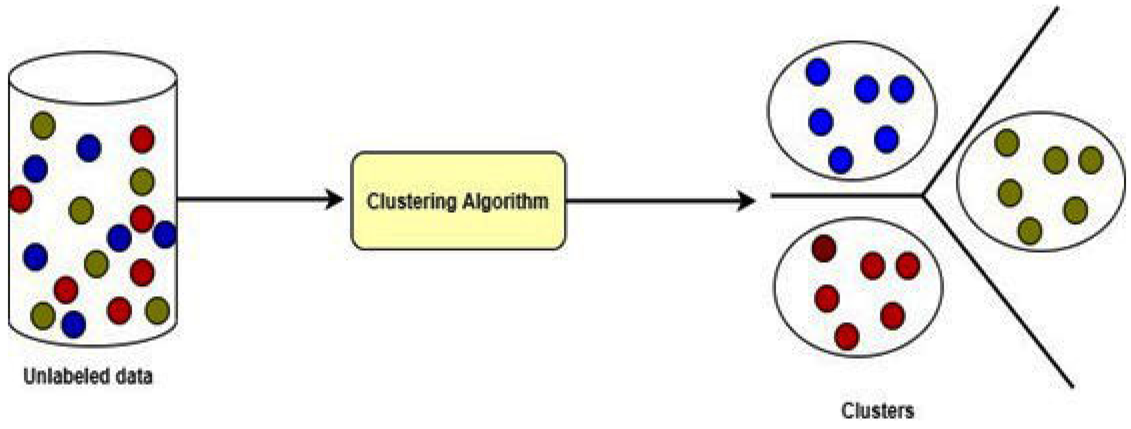


Figure 13.1: *Clustering process*

Clustering is used to figure out how a collection of unlabeled data is organized internally. But how do you know if clustering is good? It can be shown that there is no single *best* criterion that is independent of the clustering's goal. As a result, the user's responsibility is to include this condition for the clustering result to meet their requirements. To find a specific clustering solution, we must first identify the clusters' similarity measures.

Proximity measures

We need to specify a proximity measure for two data points to cluster them. The term "proximity" refers to how similar or dissimilar the samples are to one another. It can be defined as Similarity measure and distance. It is also shown in *figure*

Similarity measure **S** Large if and are similar.

Distance or dissimilarity measure **D** Small if and are similar.

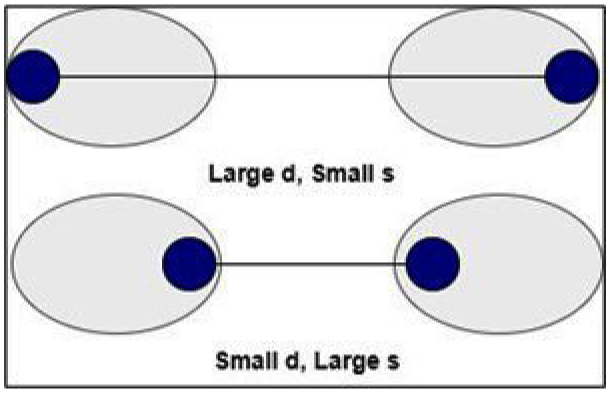


Figure 13.2: *Proximity measures*

Distance measuring methods

The selection of distance measures is an essential part of clustering. It controls the clusters' shape by defining how the similarity of two elements is determined. The Euclidean and Manhattan distances are the two most common distance measurement methods, and they are defined as follows:

Euclidean distance

$$d(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Manhattan distance

$$d(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)}$$

Where, *x* and *y* are two vectors of length

Clustering methods

Density-based methods: These techniques perceive the clusters as a dense area with some resemblance and separate them from the lower density space area. Such techniques have substantial precision and the potential to combine two clusters. Examples include **DBSCAN Spatial Clustering of Noise OPTICS Points for Clustering Structure** etc.

