

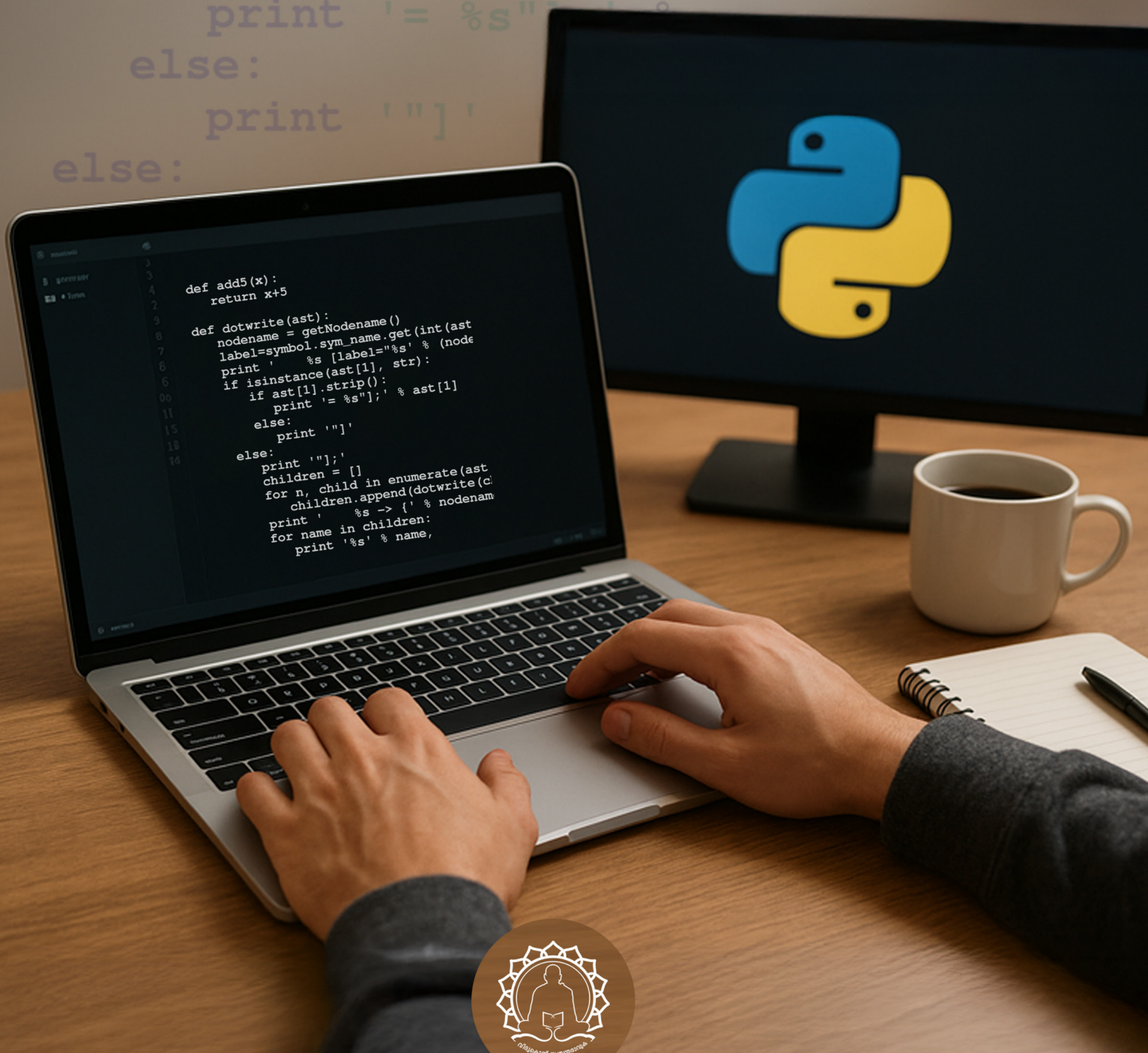
# PROGRAMMING WITH PYTHON

COURSE CODE: B21CA07DC

Bachelor of Computer Applications

Discipline Core Course

Self Learning Material



SREENARAYANAGURU  
OPEN UNIVERSITY

## SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

# SREENARAYANAGURU OPEN UNIVERSITY

## Vision

*To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.*

## Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

## Pathway

Access and Quality define Equity.

# Programming with Python

Course Code: B21CA07DC

Semester - IV

**Discipline Core Course**  
**Undergraduate Programme**  
**Bachelor of Computer Applications**  
**Self Learning Material**  
(With Model Question Paper Sets)



SREENARAYANAGURU  
OPEN UNIVERSITY

**SREENARAYANAGURU OPEN UNIVERSITY**

The State University for Education, Training and Research in Blended Format, Kerala



# PROGRAMMING WITH PYTHON

Course Code: B21CA07DC

Semester- IV

Discipline Core Course

Bachelor of Computer Applications

## Academic Committee

Dr. Aji S.  
Sreekanth M. S.  
P. M. Ameera Mol  
Dr. Vishnukumar S.  
Shamly K.  
Joseph Deril K. S.  
Dr. Jeeva Jose  
Dr. Bindu N.  
Dr. Priya R.  
Dr. Ajitha R. S.  
Dr. Anil Kumar  
N. Jayaraj

## Development of the Content

Dr. K.G. Krishnakumar  
Shamin S.  
Subi Priya Laxmi S.B.N.  
Aswathy V.S.  
Sreerekha V.K.  
Anjitha A.V.  
Dr. Kanitha Divakar  
Greeshma P.P

## Review and Edit

Dr. Ashutosh Kumar Bhatt

## Linguistics

Dr. Ashutosh Kumar Bhatt

## Scrutiny

Shamin S., Subi Priya Laxmi S.B.N.,  
Greeshma P.P., Sreerekha V.K.,  
Anjitha A.V., Aswathy V.S.,  
Dr. Kanitha Divakar.,

## Design Control

Azeem Babu T.A.

## Cover Design

Jobin J.

## Co-ordination

Director, MDDC :  
Dr. I.G. Shibi  
Asst. Director, MDDC :  
Dr. Sajeevkumar G.  
Coordinator, Development:  
Dr. Anfal M.  
Coordinator, Distribution:  
Dr. Sanitha K.K.



Scan this QR Code for reading the SLM  
on a digital device.

Edition  
July 2025

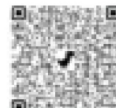
Copyright  
© Sreenarayanaguru Open University

ISBN 978-81-989642-7-4



All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

[www.sgou.ac.m](http://www.sgou.ac.m)



Visit and Subscribe our Social Media Platforms

Dear

With immense joy and excitement, I extend my heartfelt greetings to all of you and warmly welcome you to Sreenarayanaguru Open University.

Established in September 2020 as a state-driven initiative, Sreenarayana-guru Open University is dedicated to advancing higher education through open and distance learning. Our vision is guided by the principle of “ac-cess and quality define equity,” laying the foundation for a celebration of excellence in education. I am delighted to share that we are steadfast in our commitment to uphold the highest standards and refrain from com-promising on the quality of education we offer. The university draws its inspiration from the legacy of Sreenarayana Guru, a revered figure in the Indian renaissance movement. His name serves as a constant reminder for us to prioritize quality in all our academic endeavors.

Sreenarayanaguru Open University operates within the practical frame-work of the widely recognized “blended format.” Acknowledging the constraints faced by distance learners in accessing traditional classroom settings, we have curated a pedagogical approach centered on three main components: Self Learning Material, Classroom Counselling, and Virtual Modes. This comprehensive blend is poised to deliver dynamic learning and teaching experiences, maximizing engagement and effectiveness. Our unwavering commitment to quality ensures excellence across all aspects of our educational initiatives.

The University aims to offer you an engaging and stimulating educational environment that fosters active learning. The SLM is designed to offer a comprehensive and cohesive learning experience, fostering a deep interest in the study of technological advancements in IT. Careful consideration has been given to ensure a logical progression of topics, facilitating a clear understanding of the discipline’s evolution. The curriculum is thoughtfully crafted to provide ample opportunities for students to navigate through the current trends in information technology. Furthermore, this course is designed to provide essential insights into computer hardware, software classification, and foundational HTML concepts crucial for web develop-ment.

We assure you that the university student support services will closely stay with you for the redressal of your grievances during your student-ship. Feel free to write to us about anything that seems relevant regarding the academic programme.

Wish you the best.



Regards,  
Dr. Jagathy Raj V.P.

01-07-2025

# Contents

<b>Block 01</b>	<b>Introduction to Python, Data Structures and Operations</b>	<b>1</b>
Unit 1	Introduction to Python	2
Unit 2	Operators in Python	21
Unit 3	Data Types in Python	31
Unit 4	Built-in Methods of Data Structures	42
<b>Block 02</b>	<b>Decision making, Loops, Comprehensions, Functions, Modules &amp; Packages</b>	<b>53</b>
Unit 1	Decision Making and Loops	54
Unit 2	Comprehensions	68
Unit 3	Functions	78
Unit 4	Modules & Packages	94
<b>Block 03</b>	<b>File Handling, Object-Oriented Programming, Exception Handling and Regular Expressions</b>	<b>108</b>
Unit 1	File Handling	109
Unit 2	Object-Oriented Programming	116
Unit 3	Exception Handling and Regular Expressions	133
Unit 4	Regular Expressions	146
<b>Block 04</b>	<b>Database Programming, Familiarizing NumPy, Matplotlib and Pandas</b>	<b>155</b>
Unit 1	Database Programming	156
Unit 2	Familiarising NumPy	166
Unit 3	Introduction to Matplotlib	178
Unit 4	Introduction to Pandas	194
<b>Model Question Paper Sets</b>		<b>211</b>



## **BLOCK 1**

# **Introduction to Python, Data Structures and Operations**



# Introduction to Python

## Learning Outcomes

The learner will be able to :

- ◆ familiarise with the fundamentals of Python programming
- ◆ explore to set up a Python programming environment
- ◆ identify statements, variable names, input, and output function

## Prerequisites

Why Python? A Language for everyone!

Picture yourself giving simple instructions to a computer, just like talking to a friend. That's where programming languages come in. They act as a bridge between humans and computers, allowing us to communicate with computers efficiently. But not all programming languages are easy to learn. Some have complex rules, tricky symbols and a steep learning curve.

Now, what if there was a language that made coding as simple as writing a sentence in English? A language that removes unnecessary complexity and allows even beginners to create powerful programs with ease. No? That's the Language **Python**!



Python is one of the most popular and beginner-friendly programming languages in

the world today. It is free, open-source and widely used across industries - from web development to artificial intelligence. Unlike languages like C++ and Java, Python doesn't require complex syntax with braces and semicolons. Its clean and readable structure makes it easy to write, understand, and debug.

But that's not all! Python is also one of the fastest-growing languages because of its flexibility and simplicity. Whether you want to build a game, analyze data, develop websites, or automate tasks, Python is the perfect tool to get started.

Are you ready to explore the power of Python and see how effortlessly you can bring your ideas to life? Let's explore the world of Python and uncover the endless possibilities it offers!

## Keywords

Program, Programming Language, Compiler, Interpreter, Variable name

## Discussion

### Introduction

Python was created by Guido van Rossum. It is a well-designed programming language. Python is useful for accomplishing real-world tasks. Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

We can use Python for everything from website development, IoT, gaming, robotics, implementing standalone programs, and many more. Python is used widely to implement complex Internet services like search engines, cloud storage and tools, social media, and so on. For example, Google uses Python language to make the search engine better and more efficient. Google's main search algorithms are written in C++ and Python. One of the most popular languages used in Machine learning is Python. The availability of a wide collection of library functions of Python makes the programming easy and effective.

When Python is installed on a computer, it installs several components such as an interpreter and supporting library. The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications. The set of instructions written in a high-level programming language (For example, Python) is the source code and the file is called a source file.



## Python Goals

- ◆ An easy and intuitive language
- ◆ Open-source
- ◆ Source code is as understandable as plain English;

What makes Python special?

- ◆ Easy to learn
- ◆ It's easy to use for writing programs or scripting
- ◆ Easy to understand
- ◆ It's easy to obtain, install and deploy
- ◆ Python is free and open-source
- ◆ Multiplatform; Available on Windows, Unix operating systems, and macOS.

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons: The high-level data types allow you to express complex operations in a single statement. Statement grouping is done by indentation instead of beginning and ending brackets. No variable or argument declarations are necessary.

**Python is extensible:** You can add new features or use code from other languages like C or C++. This makes it flexible and powerful, allowing developers to extend Python's capabilities by integrating it with existing software or special libraries, such as those for graphics processing. If a certain functionality is not available in Python, programmers can write additional code in another language and connect it with Python to enhance performance and access special features. This makes Python a great choice for building complex applications that require advanced functions from other programming languages.

## Advantages:

Python is a high-level programming language, making it easier for developers to write and understand code without worrying about low-level hardware details. Its user-friendly data structures allow efficient data management, simplifying tasks like data manipulation and storage. For example, Python's list and dictionary structures are commonly used in web development frameworks like Django to manage user sessions and database records efficiently.

Being open-source, Python is freely available for use and modification, with a large and active community constantly contributing new tools and improvements. It is a versatile language, known for its simplicity, readability, and ease of learning, making it an excellent choice for both beginners and experienced programmers. Python supports both object-oriented and procedural programming, providing flexibility in coding styles. For instance, YouTube uses Python to handle different parts of its website, including video viewing and administrative tasks.

Python is portable, meaning it can run on different operating systems without modification, and interactive, allowing real-time code execution. It also benefits from third-party modules and extensive libraries like NumPy for numerical computations and Pandas for data analysis, making it a powerful tool for various applications. Since Python is a dynamically typed language, it automatically assigns data types based on values, reducing the need for manual declarations. Its efficiency and clean object-oriented design enhance performance, making it ideal for building prototypes with minimal coding. Additionally, Python's capabilities extend to emerging fields such as the Internet of Things (IoT) and machine learning, ensuring its relevance in modern technology. Being an interpreted language, Python executes code line by line, making debugging and testing easier for developers. Its cross-platform compatibility allows applications to run seamlessly on different operating systems, enhancing its flexibility and usability.

### **Disadvantages:**

**Global Interpreter Lock:** It is a mechanism in Python that prevents multiple threads from executing Python code at once. This can limit the parallelism and concurrency of some applications.

### **Memory consumption is very high**

**Dynamically typed:** The types of variables can change at runtime. This can make it more difficult to catch errors and can lead to bugs.

**Packaging and versioning:** Python has a large number of packages and libraries, which can sometimes lead to versioning issues and package conflicts.

**Lack of strictness :** Unlike statically typed languages such as C++ or Java, Python is dynamically typed, meaning variables do not need to be explicitly declared with a data type. While this makes coding easier, it can also lead to unexpected errors at runtime that would have been caught earlier in a strictly typed language.

Let's start the journey of learning Python programming by printing a message "hello world" program.

To write a program using the Python programming language, we need an IDE. An integrated development environment (IDE) is an application that provides facilities for software development. IDE consists of an editor to type the program, a facility to highlight the mistakes identified by the IDE, and other features to develop an application without spending much time.

A number of IDEs are available for programming in Python. Since we are new to Python programming, let's start with Jupyter Notebook Online, a simple environment for Python programming.

**Jupyter Notebooks allows:** creation and execution of Python programs by integrating code and its output into a single document. It opens the IDE in a standard web browser.



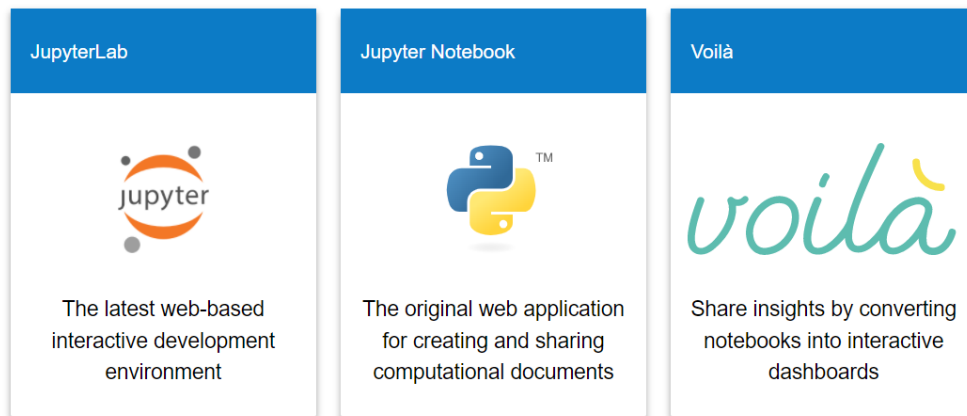


Fig 1.1.1 Python IDE

### 1.1.1 How to Start

Let's start Jupyter Notebook Online as shown in fig 1.1.2 by opening the link <https://jupyter.org/try>

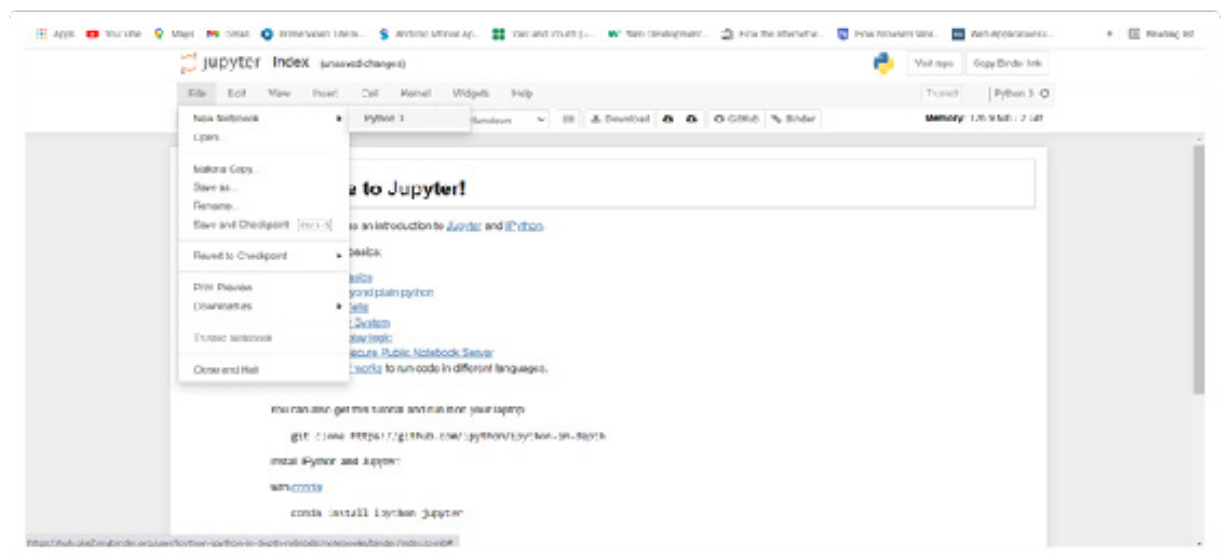


Fig 1.1.2 Python IDE

Start a new workbook as shown in fig 1.1.3. We can write the program in the cell provided by the IDE.

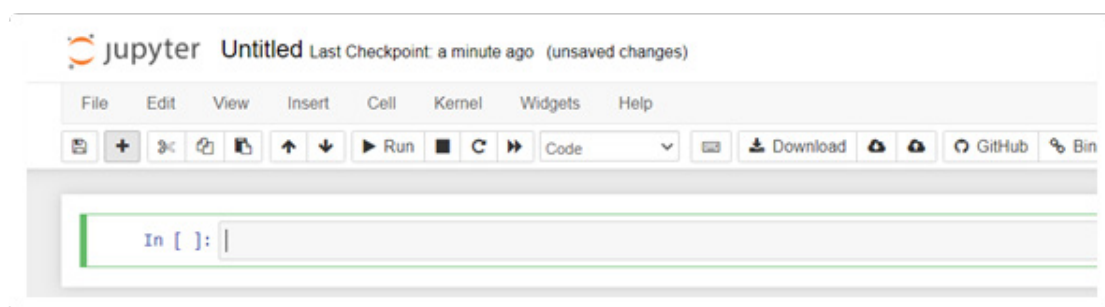


Fig 1.1.3 Python IDE

Fig 1.1.4 shows python IDE, here we can write programs

Type print (“Hello World”) as shown in fig 1.1.4 below

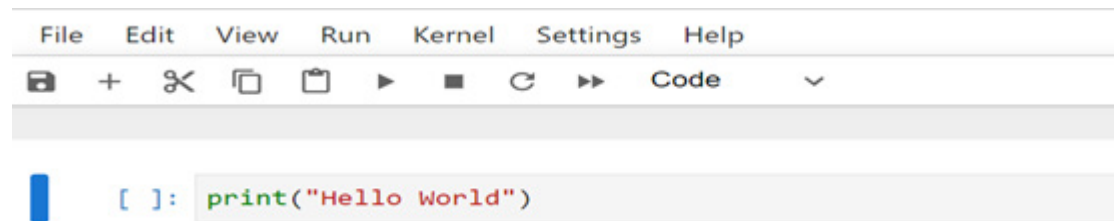


Fig 1.1.4 Python IDE

Click on the Run button and click on Run Selected Cells or click on  as shown in fig 1.1.5 to execute the program and observe the result.

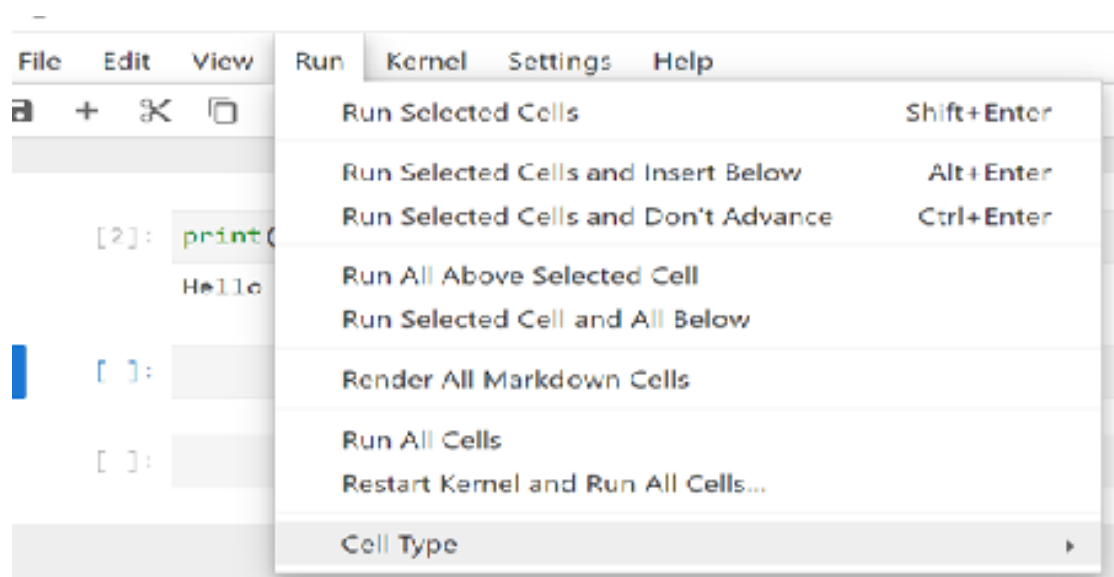


Fig 1.1.5 Python IDE

The following fig 1.1.6 shows the result will be displayed.

```
: print("Hello World")  
  
Hello World
```

Fig 1.1.6 Python IDE

Congratulations, you have created a Python program and executed the same to see the result.

**Let's try the following addition program.**

The following are the steps to use our addition application.

1. Start
2. Input the first number and store it in variable a
3. Input the second number and store it in variable b
4. Add the two numbers and store the result in variable c  
 $c = a + b$
5. Display the result stored in c
6. End

Type the program given below as shown in fig 1.1.7 in the place where we have typed hello world or create a new notebook and type the program.

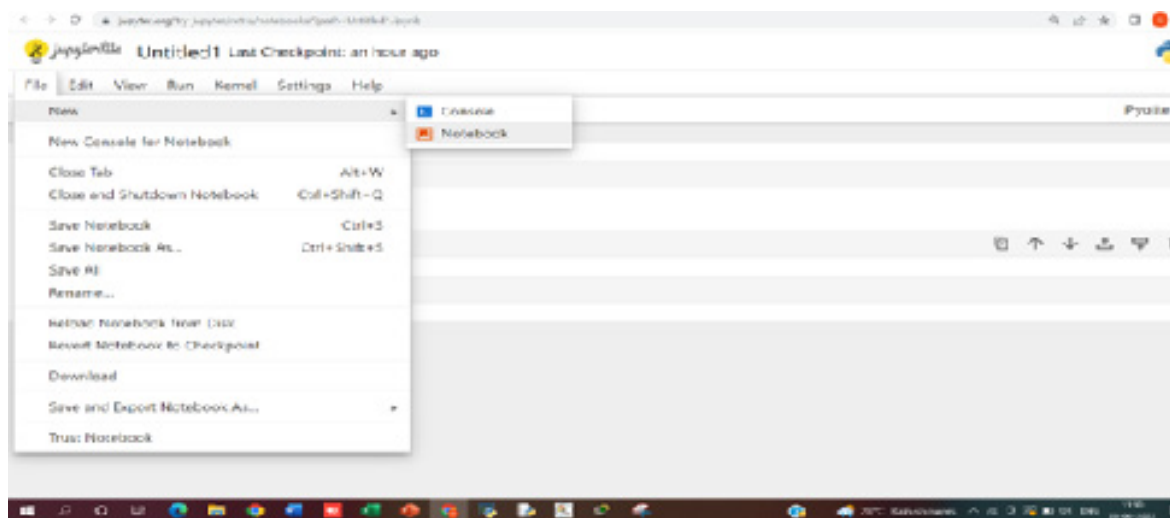


Fig 1.1.7 Python IDE

Note:

### In Python

- ◆ **int** – represents integers. Example 450, 67, 4
- ◆ **float** – represents decimal numbers. Example 30.5

**Write the following program in the IDE as shown in fig 1.1.8 and click on Run button to execute the program.**

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = a + b
print(f"The sum of {a} and {b} is {c}")
```

Output of the above program is

Enter first number: 5

Enter second number: 7

The sum of 5 and 7 is 12

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = a + b
print(f"The sum of {a} and {b} is {c}")
```

Fig 1.1.8 Python IDE

While running the program you will be prompted to enter the first number and second number. For example, enter 4 as the first number and 3 as the second number.

The result displayed will be: The sum of 5 and 7 is 12

This simple addition program has used different variable names such as a, b and c. Don't worry about the syntax and structure of this program yet—In this program, we aim to get a basic idea about Python programming .

### Download and install Python

Python is already installed with the Linux operating system. Python's infrastructure is intensively used by many Linux OS components.

You can check it by typing the command **python3** in the terminal. You will get the following window if Python is already installed.

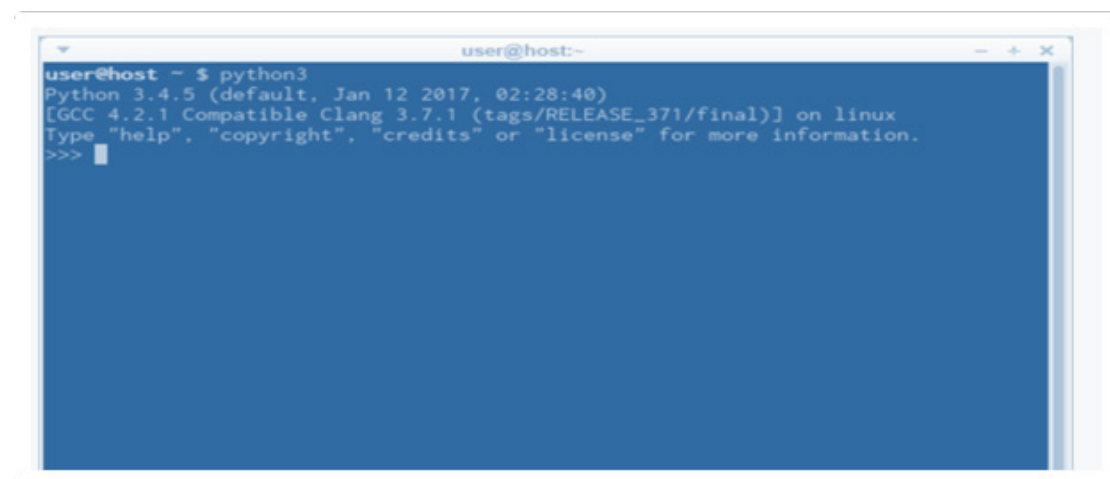


Fig 1.1.9 Python IDE

If you're a **Windows or macOS user**, download and install. Open <https://www.python.org/>

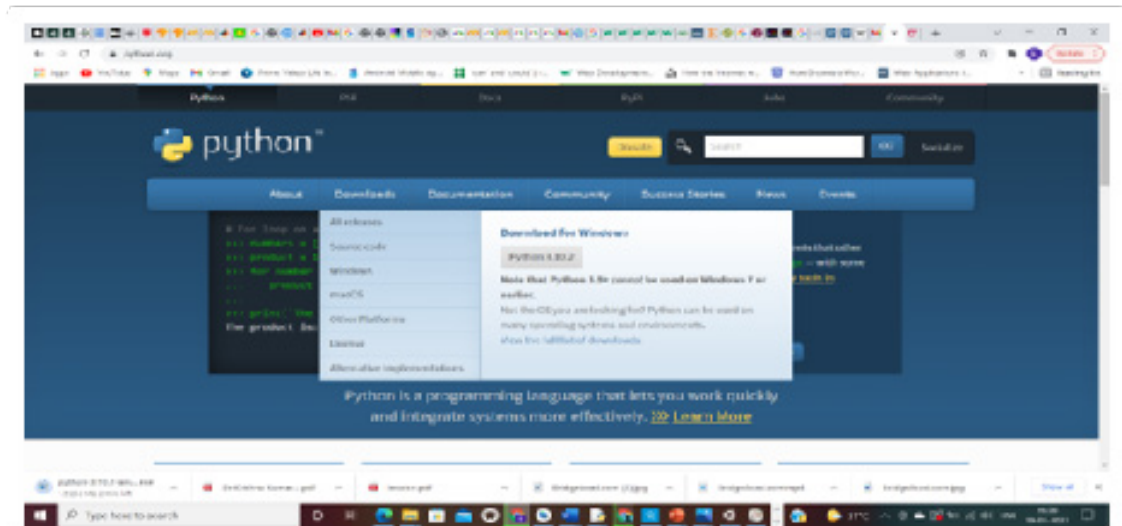


Fig 1.1.10 Python IDE

**Note:** Search in Google or any other search engine to find the installation steps. Read the instructions carefully for installation.

## 1.1.2 Python interactive mode

In Python interactive mode we can type each instruction and get it done. Type the command `python` in the command prompt of the OS to open Python interactive mode.

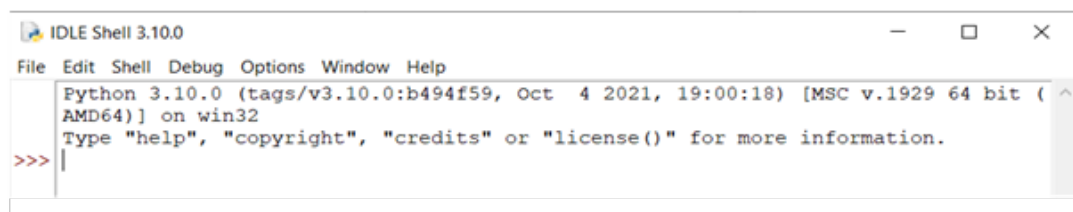


Fig 1.1.11 Python IDE

- ◆ When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is a result.
- ◆ A program usually contains a collection of statements. If there is more than one statement, the results appear one at a time as the statements execute.

**Activity:** Open the Python IDLE shell and type the following code and observe the result.

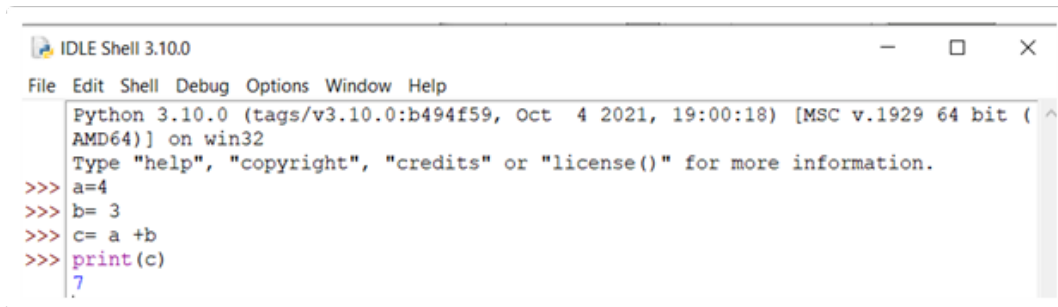


Fig 1.1.12 Python IDE

### 1.1.3 Python IDE

An IDE (Integrated Development Environment) identifies source code better than a text editor. There are several IDEs available to write Python programs.

Python 3 standard installation contains a very simple and useful application named IDLE (Integrated Development and Learning Environment). After installing Python, find IDLE somewhere under Python 3. x, and open it from the start menu of the Operating system. This is what you should see:

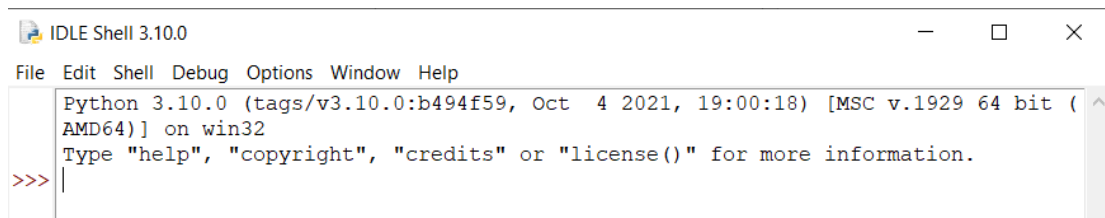


Fig 1.1.13 Python IDE

Create a new source file by clicking the File in the IDLE's menu and choosing New File.

The following is a Python program to display the Hello world message written in Python IDLE.

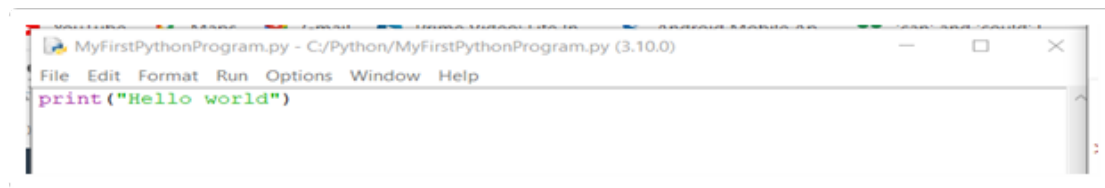


Fig 1.1.14 Python IDE

Save the file and click on Run in the IDLE's menu. Don't set any extension for the file name you are going to save. Python automatically saves the file with the .py extension. After running the program, you will see the output as shown below.

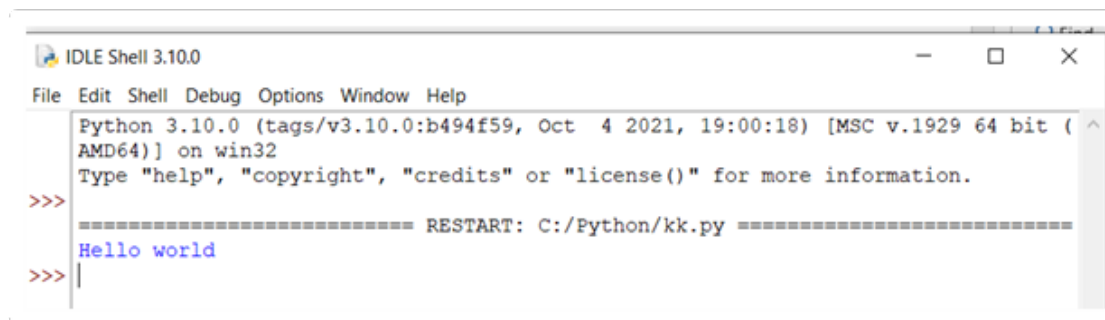


Fig 1.1.15 Python IDE

- ◆ Write `prin("hello world")`, save and run. (notice that letter t in print is missing). You will get an error message as shown below.

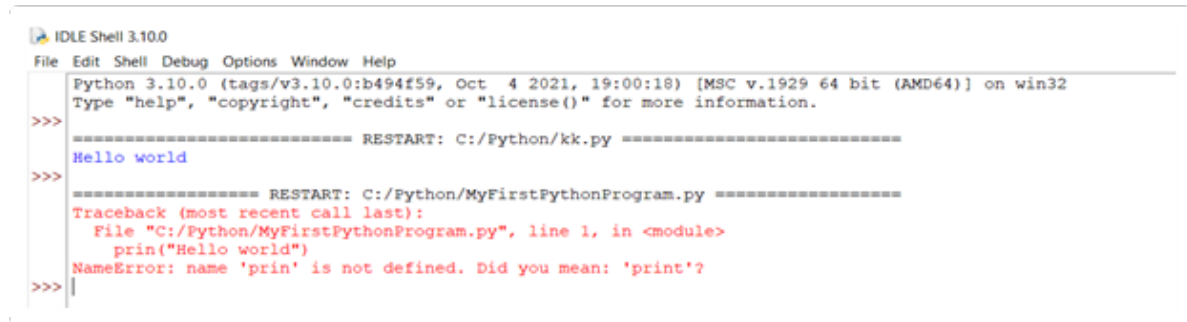


Fig 1.1.16 Python IDE

NameError is: name print is not defined. Python did not understand the word prin.