Hierarchical-based The clusters generated in this process form a hierarchy-based tree-type structure. The previously created one is used to shape new clusters. It is categorized into two parts:

Agglomerative (Bottom-up approach)

Divisive (Top-down approach)

Examples of hierarchical-based clustering methods are **CURE Using BIRCH Iterative Reducing Clustering and using** etc.

Partitioning Such techniques group the artifacts into k clusters, and each partition one cluster forms. This approach is used to optimize the similarity function of an objective criterion, such as when the distance is a significant parameter, such as K-means, **CLARANS Large Applications based on Randomized** etc.

Grid-based In this step, the data space is formulated into a finite number of cells that form a grid-like structure. All clustering operations performed on these grids are quick regardless of the number of data items, such as STING (Statistical Information Grid), wave cluster, CLIQUE (CLustering In Quest), etc.

K-means clustering

K-means is an algorithm for classifying or grouping the objects into K category numbers based on attributes/features. K is a positive integer number. The grouping is achieved by reducing the total of squares of distances between data and the corresponding vector cluster. Therefore, the object of K-mean clustering is to classify information.

The primary step of clustering with k-means is clear. We evaluate the amount of cluster K initially, and we presume the centroid or middle of these clusters. We can take any random objects as the original centroids, or the first K objects' sequence can also service such as the original vector. Then the algorithm K implies that the three steps below will be completed before convergence.

Example

```
# k-means clustering example
from numpy import unique
from numpy import where
from sklearn.datasets import make_classification
from sklearn.cluster import KMeans
from matplotlib import pyplot
# dataset defining
Data, _ = make_classification(n_samples=10000, n_features=2,
n_informative=2, n_redundant=0, n_clusters_per_class=1,
random_state=4)
# defining model
model = KMeans(n_clusters=3)
# Model fitting
model.fit(Data)

# assign a cluster
Assign = model.predict(Data)
# retrieving unique clusters
clusters = unique(Assign)
# create scatter plot for samples from each cluster
for c in clusters:
# get row indexes for samples with this cluster
ri = where(Assign == c)
# create scatter of these samples
pyplot.scatter (Data[ri, 0], Data[ri, 1])
# show the plot
pyplot.show()
```

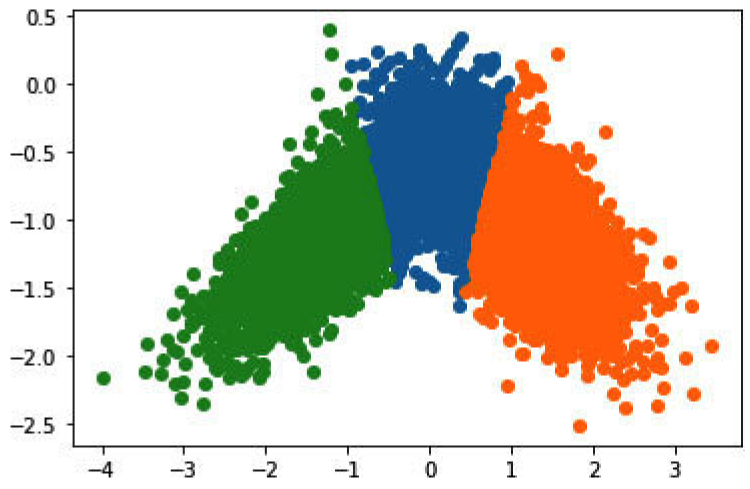


Figure 13.3: *K-mean clustering*

The number of clusters must be defined when using K-means clustering.

When clusters are of varying sizes, densities, and non-globular forms, K-means has issues.

Hierarchical Clustering (BIRCH)

Usually, hierarchical-based clustering is being used on hierarchical data, as you might get from a database of businesses or taxonomies. It constructs a tree of clusters to arrange everything from the top to bottom. It is more stringent than the other types of clustering, but it is ideal for particular types of data sets.