Write print("hello world" save and run. (Notice that the bracket is missing). You will get an error message as shown below.



Fig 1.1.17 Python IDE

Similarly, the IDE will generate error messages according to the mistakes in the program.

**The following are the most popular IDE used to develop Python applications.**

### **PyCharm:**

PyCharm is an IDE for professional developers created by JetBrains

### **2. Spyder**

Spyder is an open-source IDE written in Python, for Python.

To Do

Search in Google or any other search engine to find popular Python IDEs and compare their features.

### **Python statements**

Example 1:

The following is a Python program to input two integer numbers, add them, and display the result.

```
a = int(input("Enter first number"))  
b = int(input("Enter second number"))  
c = a + b  
  
print(f"The sum of {a} and {b} is {c}")
```

There are 4 statements in the above program. The statements are the instructions and we write to tell Python what our programs or applications should do. To add two numbers, first read the numbers and save them to memory locations or assign the two numbers to variable names. The above program will let the user enter the numbers. While executing the program, the user input two numbers. For example, 4 and 3. The output will be : The sum of 4 and 3 is 7.

The last line uses an f-string (formatted string literal) in Python to display a message that includes variable values directly within the string. The user can execute the program again by giving the next set of numbers. A statement is a unit of code that the Python interpreter can execute.

### Type and execute the above program using

- ◆ Jupyter Notebook Online
- ◆ Python IDLE shell (Interactive mode)
- ◆ IDE

### 1.1.4 Python variable name

In the programming context, variables are names that represent memory locations. The memory of a computer can store a lot of data. Each data is stored in different memory locations. From a programming perspective, these locations are identified with a variable name. In the above program (Example 1) a, b, and c are variable names.

Note: While using variable names it's better to use meaningful names related to data. For example, if you want to store and process the age of a person, use an appropriate variable name, say age to store the value. It helps in easy understanding of the program. The concept is illustrated below.

Age = 12

12



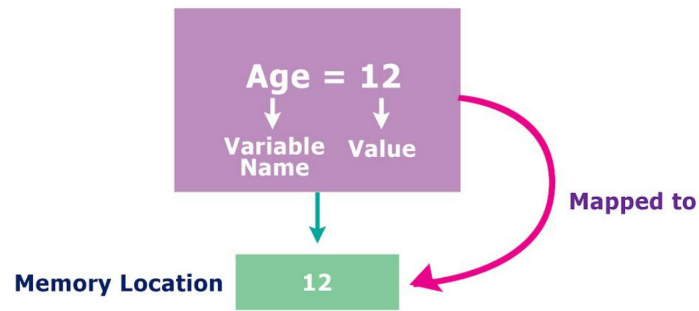


Fig 1.1.18 Memory Allocation

Here Age is the name of a memory location and the data or value stored is 12. The `print(Age)` will fetch the data from the memory location named Age and display the value or data 12. Remember **Python is case sensitive, which means Age and age are two different variable names.**

#### 1.1.4.1 Naming rules for Variables

1. Python is case sensitive, which means Age and age are two different variable names.
2. The first character of the variable name must be an alphabet or an underscore (`_`).
3. The rest of the variable name can consist of letters, underscores, or digits.
4. Special symbols (`*`, `#`, `&` etc) and space are not allowed for variable names.
5. Python keywords are not allowed as variable names.

**The following are examples of Python keywords:**

<code>assert</code>	<code>else</code>	<code>if</code>	<code>not</code>	<code>while</code>
<code>False</code>	<code>in</code>	<code>pass</code>	<code>yield</code>	<code>and</code>
<code>as</code>	<code>elif</code>	<code>global</code>	<code>nonlocal</code>	<code>try</code>
<code>break</code>	<code>except</code>	<code>import</code>	<code>or</code>	<code>with</code>
<code>def</code>	<code>for</code>	<code>lambda</code>	<code>return</code>	<code>await</code>
<code>continue</code>	<code>finally</code>	<code>is</code>	<code>raise</code>	<code>async</code>

In Python `#` represents comments. Comments will not be executed by Python. This is used by the programmer as a reference or as a note.

**Examples of correct variable names:**

`Student_ID`, `student_id`, `Location`, `NumOfChildren`

### Examples of incorrect variable names:

23Age – started with numbers.

student@age - @ symbol not allowed in a variable name.

### Input function

Python input function is **input( )**. There will be situations where the program or application has to interact with the user. The input( ) will read the input data from an input device and convert the data into a string, then return the data to the variable name.

A= input(“ Enter a number”) will prompt the user to input a number, read the number typed by the user, convert it into a string, return it to the variable A and store it in the memory location. Note that the data will be stored as a string( not a number) even though the user types a number. To convert the string to an integer, we can use the int( ) function.

```
A = int(input(“Enter a number”))
```

### Example 2:

```
Username = input(“Enter the username”)
```

```
Password = input(“Enter Password”)
```

Username and Password strings in this example.

Number\_Of\_Children =int( input(“ Enter a number”)).Here we used typecast int() to convert the input data to an integer because the number of children holds an integer value.

### print( ) function



Fig 1.1.19 Example to print a message

### Syntax : print(objects, sep=separator, end=end, file=file)

The print function is used to display the result or specified message to the screen, or another output device.

print (“Hello World”) will display the message **Hello World**

**Note:** The above program is written in Jupyter Notebook online. You may use any other IDE of your choice.

### Escape Sequences used in print()

Sometimes we need to display the output in different lines or with space in between or in a particular format.

`\n` – newline character. This will create a new line.

`\t` – the tab space character

```
print("Hello \n World")  
  
Hello  
World
```

Fig 1.1.20 Example to print a message

In the above example, if you remove `\n`, **Hello world** will be printed in one line. The following program is an example of using tabs to create space.

```
print("Hello \t \t World")  
  
Hello           World
```

Fig 1.1.21 Example to print a message

### **Activity 1: Write the following program**

```
My_name= "John"
```

```
print("My name is \n",My_name)
```

`\n` represents a new line. The text “**My Name is**” will be displayed in the first line and the name **John** will be on the next line. Run the program after removing `\n` from the code and check the output.

### **Activity 2:**

#### **1. Write the following program and check the output.**

```
print("  @")
```

```
print("  @ @")
```

```
print("  @  @")
```

Note: Run the above code in Jupyter Notebook or any other IDE and do the following activities.

- ◆ Reduce the number of `print()` functions by using the `\n` escape in the above program.

- ◆ Remove any of the quotes, and identify the response; pay attention to where Python shows an error
- ◆ Remove some of the parentheses and identify the error.;
- ◆ Use Print instead of print - what happens now?
- ◆ Change the spelling of print to print and identify the error.

Play with the code while learning, modify any part of the code, learn from your mistakes, and draw your conclusions.

2. Write the following program and check the output.

```
Mark1 = float( input("Enter a mark"))
Mark2 = float(input("Enter a Mark"))
Total = Mark1 + Mark2
print(Total)
```

## Recap

- ◆ The fundamentals of computer programming, i.e., how to create source code, how the program is executed, the definition of a programming language.
- ◆ Translating the source code using compiler and interpreter.
- ◆ The basics about Python and its features.
- ◆ Resources and different types of IDE to write Python programs.
- ◆ Python statements
- ◆ Naming rules of a variable
- ◆ Input and output function to read data and display results.

## Objective Type Questions

1. How many variables are in the following program?  

```
gender = "Male"  
print("Your Gender is ",gender)
```
2. State the purpose of a compiler.
3. What is the language used to create a source code?
4. Identify any 3 languages used to make source code.
5. Identify the Python version used in this course.
6. State the purpose of an IDE.
7. What is the significance of a translator in programming?
8. State the purpose of a variable name.

## Answers to Objective Type Questions

1. 1
2. To translate source code to object code
3. Any high-level language
4. Python, C++, Java(or any other programming languages)
5. 3.10.2
6. IDE can be used to type, edit, save, run, debug, etc.
7. Translates written in source code into machine code.
8. To identify the memory location

## Assignments

1. Write the following program and check the output.  

```
My_name = "John"  
print("My Name is ", My_name)
```
2. Write the following program and check the Python reaction.

```
My_name = "John"
```

```
print("My Name is ", My_name)
```

Here I have used My\_name to store the name. But in the print function, I missed the last letter e of the My\_name variable. Python will consider these two different variable names.

3. Change the above program by using the correct variable name in the print function

```
Print("My Name is ", My_name)
```

4. Run the above program by removing the brackets and identify the error
5. Run the above program by removing the quotes and identify the error.
6. Write a program to display your family members' names.
7. Write a program to input your family members names and display the names.
8. Write and run the following code.

```
My_name= "kkg"
```

```
print("My name is \n",My_name)
```

Replace \n with \t and see the result.

9. Write and run the following code

```
My_name= "KKG"
```

```
print("My name is \t\t",My_name)
```

```
print("My name is Python ")
```

```
print("My name is \t\t",My_name, end=" ")
```

```
print("My name is Python Guru")
```

**Use the end parameter with more space in between the quotes and observe the result.**

```
print("My name is \t\t",My_name,end=" ")
```

**Use end parameter with any characters or symbols in between the quotes and observe the result**

```
print("My name is \t\t",My_name,end="@@@@@@@@@@@@@@@@@@ ")
```

10. Write and Run the following code.

```
My_name= "KKG"
print("My name is \t \t", My_name , end =" \n")
print("Programming is fun ")
My_name= "Peter"
print("My name is \t \t",My_name , "," Hello", sep ="      ****")
print("My name is Python Guru")
```

**use sep parameter with more space in between the quotes and observe the result**

**use sep parameter with any characters or symbols in between the quotes and observe the result**

11. Write a program to input the age of your parents and display the age using the **sep** parameter.

12. Guess the output of the following code and run the code to check the output.

```
My_name= "John"
print("My name is \t \t",My_name ,"Hello" ,sep="|", end = "\n")
print("My name is Python")
```

## References

1. Learning Python by Mark Lutz, O'Reilly Media, 2013.
2. Python Crash Course by Eric Matthes, No Starch Press, 2019.
3. Fluent Python by Luciano Ramalho, O'Reilly Media, 2022.

## Suggested Reading

1. A Beginner's Guide To Learn Python In 7 Day, Author: Ramsey Hamilton
2. Python Programming For Beginners: Learn The Basics Of Python Programming (Python Crash Course, Programming for Dummies). Author: James Tudor
3. <https://www.python.org/about/gettingstarted/>



make decisions. Once you understand how these work, you'll be able to write programs that are interactive, smart, and capable of handling real-world logic.

Imagine you're creating a simple fitness app that calculates calories burned based on steps walked, checks if the user met their goal, and gives a motivational message. You would use **arithmetic operators** to calculate total calories, **comparison operators** to check if the goal is reached, and **logical operators** to decide what message to display. With the power of Python operators, you can build programs that do more than just run they can think and respond like a real assistant.

## Keywords

Logical Operators, Bitwise Operators, Comparison Operators, Membership Operators, Operator Precedence

## Discussion

### 1.2.1 Python Operators

When we withdraw money from an ATM, the amount withdrawn will be deducted from the account. The program will subtract the amount. Subtracting amount is an operation and the operator used is minus (-). **Operators** are special symbols in a programming language that carries out arithmetic, logic, and other operations. The value or data that the operator operates on is called the **operand**. The following are the types of operators (Fig 1.2.1) in Python.

1. Arithmetic Operators
2. Assignment Operators
3. Comparison (Relational) Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators



Fig 1.2.1 Tyes of Python operators

### 1.2.1.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc. Table 1.2.1 contain the basic arithmetic operators used in Python (Assume the values of **a** and **b** are 5 and 3).

Table 1.2.1 Arithmetic Operators

Operator	Name	Description	Example
+	Addition	Adds two numbers	$a + b = 8$
-	Subtraction	Subtracts one number from another	$a - b = 2$
*	Multiplication	Multiplies two numbers	$a * b = 15$
/	Division	Divides one number by another (result with decimal)	$a / b = 1.6$
%	Modulus	Gives the remainder of a division	$a \% b = 2$
**	Exponent	Raises one number to the power of another.	$a ** b = 125$
//	Floor Division	Divides and removes the decimal (gives whole number)	$a // b = 1$

### 1.2.1.2 Assignment Operators

This operator is used to assign values to variable. Table 1.2.2 specify the assignment operators is shown below.

Table 1.2.2 Assignment Operators

Operator	Description	Example
=	Assigns a value to a variable	$x = 5$
+=	Adds and assigns ( $x = x + \text{value}$ )	$x += 2$
-=	Subtracts and assigns ( $x = x - \text{value}$ )	$x -= 2$
*=	Multiplies and assigns ( $x = x * \text{value}$ )	$x *= 5$
/=	Divides and assigns ( $x = x / \text{value}$ )	$x /= 3$
//=	Floor divides and assigns ( $x = x // \text{value}$ )	$x //= 2$
%=	Modulus and assigns ( $x = x \% \text{value}$ )	$x \% = 5$
**=	Power and assigns ( $x = x ** \text{value}$ )	$x ** = 1$

### 1.2.1.3 Comparison Operators

While writing programs or applications we will use comparison operators. For example,

the ATM application will check the entered amount is less than or equal to ( $\leq$ ) the available amount. If the result is true, you will get money, otherwise, the machine will inform you that you do not have a sufficient amount in the account. All comparison operators will compare the data and return True or False.

Table 1.2.3 Comparison Operators

Operator	Description	Example
$==$	Equal to	$5 == 5$ (True)
$!=$	Not equal to	$5 != 2$ (True)
$>$	Greater than	$5 > 2$ (True)
$<$	Less than	$5 < 2$ (False)
$>=$	Greater than or equal to	$5 >= 5$ (True)
$\leq$	Less than or equal to	$6 \leq 5$ (False)

### 1.2.1.4 Logical Operators

Two or more relations that compare the data can be logically joined together using the logical operators OR and AND. For example, an application will check if the username is correct, **and** the password is correct when you log in.

#### 1. AND Operator

The **AND operator** is used to check if **multiple conditions are all true**. It returns True only if **both (or all) conditions are true**; otherwise, it returns False as shown in Table 1.2.4.

Table 1.2.4 AND Operator

X	Y	X AND Y
True	True	True
True	False	False
False	True	False
False	False	False

#### 2. OR Operator

The result of an **OR operator** is True if **at least one** of the conditions being evaluated is True. If all the conditions are False, then the result is False as shown in Table 1.2.5.

Table 1.2.5 OR Operator

X	Y	X OR Y
True	True	True
True	False	True
False	True	True
False	False	False

**Note:** X and Y should be Boolean, otherwise it will return the integer

### 3. NOT (boolean NOT) Operator

The negation of a Boolean is the opposite of its current Boolean value. Description of NOT operator is shown below table 1.2.6.

Table 1.2.6 NOT Operator

X	NOT X
True	False
False	True

#### Example:

a = True

b = False

print (a and b)

print (a or b)

print (not a)

#### Output:

False

True

False

### 1.2.1.5 Bitwise Operators

**Bitwise operators** are used to perform operations on the **binary representations** of integers. These operators work at the bit level, meaning they perform operations on individual bits (0s and 1s) of the numbers. Table 1.2.7 shows the bitwise operators used in Python.

Table 1.2.7 Bitwise Operators

Operator	Name	Description	Example	Result
&	Bitwise AND	Returns 1 if <b>both bits</b> are 1	5 & 3 → 0101 & 0011 = 0001	1
	Bitwise OR	Returns 1 if <b>at least one bit</b> is 1	5   3 → 0101   0011 = 0111	7
^	Bitwise XOR	Returns 1 if <b>bits are different</b>	5 ^ 3 → 0101 ^ 0011 = 0110	6

~	Bitwise No	Inverts all the bits (1 becomes 0, 0 becomes 1)	$\sim 5 \rightarrow \sim 0101 = 1010$ (in 2's complement)	-6
<<	Left Shift	Shifts bits to the <b>left</b> , filling with 0s	$5 << 1 \rightarrow 0101 << 1 = 1010$	10
>>	Right Shift	Shifts bits to the <b>right</b> , dropping bits	$5 >> 1 \rightarrow 0101 >> 1 = 0010$	2

### 1.2.1.6 Membership Operators

**Membership operators** are used to **check if a value exists** inside a sequence like a **string, list, tuple, or set**. Membership Operators are summarized in Table 1.2.8.

Table 1.2.8 Membership Operators

Operator	Description	Example
in	True if a value is inside a list, string, or set	'a' in 'apple' → True
not in	True if a value is not inside a list, string, or set.	'x' not in 'apple' → True

### 1.2.1.7 Identity Operators

This operator is used to **compare the memory locations** of two objects. They check whether two variables refer to the **same object** in memory, not just if their values are the same. Table 1.2.9 shows the Identity Operators used.

Table 1.2.9 Identity Operators

Operator	Description	Example
is	True if two variables point to the <b>same object</b> (same memory address)	x is y, the result is True if id(x) is the same as id(y)
is not	True if two variables point to <b>different objects</b>	x is not y; the result is True if id(x) is different from id(y)

## 1.2.2 Order of Operations

In an expression with more than one operator, the order of execution of operators depends on the rules of precedence as shown in Table 1.2.10.

Table 1.2.10 Order of Precedence

Priority	Operator	Description	Associativity
1	()	Parentheses	Left to Right
2	**	Exponentiation	Right to Left
3	~x	Bitwise NOT	Right to Left
4	*, /, //, %	Multiplication, Division, Floor Division, Modulus	Left to Right

5	+, -	Addition, Subtraction	Left to Right
6	<<, >>	Bitwise Shift Left and Right	Left to Right
7	&	Bitwise AND	Left to Right
8	^	Bitwise XOR	Left to Right
9		Bitwise OR	Left to Right
10	in, not in	Membership	Left to Right
11	not	Logical NOT	Right to Left
12	and	Logical AND	Left to Right
13	or	Logical OR	Left to Right

### Example 1:

x = 2

y = 4

z = x + y / 2

print(z)      *// will display 4.0*

In this expression + and / are the operators. y/2 will be executed first, and the result will be added to x.

### Example 2:

x = 2

y = 4

z = (x+y)/2

print(z)      *// will display 3.0*

In this expression + and / are the operators. Expressions in parentheses are evaluated first. x + y will be executed first, and the result will be divided by 2.

## Recap

- ◆ **Python Operators:** Symbols that carry out operations on operands. They help us perform different types of operations like arithmetic, comparisons, logical checks, and more.
- ◆ **Types of Operators**
  - ◆ **Arithmetic Operators:** Used for basic mathematical calculations like addition, subtraction, multiplication, etc.
  - ◆ **Assignment Operators:** Used to assign values to variables, such as = or compound assignments like += and -= that perform an operation and assign the result in one step.

- ◆ **Comparison Operators:** Used to compare two values and return True or False (e.g., ==, =, >, <).
- ◆ **Logical Operators:** Used to combine multiple conditions with and, or, and not.
- ◆ **Bitwise Operators:** Perform operations on the binary representation of integers, such as &, |, ^, and shifts (<<, >>).
- ◆ **Membership Operators:** Check if a value is a member of a sequence (like a string or list) using in and not in.
- ◆ **Identity Operators:** Compare memory addresses to check if two variables point to the same object, using is and is not.
- ◆ **Order of Operations:** For evaluating expressions with multiple operators, operators with higher precedence (like parentheses () and exponentiation \*\*) are evaluated first, followed by other operators in a specific order (e.g., multiplication \* before addition +).

## Objective Type Questions

1. What operator will be used to check if 15 is not equal to 10.
2. What operator will be used to check if 15 is equal to 10.
3. What will be the output of the following code?

```
x = ((6+5**2) -(30 *2/4))/2
print(x)
```

4. What will be the output of the following code?

```
x = ((6+5**2) -(30 *2/4))/2
print (x >6)
```

5. Identify the output of the following code.

```
mark = 30
age = 50
result = mark > 50 and age < 10
print(result)
```

6. The negation of a Boolean is the opposite of its \_\_\_\_\_
7. What does the **in operator** check in Python?
8. What type of operator is \* in Python?

9. What is the result of `5 // 2`?
10. Which operator checks if two variables refer to the same object in memory?

## Answers to Objective Type Questions

1. `!=`
2. `==`
3. 8.0 (Use the operator rule of precedence)
4. True
5. False
6. Current Boolean value
7. Membership
8. Arithmetic
9. 2
10. `is`

## Assignments

1. Write a Python program to calculate the area of a rectangle. The length of the rectangle is 7 units, and the width is 5 units. Use arithmetic operators to compute the area.
2. Write a Python program that assigns a value of 10 to a variable **a** and then performs the following operations using assignment operators:
  - a. Add 5 to a
  - b. Subtract 3 from a
  - c. Multiply a by 2
  - d. Divide a by 4.After each operation, print the value of a.

3. Write a Python program that compares two integers  $x = 12$  and  $y = 8$  using all the comparison operators ( $==, !=, >, <, >=, <=$ ). Print the result of each comparison.
4. Write a Python program that checks if a given number  $n$  is between 10 and 20 (inclusive). Use logical operators (and, or, not) to check if  $n$  satisfies the condition  $10 \leq n \leq 20$ .
5. Write a Python program that checks if a character 'a' is present in the string 'apple' and if the number 7 is in the list  $[1, 3, 5, 7, 9]$ . Use the membership operators (in, not in) and print the results.

## References

1. Chase, S. (2018). *Python for Data Analysis* (2nd ed.). O'Reilly Media.
2. Van Rossum, G., & Drake, F. L. (2001). *Python 2.0 Reference Manual*. Python Software Foundation.
3. Sweiger, A. (2019). *Automate the Boring Stuff with Python* (2nd ed.). No Starch Press.
4. Müller, A. C., & Guido, S. (2016). *Introduction to Machine Learning with Python*. O'Reilly Media.

## Suggested Reading

1. Python Crash Course, Eric Matthes (No Starch Press, 2016)
2. Zelle, J. M. (2016). *Python programming: An introduction to computer science* (3rd ed.). Franklin, Beedle & Associates.
3. Downey, A. B. (2015). *Think Python: How to think like a computer scientist* (2nd ed.). O'Reilly Media.
4. Beazley, D. M. (2009). *Python essential reference* (4th ed.). Addison-Wesley Professional
5. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.



# Data Types in Python

## Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ define different data types in Python.
- ◆ identify the index positions of elements in a Python list.
- ◆ list the steps to perform slicing on a Python list.
- ◆ explain the characteristics of Python tuples.
- ◆ list the examples of immutable and mutable data types in Python.

## Prerequisites

Imagine you are developing a simple app to manage student information at a school. For each student, you need to store different types of data like their name (text), age (number), date of birth (date), and subjects they are studying (a list of items). You may also need to check whether the student is eligible for a scholarship (True or False), store their hobbies without any duplicates (a set), or organize their details using labels like "name", "age", and "grade" (a dictionary). This real-life situation introduces the need for data types in Python, where each kind of information is stored using a suitable type such as int, float etc. Understanding data types helps learners manage and organize data efficiently in any Python program.

## Keywords

integer, float, Boolean, sequence, list, dictionary, tuple, set

## Discussion

### 1.3.1 Data types in Python

Data is processed by applications or programs in different ways. For example, in a student registration process, different types of data are collected, such as name, date of birth, address, and family monthly income, etc. Each of these data types serves a specific purpose. The name consists of alphabetic characters, the date of birth is stored as a date type, and the family's monthly income is represented as numeric data.

A data type is a classification that specifies the type of data a variable can hold in a program. It determines the memory allocation required for storing the data. For example, the memory needed to store 'KKG', 5, and 5.10 will vary because they belong to different data types. In Python, various data types as given in Fig 1.3.1.

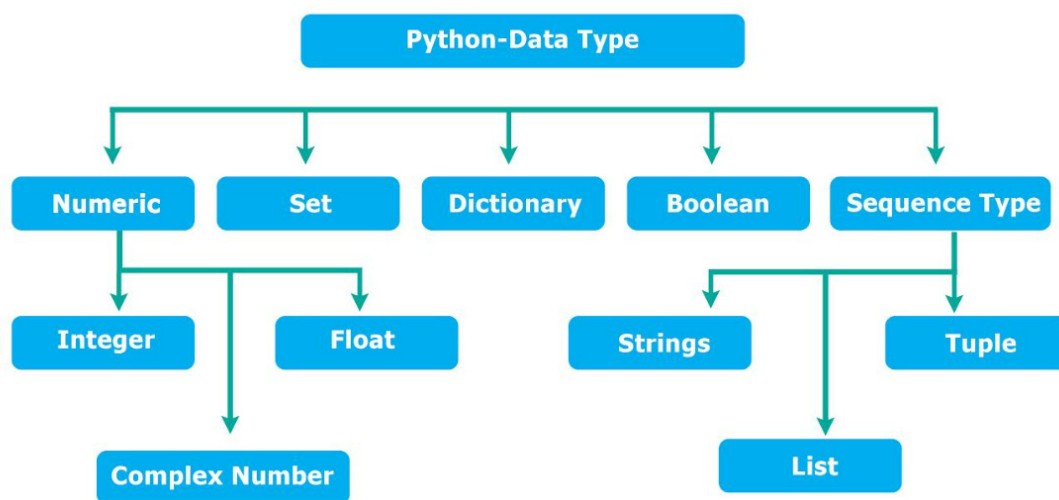


Fig 1.3.1 Classification of python data types

### 1.3.2 Numeric Data Types

In Python, the numeric data type is used to represent values that are numbers. These numbers can be integers, floating-point numbers, or complex numbers. Python provides three built-in classes for this: `int` for integers, `float` for decimal numbers, and `complex` for complex numbers.

#### 1. Integers

Represented by the 'int' class in Python, these are whole numbers that can be either positive or negative, without any decimal or fractional part. Python allows integers of any length, limited only by the available memory.

#### 2. Float

The float data type represents decimal (floating-point) numbers. It is used to store numbers that have a fractional component. Examples include 4.5 and 890.67.

#### 3. Complex numbers

In Python, complex numbers are represented in the form  $a + bj$ , where:

- ◆ a is the real part
- ◆ b is the imaginary part

Complex numbers are widely used in geometry, scientific calculations, signal processing, and calculus.

Example: 3+4j

```
a = 5
print(type(a))
b = 5.0
print(type(b))
c = 2 + 4j
print(type(c))
```

Output

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

### 1.3.3 Boolean Data Types

The Boolean data type in Python represents two values: True and False, which are commonly used in logical operations and conditional statements. It is a subclass of integers, where True is equivalent to 1 and False is equivalent to 0.

```
a = 5
b = 10
print(a > b)
print(a < b)
```

Output

```
False
True
```

### 1.3.4 Sequence Data Types

In Python, sequence data types allow storing multiple values in an ordered manner. They support indexing, slicing, and iteration. The primary sequence data types in Python are:

- ◆ String
- ◆ List
- ◆ Tuple

#### 1.3.4.1 String

A string in Python is a sequence of characters, which can include letters, numbers, symbols, and spaces. Strings are mainly used for handling text data, such as names, addresses, or messages. Strings in Python are enclosed in single (' '), double (" "), or triple (''' ' or '''' '') quotes.

Example:

```
text1 = "Hello World"
```

```
text2 = "Covid-19"
```

Even if a string contains only numbers, it is still considered a string if it is enclosed in quotes. For example “222345” is not a number, it is a string.

#### Different Types of String Representation

Python allows strings to be defined using:

##### 1. Single Quotes (')

Strings can be enclosed in single quotes.

Example:

```
Message = 'Hello World'
```

##### 2. Double Quotes (" ")

Strings can also be enclosed in double quotes

Example:

```
Message = "Hello World"
```

##### 3. Triple Quotes (''' ' or '''' '')

Triple quotes allow multiline strings. Three single quotes or double quotes can be used.

Example:

```
Message = ''' Programming is fun. Python is a high-level language. Python is used by
Facebook, Google, NASA and other companies '''
```

When we input the data from the keyboard, the number will be considered as string only. See the following example and output.

Activity 1: Run the following program and check the output

```
mark = input("Enter a mark")
```

```
print(mark)
```

```
print(type(mark))
```

Input:

Enter a mark: 67.9

Output:

67.9

<class 'str'>

### 1.3.4.2 Python List

We have already discussed variable names. Typically, only one piece of data can be represented by a single variable. For example:

```
Student_name = "KKG"
```

However, there are many situations where we need to read, store, process, and output multiple pieces of data, maybe dozens or even thousands. Imagine having to create different variable names for each value like

```
X = 20
```

```
X1 = 100
```

```
X2 = 30
```

```
...
```

```
Xn = 19
```

Using a single variable to store all these values would be easier and more convenient, and this is where lists are useful.

For example:

```
X = [20, 100, 30, ..., 19]
```

A list in Python is a sequence data type that stores an ordered collection of items. It allows you to store multiple values in a single variable. Lists are mutable, meaning you can add, remove, or change elements. They are defined using square brackets [ ] and the values are separated by commas.

For Example:

```
x = ["KKG", "pen", "beach"]
```

Here, x is a list that stores multiple string items. On the other hand, writing x = "KKG" would mean that x holds only a single string value.

A list is similar to an array in many other programming languages. Though a list can contain items of different types, it usually holds items of the same type. In a list with n items, the first item has index 0, and the last has index n-1 as in Fig 1.3.2

Index from front				
0	1	2	3	4
2	4	1	10	5
-5	-4	-3	-2	-1
Index from rear				

Fig. 1.3.2 List of 5 elements

### 1.3.4.3 Tuple

A tuple in Python is a sequence data type that stores an ordered collection of items, similar to a list. However, tuples are immutable, meaning their elements cannot be changed once they are defined. Tuples are defined using parentheses ( ).

**Example:**

```
colors = ("Red", "Green", "Blue")
print(colors)
```

Output:

```
('Red', 'Green', 'Blue')
```

### 1.3.5 Set

A set in Python is a sequence data type that stores an unordered collection of unique items. Sets are mutable, meaning you can add or remove elements, but they do not allow duplicate values. Sets are defined using curly braces { }. Set elements cannot be accessed using an index because sets are unordered and do not maintain any specific order of items. However, you can iterate through the elements using a **for** loop or check if a particular value exists in the set using the **in** keyword.

**Example:**

```
fruits = {"Apple", "Banana", "Cherry"}
print(fruits)
```

Output:

```
{'Apple', 'Banana', 'Cherry'}
```

### 1.3.6 Dictionary

A dictionary in Python is a mutable, unordered collection of key-value pairs. Unlike sequence data types, dictionaries store data in the form of keys and their corresponding values, and items are accessed using keys instead of indexes. Dictionaries are defined using curly braces {}, with each key-value pair separated by a colon (:). The dictionaries are ordered, which means that the items have a defined order, and you can refer to an item by using an index. Dictionary keys in Python are case-sensitive, meaning keys with the same name but different letter cases are considered separate and distinct entries.

The main operations on a dictionary are

#### 1.3.6.1 Creating and Printing a Dictionary

```
student = {"name": "Helen", "age": 20, "department": "Computer Science"}  
  
print(student)  
  
{'name': 'Helen', 'age': 20, 'department': 'Computer Science'}
```

#### 1.3.6.2 Accessing an Item from the Dictionary

In the following example, the dictionary is named cars and it stores car manufacturer names as keys and their respective countries as values.

```
cars = { "Maruthi": "India", "Toyota": "Japan", "KIA": "Korea" }  
  
x = cars.get("KIA")  
  
print(x)
```

**Output:**

Korea

#### 1.3.6.3 Display all Keys from a Dictionary

```
cars = { "Maruthi": "India", "Toyota": "Japan", "KIA": "China" }  
  
print(cars.keys())
```

**Output:**

['Maruthi', 'Toyota', 'KIA']

#### 1.3.6.4 Adding a new Item to a Dictionary

```
Student_Phone = {"KKG": 8608754, "John": 890744}  
  
Student_Phone["Shan"] = 989643  
  
print(Student_Phone)
```



### Output:

```
{'KKG': 8608754, 'John': 890744, 'Shan': 989643}
```

In this example:

The dictionary `Student_Phone` initially contains two entries. A new key-value pair "Shan": 989643 is added using the assignment statement. Finally, the updated dictionary is printed, showing all the stored names and phone numbers.

### 1.3.7. Differences between List, Tuple, Dictionary, and Set

Table 1.3.1 Comparison of List, Tuple, Dictionary and Set

Property	List	Tuple	Dictionary	Set
Ordered	Yes	Yes	From version 3.7	No
Indexed	Yes	Yes	Yes	No
Key and Value Pair	No	No	Yes	No
Bracket type	[ ]	( )	{ }	{ }
Changeable(add/ Remove values)	Yes	No	Yes	Yes
Allow duplicate values	Yes	Yes	No	No

## Recap

### Data Types in Python

- ◆ Data types tell Python what kind of data is being used — like text, numbers, or a group of items

### Numeric Data Types

- ◆ int: Whole numbers like 5 or 100.
- ◆ float: Decimal numbers like 3.14 or 10.5.
- ◆ complex: Numbers with a real and imaginary part, like  $3 + 4j$ .

### Boolean Data Type

- ◆ This type has only two values: True or False. It is used in comparisons like checking if one number is greater than another.

### Sequence Data Types

- ◆ String: A group of characters, like "hello". You can use single, double, or triple quotes. Input from the user is always a string.
- ◆ List: A group of values that you can change. It uses square brackets, like [1, 2, 3]. You can access items by position.
- ◆ Tuple: Similar to a list, but you cannot change the values. It uses round brackets, like (1, 2, 3).

### Dictionary

- ◆ A dictionary stores data in key-value pairs. You can quickly look up a value by its key. Example: {"name": "John", "age": 25}.

## Objective Type Questions

1. Which data type in Python is used to store whole numbers without decimals?
2. What symbol is used to define a tuple?
3. Which data type allows storing key-value pairs?
4. Which collection type is immutable: list or tuple?
5. Which data type is used to store text data?
6. What is the output type of input() function in Python?

7. Which Python numeric type includes real and imaginary parts?
8. Which symbol is used to define a list in Python?
9. Which data type in Python does not allow duplicate values and is unordered?
10. What is the Boolean value of  $5 < 3$  in Python?

## Answers to Objective Type Questions

1. int
2. ( )
3. dictionary
4. tuple
5. string
6. string
7. complex
8. [ ]
9. set
10. False

## Assignments

1. Write a Python program to create an empty list, dictionary, set, and tuple.
2. Write a Python program to store the subjects' names you are studying this semester as a list and print them.
3. Write a Python program to store your family member's phone numbers and names in a dictionary and print them.
4. Using examples, identify the differences between dictionary and list.
5. Mention any two real-time situations in which list will be more suitable than set.

6. Explain the difference between a list and a tuple in Python. Provide examples.
7. What is a dictionary in Python? Write a program to create a dictionary with at least three key-value pairs and access its elements.

## References

1. Python online documents. <https://docs.python.org/3/library/operator.html>

## Suggested Reading

1. A Beginner's Guide To Learn Python In 7 Day, Author: Ramsey Hamilton
2. Python Programming for Beginners: Learn The Basics Of Python Programming (Python Crash Course, Programming for Dummies). Author: James Tudor
3. <https://www.python.org/about/gettingstarted/>



# Built-in Methods of Data Structures

## Learning Outcomes

In this unit, you will learn:

- define built-in methods used with Python data structures like lists, tuples, sets, and dictionaries.
- describe how to use list methods such as `append()`, `insert()`, and `pop()` to manipulate data.
- explain the purpose of set methods like `add()`, `union()`, and `difference()` in managing unique collections.
- describe dictionary methods such as `update()`, `keys()`, and `values()` used to manage key-value pairs.

## Prerequisites

Before beginning this unit, learners should have a basic understanding of Python programming, including its syntax, structure, and the ability to write and run simple Python programs. They should be familiar with core data structures such as lists, tuples, sets, and dictionaries, and understand how data is stored, accessed, and manipulated using indexing and slicing especially with lists and tuples.

This foundational knowledge is essential because Python's built-in methods, used with these data structures, allow programmers to organize, access, and manipulate data more efficiently. Methods like `append()`, `pop()`, `union()`, and `update()` simplify common tasks such as adding or removing data, combining sets, or updating key-value pairs in dictionaries. Learning how to use these methods helps write cleaner, faster, and more readable code skills that are crucial for real-world applications like managing student records, processing data, or building user-focused software. Mastery of these methods forms a strong foundation for writing practical and professional Python programs.

## Keywords

List, dictionary, tuple, set, methods. append, update, pop, remove, union, difference

## Discussion

### 1.4.1 Python built-in method

Python has a set of built-in methods used for different data structures like **lists**, **sets**, **dictionaries**, and **tuples**. These methods help manage and manipulate data easily.. For example, we can use the append method to add a new student to an existing student list or remove a customer name from the customer list by using remove method.

Each data structure has its own set of built-in methods:

- ◆ **List Methods** – for ordered collections of items.
- ◆ **Set Methods** – for unordered collections of unique items.
- ◆ **Dictionary Methods** – for key-value data management.
- ◆ **Tuple Methods** – for immutable, ordered collections.

### 1.4.2 List Methods

A **list** is a mutable, ordered collection of items. List methods help you modify and access elements efficiently.

- ◆ **append()** : You can add new items at the end of the list, by using the append() method or adds an item to the **end** of the list.

#### Example

```
Student_List= ["John", "KKG", "Jane"]  
  
Student_List.append("Shan")  
  
print(Student_List)
```

**Output :** ['John', 'KKG', 'Jane', 'Shan']

- ◆ **insert(i, item)**

Insert an item at a given position(index). The first argument is the index of the element before which to insert, so Student\_List.insert(1, x) inserts **before the specified index** in the list.

#### Example

```
Student_List= ["John", "KKG", "Jane"]  
  
Student_List.insert(1,"Sam")
```



```
print(Student_List)
```

**Output :** ['John', 'Sam', 'KKG', 'Jane']

◆ **list.remove(x)**

Removes the first occurrence of the specified value.

**Example**

```
Student_List= ["John", "KKG", "Jane"]
```

```
Student_List.remove("KKG")
```

```
print(Student_List)
```

**Output :** ['John', 'Jane']

◆ **list.pop([i])**

Remove the item at the given position in the list, and return it. If no index is specified, pop() removes and returns the last item in the list.

**Example**

```
Student_List = ["John", "KKG", "Jane"]
```

```
print(Student_List.pop(1))
```

**Output :** KKG

◆ **list.clear()**

Remove all items from the list.

**Example**

```
Student_List.clear()
```

**Output:** []

◆ **list.count(x)**

Return the number of times *x* appears in the list.

```
Student_List= ["John", "KKG", "Jane", "John"]
```

**Example**

```
print(Student_List.count("John"))
```

**Output :** 2

◆ **list.reverse()**

Reverse the elements of the list in place

### Example

```
Student_List= ["John", "KKG", "Jane"]  
Student_List.reverse()  
print(Student_List)
```

**Output :** ['Jane', 'KKG', 'John']

#### ◆ list.sort()

Sort the items of the list in place.

### Example

```
Student_List= ["John", "KKG", "Jane"]  
Student_List.sort()  
print(Student_List)
```

**Output :** ['Jane', 'John', 'KKG']

### Exercise 1:

```
Student_List= ["John", "KKG", "Jane"]  
print(Student_List[0:2:1])
```

**Output :** ['John', 'KKG']

#### step 1: Add 3 more names to the list

```
Student_List.extend(["Sam", "Ravi", "Meena"])
```

#### step2 : Change the start , end and step numbers and observe the result.

```
Student_List= ["John", "KKG", "Jane"]  
print(Student_List[:])
```

What is the start, end, and step index of the above code?

**Start:** 0 (default)

**End:** len(Student\_List)

**Step:** 1 (default)

### Exercise 2:

#### Guess the output, write and run the code and observe the result

```
Student_List= ["John", "KKG", "Jane"]  
print(Student_List[-3:])
```

**Output :** ['John', 'KKG', 'Jane']



### 1.4.2.1 Reading and printing a list using loops

(Note: Looping will be discussed later)

```
Student_List= []  
for i in range(3):  
    Sname = input("Enter a name : ")  
    Student_List= Student_List+ [Sname]  
print(Student_List[:])
```

Output: Enter a name : John

Enter a name : Sam

Enter a name : Jane

['John', 'Sam', 'Jane']

### 1.4.3 Set Built-in Methods

A set is an unordered collection of unique elements. It's commonly used for operations involving membership tests, union, intersection, and difference. The following are some of the set methods available in Python.

#### 1. clear() : To delete all elements from list

```
car = {"Maruthi", "Toyota", "Mahindra"}  
car.clear()  
print(car)
```

**Output:** set()

#### 2. add() : To add an item to set

```
car= {"Maruthi", "Toyota", "Mahindra"}  
car.add("Kia")  
print(car)
```

**Output :** {'Toyota', 'Kia', 'Mahindra', 'Maruthi'}

#### 3. copy(): To make a copy of a set.

```
car= {"Maruthi", "Toyota", "Mahindra"}  
newcar = car.copy()  
print(newcar)
```

**Output :** {'Toyota', 'Mahindra', 'Maruthi'}

**4. difference(): To return a set that contains the difference between two sets.**

```
m = n.difference(x)
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.difference(y)
print(z)
```

**Output :** {'cherry', 'banana'}

**5. union() : Return the union of two sets.**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.union(y)
print(z)
```

**Output :** {'google', 'microsoft', 'apple', 'cherry', 'banana'}

**6. discard(): To remove an item from a set.**

```
fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")
print(fruits)
```

**Output :** {'apple', 'cherry'}

### 1.4.4 Dictionary built-in methods

Dictionaries store data in key-value pairs. Python provides various methods to manipulate and retrieve data from dictionaries efficiently. The following are some of the Dictionary methods.

**1. update():** To update a dictionary by adding a new item.

```
car= {"Maruthi": "2004", "Toyota": "2008 ", "Mahindra": "2007 "}
print(car)
car.update({"KIA": "2010"})
print(car)
```

**Output :** {'Maruthi': '2004', 'Toyota': '2008 ', 'Mahindra': '2007 '}  
{'Maruthi': '2004', 'Toyota': '2008 ', 'Mahindra': '2007 ', 'KIA': '2010'}

**2. clear():** To clear the items and return an empty dictionary.



```
car= {"Maruthi": "2004", "Toyota": "2008 ", "Mahindra": "2007 "}
car.clear()
print(car)
```

**Output :** {}

**3. copy():** To return the copy of an existing dictionary.

```
car= {"Maruthi": "2004", "Toyota": "2008 ", "Mahindra": "2007 "}
x= car.copy()
print(x)
```

**Output :** {'Maruthi': '2004', 'Toyota': '2008 ', 'Mahindra': '2007 '}

**4. keys() :** To return all the keys used in a dictionary.

```
car= {"Maruthi": "2004", "Toyota": "2008 ", "Mahindra": "2007 "}
x= car.keys()
print(x)
```

**Output :** dict\_keys(['Maruthi', 'Toyota', 'Mahindra'])

**5. Values():** To return the values of a dictionary

```
car= {"Maruthi": "2004", "Toyota": "2008 ", "Mahindra": "2007 "}
x= car.values()
print(x)
```

**Output :** dict\_values(['2004', '2008 ', '2007 '])

### 1.4.5 Tuples built-in Methods

A tuple is an ordered, immutable collection. Python offers limited methods for tuples because their content cannot be changed after creation. The following are some of the Dictionary methods.

**1. Count():** To find the number of times a specified word is repeated.

```
car= ("Maruthi", "Toyota", "Maruthi", "Mahindra", "Maruthi")
x= car.count("Maruthi")
print(x)
```

**Output :** 3

**2. Index():** To find the position or index of an item in a tuple.

In the following example, the tuple has 4 items. The value Mahindra is repeated two

times. The index method will display the first occurrence. The index of Maruti is zero. The index of the first occurrence of Mahindra is 1.

```
car= ("Maruthi","Toyota","Mahindra","Mahindra")
```

```
x= car.index("Mahindra")
```

```
print(x)
```

**Output : 2**

Table 1.4.1 Additional built in method

Function	Description	Example
len()	Returns the number of elements in the tuple	len((1, 2, 3)) → 3
max()	Returns the maximum value in the tuple	max((5, 1, 3)) → 5
min()	Returns the minimum value in the tuple	min((5, 1, 3)) → 1
sum()	Returns the sum of all numeric values	sum((1, 2, 3)) → 6
sorted()	Returns a sorted list from the tuple	sorted((3, 1, 2)) → [1, 2, 3]
tuple()	Converts an iterable into a tuple	tuple([1, 2, 3]) → (1, 2, 3)

## Recap

- ◆ Built-in methods in Python Lists
- ◆ Built-in methods in Python Tuple
- ◆ Built-in methods in Python Dictionary
- ◆ Built-in methods in Python Sets.
- ◆ These built-in methods will help us to do various operations using the data structures.

## Objective Type Questions

1. What is the list method used to remove all items from the list?
2. What is the list method used to remove the last item from the list?
3. What is the tuple method to count the repeated values?
4. What will be the output of the following code?  

```
mark_list= [ 30,45,36,33,45,50,48]  
print(mark_list[1:2:2])
```
5. What will be the output of the following code?  

```
mark_list= [ 30,45,36,33,45,50,48]  
print(mark_list[1:7:2])
```
6. Count() function is used to
7. The function is used to return the values of a dictionary
8. Function used to update a dictionary by adding a new item
9. Which of the following data structures is immutable?
10. Which method is used to add an element to a set?

## Answers to Objective Type Questions

1. Clear()
2. Pop()
3. Count()
4. [45]
5. [45, 33, 50]
6. Find the number of times a specified word is repeated
7. values()
8. update()
9. Tuple
10. add()

## Assignments

1. Read the marks of 10 students and add them to a list. Find the total and average marks. Display the mark list, total and average.

2. Write two programs that use list methods.

For example

```
mark = [20,14,11]
```

```
print("marks before append :", mark)
```

```
mark.append(80)
```

```
print("marks after append :", mark)
```

output : Marks before append : [20,14,11]

Marks after append : [20,14,11,80]

3. Write the output of the following.

List before execution	List Methods	Output
Mark = [ 20, 14, 11]	<b>Mark.append(80)</b>	
age = [ 2, 4, 1, 5 ]	<b>age.extend( [3,9] )</b>	
ClassAList = [ "John", "KKG"] ClassBList = ["Shan", "Jain"]	<b>ClassAList.extend(ClassBList)</b>	
Mark= [ 2, 4, 1, 5]	<b>Mark.insert(2,50)</b>	
Age= [ 22, 4, 11, 5]	<b>Age.remove(2)</b>	
Age= [ 22, 4, 11, 5]	<b>Age.remove(5)</b>	
ClassBList = ["Shan","Jain"]	<b>ClassBlist.pop()</b>	
Age = [ 2, 4, 1, 5 ]	<b>Age.sort()</b>	

## References

1. A Beginner's Guide To Learn Python In 7 Day, Author: Ramsey Hamilton
2. Python Programming For Beginners: Learn The Basics Of Python Programming (Python Crash Course, Programming for Dummies). Author: James Tudor
3. Python Programming For Beginners: Learn The Basics Of Python Programming (Python Crash Course, Programming for Dummies). Author: James Tудо

## Suggested Reading

1. <https://docs.python.org/3/tutorial/datastructures.html>
2. <https://docs.python.org/3/tutorial/>
3. <https://www.w3schools.com/python/>
4. <https://www.programiz.com/python-programming>



## **BLOCK 2**

**Decision making, Loops,  
Comprehensions, Functions,  
Modules & Packages**



# Decision Making and Loops

## Learning Outcomes

After completing this unit, the learner will be able to:

- ♦ define decision-making statements in Python
- ♦ list the different types of loops used in Python
- ♦ identify the syntax of if, if-else and elif statements
- ♦ recall the keywords used in Python loop control statements
- ♦ familiarise the types of nested structures used in conditional and looping statements

## Prerequisites

Imagine you are playing a video game. You walk through a hallway and come across two doors. One is locked, and one is open. What do you do? You check which one is open and go through it. Later, you find a treasure chest. You open it once, but what if there are five treasure chests? You repeat the action for each one, one by one. You have just made decisions and repeated actions just like Python can do in your programs. So far, you have learned how to give instructions to a computer using Python: storing information in variables, doing math, and printing results. But now, it is time to make your programs smarter. What if your code could choose what to do based on different situations? Or repeat tasks many times without you rewriting them. This is where decision-making and loops come in. Like the game character who checks doors or opens treasure chests, your program can now decide what to do and repeat actions as needed.

## Key Words

if, if else, nested if, elif, break, continue, pass, for, while

## Discussion

### 2.1.1 Decision Making

**Decision-making statements** in Python are used to control the flow of a program based on certain conditions. These statements allow a program to make choices and execute different blocks of code depending on whether a condition is **true** or **false**.

In python, there are four types of decision-making statements.

- ◆ if statement
- ◆ if else statement
- ◆ nested if statement
- ◆ elif ladder statement

#### 2.1.1.1 if statement

The if statement is the simplest decision-making statement. It decides whether certain statements need to be executed or not. It contains a body of statements that runs only when the condition given in the if statement is true. If the condition is false, then the statements are skipped over and not executed.

#### Syntax

if expression:

statement (s)

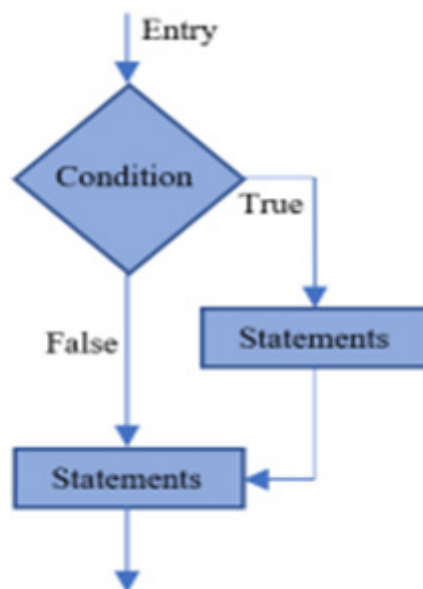


Fig 2.1.1 Flowchart of if statement

#### Example:

x = 105

```
if x > 100:  
    print ("x is greater")
```

### Output

x is greater

#### 2.1.1.2 if else statement

In if else statement if the condition of if statement is true, then all the statements which are

written under if statement will execute, otherwise the else part will execute. The else block should be right after the if block and it is executed when the expression is False.

### Syntax:

if expression:

    statement(s)

else:

    statement(s)

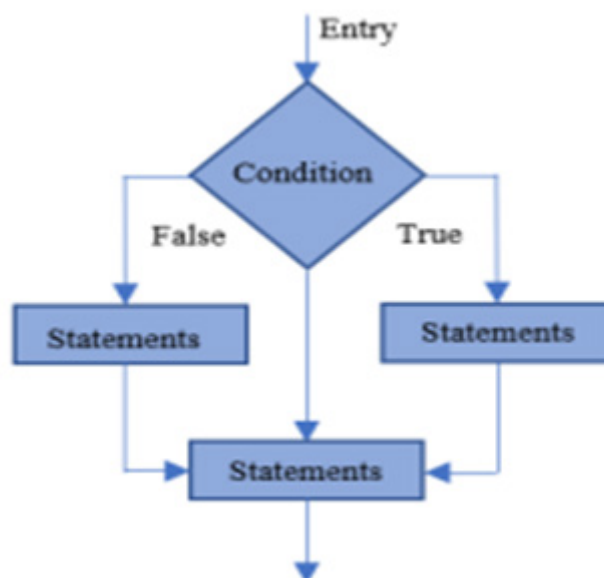


Fig 2.1.2 Flowchart of if else statement

### Example:

```
x = 100  
y = 200  
if x > y:  
    print ("x is greater")  
else:
```

```
print ("y is greater")
```

### Output

y is greater

#### 2.1.1.3 Nested if statements

A nested if statement in Python is an if statement placed inside another if or else block. This structure can have several levels, enabling programmers to check multiple conditions one after another. This can be implemented in two ways. In the first method, if statement can be placed inside the if code block and in second method, if statement can be placed inside the if-else statement.

#### Syntax: Nested if construct

if expression:

    statement(s)

if expression:

    statement(s)

#### Syntax: Nested if construct with else condition

if expression:

    statement(s)

else:

    statement(s)

if expression:

    statement(s)

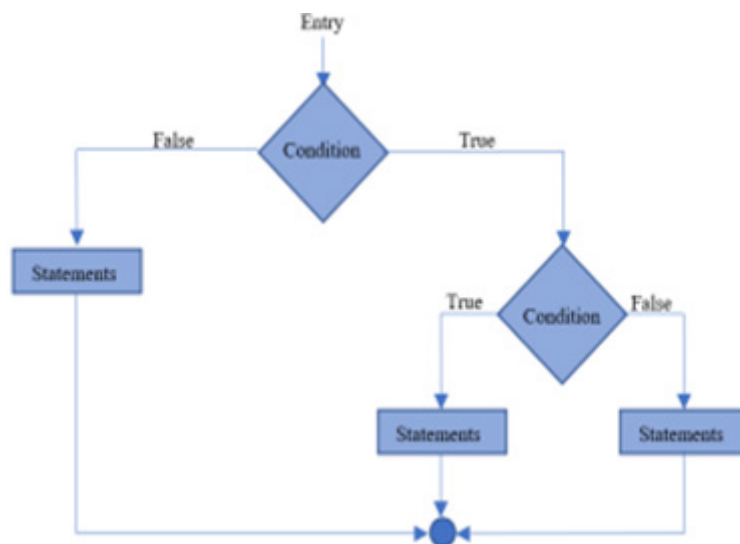


Fig 2.1.3 Flowchart of nested if statement

#### Example:

x = 10

```

if x >= 0:
    if x == 0:
        print("Zero")
    else:
        print ("Positive number")
else:
    print ("Negative number")

```

### Output:

Positive number

#### 2.1.1.4 elif ladder statements

The elif (short for else if) statement allows to check multiple conditions for TRUE and executing a set of code as soon as one of the conditions evaluates to TRUE. It is similar to an if-else statement and the only difference is that in else, it will not check the condition but in elif it will check the condition.

### Syntax:

```

if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
else:
    statement(s)

```

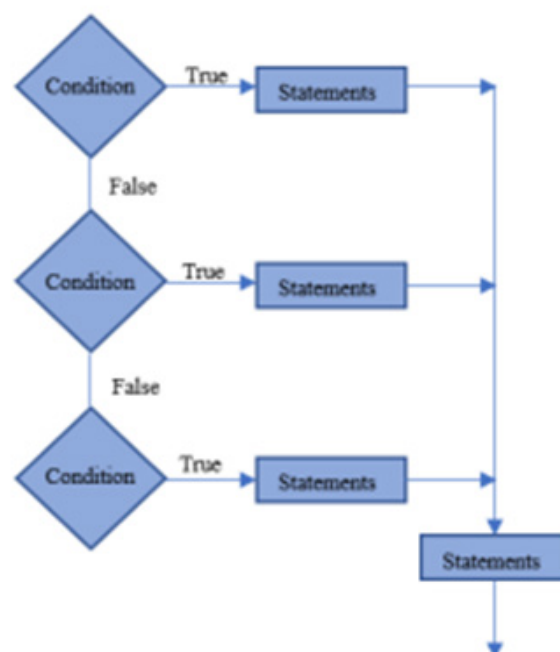


Figure 2.1.4: Flowchart of elif ladder statement

**Example:**

```
x= -2  
  
if x > 0:  
    print ("Positive number")  
  
elif x == 0:  
    print("Zero")  
  
else:  
    print ("Negative number")
```

**Output**

Negative number

### 2.1.2 Loops

A **loop** allows you to **repeat a set of instructions** as long as a certain condition is true or for a specific number of times. Instead of writing the same code over and over, use a loop to **automate repetition**.

Python supports three types of looping statements.

- ◆ for loop
- ◆ while loop
- ◆ nested loops

#### 2.1.2.1 for loop

In Python, the **for loop** allows you to iterate through elements of any sequence, such as a list, tuple, or string. It applies the same set of instructions to each item in the sequence. The loop begins with the keyword **for**, followed by a variable that takes on the value of each item, one at a time. The **in** keyword connects this variable to the sequence being looped through. A colon (:) marks the start of the loop block, and the indented code below runs once for every item in the sequence.

Before a for loop begins, the sequence is evaluated. If it is a list, it is processed first. The loop then assigns the first item to the loop variable. On each iteration, the code block runs using that item, and the loop moves to the next one continuing until all items are processed.

**Syntax:**

```
for var in sequence:  
    statement(s)
```

Here, **var** is a variable to which the value of each sequence item will be assigned during each iteration. Statements represents the block of code that you want to execute repeatedly.

The for loop in python is used to iterate over a sequence like list, tuple, string or range using range ()

Syntax : For variable in range(limit):

# Body of the loop

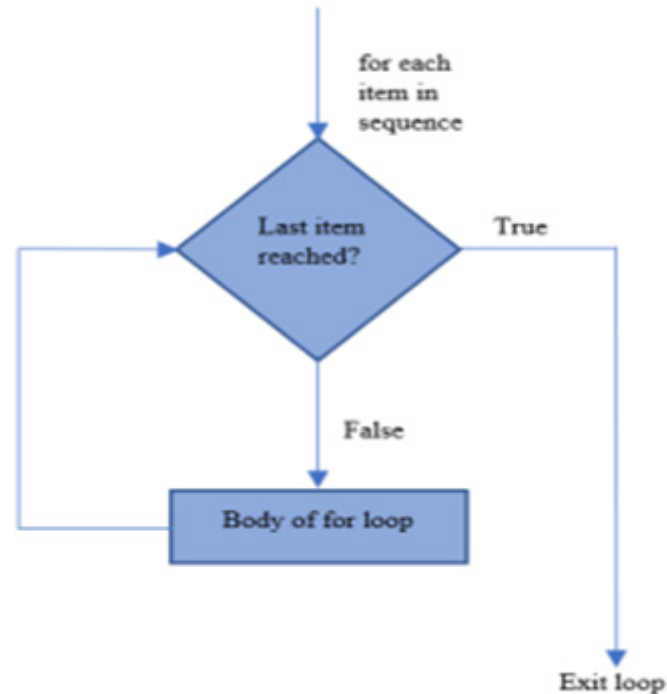


Fig 2.1.5 Flowchart of for loop

#### Example:

```
for i in range(10):  
    print (i, end = " ")
```

#### Output

0 1 2 3 4 5 6 7 8 9

#### 2.1.2.2 while loop

A **while loop** in Python repeatedly runs a block of code **as long as a given condition is true**. It starts with the **while** keyword, followed by a condition and a colon. The indented code below runs until the condition becomes false.

In the while loop, the test expression is checked first. The body of the loop is executed only if the expression evaluates to True. After the first iteration, the test expression is again checked. This process will continue until the test expression becomes False.

#### Syntax:

while expression:

statement(s)

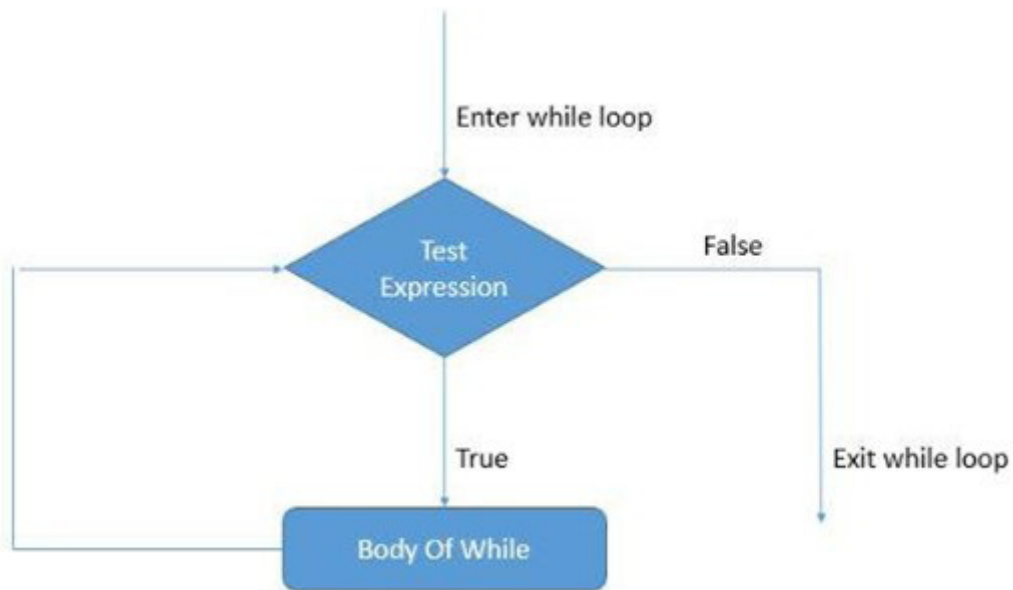


Fig 2.1.6 Flowchart of while loop

**Example:**

```
n = 10
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1
print ("The sum is", sum)
```

**Output**

The sum is 55

**2.1.2.3 Using else Statement with while Loop**

In Python, you can use the else statement in conjunction with a while loop to execute a block of code when the loop has completed all iterations and the condition evaluated in the while statement becomes False. The else block is not executed if the loop is exited prematurely by a break statement.

```
x = 0
while x < 5:
    print(x)
    x += 1
```

else:

```
print ("x is no longer less than 5")
```

### Output

1

2

3

4

X is no longer less than 5

### 2.1.2.4 Nested loops

A nested loop means loops inside the loops. The inner or outer loop can be any type, such as a while loop or for loop. For example, the outer while loop can contain an inner for loop and vice versa. There is no limitation on the chaining of loops. The inner loop will execute one time for each iteration of the outer loop. The number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop.

**Syntax:** Nested for loop within for loop

```
for outer_iterating_var in outer_sequence:
```

```
    for inner_iterating_var in inner_sequence:
```

```
        statement(s)
```

**Syntax:** Nested while loop within while loop

```
while outer_expression:
```

```
    while inner_expression:
```

```
        statement(s)
```

**Example:**

```
for i in range (1, 11):
```

```
    for j in range (1, 11):
```

```
        print (i * j, end=' ')
```

```
print ()
```

### Output

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10 12 14 16 18 20

3 6 9 12 15 18 21 24 27 30  
4 8 12 16 20 24 28 32 36 40  
5 10 15 20 25 30 35 40 45 50  
6 12 18 24 30 36 42 48 54 60  
7 14 21 28 35 42 49 56 63 70  
8 16 24 32 40 48 56 64 72 80  
9 18 27 36 45 54 63 72 81 90  
10 20 30 40 50 60 70 80 90 100

### 2.1.2.5 Loop Control Statements

Statements used to control loops and change the execution from its normal sequence are called control statements. Python supports three types of looping statements.

- ◆ break statement
- ◆ continue statement
- ◆ pass statement

#### 1. break statement

It is used to terminate the execution of the loop statement and the execution transfers to the next statement following the loop.

##### Example:

```
for i in range (10):  
    if i == 5:  
        break                // Exit the loop when i is 5  
    print(i)  
print ("Loop finished")
```

##### Output

```
0  
1  
2  
3  
4 Loop finished
```

In this example, the loop iterates from 0 to 9. However, when i becomes 5, the break statement is executed, and the loop terminates. The "Loop finished." message is then printed.

## 2.continue statement

When the program encounters a continue statement, the python interpreter ignores the rest of the statements in the loop body for the current iteration and returns the program execution to the very first statement in the loop body.

### Example:

```
for i in range (10):  
    if i % 2 == 0:  
        continue //Skip even numbers  
    print(i)
```

### Output

```
1  
3  
5  
7  
9
```

Here, when i is even (the condition  $i \% 2 == 0$  is true), the continue statement is executed. This causes the print(i) statement to be skipped, and the loop moves to the next value of i. Only the odd numbers are printed.

## 3.pass statement

The pass statement is considered as a null operation statement. The interpreter executes the pass statement like a valid python statement but nothing happens when pass is executed.

**Example:** it can be used in an if statement to skip the current iteration.

```
for i in range (5):  
    if i == 2:  
        pass // Do nothing when i is 2  
    else:  
        print(i)
```

### Output

```
0  
1  
3  
4
```

In this case, when i is 2, the pass statement is executed, which does nothing, and the loop continues to the next iteration

## Recap

- ◆ Python supports if, if-else, nested if, and elif ladder for decision making
- ◆ The if statement runs a block only if the condition is true
- ◆ The if-else statement chooses between two blocks based on a condition
- ◆ Nested if statements are used to check multiple conditions inside each other
- ◆ The elif ladder checks multiple conditions and runs the first true one
- ◆ Python has three loop types: for, while, and nested loops
- ◆ The for loop iterates over sequences like lists or strings
- ◆ The while loop repeats as long as a condition is true
- ◆ Nested loops allow placing one loop inside another for complex iteration
- ◆ The break statement exits a loop immediately
- ◆ The continue statement skips the current loop iteration
- ◆ The pass statement does nothing and acts as a placeholder

## Objective Type Questions

1. Is 'if-else' a decision-making statement?
2. What keyword would you use to add an alternative condition to an if statement?
3. What will be the output of the following code?  

```
x = [1, 2, 3, 4, 5]
sum = 0
for var in x:
    sum += var
print(sum)
```
4. Is do while is a valid loop in Python?

5. if (x >= 20), is it a valid Python if statement?
6. If the else statement is used with a while loop, the else statement is executed when the condition becomes -----
7. Which keyword is used to end a loop prematurely?
8. What control statement can be used to create empty blocks?
9. Which symbol is used to indicate the start of a block in loops and conditionals?
10. Which keyword skips the current iteration of a loop?

## Answers to Objective Type Questions

1. Yes
2. elif
3. 15
4. No
5. No
6. FALSE
7. break
8. pass
9. colon
10. continue

## Assignments

1. **Explain the different types of decision-making statements in Python** with syntax and examples
2. Write a Python program to print Fibonacci series.
3. **Write a Python script** to print a multiplication table from 1 to 10 using nested for loops.

4. Explain the difference between Python for loop and while loop.
5. **Write a Python program** that uses a for loop and continue to print only the vowels from the string "Python Programming is fun".

## References

1. Chase, S. (2018). *Python for Data Analysis* (2nd ed.). O'Reilly Media.
2. Van Rossum, G., & Drake, F. L. (2001). *Python 2.0 Reference Manual*. Python Software Foundation.
3. Sweiger, A. (2019). *Automate the Boring Stuff with Python* (2nd ed.). No Starch Press.
4. Müller, A. C., & Guido, S. (2016). *Introduction to Machine Learning with Python*. O'Reilly Media.
5. *Fluent Python* by Luciano Ramalho (2015). O'Reilly Media.

## Suggested Reading

1. Python Crash Course, Eric Matthes (No Starch Press, 2016)
2. Zelle, J. M. (2016). *Python programming: An introduction to computer science* (3rd ed.). Franklin, Beedle & Associates.
3. Downey, A. B. (2015). *Think Python: How to think like a computer scientist* (2nd ed.). O'Reilly Media.
4. Beazley, D. M. (2009). *Python essential reference* (4th ed.). Addison-Wesley Professional
5. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.



# Comprehensions

## Learning Outcomes

After completing this unit, the learner will be able to:

- ◆ define list, dictionary, set, and generator comprehensions.
- ◆ recall the syntax of list comprehension.
- ◆ identify the structure of the dictionary and set comprehensions.
- ◆ list the differences between list and generator comprehensions.
- ◆ explain the purpose of using comprehensions in Python.

## Prerequisites

Before learning comprehensions in Python, you are already familiar with basic programming concepts such as loops (for, while), conditional statements (if, else), and creating and using data structures like lists, dictionaries, and sets. You have written programs that process these structures element by element, often using loops to modify or filter the contents.

Now, imagine if you could achieve the same results using fewer lines of code, with better readability and efficiency. This is where Python comprehensions become useful. By connecting your knowledge of loops and data structures, you will now learn how to use a more elegant and concise way of constructing and transforming sequences.

## Keywords

List Comprehension, Dictionary Comprehension, Set Comprehension, Generator Expression, Python Iterables

## Discussion

### 2.2.1 What is Comprehension

Comprehensions in Python provide concise and efficient constructs that allow programmers to build new sequences from existing ones. These constructs are not only compact but also promote better readability and performance.

To understand the concept of comprehension more intuitively, let us consider the following real-life analogy.

Suppose there are five black dolls placed in a basket. Each doll has a unique number embossed on it. Alongside the basket, there is a tin of white paint, and the task is to repaint all black dolls into white ones.

There are two ways to perform this task:

- ◆ **Method 1:** Paint each doll one by one.
- ◆ **Method 2:** Paint all the dolls simultaneously.

Clearly, the second method is more efficient and convenient.

Now, let us map this scenario to a programming context. Suppose these dolls are represented as elements in a Python list, where each item is a string that combines the color and number of the doll - for example, a black doll with the number 2 is represented as "B2".

To transform all black dolls into white ones, we need to update the color code from "B" to "W" for each element. Python comprehensions allow us to perform this transformation efficiently by generating a new list from the existing one in a single, readable line of code.

#### Understanding Through an Example

Let us now implement this concept using both traditional loop-based programming and list comprehensions.

##### Without Comprehension (Using a For Loop)

In this approach, we iterate over each element of the `black_list` and append a corresponding "white" value to a new list named `white_list`.

```
black_list = ["black", "black", "black", "black", "black"]
print(black_list)
white_list = []
for item in black_list:
    white_list.append("white")
print(white_list)
```

Output:

```
['black', 'black', 'black', 'black', 'black']
```

```
['white', 'white', 'white', 'white', 'white']
```

In the first line of output, the original list `black_list` is displayed, showing all the black dolls. The second line shows the newly created `white_list`, containing all white-colored items.

### With List Comprehension

Python allows the same logic to be implemented more efficiently using list comprehension. The entire transformation can be achieved in a single line:

```
white_list = ["white" for list_item in black_list]
```

Here:

- ◆ `white_list` is the new list to be created.
- ◆ `"white"` is the value to be added for each item.
- ◆ `list_item` is a temporary variable used to iterate over the `black_list`.

The complete code is given below:

```
black_list = ["black", "black", "black", "black", "black"]
```

```
print(black_list)
```

```
white_list = ["white" for list_item in black_list]
```

```
print(white_list)
```

Output:

```
['black', 'black', 'black', 'black', 'black']
```

```
['white', 'white', 'white', 'white', 'white']
```

From the above example, it is clear that using list comprehension not only reduces the number of lines of code but also enhances clarity and simplicity.

### 2.2.2 Types of Comprehensions in Python

Python supports four main types of comprehensions:

1. List Comprehension
2. Dictionary Comprehension

3. Set Comprehension
4. Generator Comprehension

### 2.2.2.1. List comprehension

**List comprehension** is a powerful and concise feature in Python that allows the creation of new lists by processing elements from an existing iterable, such as a list, tuple, or string. It simplifies the code by replacing traditional looping constructs with a single line of expression.

Components of a List Comprehension

A list comprehension generally consists of the following components:

#### 1. Input Sequence (iterable):

This is the source of elements. It can be a list, tuple, string, or any iterable object.

#### 2. Variable (loop variable):

A temporary variable that represents each member of the input sequence during iteration.

#### 3. Predicate Expression (Optional Condition):

A filtering condition that is applied to each element of the input sequence.

Only the elements that satisfy this condition are included in the output list.

#### 4. Output Expression:

An expression that determines how each element in the output list should be formed. This can be a transformation of the input element or the element itself.

#### General Syntax

```
my_list = [<expression> for <item> in <iterable> if <condition>]
```

- ◆ **<expression>** – This is the value you want to put in the new list. It can be the item itself or something you do with the item (like a calculation).
- ◆ **<item>** – A name used to refer to each thing in the original list, one at a time.
- ◆ **<iterable>** – The original list (or similar group of items) that you are going through.
- ◆ **<condition>** – An optional rule. If you include it, only the items that meet this rule will go into the new list.

#### Example: Filtering Items Based on a Condition

Let us consider an example where we want to extract only those days from a list of week days that contain the letter "u"

#### Output:

```
['Sunday', 'Tuesday', 'Thursday', 'Saturday']
```

As a result, a new list `new_list` is created containing only the elements 'Sunday', 'Tuesday', 'Thursday', and 'Saturday', as they all contain the letter 'u'.

### 2.2.2.2 Dictionary Comprehensions

In Python, a dictionary is an unordered collection of data that stores values in the form of key-value pairs. Each key is unique and is used to retrieve its corresponding value efficiently. Python provides a convenient and elegant way to create or modify dictionaries using dictionary comprehensions, which are similar in structure and purpose to list comprehensions.

Before understanding dictionary comprehensions, it is important to recall some basic characteristics of dictionaries:

- ◆ All elements in a dictionary are enclosed within **curly braces** `{}`.
- ◆ Each element in the dictionary is represented as a **key-value pair**, where the key and value are separated by a colon `:` (e.g., `"name": "John"`).
- ◆ Keys must be unique and immutable (e.g., strings, numbers, or tuples), while values can be of any data type.
- ◆ Values are accessed using their corresponding keys, not their positions, unlike lists or tuples.

### What is Dictionary Comprehension?

Dictionary comprehension is a concise way to create a new dictionary by iterating over an iterable and applying an expression that generates key-value pairs.

It helps in:

- ◆ Creating dictionaries from lists, tuples, or ranges.
- ◆ Filtering data based on a condition.
- ◆ Applying transformations to both keys and values.

### Syntax of Dictionary Comprehension

```
dictionary = {<key_expression>: <value_expression> for <item> in  
<iterable> if <condition>}
```

Where,

- ◆ `<key_expression>` – An expression that defines the key for each dictionary item.
- ◆ `<value_expression>` – An expression that defines the value associated with the key.
- ◆ `<item>` – The loop variable representing each element in the iterable.
- ◆ `<iterable>` – The data source from which elements are drawn (e.g., a list, range, or tuple).

- ◆ `<condition>` – (Optional) A filtering condition to include only selected items.

### Example: Creating a Dictionary of Squares

```
my_dict = {x: x ** 2 for x in range(10)}  
  
print(my_dict)
```

### Output:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

In the above example, the first part of the comprehension specifies how the key-value pairs should be generated for each iteration. The second part functions similarly to a standard for loop, where we specify the iterable that Python should process.

### 2.2.2.3 Set Comprehensions

Set comprehension in Python is a concise and elegant way to construct a new set by performing operations on each element of an iterable. It is very similar to list comprehension, with one important difference, sets do not allow duplicate elements. Therefore, set comprehensions automatically eliminate duplicates from the result.

### Syntax:

```
set_name = {expression for element in iterable}
```

Where,

- ◆ **set\_name:** Refers to the name of the new set that is being created.
- ◆ **expression:** Defines the value to be included in the new set. This expression can be a direct reference to the element, or it can be a more complex expression such as a mathematical operation.
- ◆ **element:** Represents each individual item taken from the given iterable.
- ◆ **iterable:** Any Python object capable of returning its elements one at a time (e.g., list, tuple, set, or range).

### Example:

```
old_set = [10, 20, 30, 40, 50]  
new_set = {element * 2 for element in old_set}  
print("The old set is:")  
print(old_set)  
print("The newly created set is:")  
print(new_set)
```

---

### Output:

The old set is:

```
[10, 20, 30, 40, 50]
```

The newly created set is:

```
{100, 40, 10, 20, 50, 60, 30, 70, 80, 90}
```

### 2.2.2.4 Generator Comprehensions

Generator comprehensions in Python are syntactically similar to list comprehensions. However, unlike list comprehensions, which generate the entire list in memory at once, generator comprehensions produce items one at a time and on demand. This characteristic makes them significantly more memory efficient, especially when working with large datasets.

### Syntax:

```
gen_list = (<expression> for <item> in <iterable> if <condition>)
```

Where,

- ◆ **gen\_list**: Refers to the generator object that will be created.
- ◆ **expression**: Represents the operation or transformation to be applied to each element from the original iterable.
- ◆ **item**: Denotes the current item being accessed from the iterable during iteration.
- ◆ **iterable**: Any Python object capable of returning its elements one at a time (such as a list, set, tuple, or range).
- ◆ **condition(optional)**: A logical expression used to filter elements from the iterable. Only elements satisfying this condition are processed and included in the generator output.

### Example:

```
old_list = [1, 1, 2, 3, 4, 4, 5, 6, 7, 7, 9]
new_gen = (var for var in old_list if var % 2 == 0)
print("New values using generator comprehensions:")
for var in new_gen:
    print(var)
```

Output:

New values using generator comprehensions:

2

4

4

6

## Recap

- ◆ Comprehensions provide a concise way to build new sequences from existing ones.
- ◆ Python supports four main types of comprehensions:
  1. List Comprehension
  2. Dictionary Comprehension
  3. Set Comprehension
  4. Generator Comprehension
- ◆ List Comprehension : Used to create new lists from iterables.
  - ◆ General Syntax: [<expression> for <item> in <iterable> if <condition>]
- ◆ Dictionary Comprehension: Creates dictionaries using key-value pairs from iterables.
  - ◆ Syntax: {<key\_expr>: <value\_expr> for <item> in <iterable> if <condition>}
- ◆ Set Comprehension : Creates sets (unique elements only) from iterables.
  - ◆ Syntax: {expression for element in iterable}
- ◆ Generator Comprehension: Creates generator objects (yields items one by one, memory efficient).
  - ◆ Syntax: (<expression> for <item> in <iterable> if <condition>)

## Objective Type Questions

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

```
list_string = ['a','b','c','d','e','f','g','h','i','j','k']  
a = [print(k) for k in list_string if k not in "aeiou"]
```
2. Write a list comprehension for producing a list of numbers between 1 and 1000 and are divisible by 3.
3. State the main difference between list and generator comprehension.
4. What will be the output of the following Python code?  