Example

```
# Example of birch clustering
from numpy import unique
from numpy import where
from sklearn.datasets import make_classification
from sklearn.cluster import Birch
from matplotlib import pyplot
# define dataset
Data, _ = make_classification(n_samples=10000, n_features=2,
n_informative=2, n_redundant=0, n_clusters_per_class=1,
random_state=4)
# defining model
model = Birch(threshold=0.5, n_clusters=4)
# Model fitting
```

```
model.fit(Data)
# assign a cluster
Assign = model.predict(Data)
# retrieve unique clusters
clusters = unique(Assign)
# create scatter plot for samples from each cluster
for c in clusters:
# get row indexes for samples with this cluster
ri = where(Assign == c)

# create scatter of these samples
pyplot.scatter(Data[ri, 0], Data[ri, 1])
# show the plot
pyplot.show()
```

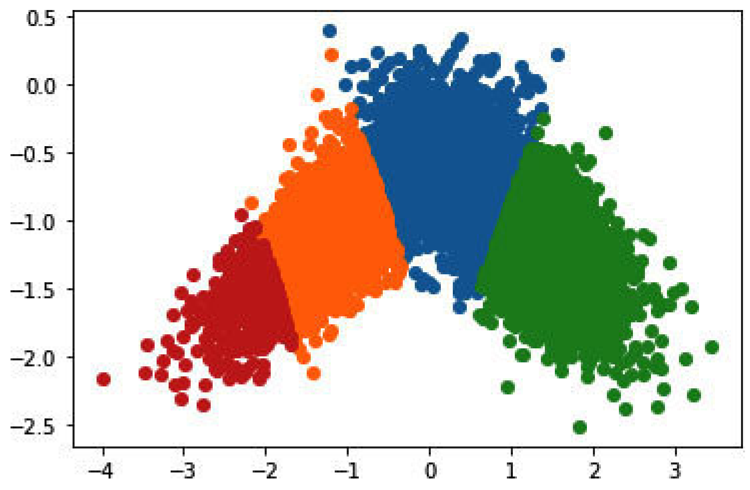


Figure 13.4: *BIRCH Clustering*

Density-based Clustering (DBSCAN)

In density-based clustering, data is clustered by high data point concentration areas, surrounded by low data point concentration areas. The algorithm essentially finds the sites with data points that are dense, and those clusters are named. The best thing about this is that there can be any form in the clusters. You are not limited to anticipated circumstances. Under this method, the clustering algorithms do not attempt to allocate cluster outliers, so they are neglected.

Example

```
# Example of DBSCAN clustering
from numpy import unique
from numpy import where
from sklearn.datasets import make_classification
from sklearn.cluster import DBSCAN
from matplotlib import pyplot
# define dataset
Data, _ = make_classification(n_samples=1000, n_features=2,
n_informative=2, n_redundant=0, n_clusters_per_class=1,
random_state=4)
# define the model
```

```
model = DBSCAN(eps=0.30, min_samples=10)
# fitting model and predicting clusters
a = model.fit_predict(Data)
# retrieve unique clusters
clusters = unique(a)
# create scatter plot for samples from each cluster
for c in clusters:
# get row indexes for samples with this cluster

ri = where(a == c)
# create scatter of these samples
pyplot.scatter(Data[ri, 0], Data[ri, 1])
# show the plot
pyplot.show()
```