```
print([if k%2==0: k; else: k+1; for k in range(4)])
```
5. ----- comprehension allows us to retrieve values based on the keys.
6. A key-value pair is often called
7. Write syntax for declaring dictionary comprehension.
8. 

```
days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",  
        "Friday", "Saturday"]  
  
new_list = [i for i in days if "u" in i]  
  
print(new_list)
```
9. Which symbols are used to define dictionary comprehensions?
10. What is the general syntax of a list comprehension with a condition?

## Answers to Objective Type Questions

1. b c d f g h j k
2. `[i for i in range(1000) if i%3==0]`
3. Memory allocation
4. `[1, 1, 3, 3]`
5. Dictionary
6. Dictionary
7. `dictionary = {expression for key: value in iterable (if conditional)}`

8. Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
9. Curly braces {} with key-value pairs
10. [expression for item in iterable if condition]

## Assignments

1. State the advantages of comprehensions in Python over loops.
2. Explain different python comprehension in detail.
3. Write the list comprehension equivalent for:  
{x : x is a whole number less than 20, x is even} (including zero)
4. Write a Python program using list comprehension to create a list of even numbers from 1 to 20.
5. Use dictionary comprehension to create a dictionary where the keys are numbers from 1 to 5 and the values are their squares.

## References

1. <https://coderpad.io/blog/development/python-list-comprehension-guide/>

## Suggested Reading

1. Guide To: Functional Python & Comprehension Constructs, Matt Harrison, 2013
2. Severance, Charles. *Python for everybody: Exploring Data using python 3*. Charles Severance, 2016.
3. Ramalho, Luciano. *Fluent Python: Clear, concise, and effective programming*. "O'Reilly Media, Inc.", 2015.



## Functions

### Learning Outcomes

After completing this unit, learners will be able to:

- ◆ familiarise the concept and importance of functions in Python.
- ◆ make aware of how to define and call user-defined functions in Python.
- ◆ explore the different types of arguments in Python functions
- ◆ gain knowledge on variable-length arguments
- ◆ identify the concepts of scope and lifetime of variables in Python functions.

### Prerequisites

Having a strong foundation in fundamental programming concepts is crucial before embarking on the study of Python functions. It is necessary to be familiar with basic programming concepts like variables, data types, control flow structures, and the fundamental syntax of Python. Additionally, a solid understanding of Python's built-in data structures such as lists, dictionaries, and tuples is essential, as functions often operate on these structures. Proficiency in manipulating and understanding their properties will greatly enhance the comprehension of Python functions.

Understanding variable scope and its impact on variable accessibility within functions is also vital. Knowledge of local and global variables, as well as their scopes, is necessary to comprehend how Python functions behave. A grasp of these concepts will provide insights into how variables are defined, accessed, and modified within functions.

Furthermore, having a basic understanding of object-oriented programming (OOP) concepts can be advantageous when exploring more advanced aspects of Python functions. Familiarity with concepts like classes, objects, methods, and inheritance will deepen the understanding of how functions can be utilized within an object-oriented paradigm.

# Key Concepts

Arguments, Global Variable, Local Variable, Recursion

## Discussion

A function is a group of statements designed to carry out a specific task. The main purpose of using functions is to combine commonly used or repetitive tasks into a single block of code. This allows us to perform the same task multiple times throughout a program without rewriting the code each time. As programs become larger and more complex, functions help keep the code organized, reusable, and easier to manage.

### 2.3.1 Advantages of Functions

1. **Modularity:** Functions break down complex programs into smaller, manageable parts, making code more understandable, testable, and maintainable. They promote code reusability and the principle of avoiding repetition.
2. **Code Organisation:** Functions provide a structured approach to programming by dividing code into logical units. This improves overall code navigation, comprehension, and reduces the likelihood of errors. Well-organized code is easier to read and maintain.
3. **Reusability:** Functions enable code reuse, allowing them to be called multiple times with different inputs. This eliminates the need for redundant code, promoting efficient programming practices.
4. **Abstraction:** Functions hide internal implementation details, allowing users to focus on what the function does and how to use it. This simplifies programming and promotes a higher-level understanding of the program's functionality.
5. **Testing and Debugging:** Functions facilitate testing and debugging efforts as they can be individually tested to ensure proper functionality. Debugging becomes more manageable as errors can be isolated to specific functions, enabling focused troubleshooting.
6. **Collaboration:** Functions promote collaboration by dividing code into independent units, enabling different team members to work on separate functions concurrently. Functions also facilitate code sharing and open-source collaboration within the Python community.
7. **Code Maintainability:** Functions enhance code maintainability by isolating specific tasks. Updates or changes can be made to individual functions, reducing the risk of introducing bugs. This simplifies the maintenance process, particularly in large and complex projects.

## 2.3.2 Types of Function

There are two types of functions

- ◆ User Defined Functions
- ◆ Built-in Functions

### 2.3.2.1 User Defined Functions

User-defined functions are those functions that we define ourselves to do certain specific tasks. A user-defined function in Python is a piece of code that you create to perform a particular task. It enables you to group a set of instructions together under a unique name, enhancing the modularity and organization of your code. This named entity becomes a custom function within your program, allowing you to call it repeatedly with different inputs.

#### Creating a Function

We can define our own functions in Python by using the "def" keyword.

The syntax of a Python function is

```
def function_name (parameter1, parameter2, ...):  
    """  
  
    Docstring: Description of the function (optional).  
  
    """
```

To define a function in Python, we use the "def" keyword, followed by the function name, which should be a valid identifier in Python. If the function takes any parameters, they are placed within parentheses and separated by commas.

A docstring, which is an optional multi-line string enclosed in triple quotes ("""), can be included right after the function definition. This docstring provides a brief description of the function's purpose, parameters, and return values.

The function body comprises the code block that specifies the behavior of the function. It starts with a colon (:) and is indented consistently. The body can contain multiple statements, all indented at the same level.

If the function is expected to return a value, the "return" statement is used to specify the value(s) to be returned. The return statement is optional, and if it is not included, the function will automatically return None. It is possible to return a single value or multiple values separated.

Here is a simple python function definition:

```
def greet(name): """Prints a greeting message."""  
  
print("Hai, " + name + "!")
```

In this example, we have defined a function called greet that takes a parameter 'name'.

The function's purpose is to print a greeting message to the console. When the function is called with a specific name, it will print "Hai, " followed by the provided name and an exclamation mark.

### Calling a Function

To call a function in Python, you simply write the function name followed by parentheses. If the function requires any arguments, you provide them inside the parentheses.

To call the above example function, we write:

```
greet("Ann")
```

Here the passed string argument is "Ann". The function is then executed, resulting in the output "Hai, Ann!" being displayed on the console.

#### 2.3.2.2 Built-in Functions

Built-in functions are an integral part of Python's standard library, offering a broad range of functionalities that streamline coding tasks. These functions are readily accessible without the need for additional installation or setup. An example of such a fundamental built-in function is `print()`, which enables us to display text or variables on the console. By simply passing the desired content as arguments, we can swiftly output information to the user. For instance, executing `print("Hello, world!")` will print the phrase "Hello, world!" on the console.

Python's built-in functions provide numerous ways to manipulate and analyze data. One such function is `sorted()`, which returns a new list containing the sorted elements from the input iterable. For instance, using `sorted([5, 2, 7, 1, 3])` will yield `[1, 2, 3, 5, 7]`, showcasing how the function arranges the elements in ascending order. Another useful built-in function is `len()`, which determines the length of an object, such as a string, list, or tuple. By employing `len()`, we can quickly ascertain the number of elements in a given collection. For example, `len("Python")` will return the value 6, representing the length of the string "Python".

#### 2.3.3 Arguments

Arguments are the values that you pass to a function during its invocation. They serve as inputs for the function to perform specific operations or calculations. Python supports various types of arguments, including:

- 1. Positional Arguments:** These arguments are provided to a function in the same order as they are defined in the function's parameter list. The values are assigned to the respective parameters based on their positions.

#### Example:

```
def add_numbers(x, y):  
    """Adds two numbers."""  
    return x + y  
result = add_numbers(3, 5)
```

```
print(result)
```

```
# Output: 8
```

In this example, 3 is assigned to x and 5 is assigned to y based on their positions.

- 2. Keyword Arguments:** With keyword arguments, you explicitly specify the parameter name followed by the corresponding value, separated by an equal sign. This allows you to pass arguments in any order, disregarding their position in the parameter list.

**Example:**

```
def add_numbers(x, y):
```

```
    """Adds two numbers."""
```

```
    return x + y
```

```
result = add_numbers(y=4, x=2)
```

```
print(result)
```

```
# Output: 6
```

Here, the function is called with keyword arguments, allowing us to specify the values explicitly.

- 3. Default Arguments:** Default arguments have predefined values assigned to them in the function's parameter list. If an argument is not supplied during the function call, the default value is used instead.

**Example:**

```
def greet(name, message="Hai"):
```

```
    """Prints a personalized greeting."""
```

```
    print(message + ", " + name)
```

```
    greet("Ann")
```

```
# Output: Hai, Ann
```

```
    greet("Balu", "Hello")
```

```
# Output: Hello, Balu
```

In this example, the message parameter has a default value of "Hai". If not provided, the default value is used.

- 4. Variable-length Arguments:** Python functions can accept a varying number of arguments. To achieve this, you can use the asterisk (\*) before a parameter name for variable-length positional arguments, or two asterisks (\*\*) for variable-length keyword arguments.

**Example:**

```
def add(*numbers):  
    """Addition of numbers."""  
    total = sum(numbers)  
    return total  
result = add(1, 2, 3, 4, 5)  
print(result)  
# Output: 15
```

The function add accepts a variable number of positional arguments using \*numbers. The arguments are treated as a tuple inside the function.

- ◆ def add(\*numbers): -This defines a function named add that accepts a variable number of arguments.
- ◆ The asterisk \* before numbers collect all positional arguments into a tuple called numbers.
- ◆ total = sum(numbers) - The built-in sum() function is used to add all the values inside the numbers tuple.
- ◆ return total - The total sum is returned from the function.

**5. Unpacking Arguments:** Arguments can be unpacked from a list or tuple using the asterisk (\*) operator. This allows you to pass the individual elements of a sequence as separate arguments to a function.

**Example:**

```
def add(a, b, c):  
    """Adds three numbers."""  
    return a + b + c  
numbers = [2, 3, 4]  
result = add(*numbers)  
print(result)  
# Output: 9
```

In this example, the elements of the numbers list are unpacked using \* and passed as separate arguments to the add function.

**Function Definition:**

- ◆ def add(a,b,c): defines a function named add that takes three parameters: a, b, and c.



- ◆ `return a + b + c` returns the sum of the three parameters.

### List Creation:

- ◆ `numbers = [2, 3, 4]` creates a list named `numbers` containing three integers.

### Argument Unpacking:

- ◆ `result = add(*numbers)` uses the `*` operator to unpack the list `numbers` into separate arguments.
- ◆ This is equivalent to calling `add(2,3,4)`.
- ◆ The `*` operator is used to unpack iterables (like lists or tuples) into separate positional arguments when calling a function. This is useful when the number of elements in the iterable matches the number of parameters the function expects.

## 2.3.4 Pass by Reference and Pass by Value

Pass by Value and Pass by Reference are two different methods used to provide arguments to functions in programming.

### 2.3.4.1 Pass by Reference (Mutable Objects)

In Python, when an object is passed as an argument to a function using pass by reference, a reference to the object's memory location is passed. This means that any modifications made to the object within the function will impact the original object outside the function as well. However, it is important to note that in Python, all variable assignments are references to objects. So, when an object is passed to a function, a reference to the object is passed as well. If the function modifies the object directly, the changes will be visible outside the function since it operates on the same underlying object.

#### Example:

```
def modify_list(lst):  
    lst.append(4) # Modifying the list within the function  
  
test_list = [1, 2, 3]  
modify_list(test_list)  
print(test_list)  
# Output: [1, 2, 3, 4]
```

#### Function Definition:

- ◆ `modify_list(lst)` is a function that takes a list `lst` as its parameter.
- ◆ Inside the function, `lst.append(4)` adds the integer 4 to the end of the list.

#### List Initialization:

- ◆ `test_list = [1, 2, 3]` creates a list with elements 1, 2, and 3.

### Function Call:

- ♦ `modify_list(test_list)` calls the function with `test_list` as the argument. Since lists are mutable in Python, any modifications made to `lst` inside the function will affect `test_list`.

In this example, the `test_list` object (a list) is passed to the `modify_list` function. The function modifies the list by appending an element. Since lists are mutable objects, the changes made to the list within the function are also reflected in the original list outside the function.

#### 2.3.4.2 Pass by Value (Immutable Objects)

When using pass by value, a copy of the object's value is passed as an argument to a function. This means that any modifications made to the object within the function do not affect the original object outside the function. However, it is important to note that in Python, objects of immutable types, such as integers, strings, and tuples, are passed by value. If the function modifies the object directly, a new object is created, while the original object remains unchanged.

#### Example:

```
def modify_number(num):  
  
    num += 1 # Modifying the number within the function  
  
test_number = 5  
  
modify_number(test_number)  
  
print(test_number)  
  
# Output: 5
```

In this example, the `test_number` object (an integer) is passed to the `modify_number` function. However, integers are immutable objects in Python. Therefore, any modifications made to the `num` variable within the function do not affect the original `test_number` object outside the function.

### 2.3.5 Scope and Lifetime of Variables

Scope and lifetime of variables determine where and for how long a variable is accessible and exists in a program.

#### 2.3.5.1 Scope

- ♦ **Global Scope:** Variables defined outside any function or class have global scope, meaning they can be accessed from anywhere within the program.
- ♦ **Local Scope:** Variables defined inside a function or block have local scope, which means they are only accessible within that specific function or block.

#### 2.3.5.2 Lifetime of Variables

- ♦ **Global Variables:** Global variables are created when the program starts and



persist throughout the entire execution of the program. They are destroyed when the program terminates.

- ◆ **Local Variables:** Local variables have a limited lifetime within the scope of the function or block in which they are defined. They are created when the function or block is entered and cease to exist when the function or block is exited.

**Example:**

```
def test_function():  
  
    local_var = 15 # Local variable within the function  
  
    print("Local variable:", local_var)  
  
    global_var = 25 # Global variable  
  
test_function()  
  
print("Global variable:", global_var)
```

#Output:

Local variable: 15

Global variable: 25

In this example, we have a function called `test_function()` that defines a local variable `local_var` with a value of 15. This variable is only accessible within the scope of the function. When the function is called, the local variable is created and printed.

We also have a global variable `global_var` defined outside the function. Global variables are accessible from anywhere in the program. It is printed after the function call.

### 2.3.6 Return Values

Return values in Python pertain to the values that a function can provide to the caller once it has executed its tasks. The return statement is employed to indicate the specific value that a function will return.

#### 1. Single Value Return

To return a single value, a function employs the return statement followed by the value that will be returned. This allows the function to provide a single result to the caller.

**Example:**

```
def multiply(a, b):  
  
    return a * b  
  
result = multiply(3, 4)  
  
print(result)
```

# Output: 12

## 2. Multiple Value Return

Functions have the ability to return multiple values by listing them separated by commas within the return statement. This allows the function to provide multiple results as a tuple or any other sequence type.

### Example:

```
def get_person_details():  
    name = "Ann"  
    age = 25  
    occupation = "Teacher"  
    return name, age, occupation  
  
person = get_person_details()  
print(person)  
  
# Output: ("Ann", 25, "Teacher")
```

## 3. Empty Return

In situations where a return statement is encountered without a value or if a function lacks a return statement, Python implicitly returns None.

### Example:

```
def is_even(number):  
    if number % 2 == 0:  
        return True  
    else:  
        return  
  
result1 = is_even(4)  
result2 = is_even(5)  
print(result1) # Output: True  
print(result2) # Output: None
```

In this example, we have a function called `is_even()` that checks whether a given number is even. If the number is divisible by 2, the function returns True using the `return True` statement. If the number is not even, the function does not explicitly provide a return value.

When we call `is_even()` with the number 4, the function returns True, indicating that 4 is an even number. We assign the return value to the variable `result1` and print it, which outputs True.



When we call `is_even()` with the number 5, which is an odd number, the function does not have a return statement for this case. In such situations, Python implicitly returns `None`. We assign the return value to the variable `result2` and print it, which outputs `None`.

### 2.3.7 Function within Functions

In Python, it is permissible to define a function within another function, which is referred to as a nested function or a function within a function. This allows for the creation of a local function that can only be accessed and invoked from within the enclosing function. The inner function has visibility and access to the variables and parameters of the outer function, forming a nested scope.

#### Example:

```
def outer_function():  
    def inner_function():  
        print("This is the inner function")  
    print("This is the outer function")  
    inner_function()  
outer_function()
```

In this example, the `outer_function` defines the `inner_function` within it. When the `outer_function` is called, it prints "This is the outer function" and then invokes the `inner_function`. The `inner_function`, in turn, prints "This is the inner function".

### 2.3.8 Anonymous Functions

Anonymous functions in Python are also known as lambda functions. They are compact and inline functions that can be defined without the traditional "def" keyword. Instead, lambda functions are created using the "lambda" keyword, followed by a parameter list, a colon (:), and an expression that defines the function's behavior.

For example, consider a lambda function that calculates the square of a given number:

```
square = lambda x: x ** 2  
result = square(5)  
print(result)  
# Output: 25
```

In this example, we define an anonymous function (lambda function) named "square" that takes an argument "x" and returns its square (`x ** 2`).

We then call the lambda function with the argument "5" and store the result in the variable "result". Finally, we print the value of "result", which outputs "25".

### 2.3.9 Recursive Function

A recursive function is a function that invokes itself during its execution. It is employed when a problem can be divided into smaller, similar subproblems. With each recursive call, the function addresses a reduced version of the problem until a base case is encountered, which serves as the stopping condition for the recursion.

#### Example:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)
```

In this example, the factorial function takes an integer  $n$  as an argument. It checks if  $n$  is equal to 0, which represents the base case. If it is, the function returns 1. Otherwise, it recursively calls itself with the argument  $n - 1$  and multiplies the result by  $n$ . This process continues until the base case is reached.

Let's use this function to calculate the factorial of 5:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
result = factorial(5)  
print(result)  
  
# Output: 120
```

In this case, we call the factorial function with the argument 5 and assign the result to the variable `result`. The factorial of 5 is calculated by recursively multiplying 5 by the factorial of 4, which further multiplies 4 by the factorial of 3, and so on, until we reach the base case. The final result, 120, is then printed.

## Recap

- ◆ A function is a set of statements that performs a specific task and avoids code repetition.
- ◆ Functions help make programs modular, organized, reusable, and easier to manage.
- ◆ Functions improve code readability, maintenance, and collaboration in large projects.
- ◆ Two main types of functions: user-defined functions and built-in functions.
- ◆ User-defined functions are created using the `def` keyword and can be reused multiple times.
- ◆ Built-in functions are provided by Python and can be used without importing anything.
- ◆ Function syntax includes the function name, optional parameters in parentheses, an optional docstring, and an indented function body.
- ◆ The `return` statement is used to return a value from a function; if not used, the function returns `None` by default.
- ◆ Functions can be called using their name followed by parentheses, with arguments passed if needed.
- ◆ Positional arguments are matched by position, keyword arguments are matched by name.
- ◆ Default arguments allow functions to be called with fewer arguments than defined.
- ◆ Variable-length arguments use `*args` for positional and `**kwargs` for keyword arguments.
- ◆ Unpacking arguments from lists or tuples can be done using the `*` operator.
- ◆ Python uses pass by reference for mutable objects like lists; changes affect the original object.
- ◆ Python uses pass by value for immutable objects like integers; changes inside the function do not affect the original.
- ◆ Global scope means the variable is accessible throughout the program; local scope means it is accessible only within the function.
- ◆ Global variables exist for the entire duration of the program; local variables are created and destroyed during function execution.
- ◆ Functions can return a single value, multiple values as a tuple, or `None` if no return is specified.

- ◆ Nested functions (functions within functions) are possible, and inner functions can access variables from the outer function.
- ◆ Anonymous functions (lambda functions) are defined using the `lambda` keyword and are useful for short, simple tasks.
- ◆ Recursive functions call themselves and must include a base case to prevent infinite recursion.

## Objective Type Questions

1. Which keyword is used to define a user-defined function in Python?
2. What symbol is used to denote a docstring in a function?
3. What is the default return value of a Python function if no return statement is used?
4. What kind of function is defined without a name in Python?
5. What type of arguments are passed using the parameter names?
6. What kind of arguments have predefined values in the function definition?
7. What is the term for a function that calls itself?
8. What do you call a function defined inside another function?
9. Which built-in function is used to find the number of elements in a list or string?
10. What is the scope of a variable defined inside a function?

## Answers to Objective Type Questions

1. `def`
2. Triple quotes (`"""`)
3. `None`
4. Lambda function
5. Keyword arguments



6. Default arguments
7. Recursive function
8. Nested function
9. len()
10. Local scope

## Assignments

1. Write a Python function called "calculate\_average" that takes in a list of numbers as an argument and returns the average of those numbers.
2. Create a Python function called "reverse\_string" that takes a string as input and returns the reverse of that string.
3. Implement a Python function named "count\_vowels" that accepts a string and returns the count of vowels (a, e, i, o, u) present in that string.
4. Write a Python function called "is\_palindrome" that takes a string as input and checks if it is a palindrome. The function should return True if the string is a palindrome and False otherwise.
5. Develop a Python function called "find\_maximum" that accepts a list of numbers and returns the largest number from that list.

## References

1. Matthes, E. (2019). Python Crash Course: A Hands-On, Project-Based Introduction to Programming. No Starch Press.
2. Ramalho, L. (2015). Fluent Python: Clear, Concise, and Effective Programming. O'Reilly Media.
3. Beazley, D., & Jones, B. K. (2013). Python Cookbook: Recipes for Mastering Python 3. O'Reilly Media.
4. <https://docs.python.org/3/tutorial/modules.html>

## Suggested Reading

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
2. Zelle, J. M. (2016). *Python programming: An introduction to computer science* (3rd ed.). Franklin, Beedle & Associates Inc.
3. Sweigart, A. (2019). *Automate the boring stuff with Python: Practical programming for total beginners* (2nd ed.). No Starch Press.
4. Beazley, D. M., & Jones, B. K. (2013). *Python cookbook: Recipes for mastering Python 3* (3rd ed.). O'Reilly Media.
5. Downey, A. (2015). *Think Python: How to think like a computer scientist* (2nd ed.). O'Reilly Media.



## Modules & Packages

### Learning Outcomes

After completing this unit, you will be able to:

- ◆ understand the concept and purpose of Python modules and packages.
- ◆ create and define your own Python modules containing functions, classes, and variables.
- ◆ import and use modules in your Python scripts for code reuse and organization.
- ◆ create and structure your own Python packages with subpackages.
- ◆ import modules and subpackages from packages using the appropriate import statements.
- ◆ understand the hierarchical structure of packages and how to navigate through subpackages.

### Prerequisites

To learn and understand Python modules and packages effectively, it is important to have certain prerequisites. First, you should have a basic understanding of Python, including variables, data types, control structures, functions, and basic object-oriented programming concepts. Additionally, a good grasp of Python syntax is necessary, which includes knowledge of defining functions and classes, using modules, import statements, and working with variables and data structures. Understanding fundamental programming concepts like code organization, code reuse, and modularization is beneficial, as it helps in comprehending the purpose and advantages of modules and packages. Familiarity with file and directory operations in Python, such as reading and writing files, navigating directories, and manipulating file paths, will be useful when dealing with modules and packages organized in a file system hierarchy. Lastly, setting up a Python environment on your machine, including the installation of Python and a suitable code editor or IDE, is necessary to write and execute Python code. By fulfilling these prerequisites, you will be better equipped to grasp the concepts, implementation, benefits, and best practices related to Python modules and packages.

## Key words

Import, module, package

## Discussion

### 2.4.1 Modules

Python modules are files that contain Python code, defining functions, classes, and variables that can be utilized in other Python programs. Their purpose is to organize and reuse code, encouraging modularity and reusability. Modules aid in separating different concerns and making code easier to maintain. They can be either built-in modules included in the Python standard library or external modules developed by the Python community and installed using tools like pip. By importing modules into our programs, we can access their functionality and utilize their defined objects to perform various tasks, saving time and effort by avoiding the need to write code from scratch.

#### 2.4.1.1 Creating a Python module

Creating a Python module follows a simple syntax. To define a module, you create a new Python file with a .py extension. Inside this file, you can define functions, classes, and variables that you intend to use in other Python programs. These definitions allow you to organize and reuse code effectively.

#### Example:

Here's an example of creating a Python module named math\_operations.py. Write the code first.

```
def add(a, b):  
    return a + b  
  
def subtract(a, b):  
    return a - b  
  
def multiply(a, b):  
    return a * b  
  
def divide(a, b):  
    if b != 0:  
        return a / b  
    else:  
        print("Error: Division by zero is not allowed.")
```

Now save the file as `math_operations.py`. The module named `math_operations.py` will be created. In this example, the module `math_operations` contains four functions: `add`, `subtract`, `multiply`, and `divide`. These functions perform basic mathematical operations and can be reused in other Python programs. In Python, both **modules** and **normal files** are files containing Python code, but they serve different purposes. A normal Python file (e.g., `script.py`) is meant to be executed as a standalone program. A module is a Python file (`.py`) that is intended to be imported and used in other scripts.

#### 2.4.1.2 The import Statement

The `import` statement is used in Python to bring modules or specific objects from modules into the current program's namespace. It allows us to access and utilize the functionality defined within the imported modules.

##### Syntax:

```
import module_name
```

##### Example:

```
import math_operations

result = math_operations.add(5, 3) [Refer section 2.4.1.1]

print(result)

# Output: 8

result = math_operations.divide(10, 2)

print(result)

# Output: 5.0
```

In the above code, we import the `math_operations` module and use its functions `add` and `divide` to perform addition and division operations respectively.

#### 2.4.1.3 Naming a Module

Choosing a suitable name for a Python module is crucial and involves the following guidelines:

1. **Descriptive:** Opt for a name that precisely describes the module's purpose and functionality. A descriptive name allows others to understand its role with minimal effort.
2. **Concise:** Keep the module name short and avoid unnecessary length. Shorter names are easier to type, remember, and fit well within code.
3. **Lowercase:** Use lowercase letters for module names. This is a common convention in Python, distinguishing modules from classes and constants.
4. **Underscores:** When using multiple words in the module name, separate them with underscores (`_`) for better readability. For example, it prefers

"my\_module" instead of "mymodule".

5. **Avoid conflicts:** Ensure that the module name doesn't conflict with any Python keywords or built-in module names. This helps prevent naming clashes and potential issues.
6. **Meaningful and self-explanatory:** Select a module name that conveys meaning and is self explanatory. This empowers developers, including yourself in the future, to grasp the module's purpose without delving into the code details. For example, if developing a module for string manipulation utilities, consider names like "string\_utils" or "str\_helpers". These names clearly convey the module's focus on string-related functionalities.

#### 2.4.1.4 Renaming a Module

To rename a module in Python, you need to perform the following steps:

1. **Change the module file name:** Modify the name of the module file by renaming it while preserving the .py extension. For instance, if the original module file was named "old\_module.py", rename it to "new\_module.py".
2. **Update import statements:** Scan through other code files and locate import statements that refer to the original module name. Update these import statements to use the new module name instead. Replace occurrences of "import old\_module" with "import new\_module".
3. **Modify module references:** If there are any references to the original module within the code files, update them to reflect the new module name. For example, if there was a function called "old\_module.some\_function()", change it to "new\_module.some\_function()".
4. **Test and validate:** Execute the code and ensure that everything functions correctly after renaming the module. Verify for any errors or unexpected behavior, and make any necessary adjustments.

#### 2.4.1.5 Variables in Module

Variables within a Python module can be accessed and used by other programs that import the module. They serve as containers for storing data that can be shared between different parts of a program or even across multiple programs. This allows for efficient data organization and sharing, promoting code modularity and reusability.

##### Example:

```
# my_module.py

my_variable = "Hello, World!"

def print_variable():

    print(my_variable)
```



In this example, the module `my_module` contains a variable named `my_variable` assigned with the string value "Hello, World!". It also includes a function `print_variable()` that prints the value of `my_variable`.

#### 2.4.1.6 Executing a Module as a Script

To execute a Python module as a script, you can utilize the special construct `if __name__ == "__main__":`. This construct allows you to differentiate between when the module is being directly executed as a script versus when it is being imported as a module.

Here's an example of how to execute a module as a script:

```
def add(a, b):  
    return a + b  
  
def subtract(a, b):  
    return a - b  
  
if __name__ == "__main__":  
    # Code block executed when the module is run as a script  
    result = add(5, 3)  
    print("Result:", result)
```

In the above example, the module defines two functions, `add` and `subtract`. The `if __name__ == "__main__"` condition is used to determine if the module is being directly executed. If it is, the code block within the condition will be executed.

To run the module as a script, you can execute the following command in the terminal or command prompt:

**`python module_name.py`**

Replace `module_name` with the actual name of your module file. This will execute the code within the `if __name__ == "__main__"` condition.

When the module is imported and used by another Python script, the code within the `if __name__ == "__main__"` block will not be executed. This allows the module to be imported and its functions to be used without any interference from the script-execution-specific code.

By using the `if __name__ == "__main__"` construct, you can execute specific code when running a module as a script while still enabling it to be imported and utilized as a module in other scripts.

#### 2.4.1.7 The Module Search Path

The module search path in Python is a collection of directories that the Python interpreter examines when attempting to import modules in a program. The module search path is determined by the `sys.path` variable, which is a list of directory locations.

When importing a module, Python follows a specific order to search for the module in different locations. The search path is checked in the following sequence:

1. **Current Directory:** Python first checks the directory where the script or interactive interpreter is executed. This allows for importing modules from the same directory as the script.
2. **PYTHONPATH environment variable:** Python examines the directories specified in the PYTHONPATH environment variable. This variable stores a list of directories, separated by colons (:) on Unix-like systems and semicolons (;) on Windows.
3. **Default module directory:** Python checks the standard library directories that are part of the Python installation. These directories contain built-in modules and other standard library modules.
4. **Third-party module directories:** If the module is not found in the previous locations, Python searches in directories typically used for installing third-party modules. These directories are commonly determined by package managers, such as site-packages or dist-packages.

The module search path can be modified programmatically by adding or modifying entries in the `sys.path` list. This can be helpful when you need to include additional directories for module searching during runtime.

To examine the current module search path, you can access the `sys.path` variable:

```
import sys
print(sys.path)
```

This will display a list of directories constituting the module search path. `dir()` function

In Python, the `dir()` function is a powerful tool for inspecting the contents of a module and retrieving a list of names, attributes, and methods defined within it. By calling `dir(module_name)`, you can explore the specific names associated with that module.

Here's an example demonstrating the usage of `dir()` on a module:

```
# Import a module

import my_module

# Display names, attributes, and methods of the module

print(dir(my_module))
```

In the above code, we import the `my_module` module and then use the `dir()` function to retrieve a list of names associated with it. The output will include functions, variables, classes, and other objects defined within the `my_module`.

Using `dir()` on a module provides valuable insights into the available functionality and objects within the module. It helps in understanding what the module offers and allows

you to utilize its attributes and methods effectively.

Keep in mind that the `dir()` function provides only the names defined within the module, without providing detailed explanations or documentation. For more information about a specific attribute or method, you can use the `help()` function, passing the module and the name as arguments (e.g., `help(my_module.some_function)`).

The `dir()` function helps explore a module's contents, allowing you to harness its capabilities for building robust and efficient Python programs.

#### 2.4.1.8 Built-in Modules

Python provides an extensive range of built-in modules that offer various functionalities to developers. These modules are part of the Python standard library and come pre-installed, eliminating the need for manual installation. Some commonly used built-in modules in Python include *math* for mathematical operations, *random* for random number generation and selection, *datetime* for manipulating and formatting dates and times, *os* for performing operating system-related tasks, *sys* for system-specific operations, *re* for working with regular expressions, *json* for JSON manipulation, *csv* for handling CSV files, *urllib* for working with URLs and HTTP operations, and *sqlite3* for interacting with SQLite databases. These built-in modules offer a wide range of functionalities and simplify common programming tasks, providing developers with efficient and effective tools for their Python programs.

#### 2.4.2 Packages

Python packages provide a way to structure and distribute code efficiently. Essentially, a package is a directory containing one or more Python modules, along with an optional special file named `__init__.py`. It facilitates the grouping of related modules thereby establishing a hierarchical organization for your code.

The primary purpose of packages is to enable code organization and reuse in a modular and scalable manner. By organizing modules into packages, you can prevent naming conflicts, enhance code maintainability, and improve code readability. Moreover, packages can be shared with others, fostering code collaboration and reuse.

To create a package, you create a directory with a unique name and include the `__init__.py` file within it. The `__init__.py` file can be empty or include initialization code that runs when the package is imported.

Packages can have sub-packages, which are essentially nested directories containing their own `__init__.py` files. This nesting capability allows for a hierarchical arrangement of packages, facilitating the organization of code at different levels of abstraction.

Package installation and management can be handled using package managers like `pip`, which is the predominant package manager in the Python ecosystem. Utilizing `pip`, you can effortlessly install, upgrade, and uninstall packages from the Python Package Index (PyPI), a community-maintained repository of Python packages.

Once a package is installed, its modules can be imported and utilized in other Python scripts through the `import` statement. This statement grants access to functions, classes, and variables defined within the package's modules.

### 2.4.2.1 Package Initialization

Package initialization in Python refers to the process of preparing a package for use when it is imported. It involves executing the code within the `__init__.py` file located in the package directory.

The `__init__.py` file serves as an indicator that the directory is a Python package. It can contain Python code that is executed during package import. This initialization code typically handles tasks like importing specific modules, setting up package-level variables, or performing any necessary initialization logic.

**Common scenarios for package initialization include:**

**Importing Modules:** The `__init__.py` file can include import statements to bring in modules within the package. This simplifies access to the package's modules and their contents when the package is imported.

**Setting Package-Level Variables:** Initialization code can define variables that are accessible at the package level. These variables can be shared among the package's modules or used for configuration purposes.

**Executing Initialization Logic:** The `__init__.py` file can contain code that carries out initialization steps required by the package. This may involve tasks such as establishing database connections, configuring logging, or registering components.

While the `__init__.py` file is optional, it provides a way to customize package behavior during import and serves as a central location for package initialization.

When a package is imported, the Python interpreter automatically executes the code within the `__init__.py` file, if present. This initialization code runs only once, regardless of how many times the package is imported in a program.

By leveraging package initialization, you ensure that your package is properly set up and ready for use upon import. This simplifies package organization and allows for better control over the package's behavior and functionality.

**Example:**

Suppose you have a package named "my\_package" with the following directory structure: my\_package/

```
__init__.py
module1.py
module2.py
```

**To initialize the package, follow these steps:**

**Create the `__init__.py` file:** Inside the "my\_package" directory, create a file named `__init__.py`. This file can be left empty or include initialization code.

**Define module files:** Within the "my\_package" directory, create two Python module files, `module1.py` and `module2.py`. Each module will contain functions or variables

related to specific functionalities.

Here's an example of how you can initialize the package:

init.py:

```
print("Initializing my_package...")
# Import modules within the package
from . import module1
from . import module2
module1.py:
```

```
def function1():
print("This is function 1 from module 1")
```

```
module2.py:
def function2():
print("This is function 2 from module 2")
```

In the `__init__.py` file, we print a message to indicate that the package is being initialized. We also import the modules `module1` and `module2` using relative imports (`from . import ...`).

Now, let's use the package in another Python script:

main.py:

```
import my_package
print("Package imported!")
my_package.module1.function1()
my_package.module2.function2()
```

When you run the `main.py` script, the output will be:

Initializing my\_package...

Package imported!

This is function 1 from module 1

This is function 2 from module 2

In this example, when the `my_package` package is imported, the `__init__.py` file is executed, and the initialization code within it is run. It prints the initialization message and imports the `module1` and `module2` modules.

You can then access the functions defined in the modules using the package name and module name as demonstrated in `main.py`.

This example demonstrates the process of package initialization in Python, where the `__init__.py` file is crucial for setting up the package during import. Remember to modify the package and module names and customize the functionality based on your specific requirements.

#### 2.4.2.2 Subpackages

Subpackages in Python provide a means of structuring and organizing code within packages in a hierarchical manner. They facilitate improved modularity, organization, and reusability of related components or functionality. Subpackages enable the creation of complex projects by establishing a multi-level package structure.

To create a subpackage, you can follow these steps:

Begin by creating the main package directory, which serves as the parent directory for the subpackage. This directory should include an `init.py` file that can either be left empty or contain initialization code specific to the package.

Inside the main package directory, create a subdirectory with a unique name that will function as the subpackage. This subdirectory should also have an `init.py` file, which can be empty or contain initialization code specific to the subpackage.

Include one or more module files (Python files) within the subpackage directory to hold the code relevant to the subpackage. These modules can consist of functions, classes, or other code elements.

By utilizing subpackages, developers can enhance code maintainability, separation of concerns, and code reuse. Subpackages allow for the logical grouping of related modules, facilitating easier navigation and utilization of specific functionality within a project.

#### Example:

Suppose you are working on a project related to geometry calculations and want to organize your code into subpackages. You can create a main package called "geometry" and include subpackages such as "shapes" and "utils".

The directory structure would look like this:

```
geometry/  
  __init__.py  
  shapes/  
    __init__.py  
    circle.py  
    rectangle.py  
  utils/  
    __init__.py
```



calculations.py

In this example, the "geometry" package serves as the main package. It contains two subpackages, "shapes" and "utils", represented by separate directories. Each subpackage has its own `__init__.py` file, indicating that they are Python subpackages.

The "shapes" subpackage includes two module files: "circle.py" and "rectangle.py". These files can contain classes and functions related to calculations and properties of circles and rectangles.

The "utils" subpackage consists of one module file: "calculations.py". This file can contain utility functions or calculations that are commonly used in geometry operations.

To import and use modules from the subpackages, you can use the dot notation: `from geometry.shapes.circle import Circle` from `geometry.utils.calculations`

```
import calculate_area

circle = Circle(radius = 5)

area = calculate_area(circle)

print(f"The area of the circle is: {area}")
```

Here, we import the Circle class from the "shapes.circle" subpackage and the calculate\_area function from the "utils.calculations" subpackage. This allows us to create a circle object and calculate its area using the imported functionality.

## Recap

- ◆ **Module** – A module is a .py file containing reusable Python code like functions and variables.
- ◆ **Creating a Module** – You create a module by saving Python code in a file with a .py extension.
- ◆ **Importing a Module** – Use the import statement to access functions and variables from a module.
- ◆ **Naming a Module** – Module names should be short, lowercase, descriptive, and avoid conflicts.
- ◆ **Renaming a Module** – Rename the file and update all related import statements and references.
- ◆ **Variables in a Module** – Variables defined in a module can be accessed from other scripts that import it.
- ◆ **Executing a Module as Script** – The `if __name__ == "__main__"` block lets you run code only when the module is executed directly.

- ◆ **Module Search Path** – Python searches for modules in a specific order defined in the sys.path list.
- ◆ **Built-in Modules** – Python includes many ready-to-use built-in modules like math, os, and json.
- ◆ **Package** – A package is a folder of related modules with an `__init__.py` file to make it importable.
- ◆ **Package Initialization** – Code in `__init__.py` runs when the package is imported and can initialize the package.
- ◆ **Subpackage** – A subpackage is a package within another package, helping organize complex projects.
- ◆ **Using Subpackages** – You can import specific parts of subpackages using dot notation like `from package.sub.module import item`.

## Objective Type Questions

1. What is a module?
2. What is a package?
3. What does `init.py` file do?
4. How do you import a module?
5. What keyword is used to import specific items from a module?
6. What is a subpackage?
7. What keyword is used to create a package?
8. What is the purpose of a package?
9. What keyword is used to import all items from a module?
10. What is the main benefit of using modules and packages?

## Answers to Objective Type Questions

1. File
2. Directory

3. Initialization
4. import
5. from
6. Nested
7. init.py
8. Organization
9. \*
10. Reusability

## Assignments

1. Create a module that contains functions for calculating the area and circumference of a circle, and import it to calculate these values for user-provided input.
2. Design a package with subpackages representing different categories of animals, each containing modules with functions to display information about specific animals, and import them to display details based on user input.
3. Create a module that includes a function to generate a random password, and import it to generate and display a password with a user-defined length.
4. Develop a package with subpackages for basic mathematical operations (addition, subtraction, etc.) and import the appropriate subpackage and module to perform calculations based on user input.

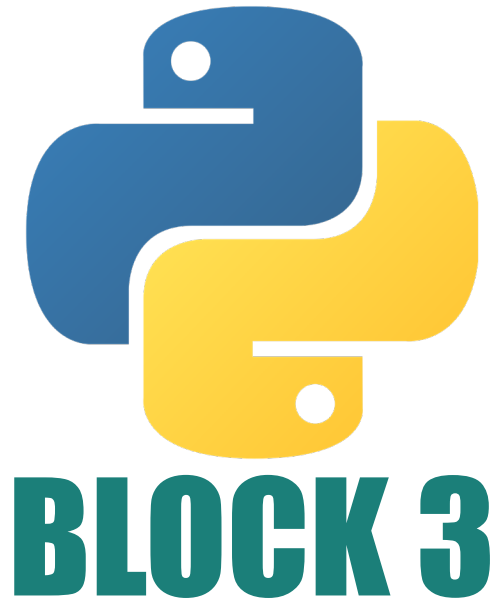
## References

1. Python Software Foundation. (2023). *The Python Standard Library*.
2. Beazley, D. M., & Jones, B. K. (2013). *Python Cookbook* (3rd ed.). O'Reilly Media.
3. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

4. Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.
5. Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.

## Suggested Reading

1. Matthes, E. (2019). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. No Starch Press.
2. Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media.
3. <https://docs.python.org/3/tutorial/modules.html>



# **File Handling, Object-Oriented Programming, Exception Handling and Regular Expressions**



# File Handling

## Learning Outcomes

After completing this unit, learners will be able to:

- ◆ define file handling in Python and its importance in managing data.
- ◆ describe the various file modes used to open files in Python.
- ◆ explain how the with statement is used for safe file operations.
- ◆ describe the purpose of cursor handling functions like seek() and tell().

## Prerequisites

Think about tracking your daily spending in a notebook. At first, writing down each expense works fine. But as time goes on, the notebook may become cluttered, difficult to navigate, or even damaged, making it hard to review or total your expenses. Now, consider using Python to manage this task more efficiently. With file handling, you can create a file on your computer to record your spending, add new entries each day, easily view past records, make updates, or delete unnecessary data. This method keeps your information organized and safe, and it simplifies tasks like calculations or summaries. Essentially, Python's file handling serves as a reliable digital notebook for managing and storing important data.

## Keywords

open(), read(), write(), tell(), Truncation, Cursor Positioning

## Discussion

### 3.1.1 Introduction to File Handling

Python also provides file handling and enables users to read and write files as well as perform a variety of other operations on files. Like many other concepts in Python, the idea of file management has been extended to a number of other languages, but their implementations are either difficult or time-consuming. Python handles text and binary files differently, and this is crucial. A text file is created by the character sequences in each line of code. A unique character known as the EOL or End of Line character, such as the comma (,) or newline character, is used to end each line of a file. It signals the interpreter that a new line has started and ends the current one. The reading and writing files will come first.

### 3.1.2 File Handling Functions

File handling functions in Python are built-in functions that enable you to carry out various file operations like creating, opening, reading, writing, appending, and closing files. These functions are essential in accessing and manipulating data stored in external text or binary files, making file input/output (I/O) tasks more efficient and manageable.

#### 3.1.2.1 open() function

We must first open the file before we can read from it or write to it. To do this, we should use the built-in Python function open(), but we must first give the mode, which denotes the goal of the opening file.

```
f = open(filename, mode)
```

r : Read an existing file by opening it.

w : initiates a write operation on an open file. The data will be replaced if it already exists in the file.

a : Start an add operation on an existing file. Existing data won't be replaced by it.

r+ : To both read from and write to the file. It will replace any prior data in the file.

w+ : Opens the file for both reading and writing. If the file already exists, its content will be overwritten and if it does not exist, it will create a new file.

a+ : To add to and read from the file with data. Existing data won't be replaced by it.

```
file = open('myfile.txt', 'r') # This will print each line one by one in the file
```

```
for i in file:
```

```
    print (i)
```

The file will be opened in read-only mode by the open command, and every line in the file will be printed by the for loop.

### 3.1.2.2 read() mode

In Python, there are various methods for reading files. Use `file.read()` to extract a string containing every character in the file if necessary (). The complete code would operate as follows:

```
file = open("myfile.txt", "r")  
  
print (file.read())
```

The following code will cause the interpreter to read the first five characters of any stored data and return them as a string, which is another way to read a file:

```
file = open("myfile.txt", "r")  
  
print (file.read(5))
```

### 3.1.2.3 Creating a file using write() mode

```
file = open("myfile.txt", "w")  
  
file.write("Writing new content")  
  
file.write("It allows us to write in a particular file")  
  
file.close()
```

`Close()` terminates all resources that are currently in use and releases this specific program from the system.

```
file = open('myfile.txt','a')  
  
file.write("This will add this line")  
  
file.close()
```

The following additional commands are used in file handling to accomplish different tasks:

`rstrip()`: This function removes all right-side spaces from each line of a file.

`lstrip()`: This function removes any left-side spaces from each line of a file.

### 3.1.2.4 Renaming a File

We will use the `rename()` method from the `os` module to rename our file. Two arguments are required by the `rename()` method:

1. The string type of the current file name.
2. The newly renamed file's name, which must be supplied as a string type.

```
import os  
  
os.rename('myfile.txt', 'outfile.txt')
```

### 3.1.2.5 Delete a File

You must import the OS module and call its `os.remove()` function in order to delete a file:

```
import os  
  
os.remove("myfile.txt")
```

#### ◆ Check if File exist:

Before attempting to delete a file, you might want to verify if it already exists to avoid receiving an error:

```
import os  
  
if os.path.exists("myfile.txt"):  
    os.remove("myfile.txt")  
  
else:  
    print("The file does not exist")
```

#### ◆ Delete Folder

The `os.rmdir()` function can be used to completely remove a folder:

```
import os  
  
os.rmdir("myfolder")
```

## 3.1.3 Cursor Positioning Methods

Cursor positioning methods are techniques or functions used in programming, text editing, or terminal applications to shift the cursor to a desired location on the screen or within a file.

### 3.1.3.1 seek() method

In Python, you can move the cursor to a specific location using the `seek()` function.

```
file = open("myfile.txt", 'r')  
  
file.seek(0)  
  
file.close()
```

The cursor will then be moved to index position 0, and file reading will once more begin at the beginning.

### 3.1.3.2 The tell() method

The `tell()` method in Python prints the current position of our cursor.

```
file = open("myfile.txt", 'r') # Open the file in read mode
```

```
file.tell()      # Initially, the file pointer is at the start of the file, This will print 0
file.read(5)     # Read the first 5 characters
print(file.tell()) # Now the file pointer has moved 5 bytes forward, This will print 5
file.read(6)     # Read another 6 characters (total 11 characters read so far)
print(file.tell()) # The file pointer has now moved 6 more bytes forward (total of 11
bytes), This will print 11.
```

### 3.1.4 Truncating a File

In Python, it is also possible to truncate a file. We can truncate the file to the required length by using the `truncate()` method.

```
file = open('myfile.txt', 'w')
file.truncate(20)
```

## Recap

- ◆ Python provides built-in support for file handling to perform operations like creating, opening, reading, writing, appending, and closing files.
- ◆ Text and binary files are handled differently in Python.
- ◆ The `open()` function is used to open a file and requires a filename and mode ('r', 'w', 'a', 'r+', 'w+', 'a+').
- ◆ The `read()` method is used to read the contents of a file.
- ◆ The `write()` method allows writing data to a file; it replaces existing content if in 'w' mode.
- ◆ The `append()` mode adds new content to the end of the file without removing existing data.
- ◆ The `close()` method should be used to release system resources after file operations.
- ◆ The `os.rename()` function renames a file.
- ◆ The `os.remove()` function deletes a file, and `os.rmdir()` removes a folder.
- ◆ `os.path.exists()` checks if a file exists before performing actions on it.
- ◆ The `seek()` method moves the file cursor to a specified location.
- ◆ The `tell()` method returns the current position of the file cursor.
- ◆ The `truncate()` method is used to reduce the file size to a given number of bytes.

## Objective Type Questions

1. Which function is used to open a file in Python?
2. What mode is used to read a file?
3. Which mode is used to write to a file and overwrite existing content?
4. Which method moves the cursor to a specific position in a file?
5. Which method returns the current position of the cursor in a file?
6. Which function is used to close a file?
7. Which method removes all right-side spaces from a line?
8. What method is used to delete a file?
9. Which method is used to reduce the size of a file?
10. What mode is used for both reading and writing, overwriting content if the file exists?

## Answers to Objective Type Questions

1. open
2. r
3. w
4. seek
5. tell
6. close
7. rstrip
8. remove
9. truncate
10. w+

## Assignments

1. Write a Python program to create a file, write some content into it, and then read and display the content.
2. Explain the different file access modes in Python with suitable examples for each.
3. Write a Python script to append a new line of text to an existing file and display the updated content.
4. Using the os module, write a Python program to rename and then delete a file, ensuring the file exists before each operation.
5. Demonstrate the use of seek() and tell() methods in a Python program to manipulate and track the cursor position within a file

## References

1. Matthes, E. (2019). Python Crash Course: A Hands-On, Project-Based Introduction to Programming. No Starch Press.
2. Ramalho, L. (2015). Fluent Python: Clear, Concise, and Effective Programming. O'Reilly Media.
3. Beazley, D., & Jones, B. K. (2013). Python Cookbook: Recipes for Mastering Python 3. O'Reilly Media.
4. <https://docs.python.org/3/tutorial/modules.html>

## Suggested Reading

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
2. Zelle, J. M. (2016). *Python programming: An introduction to computer science* (3rd ed.). Franklin, Beedle & Associates Inc.
3. Sweigart, A. (2019). *Automate the boring stuff with Python: Practical programming for total beginners* (2nd ed.). No Starch Press.
4. Beazley, D. M., & Jones, B. K. (2013). *Python cookbook: Recipes for mastering Python 3* (3rd ed.). O'Reilly Media.
5. Downey, A. (2015). *Think Python: How to think like a computer scientist* (2nd ed.). O'Reilly Media.



# Object-Oriented Programming

## Learning Outcomes

At the end of this unit, the learner will be able to:

- ♦ define object-oriented programming (OOP) in the context of Python.
- ♦ describe the main OOP principles (class, object, inheritance, polymorphism, encapsulation, and abstraction).
- ♦ identify the role of the `__init__` method and `self`-keyword in class construction.
- ♦ recognise the different types of inheritance used in Python with examples.
- ♦ recall the syntax for creating a class and instantiating objects in Python.

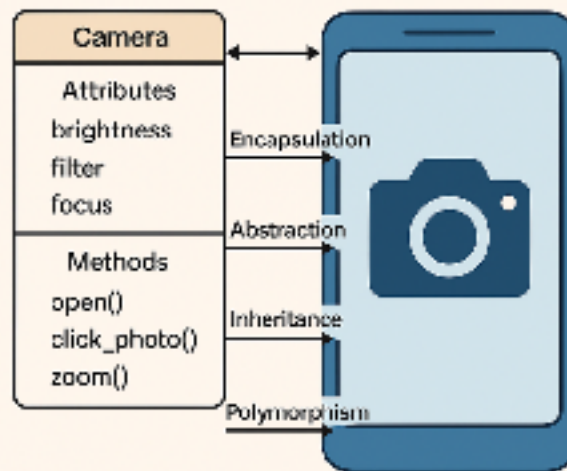
## Prerequisites

Have you ever thought about how your favorite apps like WhatsApp or a ride-booking service like Uber work behind the scenes? These apps are made of different parts that work together just like real-world objects. Each user, message, or vehicle is treated as a separate "thing" with properties and actions. This is the core idea behind **Object-Oriented Programming (OOP)** organizing code around real-world entities. Before diving into OOP, it's helpful to understand basic Python concepts like variables, data types, functions, and control structures such as loops and conditionals.

Now, imagine a smartphone. It has apps (objects), each with specific functions. The Camera app can open, click photos, and zoom. These functions are like methods, and the app itself holds data like brightness, filter, and focus called attributes. In Python, we model such real-world items using classes (blueprints) and objects (actual instances). This structure helps developers manage complexity by grouping related behavior and data together in a meaningful way.

With the help of OOP concepts, you can write code that is not only cleaner but also reusable and flexible. Features like encapsulation keep data safe, abstraction hides unnecessary details, inheritance lets you reuse code, and polymorphism allows one interface to work in multiple ways. Understanding these concepts allows you to build powerful and realistic applications in Python, just like the ones you use daily. This unit will open your mind to a more logical, scalable way of programming and once you start thinking in objects, programming becomes both easier and more exciting.

## Object-Oriented Programming Concepts



## Keywords

Class, Objects, Polymorphism, Encapsulation, Inheritance, Data Abstraction

## Discussion

### 3.2.1 Introduction to Object-Oriented Programming in Python

Object-Oriented Programming (OOP) is a powerful programming paradigm that helps organize code by modeling real-world entities as software objects. Unlike procedural programming, where code is written as a sequence of steps, OOP structures programs around **classes** (blueprints) and **objects** (instances of those classes). This approach makes code more modular, reusable, and easier to maintain. Python, as a high-level and beginner-friendly language, fully supports OOP and provides simple syntax to define and use classes and objects effectively.

In Python's OOP, key concepts such as **encapsulation**, **abstraction**, **inheritance**, and **polymorphism** play a vital role. These concepts allow developers to hide internal details, simplify interfaces, reuse existing code, and design flexible applications. For example, a class called `Car` can represent vehicles with shared properties like color and speed, while different types of cars (like `ElectricCar` or `SportsCar`) can inherit from it and extend or modify behaviors. Learning OOP in Python provides a strong foundation for building scalable software and real-world applications.

### 3.2.2 Concepts of Object-Oriented Programming (OOPs)

Object-Oriented Programming (OOP) in Python is a method of structuring code that

models real-world entities using **classes** and **objects**. Python makes it easy to implement OOP due to its simple and readable syntax. By organizing code into reusable and self-contained objects, Python's OOP approach helps developers write clear, efficient, and maintainable programs. These concepts are especially useful in building complex applications like games, web apps, or data processing systems by breaking them down into smaller, manageable components. The core OOP concepts in Python (Fig 3.2.1) are:

- ◆ **Class:** A template that defines the structure and behavior of objects.
- ◆ **Object:** An instance of a class with specific data and functionality.
- ◆ **Encapsulation:** Bundling data and methods together and restricting direct access.
- ◆ **Abstraction:** Hiding unnecessary details and exposing only the essential features.
- ◆ **Inheritance:** Creating new classes from existing ones to promote code reuse.
- ◆ **Polymorphism:** Allowing different objects to respond uniquely to the same method call.

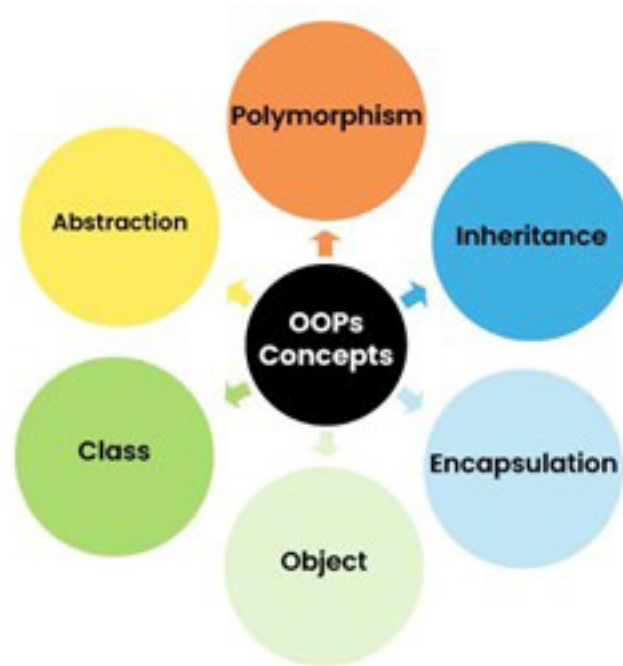


Fig 3.2.1 Core OOP Concepts in Python

### 3.2.3 Class

A class is a group of related items. The models or prototypes used to generate objects are included in classes. It is a logical entity with a few methods and characteristics. Consider the following scenario to better appreciate the need for generating classes. Suppose you needed to keep track of the number of dogs that might have various characteristics, such as breed or age. If a list is utilized, the dog's breed and age might

be the first and second elements, respectively. What if there were 100 different breeds of dogs? How would you know which ingredient should go where? What if you wanted to give these dogs additional traits? This is unorganized and just what classes need.

A class is a collection of objects. i.e., Classes are blueprints for creating objects. A class defines a set of attributes and methods that the created objects (instances) can have. Some key points on Python class:

- ◆ The keyword **class** is used to create classes.
- ◆ The variables that make up a class are known as attributes.
- ◆ With the dot (.) operator, attributes can always be retrieved and are always public. (For example: **Myclass.Myattribute**)

### Class definition Syntax:

```
class ClassName:
```

```
    # Statement-1
```

```
    .
```

```
    .
```

```
    .
```

```
    # Statement-N
```

**Example:** Creating an empty Class in Python

```
class Dog:
```

```
    .....
```

Using the **class** keyword, we built a class with the name Dog in the example above.

### 3.2.4 Objects

The object is an entity that is connected to a state and activity. Any physical device, such as a mouse, keyboard, chair, table, pen, etc., may be used. Arrays, dictionaries, strings, floating-point numbers, and even integers are all examples of objects. Any single string or integer, more specifically, is an object. A list is an object that may house other things, the number 12 is an object, the text "Hello, world" is an object, and so on. You may not even be aware of the fact that you have been using items.

An **object** is a real-world entity created from a class. It contains data (attributes) and functions (methods) defined by the class. i.e., an object is an instance of a class. An object includes:

- ◆ **State:** An object's properties serve as a representation of it. Additionally, it reflects an object's characteristics.
- ◆ **Behavior:** The methods of an object serve as a representation of behavior. It



also shows how one object reacts to other objects.

- ◆ **Identity:** It gives a thing a special name and makes it possible for one object to communicate with another.

Let's look at the example of the class dog to better understand the state, behavior, and identity. The identity may be regarded as the dog's name. Breed, age, and color of the dog are examples of states or attributes. You may infer from the behavior whether the dog is eating or sleeping.

**Syntax:** object\_name = ClassName(arguments)

**Example:** Creating an object

```
d = Dog()
```

This will create an object with the class Dog, named “d”, as stated above. Let's first grasp the fundamental terms that will be utilized while working with objects and classes before delving further into them.

## 1. Using Self

- ◆ An additional initial parameter in the method declaration is required for class methods. When we call the method, we don't supply a value for this parameter; Python does.
- ◆ Even if we have a method that doesn't require any parameters, we still need one.
- ◆ This is comparable to this Java reference and this C++ pointer.

This is the primary purpose of the special self. When we invoke a method of this object as myobject.method(arg1, arg2), Python automatically converts it to MyClass.method(myobject, arg1, arg2).

## 2. \_\_init\_\_ method

The **\_\_init\_\_** method is similar to constructors in C++ and Java. As soon as a class object is created, it is executed. Any initialization you want to perform on your object can be done with the method.

**Example: Creating a class and object with class and instance attributes**

class Dog:

```
    attr1 = "mammal"
```

*# class attribute*

```
    def __init__(self, name):
```

*# instance attribute*

```
        self.name = name
```

```
Rodger = Dog("Rodger")
```

*# object instantiation*

```
Tommy = Dog("Tommy")
```

```

print("Rodger is a ", Rodger.__class__.attr1)          # Accessing class attributes
print("Tommy is also a ", Tommy.__class__.attr1)
print("My name is ", Rodger.name)                    # Accessing instance attributes
print("My name is ", Tommy.name)

```

**Output:**

```

Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy

```

**Example: Creating Class and objects with methods**

```

class Dog:
    attr1 = "mammal"                                # Class attribute
    def __init__(self, name):                        # Instance attribute
        self.name = name
    def speak(self):
        print("My name is", self.name)

Rodger = Dog("Rodger")    # Object instantiation (outside the class definition)
Tommy = Dog("Tommy")

Rodger.speak()            # Accessing class methods
Tommy.speak()

```

**Output:**

```

My name is Rodger
My name is Tommy

```

### 3.2.5 Inheritance

The ability of one class to derive or inherit properties from another class is known as inheritance. The class from which the properties are being derived is referred to as the **base class** or **parent class**, and the class that inherits those properties is referred to as the **derived class** or **child class**.

**Why Use It?**

- ◆ To reuse code.
- ◆ To organize similar classes better.

- ◆ To avoid writing duplicate functions.

**The advantages of inheritance include:**

- ◆ It accurately depicts relationships in the real world.
- ◆ It offers a code's reusability. We don't need to keep writing the same code. Additionally, it enables us to expand a class's features without changing it.
- ◆ Because of its transitive nature, if a class B inherits from a class A, then all of class B's subclasses will also automatically inherit from class A.

**Real-life example:**

A **Car** and a **Truck** are both types of **Vehicles**. They can **share common features** like `start()`, `stop()`.

### 3.2.5.1 Types of Inheritance

Inheritance is a powerful feature in Python's object-oriented programming that allows a class (called the child or subclass) to reuse the properties and behaviors of another class (called the parent or superclass). This promotes code reuse, reduces redundancy, and helps in building a clear and logical class hierarchy.

Python supports several types of inheritance, each serving a different purpose depending on the relationship between classes. The main types (Fig 3.2.2) include:

- a. Single Inheritance
- b. Multiple Inheritance
- c. Multilevel Inheritance
- d. Hierarchical Inheritance
- e. Hybrid Inheritance

Each type allows you to build more flexible and modular programs by organizing your code using real-world relationships.

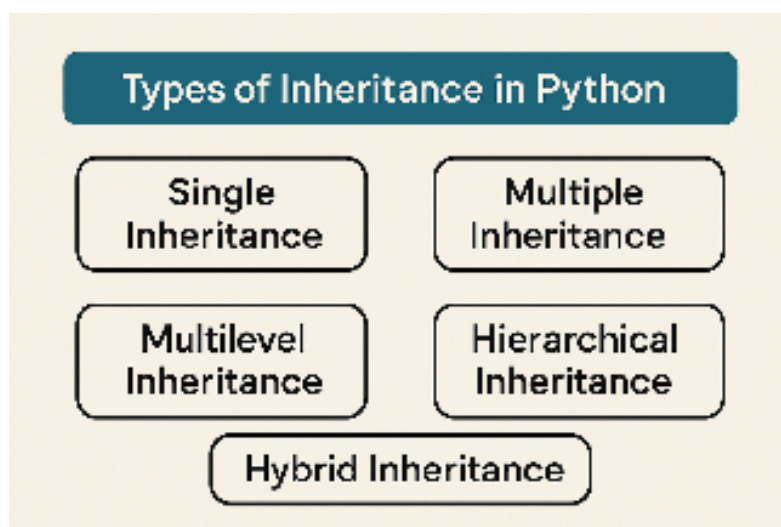


Fig 3.2.2 Types of Inheritance in Python

## 1. Single Inheritance

**Single inheritance** is the most basic type of inheritance in Python. It occurs when a **child class** inherits from **one parent class**. This allows the child class to reuse the attributes and methods of the parent class, reducing code duplication and promoting reusability. Single inheritance is helpful when there is a clear and straightforward relationship between two classes.

### *Example:*

```
class Animal:
    def speak(self):
print("Animal speaks")
    class Dog(Animal):
        def bark(self):
            print("Dog barks")
d = Dog()
d.speak()
d.bark()
```

### *Output:*

```
Animal speaks
Dog barks
```

## 2. Multilevel Inheritance

Multilevel inheritance in Python refers to a situation where a class is derived from another class, which is itself derived from a third class. In other words, the inheritance chain goes on for multiple levels. This type of inheritance helps build a hierarchical relationship where each level can add new features or extend the existing ones from the class it inherits. It allows better structure and deeper reuse of code across related classes. i.e., A derived class can inherit properties from an immediate parent class, which in turn can inherit properties from its parent class, thanks to multi-level inheritance.

### **Example :**

```
class Animal:
    def speak(self):
print("Animal speaks")
class Dog(Animal):
    def bark(self):
```

```
print("Dog barks")

class Puppy(Dog):
    def weep(self):
print("Puppy weeps")

p = Puppy()
p.speak()
p.bark()
p.weep()
```

***Output:***

Animal speaks

Dog barks

Puppy weeps

### 3. Hierarchical Inheritance

**Hierarchical inheritance** occurs when **multiple child classes inherit from a single parent class**. This means one base class is shared by many derived classes. Each child class gets access to the attributes and methods of the same parent, allowing code reuse across multiple subclasses. Hierarchical inheritance is useful when different types of objects share common behavior but also have their own unique features.

**Example:**

```
class Animal:
    def speak(self):
print("Animal speaks")

class Dog(Animal):
    def bark(self):
print("Dog barks")

class Cat(Animal):
    def meow(self):
print("Cat meows")

d = Dog()
c = Cat()
d.speak()
```

```
d.bark()
c.speak()
c.meow()
```

***Output:***

```
Animal speaks
Dog barks
Animal speaks
Cat meows
```

#### **4. Multiple Inheritance**

**Multiple inheritance** is a type of inheritance in Python where a **child class inherits from more than one parent class**. This means the child class can access the features (attributes and methods) of all its parent classes. Multiple inheritance is powerful for combining functionalities from different sources, but it can also introduce complexity, especially when parent classes have methods with the same name.

***Example:***

***# Parent class 1***

```
class Father:
    def skills(self):
print("Father: Gardening, Carpentry")
```

***# Parent class 2***

```
class Mother:
    def skills(self):
print("Mother: Cooking, Painting")
```

***# Child class inherits from both Father and Mother***

```
class Child(Father, Mother):
    def skills(self):
print("Child inherits skills from both parents:")
Father.skills(self)  # Calling Father's version
Mother.skills(self)  # Calling Mother's version
print("Child: Coding")
```

***# Create object of Child***



```
c = Child()
```

```
c.skills()
```

**Output:**

Child inherits skills from both parents:

Father: Gardening, Carpentry

Mother: Cooking, Painting

Child: Coding

### 3.2.6 Polymorphism

Polymorphism is an important concept in object-oriented programming that means “many forms.” In Python, polymorphism allows objects of different classes to be treated as if they are of the same class, especially when they share a common interface or method name. This makes the code more flexible and reusable, as the same function or method can work with different types of objects. Polymorphism helps in designing systems where components can be easily extended or replaced without changing existing code.

#### Why Use It?

- ◆ To write flexible code.
- ◆ So different objects can be used interchangeably even if they behave differently.

#### Real-life example:

The word "run" means:

- ◆ A human can run.
- ◆ A computer program can run.
- ◆ A car engine can run.

They all use the word “run” but **act differently!**

#### Example:

```
class Bird:
```

```
    def fly(self):
```

```
print("Birds can fly")
```

```
class Ostrich(Bird):
```

```
    def fly(self):
```

```
print("Ostrich cannot fly")
```

```
b = Bird()
```

```
o = Ostrich()

b.fly() # Bird's version

o.fly() # Ostrich's version
```

### Output:

```
Birds can fly

Ostrich cannot fly
```

### 3.2.7 Encapsulation

One of the core ideas in object-oriented programming is encapsulation (OOP). It explains the concept of data wrapping and the techniques that operate on data as a single unit. This restricts direct access to variables and procedures and can avoid data alteration by accident. A variable can only be altered by an object's method in order to prevent inadvertent modification. These variables fall under the category of private variables. A class, which encapsulates all the data that is contained in its member functions, variables, etc., is an example of encapsulation.

### Why Use It?

- ◆ To protect data from being changed by accident.
- ◆ To make code clean and secure.
- ◆ To control how data is accessed.

Consider a real-life example, think of an ATM machine. You press buttons to withdraw money, but you don't need to know how it calculates or verifies inside, that's encapsulation!

### Example:

```
class Person:

    def __init__(self):

self.__age = 0 # Private variable

    def set_age(self, age):

self.__age = age

    def get_age(self):

    return self.__age

p = Person()

p.set_age(25)

print(p.get_age()) # Correct way to access private data
```



**Output:**

25

`__age` is private. We **can't access it directly**, only through `set_age()` and `get_age()` methods.

### 3.5.8 Abstraction

In real life, we often interact with systems without knowing how they work internally. For example, when we drive a car, we use the steering wheel, brake, and accelerator but we don't need to know how the engine works. This idea is called **abstraction**, and it's used in Python programming to make complex systems easier to use. It hides unnecessary details and shows only the essential features, helping developers focus on **what an object does, not how it does it**.

Abstraction is the process of hiding the internal implementation details and showing only the required features of an object. In Python, abstraction is achieved using abstract classes and abstract methods, provided by the `abc` (Abstract Base Class) module. The main purpose of abstraction is:

- ◆ To reduce complexity and make code easier to manage.
- ◆ To hide internal implementation from the user.
- ◆ To focus on what an object does, not how it works.
- ◆ To provide a clear structure for creating reusable and extendable code.
- ◆ To improve security by hiding sensitive logic.

**Consider real-Life example, imagine using an ATM:**

- ◆ You insert your card, enter the PIN, and withdraw money.
- ◆ You don't need to know **how** the ATM checks your account or dispenses the cash.

This is abstraction, you use the features without seeing the background process.

**Example:**

```
from abc import ABC, abstractmethod
```

```
class Vehicle(ABC):          # Abstract class
```

```
    @abstractmethod
```

```
    def start_engine(self):
```

```
        pass
```

```
class Car(Vehicle):          # Subclass
```

```
    def start_engine(self):
```

```
print("Car engine started.")  
  