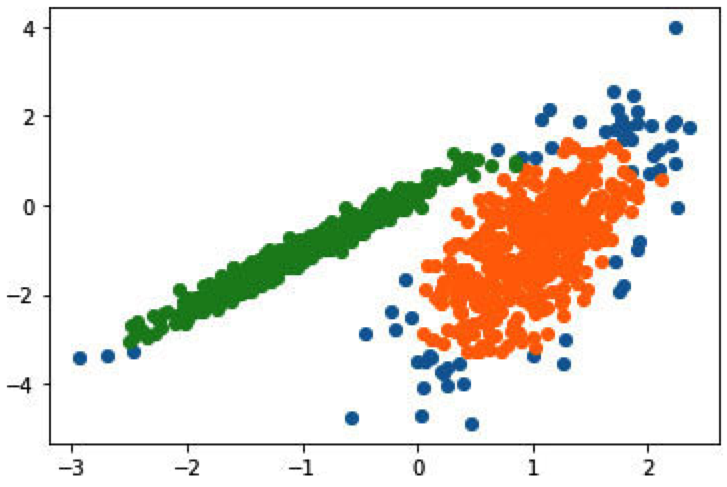


Figure 13.5: *DBSCAN Clustering*

OPTICS

OPTICS clustering is a simplified variant of DBSCAN. OPTICS stands for Ordering Points To Identify the Clustering Structure. It does not generate an actual clustering of a data set; instead, it provides an augmented ordering of the database representing the density-based clustering structure. This cluster-ordering includes knowledge that corresponds to density-based clustering's across a wide variety of parameter settings. Python code for OPTICS clustering is given below.

Example

```
# Example of OPTICS clustering
from numpy import unique
from numpy import where
from sklearn.datasets import make_classification
from sklearn.cluster import OPTICS
from matplotlib import pyplot
# define dataset
Data, _ = make_classification(n_samples=1000, n_features=2,
n_informative=2, n_redundant=0, n_clusters_per_class=1,
random_state=4)
# define the model
```

```
model = OPTICS(eps=0.5, min_samples=9)
# fit model and predict clusters
Assign = model.fit_predict(Data)
# retrieve unique clusters
clusters = unique(Assign)
# create scatter plot for samples from each cluster
for c in clusters:
# get row indexes for samples with this cluster

row_in = where(Assign == c)
# create scatter of these samples
pyplot.scatter(Data[row_in, 0], Data[row_in, 1])
# showing the plot
pyplot.show()
```

Output:

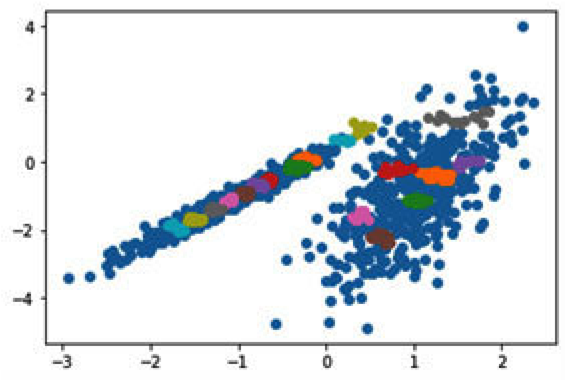


Figure 13.6: *OPTICS Clustering*

Conclusion

In this chapter, we discussed the concepts and various methods of clustering. Various examples and Python code are given for implementing clustering methods. Based on these examples, you can write your code for various clustering methods by setting different parameters and different datasets.

Points to remember

Clustering is an unsupervised problem of defining natural groups in the input data's feature space.

There are many clustering algorithms available, and no single approach is best for all datasets.

The term *proximity* refers to the degree of similarity or dissimilarity between the samples.

Multiple choice questions

Which of the following is required by K-means clustering?

Distance metric

Number of clusters

Initial guess of cluster centroids

All the above

Which of the following is NOT a clustering method?

K-Mean

Decision Tree

DBSCAN

OPTICS

For clustering, we do not require.

Labeled data

Unlabelled data

Numerical data

Categorical data

Which of the following(s) are applications of unsupervised machine learning?

Anomaly detection

Clustering

Association mining

All the above

Which of the following uses the merging approach?

Hierarchical clustering

Partitional clustering

Density-based clustering

All of the above

Answers

d

b

a

d

a

Questions

What do you mean by unsupervised learning?

Discuss various issues of unsupervised learning?

What are the distance measuring methods?

Write the difference between clustering and classification.

Write some advantages and disadvantages of K-Mean clustering.

Write applications of clustering in different fields.

Explain clustering methods.

Discuss various parameters required for implementing a clustering method.

Index