my_car = Car()                                # Using the class  
  
my_car.start_engine()
```

**Output:**

Car engine started.

Vehicle hides the engine starting process. Car gives the actual detail. The user only needs to call start\_engine() without knowing how it works internally.

## Recap

- ◆ OOP is a programming style that organizes code using classes and objects to model real-world entities.
- ◆ Class is a blueprint for creating objects with shared structure and behavior.
- ◆ Object is an instance of a class representing individual data and functionality.
- ◆ self refers to the current object and is used to access variables and methods within the class.
- ◆ \_\_init\_\_() is the constructor method automatically called when an object is created.
- ◆ Class attributes are shared across all instances, while instance attributes are unique to each object.
- ◆ Encapsulation hides the internal state of objects using private variables and methods.
- ◆ Abstraction is the process of hiding the internal implementation details and showing only the required features of an object.
- ◆ Inheritance allows one class (child) to acquire attributes and methods from another (parent).
- ◆ Polymorphism enables the same method to behave differently across different classes.
- ◆ Single Inheritance means a class inherits from one parent class.
- ◆ Multilevel Inheritance is when a class inherits from a child class that already has a parent.
- ◆ Hierarchical Inheritance means multiple classes inherit from a single base class.

- ◆ Multiple Inheritance lets a class inherit from two or more parent classes.
- ◆ Class methods operate on class-level data, while instance methods work with instance-specific data.
- ◆ OOP in Python makes code more reusable, organized, and easier to maintain.

## Objective Type Questions

1. What does OOP stand for?
2. Which keyword is used to define a class in Python?
3. What is an object in Python?
4. Which special method acts as a constructor in Python?
5. What does the self keyword represent in a class method?
6. What is inheritance in OOP?
7. What is polymorphism in Python?
8. Which module is used to define abstract classes in Python?
9. What is the purpose of abstraction in OOP?
10. What is encapsulation?
11. Which inheritance type allows a class to inherit from more than one parent?
12. Which of the following is not an OOP concept: Inheritance, Encapsulation, Compilation, Polymorphism?

## Answers to Objective Type Questions

1. Object-Oriented Programming
2. class
3. An instance of a class
4. `__init__()`

5. The current instance of the class (object)
6. The process where one class can use the properties and methods of another class
7. The ability to use the same method name in different classes with different behavior
8. abc
9. To hide the internal details and show only necessary features
10. Wrapping data and methods together in a single unit (class)
11. Multiple Inheritance
12. Compilation

## Assignments

1. What is Object-Oriented Programming (OOP)? How does Python implement OOP concepts? List and explain the major pillars of OOP with real-life examples.
2. Differentiate between a class and an object. What is the purpose of the self keyword in Python? How does it relate to this in other languages like Java or C++?
3. Explain the role of the `__init__()` method in Python classes. What are instance attributes and class attributes? How do you access them?
4. What is encapsulation? How can we implement private members in Python?
5. Define inheritance and explain the types of inheritance supported in Python.
6. What is polymorphism? How does Python support polymorphism through method overriding?
7. Explain data abstraction with an example. How does it differ from encapsulation?

## References

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
2. Downey, A. B. (2015). *Think Python: How to think like a computer scientist* (2nd ed.). O'Reilly Media.
3. Goldwasser, M. H., & Letscher, D. (2007). *Object-oriented programming in Python*. Prentice Hall.

## Suggested Reading

1. Matthes, E. (2019). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming* (2nd ed.). No Starch Press.
2. Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media.
3. Bader, D. (2017). *Python Tricks: A Buffet of Awesome Python Features*. Dan Bader Press.
4. Real Python - Functions: <https://realpython.com/tutorials/functions/>



# Exception Handling and Regular Expressions

## Learning Outcomes

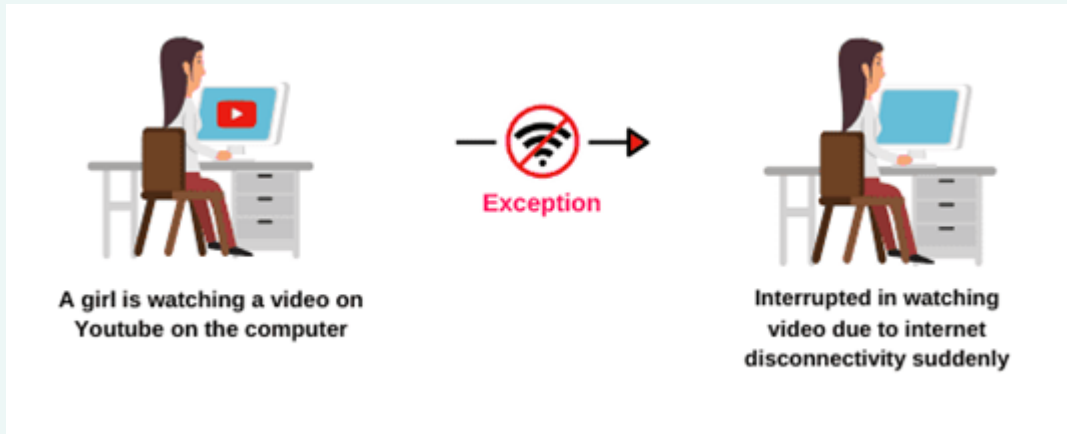
After completion of this unit, the learner will be able to:

- ◆ define exception and error in the context of Python.
- ◆ list common types of errors in Python.
- ◆ identify the purpose of exception handling in Python.
- ◆ recall the syntax for using try and except blocks.
- ◆ identify the keywords used in Python exception handling.

## Prerequisites

In your previous Python lessons, you have learned how to write programs using variables, conditional statements, loops, and functions. These tools help you create useful programs that take input, perform actions, and give output. However, sometimes unexpected situations can occur while a program is running such as dividing by zero or trying to open a file that doesn't exist. These situations cause errors that can stop the program from working properly. When an error happens, the program usually ends suddenly, which can be confusing or frustrating for users. To prevent this, Python provides a way to handle errors more safely and clearly using something called exception handling. Exception handling helps your program deal with errors without crashing. This makes your code more reliable and easier to understand, especially when working on real-world applications.

Suppose you are watching a video on Youtube, suddenly, internet connectivity is disconnected or not working. In this case, you are not able to continue watching the video on Youtube. This interruption is nothing but an exception. How to handle this exception efficiently is important. For further progress we need to overcome this. Exceptions are unexpected events or errors that occur during the execution of a program. They can be caused by various factors, such as invalid input, resource unavailability, or programming mistakes. Python provides a robust exception handling mechanism that allows you to catch and handle these exceptions gracefully, preventing your program from crashing.



In this topic, you will learn the difference between errors and exceptions, understand why exception handling is important, and explore how to use Python's try, except, and other related statements. These skills will help you write programs that can detect and respond to problems in a smart and controlled way, improving the quality and stability of your code.

## Keywords

Exception, Try, Catch, Except, Finally, Errors, Assert, Raise.

## Discussion

### 3.3.1 Introduction to Exception Handling Mechanism in Python

In programming, errors are common and often unavoidable, especially when working with user input, files, or external systems. These errors can stop a program from running and may lead to a poor user experience. To manage these situations effectively, Python provides a feature called exception handling, which allows programmers to detect and respond to errors in a safe and controlled way. Instead of letting the program crash, you can use exception handling to show a helpful message or take corrective action.

Python uses specific keywords like try, except, else, and finally to handle exceptions. With these tools, you can test a block of code for errors, catch and handle those errors, and even perform clean-up actions no matter what happens. This helps make your code more reliable, especially in real-world applications where unexpected problems are likely to occur. By learning exception handling, you will be able to write programs that are more robust, user-friendly, and professional.

### 3.3.2 Errors

Errors are problems in a program that cause it to stop running or behave unexpectedly. In Python, errors can be broadly categorized into two types:

## 1. Syntax Errors

## 2. Exceptions (Runtime Errors)

### 3.3.2.1 Syntax Errors

As the name implies, this error results from incorrect syntax in the code. It results in the program's termination. Syntax errors occur when the Python code is not written correctly according to the rules of the language. These are detected before the program runs (during the compilation or interpretation phase). Common causes are missing colons, unmatched parentheses, incorrect indentation.

#### Example:

```
x=33
if x > 10
print("x is greater than 10")
```

#### Output:

**Error:**SyntaxError:expected':'

### 3.3.2.2 Exceptions

Exceptions are errors that occur during the execution of a program. Unlike syntax errors, exceptions occur when the code is syntactically correct but fails during execution due to an unexpected condition. If not handled, exceptions can crash the program. Common types of built-in exceptions are shown in Table 3.3.1.

Table 3.3.1 Common types of Built-in Exceptions

Exception	Description
<b>ZeroDivisionError</b>	Raised when a number is divided by zero
<b>TypeError</b>	Raised when an operation is used on the wrong data type
<b>ValueError</b>	Raised when a function receives an argument of the right type but inappropriate value
<b>IndexError</b>	Raised when a list index is out of range
<b>KeyError</b>	Raised when a dictionary key is not found
<b>FileNotFoundError</b>	Raised when trying to access a file that does not exist
<b>ImportError</b>	Raised when Python cannot find a module or function to import

#### Example:

```
a = [1, 2, 3]
print(a[5])
```

#### Output:

**Error:**IndexError:list index out of range



### 3.3.3 Exception Handling

Python exception handling handles errors that occur during the execution of a program. Exception handling allows the user to respond to the error, instead of crashing the running program. It enables you to catch and manage errors, making your code more robust and user-friendly. Exception handling in Python is done using the **try**, **except**, **else** and **finally** blocks.

#### Syntax:

##### *try:*

*# Code that might raise an exception*

##### *except SomeException:*

*# Code to handle the exception*

##### *else:*

*# Code to run if no exception occurs*

##### *finally:*

*# Code to run regardless of whether an exception occurs*

The **try block** tests a block of code for errors. Python will “try” to execute the code in this block. If an exception occurs, execution will immediately jump to the **except block**. Then the **except block** enables us to handle the error or exception. If the code inside the try block throws an error, Python jumps to the except block and executes it. To handle specific exceptions, use a general exception code to catch all exceptions. The **else block** is optional and if included, must follow all except blocks. The else block runs only if no exceptions are raised in the try block. This is useful for code that should execute if the try block succeeds. The **finally block** always runs, regardless of whether an exception occurred or not. It is typically used for cleanup operations (closing files, releasing resources).

#### Example:

##### *try:*

`n = 0`

`res = 100 / n`

##### *except ZeroDivisionError:*

`print("You can't divide by zero!")`

##### *except ValueError:*

`print("Enter a valid number!")`

##### *else:*

`print("Result is", res)`

##### *finally:*

`print("Execution complete.")`

**Output:**

You can't divide by zero!

Execution complete.

The **try block** asks for user input and tries to divide 100 by the input number. The **except blocks** handle `ZeroDivisionError` and `ValueError`. The **else block** runs if no exception occurs, displaying the result. The **finally block** runs regardless of the outcome, indicating the completion of execution.

### 3.3.3.1 The try and except statement

The **try-except** statement is used to **handle exceptions** (errors) that occur during program execution. Instead of letting the program crash when an error occurs, you can **catch the error** and decide what to do. The most simple way of handling exceptions in Python is by using the try and except block.

Run the code under the **try** statement.

When an exception is raised, execute the code under the **except** statement.

**Syntax:**

try:

*# code that may cause exception*

except:

*# code to run when exception occurs*

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by an except block. When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

**Example:**

try:

print(x)

except:

print ("An exception occurred")

**Output:**

An exception occurred

In the above example, the **try** block will generate an exception, because **x** is not defined. Let's go through another example,

**Example:**

try:

numerator = 10

```

denominator = 0
result = numerator/denominator
print(result)
except:
    print("Error: Denominator cannot be 0.")

```

### Output:

**Error:** Denominator cannot be 0.

In the above example, program code is to divide a number by 0. Here, this code generates an exception. To handle the exception, we have put the code, `result = numerator/denominator` inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped. The `except` block catches the exception and statements inside the `except` block are executed. If none of the statements in the try block generates an exception, the `except` block is skipped.

### 3.3.3.2 Catching Specific Exception

For each try block, there can be zero or more `except` blocks. Multiple `except` blocks allow us to handle each exception differently. The argument type of each `except` block indicates the type of exception that can be handled by it. You can use the try and except statements to identify specific exceptions. You can have zero or more except blocks for a try statement. With multiple except blocks, you can handle each exception differently based on its type. The argument type of every except block shows the exception type that it can handle.

### Syntax:

```

try:
    # statement(s)
except IndexError:
    # statement(s)
except ValueError:
    # statement(s)

```

### Example:

```

try:
    even_numbers = [2,4,6,8]
    print(even_numbers[5])
except ZeroDivisionError:
    print("Denominator cannot be 0.")
except IndexError:

```

```
print("Index Out of Bound.")
```

### Output:

Index Out of Bound

In this example, we have created a list named `even_numbers`. Since the list index starts from 0, the last element of the list is at index 3. Notice the statement,

```
print(even_numbers[5])
```

Here, we are trying to access a value to the index 5. Hence, `IndexError` exception occurs. When the `IndexError` exception occurs in the try block,

The `ZeroDivisionError` exception is skipped.

The set of code inside the `IndexError` exception is executed.

#### 3.3.3.3 The try and else clause

We have learned about **try** and **except**, and now we will be learning about the **else** statement. When the try statement does not raise an exception, code enters into the else block. It is the remedy or a fallback option when you expect a part of your script will produce an exception. It is generally used in a brief setup or verification section where you don't want certain errors to hide.

If the code block within the try block runs without errors, we can run a certain code block in a few situations. We use the else clause in such cases with the try and except statements. The code in the else block is executed only when no exception occurs in the try block. In some situations, we might want to run a certain block of code if the code block inside try runs without any errors. For these cases, you can use the optional else keyword with the try statement.

### Syntax:

```
try:  
    # Code that can raise an exception
```

### except ExceptionType:

```
# Code to handle exception
```

### else:

```
# Code to execute when no exception occurs
```

### Example:

```
try:  
    print("Hello")  
  
except:  
    print("Something went wrong")
```

**else:**

```
print("Nothing went wrong")
```

**Output:**

Hello

Nothing went wrong

### 3.3.3.4 Finally keyword

The finally keyword is available in Python, and it is always used after the try and except blocks. The final block is always executed after the try block has terminated normally or after the try block has terminated for some other reason. In Python, we use the finally keyword with try and except when we have code that is executed regardless of whether try raises an exception. The code within finally runs after try and except blocks. It is useful for tasks like resource cleanup.

**Syntax:**

try:

```
# Some Code....
```

**except:**

```
# optional block
```

```
# Handling of exception (if required)
```

**else:**

```
# execute if no exception
```

**finally:**

```
# Some code .....(always executed)
```

**Example:**

**try:**

```
k = 5/0 # raises divide by zero exception.
```

```
print(k)
```

```
# handles zero division exception
```

**except ZeroDivisionError:**

```
print("Can't divide by zero")
```

**finally:**

```
# this block is always executed
```

```
# regardless of exception generation
```

```
print('This is always executed')
```

**Output:**

Can't divide by zero

This is always executed

### 3.3.3.5 Raise an Exception

We use the raise statement when we forcefully want a specific exception to occur. It helps us raise an exception manually and explicitly signal an error condition. The sole argument in the raise statement shows the exception we want to raise, which can be an exception instance or exception class.

**Syntax:**

```
raise ExceptionType ("Optional error message")
```

In the given syntax, **raise** is the keyword to raise an exception. ExceptionType mentions the type of exception to raise. Optional error message provides additional information regarding the exception.

**Example:**

```
age = int(input("Enter your age: "))
if age < 0:
    raise ValueError("Age cannot be negative")
print("Your age is:", age)
```

**Output:**

Enter your age: -3

ERROR!

Traceback (most recent call last):

File "<main.py>", line 4, in <module>

ValueError: Age cannot be negative

Enter your age: 5

Your age is: 5

### 3.3.3.6 Assert statement

The assert statement is used to test if a condition is True. If it's not, Python raises an AssertionError. It's mainly used for debugging and testing.

**Syntax:**

```
assert condition, "Optional error message"
```

- ◆ If the condition is True → nothing happens.



- ◆ If the condition is False → raises AssertionError.

In the given syntax, the keyword **assert** performs an assertion check. The condition specifies the expression to be tested if it's true or not. Optional error message is displayed if the condition is false.

Examples:	Output:
<pre>x = 10 assert x &gt; 0, "x must be positive" print("x is positive")</pre>	<pre>x is positive</pre>
<pre>x = -3 assert x &gt; 0, "x must be positive" print("x is positive")</pre>	<pre>ERROR! Traceback (most recent call last): File "&lt;main.py&gt;", line 2, in &lt;module&gt; AssertionError: x must be positive</pre>

## Recap

- ◆ In Python, errors are problems in the code that can either be syntax errors or runtime errors.
- ◆ Exception handling allows programs to deal with unexpected errors during execution.
- ◆ An exception is an event that disrupts the normal flow of a program.
- ◆ The try block contains code that might cause an exception.
- ◆ The except block defines how to respond to specific exceptions.
- ◆ The else block runs if no exceptions occur in the try block.
- ◆ The finally block runs code that should execute whether an exception occurs or not.
- ◆ The raise keyword is used to trigger exceptions manually.
- ◆ The assert statement checks a condition and raises an error if the condition is false.
- ◆ Exception handling makes programs more robust and prevents unexpected crashes.

## Objective Type Questions

1. \_\_\_\_\_ handling helps manage unexpected errors during program execution.
2. Code that might raise an error is written inside the \_\_\_\_\_ block.
3. The \_\_\_\_\_ block is used to catch and handle exceptions.
4. To ensure code runs no matter what, we use the \_\_\_\_\_ block.
5. The \_\_\_\_\_ keyword is used to manually trigger an exception.
6. What do you call an event that disrupts normal program flow?
7. Which keyword is used to define an exception handling block?
8. Which block is executed if no exception occurs?
9. Which block runs whether an exception occurs or not?
10. What keyword is used to manually raise an exception?
11. What statement is used for debugging by testing conditions?
12. What error is raised when an assert fails?
13. What type of error occurs due to mistakes in program syntax?
14. What type of error occurs when the program is running?

## Answers to Objective Type Questions

1. Exception
2. try
3. except
4. finally
5. raise
6. Exception
7. try

8. else
9. finally
10. raise
11. assert
12. AssertionError
13. SyntaxError
14. RuntimeError

## Assignments

1. Explain the concept of Exception Handling in Python. How does Python handle different types of errors using try, except, else, and finally blocks? Illustrate with examples how each block is used in practice to manage different scenarios of exception handling.
2. Compare and contrast the raise and assert keywords in Python. How do they differ in their use, and in which scenarios would you prefer to use each? Provide examples demonstrating their usage in exception handling.
3. Illustrate the importance of exception handling in ensuring the reliability of Python programs. Discuss the role of the finally block in Python programming and provide suitable examples.
4. Discuss the various types of errors that occur in Python. How can exception handling techniques such as try and except be used to manage runtime errors effectively? Provide examples for each type of error.

## References

1. Eric Matthes (2019), "Python Crash Course: A Hands-On, Project-Based Introduction to Programming" (2<sup>nd</sup> ed.), published by No Starch Press.
2. Ramalho, L. (2015), "Fluent Python: Clear, Concise, and Effective Programming" (1<sup>st</sup> edition), published by O'Reilly Media.
3. Mark Lutz (2013), "Learning Python: A comprehensive guide for beginners to learn Python, including detailed coverage of exception handling" (5<sup>th</sup> Edition), published by O'Reilly Media.

## Suggested Reading

1. Bader, D. (2017). Python Tricks: A Buffet of Awesome Python Features. Dan Bader Press.
2. Real Python - Functions: <https://realpython.com/tutorials/functions/>
3. Python Course - Exception Handling :<https://docs.python.org/3/tutorial/errors.html>



## Regular Expressions

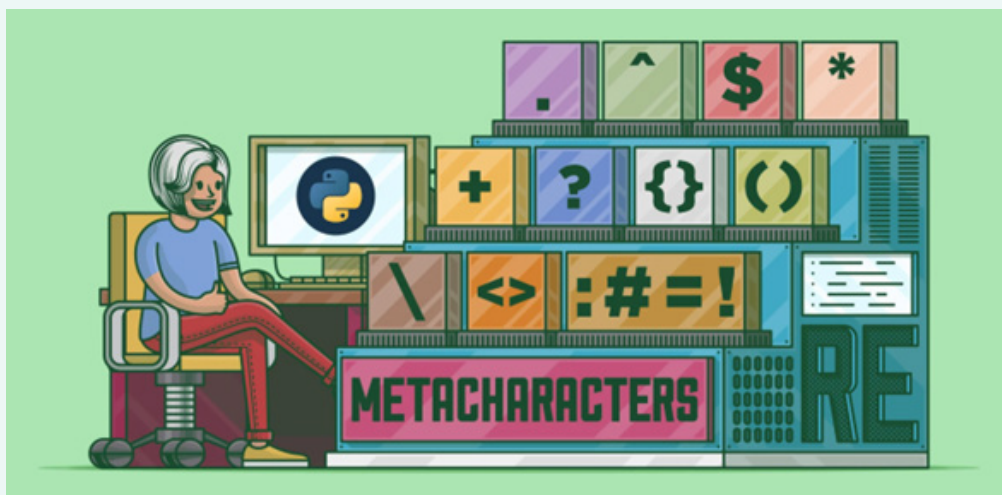
### Learning Outcomes

After completion of this unit, the learner will be able to:

- ◆ recognise the role of regular expressions in Python
- ◆ list the functionality of Python's re module and its core functions
- ◆ identify and interpret special sequences in Python regular expressions
- ◆ recall the importance of Regular Expressions in text manipulation

### Prerequisites

Before moving into the internal details of regular expressions (regex) in Python, it's essential to have a foundational understanding of Python's syntax and string operations. Familiarity with basic programming concepts such as variables, loops, and conditionals will facilitate the learning process. Knowledge of Python's built-in string methods like `.find()`, `.replace()`, and `.split()` is beneficial, as these methods often serve as precursors to more advanced regex functionalities. For instance, before using regex to validate an email address, one might first use string methods to check for the presence of an "@" symbol. This progression from basic string operations to regex ensures a smoother transition and deeper comprehension.



Furthermore, grasping the significance of regex in real-world applications can enhance motivation and contextual understanding. Regular expressions are invaluable tools for tasks such as data validation, text parsing, and information extraction. For example, consider a scenario where a company needs to extract all phone numbers from a large dataset. Using regex patterns like `\d{3}-\d{3}-\d{4}`, one can efficiently identify and extract phone numbers formatted as "123-456-7890" from the text. Recognizing the practical applications of regex underscores its importance and encourages a more engaged learning experience.

## Keywords

Regex, Module, Meta characters, Special sequences, Functions.

## Discussion

### 3.4.1 Introduction to Regular expressions in Python

Regular expressions (regex) are a powerful tool for pattern matching and text manipulation in Python. They allow developers to define search patterns using a sequence of characters, enabling complex string operations such as validation, searching, and extraction. Python's built-in `re` module provides a comprehensive suite of functions to work with regular expressions, making it an essential component for tasks involving text processing.

In Python, regular expressions are utilized to match specific patterns within strings. The `re` module offers various functions like `re.match()`, `re.search()`, and `re.findall()` to perform these operations. By using special characters and syntax, regular expressions can identify patterns such as email addresses, phone numbers, or dates within text. Understanding and leveraging regular expressions can significantly enhance a developer's ability to handle and manipulate textual data efficiently.

#### 3.4.1.1 Definition

In Python, a Regular Expression (RegEx) is a sequence of characters that forms a search pattern. This pattern can be used to check if a string contains a specified search pattern, allowing for complex string matching and manipulation. Python's built-in `re` module provides support for working with regular expressions, offering a range of functions to perform pattern matching operations.

A `re` module in Python supports the use of RegEx, which performs the primary function of offering a search, where it takes a regular expression and a string. It returns the first match, and in case of no match, it returns `None`. Regular expression in Python is a powerful tool for matching text based on predefined patterns. Many programming languages support regular expression for its several uses, which are listed below:

- ◆ Identify patterns in a string/ file.
- ◆ Search for a substring in a string.
- ◆ Replace a section of a string with another string.
- ◆ Validate email format.
- ◆ Split a string into substrings.

#### **Example:**

```
import re

text = "The rain in Spain stays mainly in the plain."

pattern = r"Spain"

match = re.search(pattern, text)

if match:

    print("Match found:", match.group())

else:

    print("No match found.")
```

#### **Output:**

Match found: Spain

### **3.4.1.2 Importance of Regular Expressions**

Regular expressions (regex) are a powerful tool in Python for efficiently searching, validating, and manipulating text. By defining specific patterns, regex allows developers to perform complex string operations with concise syntax. For instance, validating an email address format can be achieved using a single regex pattern, rather than multiple lines of code. This capability is particularly beneficial in data processing tasks, where large volumes of text need to be parsed or cleaned. The Python `re` module provides a suite of functions that facilitate these operations, making regex an indispensable tool for tasks such as input validation, data extraction, and text parsing.

Moreover, regex enhances code readability and maintainability by abstracting complex string matching logic into reusable patterns. This abstraction reduces the need for verbose and error-prone code, leading to more efficient development processes. For example, extracting all email addresses from a document can be accomplished with a straightforward regex pattern, eliminating the need for intricate string manipulation techniques. As a result, regex not only streamlines text processing tasks but also contributes to cleaner and more maintainable codebases. Its versatility and efficiency make it a fundamental skill for Python developers working with text data.

### **3.4.2 Regex Module in Python**

In Python, a built-in module called 're' is used to work with regular expressions in Python.

The 're' module allows us to search, match, and work with text using specific patterns. We can import the Python regular expression module using the import statement. Here are the main ideas and features of the 're' module. The strength and versatility of the 're' module make it a must-have tool to process text and match patterns. Once you gain a fair understanding of functions and principles, you can master text manipulation tasks efficiently.

### 3.4.2.1 Import the Module

Start by importing the 're' module to use the RegEx function. To import the re module, include the following line at the beginning of your Python script:

**Syntax:**

```
import re
```

### 3.4.2.2 Metacharacters

In Python's re module, special characters (also known as metacharacters) are symbols that have a specific meaning within regular expressions. These characters enable the creation of complex search patterns for matching, searching, and manipulating strings. Here's an overview of commonly used special characters: as shown in Table 3.4.1.

Table 3.4.1: Overview of commonly used metacharacters

Metacharacters	Description	Example Pattern	Matches
. - Dot	Matches any character except a newline (\n)	a.c	"abc", "axc", "a c"
^ - Caret	Matches the start of the string	^Hello	"Hello world" (but not "Say Hello")
\$ - Dollar	Matches the end of the string	world\$	"Hello world" (but not "worldwide")
* - Star	Matches 0 or more repetitions of the preceding pattern	a*b	"b", "ab", "aaab"
+ - Plus	Matches 1 or more repetitions of the preceding pattern	a+b	"ab", "aaab"
? - Question Mark	Matches 0 or 1 repetition of the preceding pattern	a?b	"b", "ab"
{ } - Braces	It shows the total number of occurrences of patterns preceding regex to match.	a{2,4}	"aaab", "baaaac", "gaad"
[ ] - Square Brackets	Denotes a character class; matches any one of the enclosed characters	[aeiou]	"a", "e", "i", "o", "u"
( ) - Group	Groups patterns together; creates capturing groups	(abc)+	"abc", "abcabc"
\ - Backslash	Escapes a special character to match it literally	\.	"."

### 3.4.2.3 Special Sequences

In Python's `re` module, special sequences are predefined patterns that simplify common matching tasks, enhancing the expressiveness and efficiency of regular expressions. These sequences are denoted by a backslash (`\`) followed by a character and are widely used for pattern matching involving character types, positions, and assertions. Detailed description of special sequences are depicted in the table 3.4.2.2 listed below.

Table 3.4.2: Special Sequences used in Python

Sequence	Description	Example Pattern	Matches
<code>\d</code>	Matches any decimal digit; equivalent to <code>[0-9]</code>	<code>\d{3}</code>	"123", "456"
<code>\D</code>	Matches any character that is not a decimal digit; equivalent to <code>[^0-9]</code>	<code>\D+</code>	"abc", "XYZ"
<code>\w</code>	Matches any alphanumeric character (letters and digits) and underscore; equivalent to <code>[a-zA-Z0-9_]</code>	<code>\w{5}</code>	"hello", "Python"
<code>\W</code>	Matches any character that is not alphanumeric or underscore; equivalent to <code>[^a-zA-Z0-9_]</code>	<code>\W+</code>	!", "@", " "
<code>\s</code>	Matches any whitespace character (spaces, tabs, newlines); equivalent to <code>[\t\n\r\f\v]</code>	<code>\s+</code>	" ", "\t", "\n"
<code>\S</code>	Matches any character that is not a whitespace character; equivalent to <code>[^\t\n\r\f\v]</code>	<code>\S{3}</code>	"abc", "123"
<code>\b</code>	Matches a word boundary; the position between a word and a non-word character	<code>\bword\b</code>	"word" in "wordplay" but not in "sword"
<code>\B</code>	Matches a non-word boundary; the position between two word characters or two non-word characters	<code>\Bend</code>	"bend" in "bendable" but not in "end"
<code>\A</code>	Matches the start of the string	<code>\AHello</code>	"Hello" at the beginning of a string
<code>\Z</code>	Matches the end of the string	<code>world\Z</code>	"world" at the end of a string

### 3.4.2.4 Python RegEx Functions

Python's `re` module provides a suite of functions to perform operations using regular expressions. These functions enable tasks such as searching, matching, splitting, and replacing strings based on patterns. Here's an overview of the most commonly used functions shown in Table 3.4.2.3.

Table 3.4.3: Python RegEx Functions

Function	Description	Example Usage
<code>re.match(pattern, string)</code>	Determines if the regular expression pattern matches at the beginning of the string. Returns a match object if found; otherwise, returns None.	<code>re.match(r'^\d{3}', '123abc')</code> returns a match object for '123'.
<code>re.search(pattern, string)</code>	Scans through the string looking for the first location where the regular expression pattern matches. Returns a match object if found; otherwise, returns None.	<code>re.search(r'abc', 'xyzabc')</code> returns a match object for 'abc'.
<code>re.findall(pattern, string)</code>	Returns a list of all non-overlapping matches of the pattern in the string.	<code>re.findall(r'\d+', 'abc 123 def 456')</code> returns ['123', '456'].
<code>re.split(pattern, string)</code>	Splits the string by the occurrences of the pattern. Returns a list of substrings.	<code>re.split(r'\s+', 'Hello World')</code> returns ['Hello', 'World'].
<code>re.sub(pattern, repl, string)</code>	Replaces occurrences of the pattern in the string with a replacement string. Returns the modified string.	<code>re.sub(r'abc', 'XYZ', 'abc def abc')</code> returns 'XYZ def XYZ'.
<code>re.escape(string)</code>	This function escapes all special characters in a string and treats them as literals.	<code>re.escape(hello)</code>
<code>re.compile()</code>	This function compiles regular expressions into pattern objects for repeated use. It is useful for improving performance when the pattern is repeated.	<code>re.compile(r'\d{3}')</code>
<code>re.subn(pattern, repl, string)</code>	Similar to <code>re.sub()</code> , but also returns the number of substitutions made.	<code>re.subn(r'abc', 'XYZ', 'abc def abc')</code> returns ('XYZ def XYZ', 2).

## Recap

- ◆ A regular expression (regex) in Python is a sequence of characters used to define a pattern for searching, matching, and manipulating strings.
- ◆ Regex is important in Python for tasks like validating data, searching through text, replacing substrings, and extracting useful information.
- ◆ The re module in Python provides functions and tools for working with regular expressions. It includes methods like match(), search(), and sub().
- ◆ Common functions in the re module include re.match() for matching patterns at the start of a string, re.search() for finding a match anywhere in the string, and re.findall() for extracting all matches.
- ◆ Special characters in Python regex include . (matches any character), ^ (anchors match to the beginning of the string), and \$ (anchors match to the end of the string).
- ◆ Metacharacters such as \d (digit), \w (word character), and \s (whitespace) are used in Python regex to match specific types of characters.

## Objective Type Questions

1. A regular expression in Python is defined using the \_\_\_\_\_ module.
2. The \_\_\_\_\_ character in a regex pattern matches any single character except a newline.
3. The function used to search for a pattern anywhere in a string in Python is \_\_\_\_\_.
4. To match a digit in a regex pattern in Python, the metacharacter \_\_\_\_\_ is used.
5. The \_\_\_\_\_ character in Python regex is used to match the end of a string.
6. What module in Python provides support for regular expressions?
7. Which function in Python is used to replace a pattern with a string?
8. What metacharacter matches any whitespace character in a regex?
9. Which metacharacter is used for grouping parts of a regex pattern?
10. Which quantifier matches one or more occurrences of the preceding element in Python regex?

## Answers to Objective Type Questions

1. re
2. dot (.)
3. re.search()
4. \d
5. \$
6. re
7. re.sub()
8. \s
9. ()
10. +

## Assignments

1. Define regular expressions. Explain their importance in Python programming. Give an example.
2. List and explain different functions provided by Python's re module.
3. What are special characters and metacharacters in regular expressions? Identify and describe each character.

## References

1. Jeffrey Friedl (2006), "Mastering Regular Expressions" (3rd Edition), published by O'Reilly Media.
2. David Beazley and Brian K. Jones (2013), "Python Cookbook" (3rd Edition), published by O'Reilly Media.
3. Arun Saha (2015), "Python Regular Expressions" (1st Edition), published by Packt Publishing.

## Suggested Reading

1. Python RegEx- [https://www.w3schools.com/python/python\\_regex.asp](https://www.w3schools.com/python/python_regex.asp)
2. GeeksforGeeks–PythonRegex-<https://www.geeksforgeeks.org/python-regular-expression-re-module/>
3. RealPython–RegularExpressions with Python-<https://realpython.com/regex-python/>



## **BLOCK 4**

**Database  
Programming,  
Familiarizing NumPy,  
Matplotlib and Pandas**



# UNIT 1 Database Programming

## Learning Outcomes

After completing this unit, the learner will be able to:

- ♦ describe the purpose of Database
- ♦ identify the purpose of SQL in database management.
- ♦ explain how Python connects to a MySQL database.
- ♦ make aware of Create, Retrieve, Update, and Delete (CRUD) statements

## Prerequisites

Consider a simple student management system where user inputs such as student names and grades need to be stored for future reference. Without a database, each time the application runs, it would lose all previously entered data. By integrating Python with a database like MySQL or SQLite, developers can ensure that data is saved and can be accessed or modified as needed. Python's libraries, such as `mysql.connector` or `sqlite3`, provide straightforward interfaces to connect to these databases, execute SQL queries, and manage data efficiently. This approach not only preserves data across sessions but also allows for complex operations like data filtering, sorting, and aggregation, which are crucial for dynamic applications.

Python, as a high-level programming language, offers extensive support for diverse databases. With Python, we can establish connections and execute queries for a specific database without the need to manually write raw queries in the terminal or shell of that particular database. The only requirement is to have the desired database installed on our system.

## Key Concepts

Database, MySQL, Create, Retrieve, Update, Delete

## Discussion

### 4.1.1 Introduction

This unit begins with database programming basics. We have written and executed many programs in the previous classes. For example, calculator application. While running the program, we have given the numbers to add. After that, the programs will show the output. When we run the program again, we need to input the data( numbers to add) again. What happened to the data we entered earlier? The data was saved temporarily during the execution of the program. We have to use a database or file to save the data for future use. Database software will save the data permanently. Python supports file handling. The file is the place to store data or information. We can create, update, delete, search and do other file handling operations in Python. Using a file handling method in any programming language has many limitations and issues. The alternate way to save the data is using database programs. The popular database programs are MySQL, Oracle, Microsoft Database software that is used to save and manipulate the data. A database management system (DBMS) is a comprehensive database application to work with the database. DBMS allows the users to store, retrieve, update and manage the data in an organized and optimized manner. The language used to store, retrieve, update and manage the data is called Structured Query Language (SQL), Microsoft Access, etc.

MySQL is an open-source relational database management system. In this course, we will discuss MySQL. Some of the applications that use MySQL are Twitter, LinkedIn, Facebook, YouTube, etc. For example, in the Facebook sign up process, we will fill the following form and click on signup.

**Sign Up** ×

It's quick and easy.

First name ! Surname

Mobile number or email address

New password

Date of birth ?

31 Jan 2022

Gender ?

Female Male Custom

By clicking Sign Up, you agree to our [Terms](#), [Data Policy](#) and [Cookie Policy](#). You may receive SMS notifications from us and can opt out at any time.

**Sign Up**

Fig. 4.1.1 Sample Signup form

The application will verify the data entered and save the data to the database. See Fig 4.1.1 for Signup form. The verification part is done by the scripting or programming language and the saving data to the database is done by the SQL. In conclusion, to develop an application, we need a programming language or a scripting language and database program to save the data.

Upon clicking on Sign Up, the Facebook application will send the user data to Facebook's server on which the database resides.

A database is a collection of data organized as tables, records, and fields. For example, the University database has a student table, Course table, Examination table, etc. The student table consists of student records. The following table has four fields and 2 records.

Student ID	Student name	City	Phone number
SNOU123	KKG	Kollam	97642789
SNOU222	Shan	Trivandrum	89902233

Install MySQL on your computer.

### 4.1.2 Example of database programming in Python

Program to create a database. Type the following code in Python IDLE, save and run.

```
import mysql.connector

MyFirstDB_Connection = mysql.connector.connect(host="localhost",user="root", password="Kxxxxxx!")

mycursor = MyFirstDB_Connection.cursor()

mycursor.execute("CREATE DATABASE MyFirstDB")
```

The first step is to import the mysql.connector. This connector is a self-contained Python driver for communicating Python with MySQL servers. Second step is to connect Python with MySQL.

MyFirstDB\_Connection is the name of the connection. It could be any name. The host is the place the server is located. In our case, MySQL is installed on the same computer we are working with. Hence the host is the localhost or IP address of the local host. During the Facebook sign-up process the connection will be made to Facebook's server on which their database is created and the host will be the name of the Facebook's domain or IP address. By default, the username is root. We can create users in MySQL and use it. Password is the password set for the user root. MySQL cursor interacts with the MySQL server to execute operations such as SQL statements. The name of the database created in the above example is MyFirstDB and the name of the cursor is my cursor. It could be any name. The cursor permits row-by-row processing of the result

sets. It is used to fetch the results returned from a query.

The following program will display all databases created.

```
import mysql.connector

# Connect to MySQL server

MyFirstDB_Connection = mysql.connector.connect(host="localhost", user="root",
password="Kxxxxxl" # Replace with your actual password)

# Create a cursor object to execute SQL queries

myc = MyFirstDB_Connection.cursor()

# Execute a query to list all databases

myc.execute("SHOW DATABASES")

# Loop through and print each database name

for k in myc:

    print(k)
```

This program will connect with MySQL, execute the SQL command to show databases, and display the database names retrieved and stored in the cursor one by one using a for loop. k is a variable name. it could be any name.

The following is a program to create a table in the database. Remember, the name of the database we created is MyFirstDB.

```
Import mysql.connector

MyFirstDB_Connection=mysql.connector.connect(host="localhost",user="root",
password= "Kxxxxxl", database = " MyFirstDB")

myc = MyFirstDB_Connection.cursor()

myc.execute("CREATE TABLE Student(StudentID VARCHAR(100), Student_Name
VARCHAR(255), \ City VARCHAR(100), Phone_No VARCHAR(10))")
```

Note: \ is used to write the SQL statement in multiline. Varchar(100) represents a variable character data type that can store a maximum of 100 characters. (The name John contains 4 characters)

There are different types of data in MySQL. For example, INTEGER will be used to store the number of students

Refer MySQL manual <https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

The following program displays the tables created in the MyFirstDB database.

```
Import mysql.connector
```



```
MyFirstDB_Connection=mysql.connector.connect(host= "localhost",user= "root",  
password= "Kxxxxxl",database = "MyFirstDB")
```

```
myc = MyFirstDB_Connection.cursor()
```

```
myc.execute("SHOW TABLES")
```

```
for t in myc:
```

```
print(t)
```

The following program inserts data into the tables.

```
Import mysql.connector
```

```
MyFirstDB_Connection=mysql.connector.connect(host= "localhost",user= "root",  
password= "Kxxxxxl",database = "MyFirstDB")
```

```
Insert_Student = (INSERT INTO student(StudentID, Student_Name, City, Phone_NO)\  
values (%s,%s,%s))
```

```
Student_Data = ("sgoul", "Diya", "Kollam", "9283458929")
```

```
myc = MyFirstDB_Connection.cursor()
```

```
myc.execute(Insert_Student, Student_Data)
```

```
MyFirstDB_Connection.commit()
```

The following program retrieves the data from the table and displays it.

```
Import mysql.connector
```

```
MyFirstDB_Connection=mysql.connector.connect(host= "localhost",user= "root",  
password= "Kxxxxxl",database = "MyFirstDB")
```

```
myc = MyFirstDB_Connection.cursor()
```

```
myc.execute("SELECT * FROM Student")
```

```
Result = myc.fetchall()
```

```
for k in Result:
```

```
print(k)
```

The following program reads the student data from the input device and inserts it into the table.

```
Import mysql.connector
```

```
SID = input("Enter student ID : ")
```

```
Sname = input("Enter student name : ")
```

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

```
phone = input("Enter phone number : ")
```

```
MyFirstDB_Connection=mysql.connector.connect(host= "localhost",user= "root",  
password= "Kxxxxxl",database = "MyFirstDB")
```

```
Insert_Student = (INSERT INTO student(StudentID, Student_Name,City,Phone_NO)\  
values (%s,%s,%s,%s))
```

```
Student_Data = (SID,Sname,city,phone)
```

```
myc = MyFirstDB_Connection.cursor()
```

```
myc.execute(Insert_Student,Student_Data)
```

```
MyFirstDB_Connection.commit()
```

In this program, I have used four variable names to input the data.

```
Student_Data = (SID,Sname,city,phone)
```

Note that the variable names in the list are not in quotes(‘ ’ )

We click on the sign-up button after entering the data in the Facebook registration process. The data we entered in the registration form will be saved to the table.

Activity: Create a login table and insert data into that.

username	password
Krishna	56Urte
Joy	J9otrw2!

Note: username must be the primary key. The values will not be repeated.

```
CREATE TABLE login(username VARCHAR(100) PRIMARY KEY, password  
VARCHAR(255))
```

The following is a sample code for the Login module.

```
Import mysql.connector
```

```
User_Name = input("Enter user name : ")
```

```
pass = input("Enter password : ")
```

```
MyFirstDB_Connection=mysql.connector.connect(host= "localhost",user= "root",  
password= "Kxxxxxl",database = "MyFirstDB")
```

```
myc.execute("SELECT * FROM login where username=%s and password=%s",  
(User_Name,pass))
```

```
Result = myc.fetchone()
```

if Result:



```
print("Login successful")
```

```
else:
```

```
    print("Invalid user name or password")
```

The user will be prompted to enter the username and password. Pass the username(I used the variable name User\_Name) and password(Pass is the variable name) to the SQL query as shown in the code. User\_Name and Pass are two variable names and username and password are the fields in the login table.

Activity: Insert the username Krishna two times and observe the result.

We can use the DELETE query to delete a record and UPDATE to make changes to the existing records.

```
DELETE FROM student WHERE city = 'Kollam'
```

```
UPDATE student SET city = ' Kottayam' WHERE Student_ID = 'SNOU123'
```

Note: Refer to the MySQL Manual to learn more about SQL.

Python supports various databases like MySQL, SQLite, Sybase, Oracle, etc. Python also supports the NoSQL database MongoDB. NoSQL ("not only SQL") is a non-tabular database and stores data differently than relational tables such as MySQL, SQLite, Sybase, Oracle

## Recap

- ◆ Data entered during program execution is stored temporarily and lost after the program ends.
- ◆ To retain data permanently, we use files or databases.
- ◆ File handling in Python allows data storage but has limitations for complex data management.
- ◆ Databases like MySQL provide efficient ways to store, retrieve, and manage structured data.
- ◆ MySQL is a widely used open-source relational database management system.
- ◆ Python connects to MySQL using the `mysql.connector` module.
- ◆ We can create databases and tables in MySQL using SQL commands executed through Python.
- ◆ Data can be inserted into tables using the `INSERT INTO` statement.
- ◆ Data retrieval is done using the `SELECT` statement, and results can be processed in Python.
- ◆ We can update or delete records using `UPDATE` and `DELETE` statements, respectively.

## Objective Type Questions

1. \_\_\_\_\_ is an interface for connecting to a MYSQL database server from Python
2. Collection of data organized as tables, records, and fields is called
3. \_\_\_\_\_ is used to write the SQL statement in multiline.
4. Which query is used to delete records of a database.
5. UPDATE command is used to
6. How to establish connection with MySQL in Python
7. Which of the following are valid Cursor methods used to execute SQL statements and retrieve query results?
8. `commit()` method should be executed to

9. What is used to store data permanently in programming?
10. Which Python module is used to connect with MySQL?
11. What is the default username in MySQL?
12. Which SQL statement is used to create a table?
13. Which SQL command displays all the databases?
14. Which SQL keyword is used to add records to a table?
15. What is the name of the variable used to execute SQL queries in Python?
16. What type of database is MySQL?
17. Which function is used to retrieve all data from the table?
18. What is the default host name when MySQL is installed on the same system?
19. What is the SQL command to delete records?
20. What is the primary key in the login table?
21. Which type of database is MongoDB?

## Answers to Objective Type Questions

1. MySQLdb
2. Database
3. \
4. DELETE
5. Make changes to the existing records.
6. Import mysql.connector
7. cursor.run()
8. Finalize transactions
9. Database
10. mysql.connector

11. root
12. CREATE
13. SHOW
14. INSERT
15. cursor
16. Relational
17. fetchall
18. localhost
19. DELETE
20. username
21. NoSQL

## Assignments

1. How to install database in python
2. Create Employee database with following field employee id, employee name, designation, address, date of birth, basic pay

## Suggested Reading

1. Matthes, E. (2019). Python Crash Course: A Hands-On, Project-Based Introduction to Programming (2nd ed.). No Starch Press.
2. Ramalho, L. (2015). Fluent Python: Clear, Concise, and Effective Programming. O'Reilly Media.
3. Bader, D. (2017). Python Tricks: A Buffet of Awesome Python Features. Dan Bader Press.
4. Real Python - Functions: <https://realpython.com/tutorials/functions/>



## Familiarising NumPy

### Learning Outcomes

After completing this unit, the learner will be able to:

- ◆ familiarise the importance of NumPy in data science and numerical computing.
- ◆ learn how to create and manipulate 1D, 2D, and 3D arrays using NumPy.
- ◆ identify the indexing and slicing techniques to access and modify array elements.
- ◆ perform array operations such as reshaping, sorting, searching, and inserting elements.
- ◆ utilize NumPy functions for efficient data processing and analysis.

### Prerequisites

Imagine a small business owner who reviews monthly sales to decide the best time to offer discounts. Or consider a fitness app that tracks how many steps a user walks each day and then calculates the weekly average. These examples show how data is collected, processed, and analyzed to make informed decisions. To understand how such tasks are done using Python, learners need to be prepared with certain foundational skills.

Before diving into Python tools for data science, learners should have a basic understanding of programming concepts. This includes working with variables, common data types like lists and tuples, and using control structures such as loops and functions. They should also know how to install and import Python libraries using tools like pip.

In addition to programming, a grasp of basic mathematics and statistics is essential. Concepts such as mean, median, variance, and basic algebra are frequently applied in data analysis. Having a clear understanding of how Python handles collections of data like lists and dictionaries will also make it easier to learn how arrays work in libraries such as NumPy. Together, these skills provide a strong starting point for exploring the powerful tools Python offers for data science.

## Keywords

Data Collection, Data Analysis, Python libraries, Insight Extraction, Types of Arrays, Array operations.

## Discussion

Before starting the tools or libraries such as NumPy, pandas, etc., let us discuss the basics of data science. Data science is the science of analyzing raw data to derive insight from raw data that will aid decision-making. Data science uses mathematics, statistics, and Machine learning techniques to derive insight from raw data. As we all know, the data is collected and saved somewhere while using applications. For example, the data is collected and saved when we browse through the website or e-commerce applications. This raw data will be analyzed to support business decision-making such as introducing new products, giving discounts, hiring decisions, etc. NumPy is crucial for harnessing Python's full potential in numerical computing, as its robust array-based operations, vectorized functions, and broadcasting capabilities make it an indispensable asset for scientific computing, data analysis, and machine learning applications.

### 4.2.1 What do we do with the data?

- ◆ Collect data
- ◆ Process the data and clean to remove data errors.
- ◆ Summarise the data. For example, find the average, median, variance, etc.
- ◆ Visualise the data. For example, plot a graph to identify the relationships and trends.

Derive insight from data.

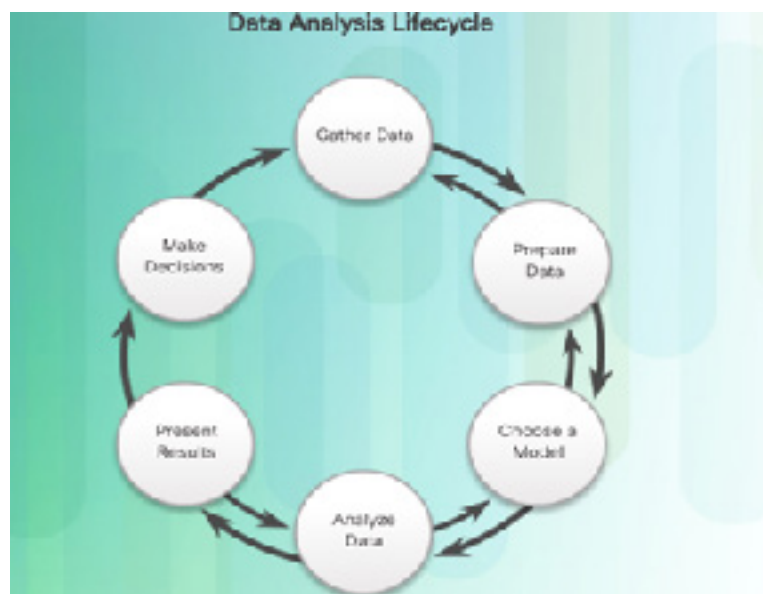


Fig 4.2.1 Data Analysis Lifecycle

## 4.2.2 What is the significance of Python programming in Data science?

- ◆ Python provides various tools and libraries which are essential for data science. Using library functions will make the programming task easier. Otherwise, we have to write the programs from scratch. Python provides functions to read data from local files, databases, and even the cloud. Python has a robust user community to update the libraries according to the new requirements. NumPy is one of the Python libraries for scientific computing. Pandas library provides libraries for data wrangling and manipulation. The following units focus on
- ◆ Familiarising NumPy
- ◆ Introduction to matplotlib
- ◆ Introduction to pandas

## 4.2.3 Python NumPy

” NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms basic linear algebra, basic statistical operations, random simulation and much more.” NumPy is a robust Python library that facilitates efficient numerical computing through its support for multi-dimensional arrays, matrices, and an extensive array of mathematical functions.

NumPy is used for working with arrays. Arrays are a collection of data similar to a list but with more features and advantages. Lists are slow to process compared to arrays. NumPy library is used in data science, linear algebra, matrices, and Fourier transform. It provides library functions to work with ndarray (N-dimensional array).

To Install NumPy the following command is used: `pip install numpy`

## 4.2.4 Pip install numpy

To import the numpy to our applications using the import keyword

```
import numpy
```

Usually, we can import using an alias.

```
import numpy as kk
```

where kk is the alias name

## 4.2.5 Creating an array

The array can be created by using different mechanisms. Such as Conversion from list and tuple structures, using Intrinsic NumPy.

Activity 1: Creating an array using NumPy.

The following is an example of creating an array using Lists.

**Input:**

```
import numpy as kk  
MyFirstArray = kk.array([11,22,32,42,2])  
print(MyFirstArray)
```

**Output:**

```
[11 22 32 42  2]
```

**Activity 2:** Create an array using intrinsic function – `arange()`

**Input:**

```
import numpy as kk  
MyFirstArray = kk.arange(5)  
print(MyFirstArray)
```

**Output:**

```
[0 1 2 3 4]
```

**Activity 3:** Creating a two-dimensional array using the list.

**Input:**

```
import numpy as kk  
MyArray = kk.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
print(MyArray)  
  
print("Dimension=", MyArray.ndim) # NumPy Arrays have a "ndim" property. It  
returns the number of dimensions in the array.
```

**Output:**

```
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]
```

```
Dimension= 2
```

**Activity 4:** Creating a three-dimensional array using the list.

**Input:**

```
import numpy as kk
```

```
MyArray = kk.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(MyArray)

print("Dimension=", MyArray.ndim) # NumPy Arrays have a "ndim" property. It
returns the number of dimensions in the array.
```

**Output:**

```
[[[ 1  2  3]
  [ 4  5  6]]
 [[ 7  8  9]
  [10 11 12]]]
Dimension= 3
```

**Exercise 1:** Create a 1-dimensional array of 5 elements and initialize it with the values 10, 20, 30, 40, and 50.

**Exercise 2:** Create a 2-dimensional array of size 3x4 and initialize it with the values 1-12.

**Exercise 3:** Create a 3-dimensional array using Lists.

## 4.2.6 Indexing an array

### One dimensional array

Indexing and arrays can be done using the standard Python `x[obj]` syntax, where `x` is the array and `obj` the selection.

The following are the indexing available depending on *obj*:

- ◆ basic indexing
- ◆ field access
- ◆ advanced indexing

`mark = np.array([20, 25, 12, 30])` will create a one dimensional array with 4 elements.

mark(0)	mark(1)	mark(2)	mark(3)
20	25	12	30
<code>\mark(-4)</code>	Negative Indexing <code>mark(-3)</code>	<code>mark(-2)</code>	<code>mark(-1)</code>

**Input:**

```
import numpy as kk
```

```
mark = kk.array([20,25,12,30])
print(mark[1])
print(mark[-4])
```

The result of the above code will be 25. As shown in the table mark(1) is 25 the index starts from zero to n-1. mark(-4) will display 20.

Two-dimensional array index

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[1,3])
```

Output:

9

**Three-dimensional array index**

**3-D array**

```
arr = [
    [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ],
    [ [10, 11, 12], [13, 14, 15], [16, 17, 18] ],
    [ [19, 20, 21], [22, 23, 24], [25, 26, 27] ],
]
```

(0, 0, 0)

1	2	3
4	5	6
7	8	9

```
arr[0][0][0] => 1
arr[1][0][0] => 2
arr[0][1][0] => 4
arr[0][0][1] => 10
```

```
ar_3d = [[[1, 2, 3, 4],
          [5, 6, 7, 8],
          [9, 10, 11, 12]],
         [[13, 14, 15, 16],
          [17, 18, 19, 20],
          [21, 22, 23, 24]]]

print(ar_3d[1][1][1])      #18
print(ar_3d[1][0][0])      #13
```

### 4.2.7 Reshaping an array

Reshaping allows modifying the structure of an array by adding, removing, or adjusting dimensions, effectively changing the arrangement of its elements.

```
import numpy as kk
m= kk.arange(10)
print(m)
```

```
y= m.reshape(2,5)
print("After Reshaping")
print(y)
```

**Output:**

```
[0 1 2 3 4 5 6 7 8 9]
```

After Reshaping

```
[[0 1 2 3 4]
```

```
[5 6 7 8 9]]
```

Reshaping an array to a 3-dimensional array

```
import numpy as kk
mark= kk.arange(12)
mark= mark.reshape(2,3,2)
print(mark)
```

```
mark = [0,1,2,3,4,5,6,7,8,9,10,11]
```

This will create a one-dimensional array with 12 elements first and reshape it to a 3D array with X value 2, Y value 3, and Z value 2. ( $2 \times 3 \times 2 = 12$ )

X value 2 means the index will be 0 to 1. Similarly, Y value 3 means, the index will be 0 to 2.

The output will be

```
[[[0 1]
 [2 3]
 [4 5]]
 [[6 7]
 [8 9]
 [10 11]]]
```

**Exercise 4:**

Fill the table if  $x=3$ ,  $y=2$  and  $z=2$

**Exercise 5:** Predict the output of the following code.

```
a) import numpy as kk
m= kk.arange(12)
```

```
print(m)
y= m.reshape(2,3,2)
print(y[1,0,0])
b) import numpy as kk
mark= kk.arange(10)
y= mark.reshape(3,2,2)
print(mark[1,0,0])
```

### 4.2.8 Slicing arrays

Slicing in Python means taking items from one index to another index.

```
import numpy as kk
mark= kk.arange(12)
print(mark[1:5])
```

The output of the above code is [1 2 3 4]. Remember the index starting from zero. In this example, the start index is 1 and the end index is 4 ( 5-1). Since the increment is not mentioned, the default value 1 is taken.

```
import numpy as kk
mark= kk.arange(12)
print(mark[1:10:3])
```

The increment value is 3 in the above code. The result will be [1 4 7].

**Exercise 6: find the output of the following code.**

```
import numpy as kk
mark= kk.array([20,30,40,7,3,4,5,6])
print(mark[1:5:2])
```

### 4.2.9 Searching from an array

```
import numpy as kk
mark= kk.array([20,30,40,7,3,4,5,6])
x=kk.where(mark==40)
print(x)
```

The result will be (array([2], dtype=int64),). The index of element 40 of the mark array is 2.

### 4.2.10 Sorting an array

```
import numpy as kk  
  
mark= kk.array([20,30,40,7,3,4,5,6])  
  
x=kk.sort(mark)  
  
print(x)
```

[ 3 4 5 6 7 20 30 40] is the result of the above code. The mark array is sorted and stored in another array named x.

### 4.2.11 Insert an element into an array

```
import numpy as kk  
  
mark= kk.array([20,30,40,7,3,4,5,6])  
  
mark=kk.insert(mark,2,89)  
  
print(mark)
```

The new array will be [20 30 89 40 7 3 4 5 6]. Inserted 89 as a 3<sup>rd</sup> element

## Recap

- ◆ Python is widely used in data science due to its simplicity and the availability of powerful libraries.
- ◆ Real-world data is collected through websites, applications, and devices, and then stored for analysis.
- ◆ Data science involves steps like data collection, cleaning, summarizing, visualizing, and insight generation.
- ◆ NumPy is a core Python library used for scientific and numerical computing.
- ◆ NumPy provides an efficient way to work with multi-dimensional arrays using the ndarray object.
- ◆ Arrays in NumPy are more powerful and faster than Python lists for numerical tasks.
- ◆ You can create arrays using functions like `array()`, `arange()`, or by converting lists and tuples.
- ◆ Indexing allows you to access specific elements in 1D, 2D, or 3D arrays using positive or negative indices.
- ◆ Slicing helps you retrieve a range of values from an array using the format `[start: stop: step]`.

- ◆ Reshaping changes the structure of an array (e.g., from 1D to 2D or 3D) using `reshape ()`.
- ◆ You can search for specific values in an array using `numpy. where ()`.
- ◆ Arrays can be sorted in ascending order using `numpy. sort ()`.
- ◆ The `insert ()` function allows you to add elements at a specific index in the array.
- ◆ Learning NumPy sets the foundation for using other Python libraries like `pandas`, `matplotlib`, and `scikit-learn` in data science.

## Objective Type Questions

1. What does NumPy stand for?
2. Which command is used to install NumPy?
3. What is the default data structure used in NumPy?
4. NumPy arrays are:
5. What will `np. arange (4)` return?
6. Which function is used to reshape a NumPy array?
7. What does `ndim` represent in a NumPy array?
8. What does `np.where(array==value)` return?
9. Which function is used to sort a NumPy array?
10. How do you create a 1D array with values `[1,2,3,4,5]` in NumPy?
11. Which method is used to insert an element into a NumPy array?
12. What does `print(arr[1:4])` do in slicing?
13. What function would you use to create an array from 0 to 9?
14. What function returns the average of an array?
15. Predict the result:

```
a = np.array([[1,2],[3,4]])
```

```
b = np.array([[5,6],[7,8]])
```

```
print(a + b)
```



## Answers to Objective Type Questions

1. Numerical Python
2. `pip install numpy`
3. `ndarray`
4. Faster than lists.
5. `[0 1 2 3]`
6. `reshape ()`
7. Number of dimensions
8. The index/indices where the value is found.
9. `sort ()`
10. `np.array([1,2,3,4,5])`
11. `np.insert()`
12. Prints elements from index 1 to 3.
13. `np.arange(10)`
14. `np.mean(arr)`
15. `[[6 8], [10 12]]`

## Assignments

1. Write the significance of data science?
2. Write a program to create an 1D, 2D and 3D array using NumPy
3. Write a program to search, sort and insert elements in an array.
4. Write a note on Searching, Sorting, and Modifying Arrays in NumPy

## References

1. Robert Johansson, 2nd Edition (2019), “Numerical Python: A Practical Techniques Approach for Industry”.
2. Joel Grus, 2nd Edition (2019), “Data Science from Scratch: First Principles with Python”.
3. Real Python - Functions: <https://realpython.com/tutorials/functions/>

## Suggested Reading

1. Matthes, E. (2019). “Python Crash Course: A Hands-On, Project-Based Introduction to Programming” (2nd ed.). No Starch Press.
2. Ramalho, L. (2015). “Fluent Python: Clear, Concise, and Effective Programming”. O'Reilly Media.
3. Bader, D. (2017). “Python Tricks: A Buffet of Awesome Python Features”. Dan Bader Press.



# Introduction to Matplotlib

## Learning Outcomes

After the successful completion of this unit, the learner will be able to:

- ◆ identify the data analysis life cycle.
- ◆ familiarise the procedure for how to install and use Matplotlib in Python.
- ◆ visualise different types of graphs like line charts, bar charts, and pie charts.
- ◆ attain the ability to customize the appearance of graphs to make them more clearer.
- ◆ create visually appealing and informative presentations using graphs.

## Prerequisites

Matplotlib is a widely used plotting library in Python that provides a flexible and comprehensive set of tools for data visualizations.. If you are curious about questions like "Which month had the highest sales?" or "Which product was most popular?", then you will enjoy using Matplotlib to find answers. It is essential to develop a solid foundation in Python programming. This includes understanding variables, loops, conditional statements, functions, and working with lists or arrays. This basic knowledge will help you to write code for graphs and charts. You should also know how to work with data in Python. Learn how to store data in lists or arrays. Libraries like NumPy or Pandas make this easier. These tools help you clean and organize data. For example, a shop owner can use a list to store daily sales and then turn it into a graph.

Good knowledge of data storage will help you to organize and prepare your data before making any graph.. Visualisation often involves showing data in terms of parts of a whole, increases and decreases, or comparing quantities. For example, when you make a pie chart of monthly expenses, you need to understand how each expense is a part of the total. You do not need to know very advanced math, but basic math skills will make understanding charts easier.

Data visualisation is not only about creating pictures. It is about finding hidden patterns in numbers. For example, by looking at a line chart of monthly temperatures, you can understand how seasons change over the year. If you are curious to explore and explain such facts, you will enjoy studying Matplotlib and data visualisation.

# Key Concepts

Data analysis, Data Visualisation, Matplotlib, Plotting, Pyplot, Graph Customization

## Discussion

Data science is the study of huge amounts of data. It is the art and science of turning data into knowledge and action. For example, a textile manufacturer can use its sales data to plan its future business model. There may be millions of data about sales, raw materials, customer complaints, salary, etc. Data science can support people to recognize their environments, study existing issues, and reveal previously hidden opportunities. If the data is in numbers, it is difficult to analyze and interpret data. Visualising the data using a graphical representation allows a quick interpretation and analysis of data.

### 4.3.1 What do we do with the data?

1. **Collect** the data systematically from reliable sources.
2. **Clean** the data by removing errors, duplicates, and handling missing values.
3. **Organise** the data into a structured format (tables, databases, etc.).
4. **Analyse** the data to discover patterns, trends, or relationships.
5. **Visualise** the data using graphs, charts, or other visual tools.
6. **Interpret** the results to draw meaningful conclusions.
7. **Store** the data securely for future use or reference.
8. **Share or report** the findings with stakeholders or publish results if needed.

Decision-makers rely on data analytics to extract the required information from data at the right time, in the right place, to make the right decision. This information can tell many different narratives, depending on how the data is analysed. For example, in business, a data scientist may discover market trends that enable a business organization to take decisions to improve the business.

### 4.3.2 Data Analysis Lifecycle

- ◆ **Collect the data** – Example: Collect the sales data from 2021 to 2022
- ◆ **Preparing the data** - Transform the data into an appropriate format.
- ◆ **Choosing a model** - Choosing an analysis technique that will answer the question with the available data.
- ◆ **Analysing the data** – Check whether the model and the analyzed data are reliable.
- ◆ **Presenting the results** - For example, a graph that represents the yearly



sales.

- ◆ **Making decisions** - The final step in the data analysis lifecycle is take the accurate decisions.

### 4.3.3 Data Visualization

There are many tools and libraries available to visualize the data. Matplotlib in Python is a tool used to present and visualize the data. Data visualization is used to understand the data and study the effect of data by making graphs or charts. This will facilitate deriving insight from data to make decisions. Python offers multiple graphing libraries. The following are some of the popular matplotlib libraries.

- ◆ matplotlib: To create 2D graphs and plots
- ◆ pandas' visualization
- ◆ seaborn: Provides a high-level interface to draw informative statistical graphics.

The following is an example of matplotlib visualization.

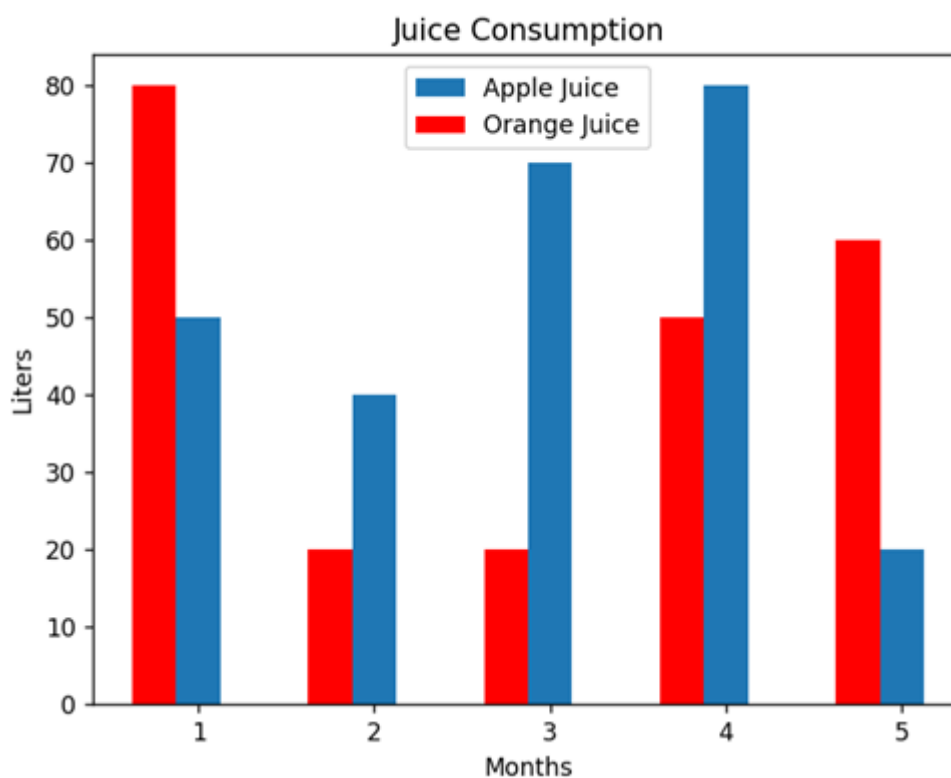


Fig: 4.3.1 Data visualization to shows the consumption of orange juice and apple juice in different months.

Matplotlib is a library to visualize the data to know the trends. It is a plotting library that can be used to create a range of plots from simple line plots to complicated 2D/3D plots. The data alone is not meaningful information, the data must be analyzed and then presented in a form that can be interpreted. This is useful to the decision-makers to take the right action.

The data should be analyzed, studied, and used to produce insights that can guide decision-making, such as the decision to start a new manufacturing unit, introducing new products etc.

Matplotlib provides an easy visual approach to present our findings using graphs/plots. The following are examples of the visualizations that can be plotted using Matplotlib

- ◆ Bar Graph
- ◆ Histogram
- ◆ Line Chart
- ◆ Pie Chart
- ◆ Scatter Plot
- ◆ Area plot



Fig 4.3.2 Different types of graphs

### 4.3.3 Installing Matplotlib

To install **Matplotlib** follow these steps:

`pip install matplotlib.`

Once Matplotlib is installed, import it in your applications by adding the following command:

```
import matplotlib
```

The version string is stored under `__version__` attribute.

A screenshot of a Python IDLE Shell window titled 'IDLE Shell 3.10.0'. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The shell displays the following text:

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import matplotlib
>>> print(matplotlib.__version__)
3.5.1
```

Fig 4.3.3 Sample output screen

For plotting using matplotlib, import its pyplot module using the following command:

```
import matplotlib.pyplot as plt
```

#plt is an alias or alternative name for the matplotlib.

#pyplot. *Pyplot is the popular matplotlib submodule.*

The plot () function of the pyplot module is used to create a chart.

The show () is used to display the figure created using the plot () function.

### 4.3.4 Graph customization

Pyplot library gives us some functions, which can be used to customize charts such as adding titles or legends. Some of the options are given below:

Table 4:3:1 Graph customization

Options	Explanation	Syntax
Title	Display the title of the plot	Plt.title(“Mark”)
Grid	Show the grid lines on the plot	Plt.grid()
Legend	Place a legend on the axis	Legend()
Savefig	Used to save the figure	<b>savefig()</b>
Xlabel	Set the label for X-axis	Xlabel()
Ylabel	Set the label for Y-axis	Ylabel()
Xticks	Get or set the current tick locations and labels of the X-axis.	plt.xticks()
Yticks	Get or set the current tick locations and labels of the Y-axis.	plt.yticks()

Other attributes for customization of chart

#### 1. Marker

In Matplotlib, markers are used to visually highlight each data point with shapes like circles, squares, and triangles.

#### 2. Color

Table 4.3.2 Some of the character color code

Character code	Color name
‘b’	Blue
‘g’	Green
‘r’	Red
‘k’	Black
‘w’	White
‘y’	Yellow

#### 3. Line width and Line style

The following are line style

- ◆ solid

- ◆ dashed
- ◆ dotted
- ◆ dashdot
- ◆ none
- ◆ The following are examples of line color options
 

Example: `plt.plot(xpoints, ypoints , linestyle = 'dotted', color = 'r')`

`plt.plot(xpoints, ypoints , linestyle = 'dotted', color = 'g')`
- ◆ We can also use Hexadecimal color codes as follows
 

Example: `plt.plot(xpoints, ypoints , linestyle = 'dotted', color = '#4000ff')`
- ◆ Using the linewidth option, we can change the line width.
 

Example: `plt.plot(xpoints, ypoints , linestyle = 'dotted', linewidth = '20.5', color = '#4000ff')`
- ◆ Use the title option to insert a title.
 

Example, `plt.title("My Title")`
- ◆ Use the grid option to display the grid
 

Example: `plt.grid()`

### Example:

The following code will make a graph.

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 2, 3, 4])
plt.ylabel('Y Label')           #label to y -axis
plt.xlabel('X Label')           #label to x-axis
plt.show()
```

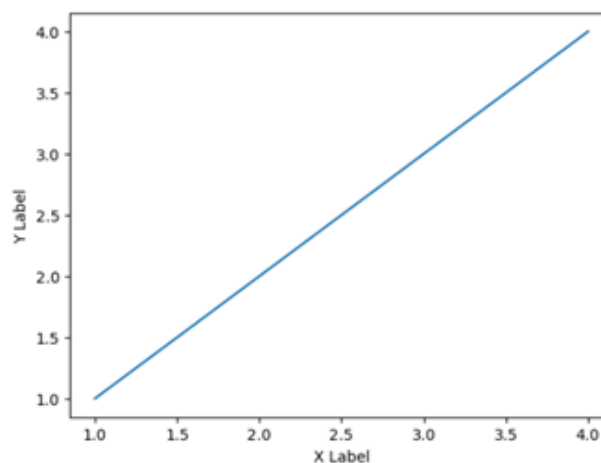


Fig: 4.3.4 Line chart

◆ The same program may also be written as

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([1, 4])
ypoints = np.array([1, 4])
plt.ylabel('Y Label')
plt.xlabel('X Label')
plt.plot(xpoints, ypoints)
plt.show()
```

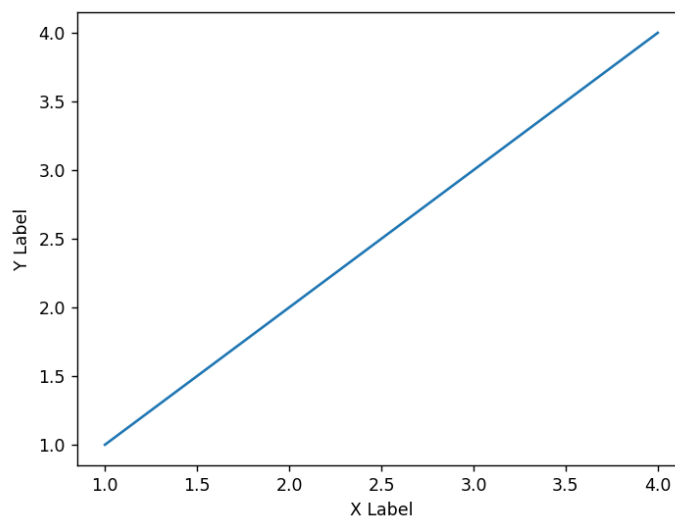


Fig 4.3.5 Data visualisation in line chart

To make the dotted line, change the code as follows

```
plt.plot(xpoints, ypoints , linestyle = 'dotted').
```

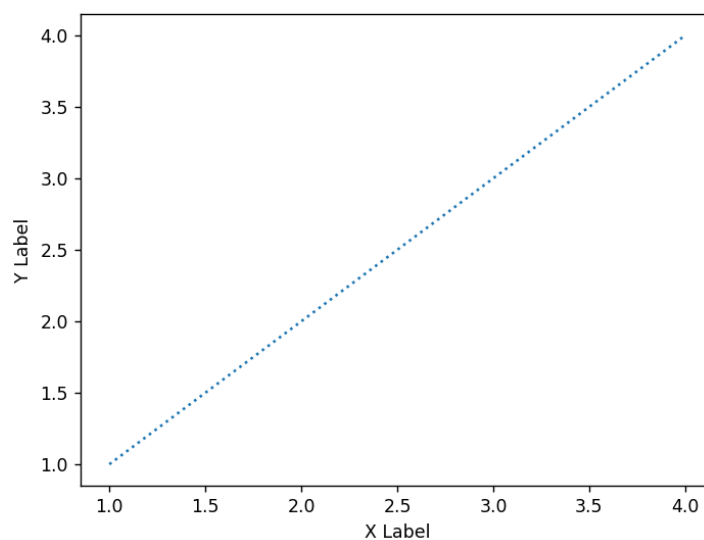


Fig: 4.3.6 Sample line chart with different line style

To display graph with grid lines

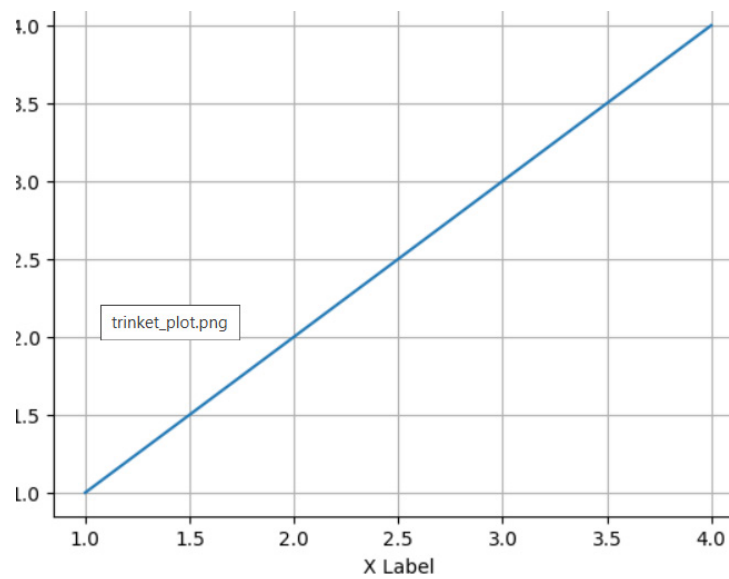


Fig 4.3.7 Visualisation with grid lines

### 4.3.5 Scatter graph

With Pyplot, use the scatter () function to draw a scatter plot.

#### Example

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([1,31,5,62,1,3, 4])
ypoints = np.array([11,32,52,6,1,3, 4])
plt.ylabel('Y Label')
plt.xlabel('X Label')
plt.title(' My Graph')
plt.grid()
plt.scatter(xpoints, ypoints , color = '#4000ff')
plt.show()
```

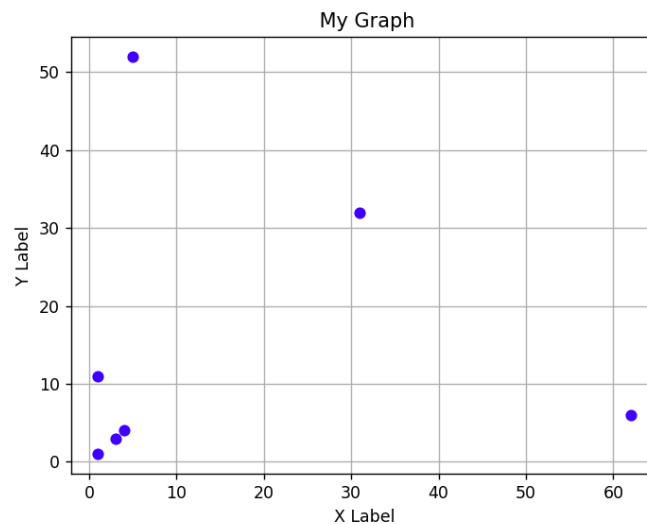


Fig 4.3.8 Scatter chart data visualisation

### 4.3.6 Bar Charts

Plt.bar() will generate a bar chart.

**Example,**

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([1,31,5,62,1,3, 4])
ypoints = np.array([11,32,52,6,1,3, 4])
plt.ylabel('Y Label')
plt.xlabel('X Label')
plt.title(' My Graph')
plt.grid()
plt.bar(xpoints, ypoints , color = '#4000ff')
plt.show()
```

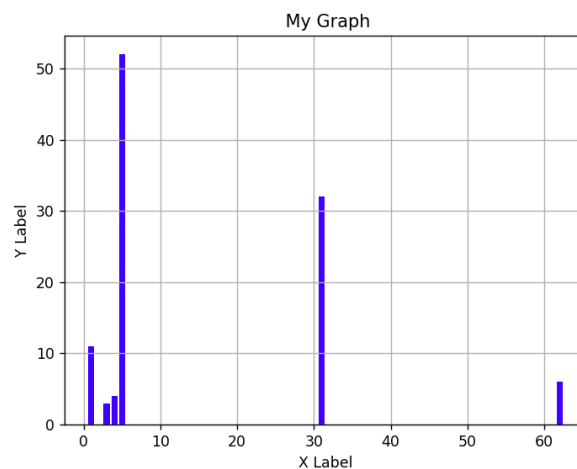


Fig: 4.3.9 Data visualisation for bar graph

If you want the bars to be displayed horizontally instead of vertically, use the `barh()` function:

**Example:**

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array(["An", "Bc", "Cs", "DD"])
y = np.array([38, 41, 12, 40])
plt.barh(x, y)
plt.show()
```

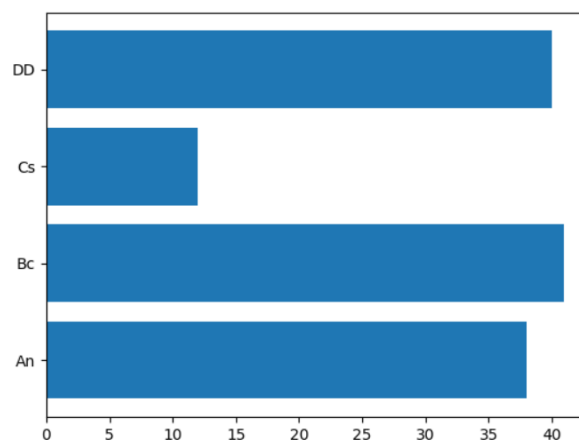


Fig 4.3.10 Horizontal bar graph data visualisation

### 4.3.7 Pie Chart

A pie chart is a circular statistical graphic image. It is used to represent a single series of data. The area of the chart represents the total percentage, with each slice showing a portion of the whole. The area of the chart represents the total percentage, with each slice showing a portion of the whole. Pie charts can be generated using `plt.pie()`. ( `plt` is variable name)

**Example**

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([10,31,5,62])
lab = ["Books", "Journals", "Magazines", "Daily"]
plt.title('My Pie Chart')
plt.pie(xpoints, labels =lab)
```

```
plt.show()
```



Fig 4.3.11 Data visualization in pie chart

Add Percentage Values (autopct)

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([10,31,5,62])
lab = ["Books", "Journals", "Magazines", "Daily"]
plt.title('My Pie Chart')
plt.pie(xpoints, labels =lab, autopct='%1.1f%%')
plt.show()
```



Fig 4.3.12 Data visualization in pie chart using percentage values

Exploding a Slice (explode)

```
import matplotlib.pyplot as plt
```

```

import numpy as np
xpoints = np.array([10,31,5,62])
exp = (0.2, 0, 0, 0) # Only "Books" slice exploded outward
lab = ["Books", "Journals", "Magazines", "Daily"]
plt.title('My Pie Chart')
plt.pie(xpoints, labels =lab, autopct='%1.1f%%',explode=exp)
plt.show()

```



Fig 4.3.13 Data visualization in pie chart using explode

Exercise 1: Plot a scatter plot for  $x = [5, 7, 8, 7, 2, 17, 2, 9, 4, 11]$   $y = [99, 86, 87, 88, 100, 86, 103, 87, 94, 78]$

Exercise 2: Create a bar chart for categories ['A1', 'B1', 'C1', 'D1'] with values [5, 7, 3, 8].

Exercise3: Plot a pie chart for the favorite fruits among people: Apples (30), Bananas (20), Cherries (25), Grapes (25). Use different colors and label each section.

## Recap

- ◆ Data science is the study of huge amounts of data to turn it into useful knowledge.
- ◆ Data visualisation makes it easier to understand numbers and patterns.
- ◆ Data analysis helps decision-makers find the right information at the right time.
- ◆ The data analysis lifecycle includes collecting, preparing, modeling, analyzing, presenting, and decision-making.
- ◆ Popular Python libraries for visualization are matplotlib, seaborn, and pandas.
- ◆ Data visualisation tools like Matplotlib help in creating graphs and charts.
- ◆ Matplotlib can create bar charts, histograms, line charts, pie charts, scatter plots, area plots etc.
- ◆ To install Matplotlib, use the command: `pip install matplotlib`.
- ◆ Import Matplotlib in Python using `import matplotlib.pyplot as plt`.
- ◆ The `plot()` function is used to create simple charts.
- ◆ The `show()` function is used to display the charts.
- ◆ Graphs can be customized by adding titles, labels, legends, grids, and colors.
- ◆ Markers are symbols used to highlight points in graphs, like circles or squares.
- ◆ Colors can be added by using character codes like 'r' for red or 'b' for blue.
- ◆ Line styles such as dotted, dashed, or solid can change the look of line charts.
- ◆ Scatter plots are created using the `scatter()` function to show relationships between two variables.
- ◆ Bar charts represent data with rectangular bars and can be made vertical or horizontal.
- ◆ Pie charts show the percentage share of different categories in a whole.
- ◆ Customizations like exploding slices, adding percentages, colors, and titles make pie charts more meaningful.
- ◆ In a pie chart, each slice represents a portion of the total data.
- ◆ The `autopct` parameter in pie charts is used to display percentage values on slices.

- ◆ The explode parameter in pie charts is used to separate one or more slices for emphasis.
- ◆ Customizing graphs helps make the information clearer and more attractive.
- ◆ Good visualisation turns complex data into simple, easy-to-understand stories.

## Objective Type Questions

1. What is the study of large volumes of data called?
2. What is used to remove errors and duplicates in data?
3. Which library is widely used for plotting in Python?
4. Which function is used to display a plot?
5. Which function is used to create a basic plot in matplotlib?
6. What do you add to a graph to display its main heading?
7. What feature adds background lines to a graph?
8. Which function is used to save a plot in a file?
9. What is the command to label the x-axis?
10. Which graph is used to display parts of a whole?
11. What is the Python command to add legends to a graph?
12. What keyword is used to set line styles like dotted or dashed?
13. What type of graph uses rectangular bars to represent data?
14. What function is used to create a scatter plot?
15. Which submodule of matplotlib is most commonly used for plotting?
16. What is used to explode a slice in a pie chart?
17. What command is used to display percentage values in pie charts?
18. Which graph is used to show trends over time?
19. Which library provides a high-level interface for statistical graphs?
20. What type of visualization shows the relationship between two variables using points?

## Answers to Objective Type Questions

1. Data Science
2. Cleaning
3. Matplotlib
4. show()
5. plot()
6. Title
7. Grid
8. savefig()
9. xlabel()
10. Pie chart
11. Legend ()
12. linestyle
13. Barchart
14. Scatter ()
15. pyplot
16. explode
17. autopct
18. Linechart
19. Seaborn
20. Scatterplot

## Assignments

1. Define data visualization. Why is it important in data science? Explain with a suitable example.
2. Write a Python program using Matplotlib to plot a line chart with custom labels, grid, title, and color.

3. Create a pie chart using Matplotlib to show the distribution of marks in 4 subjects. Include labels, percentage values, and explode the largest slice.
4. Write a program to display a scatter plot for student height vs weight with proper axis labels and title.
5. Explain the use of the following Matplotlib functions with syntax and output: plot(), xlabel(), ylabel(), title(), show()

## References

1. Robert Johansson, 2nd Edition (2019), “Numerical Python: A Practical Technique Approach for Industry”.
2. Joel Grus, 2nd Edition (2019), “Data Science from Scratch: First Principles with Python”.

## Suggested Reading

1. Matthes, E. (2019). Python Crash Course: A Hands-On, Project-Based Introduction to Programming (2nd ed.). No Starch Press.
2. Bader, D. (2017). Python Tricks: A Buffet of Awesome Python Features. Dan Bader Press.
3. Ramalho, L. (2015). Fluent Python: Clear, Concise, and Effective Programming. O'Reilly Media.
4. Real Python - Functions: <https://realpython.com/tutorials/functions/>



## Introduction to Pandas

### Learning Outcomes

After completing this unit, you will be able to:

- ◆ identify the main data structures used in Pandas.
- ◆ recognise the difference between a Pandas Series and DataFrame.
- ◆ recall the Pandas functions used to import, export, and manipulate real-world datasets.
- ◆ identify the correct syntax for creating a DataFrame from a dictionary.
- ◆ name the function used to handle missing values in Pandas.
- ◆ list advantages and limitations of using Pandas with large datasets.

### Prerequisites

Studying Pandas is crucial for students interested in data science. In this chapter, explore the powerful capabilities of Pandas, a library in Python designed for efficient data analysis and manipulation. Pandas makes working with structured data and providing versatile data structures like Series and DataFrame. These structures allow us to handle, clean, and analyze data efficiently. In this chapter also discussed Panel Data, the three-dimensional structure used to analyze multi-dimensional data, which was particularly useful in business and longitudinal studies. Pandas is a powerful tool for organizing, cleaning, and analyzing large datasets. In real-world scenarios, data is often unstructured, so it is difficult to analyze the data. Pandas provides functions that make it easy to clean and transform data into usable formats. For example, a business needs to clean up customer data by removing duplicates or filling in missing information. Pandas allows this task to be done more efficiently.

Before diving into Pandas, students should have a basic understanding of Python programming. It is important to understand the basics of Python data structures, such as lists, dictionaries, and tuples. Basic skills like calculating averages, percentages, and working with simple algebra will make it easier to interpret and manipulate data. Raw data often contains errors such as missing values, duplicates, or inconsistent formatting. Data cleaning is an essential step in the data analysis process. Clean data ensures that the

analysis or model built is accurate and reliable. If there are missing values, duplicates, or incorrect entries in the dataset, it can lead to faulty conclusions. For example, if a sales report has repeated entries for the same transaction, it could overestimate the total sales and lead to incorrect business decisions. So, data cleaning is necessary to ensure that the data used for analysis is accurate, consistent, and useful.

Consider a scenario where a retail company wants to analyse the sales data. The dataset includes columns like product name, quantity sold, sales price, and region. The data might have missing values, duplicates, or errors in certain entries. Using Pandas,

- ◆ Import the dataset from a CSV file.
- ◆ Clean the data by removing duplicates or filling in missing values.
- ◆ Group the data by region or product category to analyse trends.
- ◆ Generate visualizations to show the sales trends over time.

This hands-on approach to solving real-world problems makes learning Pandas highly relevant and practical. It allows students to develop the skill in data science, business analysis, and other fields of decision-making.

## Key words

Data integrity, Data cleaning, Duplication, Missing data, Visualization, Cross-sectional Data, Time Series Analysis, Data Standardization

## Discussion

Pandas is a powerful, high-level data manipulation tool extensively used in data analysis. It is designed to simplify the process of working with structured data. It offers a wide range of built-in functions for importing, exporting, and processing datasets. It also provides a convenient and unified interface for performing both data analysis and visualization tasks. One of the key strengths of Pandas is its well-defined data structures, which include Series (a one-dimensional labeled array), DataFrame (a two-dimensional table with labeled rows and columns), and Panel (a three-dimensional structure, though now deprecated in recent versions). In pandas, these data structures help to make the data analysis more organized, efficient, and effective, especially when dealing with large or complex datasets.

### 4.4.1 Panel panda

In Pandas, a Panel was used to store and work with three-dimensional data. The three axes had special names that helped to understand the data analysis easier. This was especially helpful when working with panel data, often used in business analysis. Panel

(pandas) is the old 3D data structure in pandas. Panel data is the dataset style of the same subjects tracked across time.

#### 4.4.1.1 Panel data

Panel data is a dataset that contains observations of multiple entities (such as individuals, companies, or countries) measured over multiple time periods. Panel data is sometimes referred to as longitudinal data. Panel data contains observations about different cross-sections. The name pandas originated from Panel data and Python Data Analysis. Pandas let us analyze big data and generate conclusions based on various theories. Using pandas, we can find the correlation, average, minimum, maximum and more. The following are examples of panel data.

- ◆ Unemployment across different states
- ◆ Stock prices by the firm
- ◆ GDP across multiple countries
- ◆ Income dynamic studies
- ◆ International current account balances

Table 4.4.1 How Panel data is different from other data types

Type of Data	What It Means	Example
<b>Cross-sectional</b>	Data collected once from many subjects	Income of 100 people in 2025
<b>Time series</b>	Data collected over many time points from one subject	Stock price of Apple from 2020–2025
<b>Panel data</b>	Data collected over time from many subjects	Incomes of 100 people from 2020–2025

#### 4.4.2 Advantages of pandas

##### 1. Easy data handling

Pandas provides a simple and built-in interface that makes it easy to handle, clean, and transform data. This simplicity makes it accessible to both beginners and advanced users, ensuring that users of all skill levels can effectively work with data.

##### 2. Powerful data analysis tools

It offers comprehensive tools for data manipulation, such as filtering, grouping, merging, and aggregation. These tools make complex data analysis tasks easier.

##### 3. Support for various file formats

Pandas can read from and write to numerous data formats, including CSV, Excel, SQL databases, JSON, and HDF5. This allows a seamless amount of data import and export.

##### 4. Labelled data structures

Pandas uses Series and DataFrame objects, which support labelled axes (rows and columns), improving data organization and access.

## 5. Handling missing data

It provides efficient methods for detecting, filling, or removing missing or null values in datasets.

## 6. High performance

Pandas is built on top of NumPy, ensuring high performance for operations on large datasets.

## 7. Time series support

Pandas has strong support for time-series data. It can easily handle date and time indexing. It also allows users to resample data and analyse it over different time intervals.

## 8. Compatibility with other libraries

Pandas works well with a variety of other Python libraries, such as NumPy, SciPy, scikit-learn, TensorFlow etc.

### 4.4.3 Installation

To Install pandas using the following command:

```
pip install pandas
```

Once Pandas is installed, import it in your applications by adding the import keyword:

```
import pandas
```

```
import pandas as pd    #pd is alias to the pandas library
```

### 4.4.4 Data Structures in Pandas

In Pandas, a data structure refers to a specialized format for organizing, storing, and managing data efficiently. These structures are designed to facilitate easy data manipulation, analysis, and visualization. The two primary data structures provided by Pandas are Series and DataFrame.

#### 1. Series

A Series is a one-dimensional array-like object that can hold data of any type (integers, strings, floats, etc.).

##### Example:

```
import pandas as pd
```

```
s = pd.Series([10, 20, 30, 40])    # Creating a Series from a list
```

```
print(s)
```

Example with custom index:

```
import pandas as pd
```

```
data = [100, 200, 300]
```

```
i = ['a', 'b', 'c']
```

```
s = pd.Series(data, index=i)
print(s)
```

## 2. DataFrame

DataFrame is a two-dimensional labeled data structure with columns that can contain different data types.

### Example:

```
import pandas as pd
data = {
    'Name': ['Ann', 'Bobb', 'Chat'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data) # Creating a DataFrame from a dictionary
print(df)
```

## 3. Read CSV Files

A simple way to store big data sets is to use CSV files (comma separated files).

CSV files contain plain text and is a well known format that can be read by everyone including Pandas. The `read_csv()` method of pandas library is used to read data from CSV files.

```
import pandas as pd
df = pd.read_csv('data1.csv')    #create a sample csv file
```

### Steps to create a csv file

Open **Notepad** or any text editor

Type your data like this:

```
Name, Age, City
John, 25, New York
Alice, 30, Los Angeles
Bob, 22, Chicago .....
```

Save the file  
with a **.csv** extension.

```
print(df.to_string())    #use to_string() to print the entire DataFrame.
```

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
5	60	102	127	300.5
6	60	110	136	374.0
7	45	104	134	253.3
8	30	109	153	195.1
9	60	98	124	269.0
10	60	103	147	329.3
11	60	100	120	250.7
12	60	106	120	345.3
13	60	104	132	379.3
14	60	98	123	275.0
15	60	98	120	215.2
16	60	100	120	300.0
17	45	90	112	NaN
18	60	105	123	323.0
19	45	97	125	243.0
20	60	108	131	364.2

Fig 4.4.1 Sample of a CSV file with complete data

```
import pandas as pd
```

```
df = pd.read_csv('data.csv')
```

```
print(df) #If you have a large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows:
```

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
..	...	...	...	...
164	60	105	140	290.8
165	60	110	145	300.4
166	60	115	145	310.2
167	75	120	150	320.4
168	75	125	150	330.4

[169 rows x 4 columns]

Fig 4.4.2 Sample of a CSV file with specific rows and columns

### 4.4.5 Data Cleaning

While collecting data there are chances to have invalid data or missing data. For example, the age of a student may be entered as a string instead of numbers. The data cleaning process will eliminate such errors. Wrong data will provide incorrect or incomplete output. The wrong or bad data could be

- ◆ Empty cells
- ◆ Invalid data format
- ◆ Duplicate values
- ◆ Wrong data

#### 4.4.5.1 Advantages of data cleaning

Data cleaning is a crucial process in data management that ensures datasets are accurate, reliable, and ready for analysis. The following figure illustrates five major benefits of data cleaning:



Fig 4.4.3 Key benefits of data cleaning

1. **Error-Free Data:** Cleaning helps to remove incorrect, duplicate, or corrupted data.
2. **Data Quality:** Cleaned data ensuring the data is meaningful, valid, and useful. It enhances the overall quality of data,
3. **Accuracy and Efficiency:** Well-prepared data improves the accuracy of results and increases the efficiency of data processing tasks.
4. **Complete Data:** Data cleaning helps in filling missing values and removing incomplete entries, it ensures the dataset is comprehensive.
5. **Maintains Data Consistency:** Consistent data formats and standardized entries across datasets are achieved through cleaning the data.

#### 4.4.5.2 Data Cleaning life cycle



Fig 4.4.4 Data Cleaning cycle

The data cleaning cycle outlines the structured steps for data analysis. Each step in the cycle plays a crucial role in ensuring data integrity:

1. **Import Data:** Raw data is collected from various sources and loaded into the system for processing.
2. **Merge Datasets:** Data from multiple sources are combined to form a unified dataset.
3. **Rebuild Missing Data:** Techniques such as imputation or referencing other sources are used to fill in missing values.
4. **Standardize:** Data values are reformatted to a common structure ensuring uniformity.
5. **Normalize:** Redundant or inconsistent data is adjusted for scale and format.
6. **De-Duplicate:** Duplicate records are identified and removed to avoid redundancy and enhance accuracy.
7. **Verify and Enrich:** The data is validated for accuracy and may be enhanced with additional relevant information.
8. **Export Data:** The cleaned data is exported for further analysis, reporting, or storage.

#### 4.4.5.3 Pandas Data Cleaning

##### 1. Drop rows with missing values

In Pandas, drop rows with missing values using the `dropna()` function.

```
DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
```

### Syntax:

Table 4.4.2 Dataframe dropna() properties

Properties	Explanations
axis	Determines if you want to drop rows (axis=0) or columns (axis=1). Default: 0 (rows)
how	<ul style="list-style-type: none"><li>◆ Decides whether to drop based on presence of any or all missing values in the row/column.</li><li>• 'any': Drop if any value is missing.</li><li>• 'all': Drop only if all values are missing.</li></ul>
thresh	Minimum number of non-NA values required to keep the row/column. Overrides how if set. Example: thresh=3 → keep only rows/columns with at least 3 non-null entries.
subset	A list of column labels to consider when checking for missing values. Other columns are ignored.
inplace	If True, modifies the original DataFrame in place. If False, returns a new DataFrame.

### Example:

```
import pandas as pd

d = {
    'A1': [1, 2, 3, None, 5],
    'B1': [None, 2, 3, 4, 5],
    'C1': [1, 2, None, None, 5]
}

df = pd.DataFrame(d)
print("Original Data:\n", df)
print()

df_cleaned = df.dropna() # use dropna() to remove rows with any missing values
print("Cleaned Data:\n", df_cleaned)
```

**Output:****Original Data:**

	A1	B1	C1
0	1.0	NaN	1.0
1	2.0	2.0	2.0
2	3.0	3.0	NaN
3	NaN	4.0	NaN
4	5.0	5.0	5.0

**Cleaned Data:**

	A1	B1	C1
1	2.0	2.0	2.0
4	5.0	5.0	5.0

**1. Fill Missing Values**

To fill the missing values in Pandas, we use the `fillna()` function.

**Example:**

```
import pandas as pd

d = {
    'A1': [1, 2, 3, None, 5],
    'B1': [None, 2, 3, 4, 5],
    'C1': [1, 2, None, None, 5]
}

df = pd.DataFrame(d)

print("Original Data:\n", df)

df.fillna(0, inplace=True)  # filling NaN values with 0

print("\nData after filling NaN with 0:\n", df)
```

**Output:**

Original Data:

```

      A1  B1  C1
0  1.0 NaN  1.0
1  2.0  2.0  2.0
2  3.0  3.0 NaN
3  NaN  4.0 NaN
4  5.0  5.0  5.0

```

Data after filling NaN with 0:

```

      A1  B1  C1
0  1.0  0.0  1.0
1  2.0  2.0  2.0
2  3.0  3.0  0.0
3  0.0  4.0  0.0
4  5.0  5.0  5.0

```

Instead of filling with 0, use aggregate functions to fill missing values.

### Example:

```

import pandas as pd

d = {
    'A1': [1, 2, 3, None, 5],
    'B1': [None, 2, 3, 4, 5],
    'C1': [1, 2, None, None, 5]
}

df = pd.DataFrame(d)

print("Original Data:\n", df)

df.fillna(df.mean(), inplace=True) # filling NaN values with the mean of each column

print("\nData after filling NaN with mean:\n", df)

```

### Output:

```
Original Data:
   A1  B1  C1
0  1.0 NaN  1.0
1  2.0  2.0  2.0
2  3.0  3.0 NaN
3  NaN  4.0 NaN
4  5.0  5.0  5.0

Data after filling NaN with mean:
   A1  B1  C1
0  1.00  3.5  1.000000
1  2.00  2.0  2.000000
2  3.00  3.0  2.666667
3  2.75  4.0  2.666667
4  5.00  5.0  5.000000
```

Fig 4.4.5 Sample of output screen

The following program is an example of data cleaning and visualization by making a plot

```
import pandas as pd
import matplotlib.pyplot as plt

dataframe = pd.read_csv('data1.csv')    #create a sample of csv file
dataframe.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)

print(dataframe.to_string())

dataframe.plot()

plt.show()
```

### 4.4.6 Converting data type

Pandas has many built-in functions for converting the data types. The following are example functions used in pandas to convert data

`to_datetime`

Convert argument to datetime.

`to_timedelta`

Convert argument to timedelta.

`to_numeric`

Convert argument to a numeric type.

`numpy.ndarray.astype`

Cast a numpy array to a specified type.

The following program converts date to numeric format.

```
import pandas as pd
dataframe = pd.read_csv('weather.csv')
dataframe['Date'] = pd.to_numeric(dataframe['Date'])
print(dataframe.to_string())
```

#### Output

	Date	Temperature	Humidity
0	20240401	28	60
1	20240402	30	55
2	20240403	31	58

Fig 4.4.6 Sample of output screen

#### 4.4.7 Printing duplicate values

The following code will check and print duplicate values from the 'data.csv' file. Returns True for every row that is a duplicate, otherwise False

```
import pandas as pd
import matplotlib.pyplot as plt
dataframe = pd.read_csv('data.csv')    #create a sample csv file
print(dataframe.duplicated())
```

#### Output:

```
>>>
===== RESTART: C:/Python/plt5.py =====
0      False
1      False
2      False
3      False
4      False
...
722    False
723    False
724    False
725    False
726    False
Length: 727, dtype: bool
```

Fig 4.4.7 Duplicate values

The following code will remove duplicate values

```
import pandas as pd
import matplotlib.pyplot as plt
dataframe = pd.read_csv('data1.csv')
print(dataframe.duplicated())
print(dataframe.drop_duplicates()) # Duplicate values removed from the data set.
```

## Recap

- ◆ Pandas is a popular Python library used for working with structured data.
- ◆ It is very helpful for data analysis and data science projects.
- ◆ Pandas makes it easy to read, write, and process data.
- ◆ It supports many file formats like CSV, Excel, SQL, and JSON.
- ◆ The main data types in pandas are Series, DataFrame, and Panel.
- ◆ A Series is a one-dimensional data structure like a single column.
- ◆ A DataFrame is a two-dimensional structure like a table with rows and columns.
- ◆ Panel is a three-dimensional structure, but it is no longer used in newer versions of pandas.
- ◆ Panel data means data collected over time from many people or groups.
- ◆ Examples of panel data include stock prices of companies and GDP of countries over years.
- ◆ To read CSV files using the read\_csv() function.
- ◆ To see the full data, use to\_string() to print all rows.
- ◆ Sometimes data may have missed or incorrect values.
- ◆ To remove rows with missing values using dropna().
- ◆ To fill missing values using fillna() with a number or a function like mean.
- ◆ To check for duplicate rows, we use the duplicated () function.
- ◆ To remove duplicates, use drop\_duplicates().
- ◆ To change data types using functions like to\_numeric() and to\_datetime().
- ◆ To draw graphs and charts using plot () and the matplotlib library.

- ◆ Cleaning data is important to get correct results during analysis.
- ◆ Clean data has no missing values, no wrong formats, and no duplicate rows.
- ◆ Data cleaning makes data more accurate, complete, and easy to use.
- ◆ Pandas is also fast and works well with other Python libraries like NumPy and scikit-learn.
- ◆ Pandas is a very useful tool for both students and professionals working with data.

## Objective Type Questions

1. Pandas stands for
2. Important library used for analyzing data
3. What are the two main data structures in Pandas?
4. What is a Series in Pandas?
5. What is a DataFrame in Pandas?
6. What was the three-dimensional data structure in Pandas called?
7. What command installs Pandas in Python?
8. Which Pandas method is used to display the first few rows of a DataFrame?
9. Which Pandas method shows the last few rows?
10. Which function finds the maximum value in a DataFrame column?
11. Which function finds the minimum value in a DataFrame column?
12. How do you check for missing data in Pandas?
13. Which method is used to remove missing values?
14. Which method fills missing values in Pandas?
15. What does `drop_duplicates()` do?
16. What type of data does Panel data represent?
17. What is cross-sectional data?
18. What is time series data?
19. What does the subset parameter in `dropna()` specify?

20. What is the use of the thresh parameter in dropna()?
21. How does Pandas handle empty cells by default?
22. Which function reads Excel files in Pandas?

## Answers to Objective Type Questions

1. Panel Data
2. Panda
3. Series and DataFrame.
4. A one-dimensional labeled array.
5. A two-dimensional labeled data structure with rows and columns.
6. Panel.
7. pip install pandas.
8. head().
9. tail().
10. max().
11. min().
12. isnull().
13. dropna().
14. fillna().
15. Removes duplicate rows.
16. Data collected over time from multiple subjects.
17. Data collected once from many subjects.
18. Data collected over time from a single subject.
19. Specific columns to check for missing values.
20. Sets the minimum number of non-NA values to keep a row.
21. Represents them as NaN (Not a Number).
22. read\_excel().

## Assignments

1. What are the advantages of using Pandas for data handling compared to traditional spreadsheet software like Excel? List five points with examples.
2. Discuss the key steps involved in the Data cleaning life cycle.
3. Define and explain the properties of the `dropna()` function in Pandas.
4. Differentiate between `drop_duplicates()` and `duplicated()` methods in Pandas. Write an example showing how they can be combined for data cleaning.
5. Create a Pandas DataFrame that simulates unemployment rates across five different states over three years. Perform the following:
  - ◆ Introduce missing values manually.
  - ◆ Clean the data by filling missing values with the column mean.

## References

1. VanderPlas, J. (2016). Python Data Science Handbook: Essential Tools for Working with Data. O'Reilly Media.
2. McKinney, W. (2017). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (2nd ed.). O'Reilly Media.
3. Chen, D. (2020). Pandas for Everyone: Python Data Analysis. Addison-Wesley Professional.

## Suggested Reading

1. McKinney, W. (2018). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (2nd ed.). O'Reilly Media.
2. The Pandas Development Team. (n.d.). Pandas documentation. Retrieved from <https://pandas.pydata.org/docs/>
3. Molin, S. (2019). Hands-On Data Analysis with Pandas: Efficiently perform data collection, wrangling, analysis, and visualization using Python. Packt Publishing.
4. Harrison, M. (2021). Effective Pandas: Patterns for Data Manipulation. MetaSnake.



**SREENARAYANAGURU OPEN UNIVERSITY**

QP CODE: .....

Reg. No : .....

Name : .....

**Model Question Paper- set-I**

End Semester Examination

BACHELOR OF COMPUTER APPLICATIONS

B21CA07DC: PROGRAMMING WITH PYTHON

(CBCS - UG)

2024-25 - Admission Onwards

Time: 3 Hours

Max Marks: 70

---

**Section A**

*Answer any 10 questions. Each carries one mark (10×1= 10)*

1. Which keyword is used to define a variable in Python?
2. What is the result of the expression `5 // 2` in Python?
3. Write any two escape sequences used in `print()`.
4. Write any four built in methods of List.
5. What do you call the process of hiding internal details and showing only functionality?
6. Which keyword is used to handle exceptions in Python?
7. \_\_\_\_\_ is an interface for connecting to a MYSQL database server from Python.
8. Which method is used to insert an element into a NumPy array?
9. What control statement can be used to create empty blocks?
10. Which symbols are used to define dictionary comprehensions?
11. Which mode is used to write to a file and overwrite existing content?
12. Which block is executed if no exception occurs?



13. What is the purpose of a compiler?
14. Write down any two logical operators in Python.
15. How to store multiple values using a single variable name?

### Section B

*Answer any 5 questions. Each carries two marks (5×2=10)*

16. What is the difference between a list and a tuple in Python?
17. What is a set in python? Write any four built in methods of set.
18. Explain the use of the seek() and tell() functions in file handling.
19. Predict the result:

```
a = np.array([[1,2],[3,4]])  
b = np.array([[5,6],[7,8]])  
  
print(a + b)
```

20. Differentiate Local and Global variables.
21. Write a Python program to create a file, write some content into it?
22. What are Anonymous Functions?
23. What is the difference between a module and a package in Python? Give one example of each.
24. Write about the concept of encapsulation.
25. Write the SQL command to display all the tables in a selected database using Python.

### Section C

*Answer any 5 questions. Each carries four marks (5 x 4 = 20)*

26. Write the advantages of Functions.
27. Write about Numeric data types.
28. What is polymorphism in Python? Explain with a suitable example showing both method overloading and method overriding.

29. Compare bar chart and pie chart with examples of when to use each.
30. Explain the different types of comprehensions in python.
31. Define inheritance and explain the types of inheritance supported in Python.
32. What is an arithmetic operator? Write about the different arithmetic operators in Python.
33. Explain any two built-in methods of Dictionary in Python with examples.
34. Write the syntax and example for if else and elif ladder.
35. Explain the different loop control statements.

### Section D

***Answer any 2 questions. Each carries fifteen mark (2 x 15 = 30)***

36. Explain in detail about creating, calling and passing arguments to functions.
37. Write a detailed note on data cleaning in Pandas.
38. Explain exception handling in Python, including the difference between errors and exceptions, the use of try-except blocks, else and finally blocks, raising exceptions, and common built-in exception types with examples.
39. What is Object-Oriented Programming (OOP)? Explain how Python implements OOP concepts with suitable examples.



**SREENARAYANAGURU OPEN UNIVERSITY**

QP CODE: .....

Reg. No : .....

Name : .....

**Model Question Paper- set-II**

End Semester Examination

BACHELOR OF COMPUTER APPLICATIONS

B21CA07DC: PROGRAMMING WITH PYTHON

(CBCS - UG)

2024-25 - Admission Onwards

Time: 3 Hours

Max Marks: 70

---

**Section A**

*Answer any 10 questions. Each carries one mark*      **(10×1= 10)**

1. Who developed the Python programming language?
2. Which Python numeric type includes real and imaginary parts?
3. What is the condition in bitwise OR to return 1?
4. What is the use of 'in' operator check in Python?
5. What does NumPy stand for?
6. Which keyword is used to define a function in Python?
7. What does OOP stand for?
8. What operator will be used to check if 15 is not equal to 10.
9. What is a module in Python?
10. Which function is used to save a plot in a file?
11. Which operator is used to check membership in a sequence in Python?
12. The process of correcting or removing inaccurate, incomplete, or irrelevant data in Python is called .....



13. Which function in Python is used to replace a pattern with a string?
14. Which block is executed if no exception occurs?
15. What is the tuple method to count the repeated values?

### **Section B**

***Answer any 5 questions. Each carries two marks (5×2=10)***

16. What is the purpose of the range() function in loops? Write a simple example using range().
17. How to create a file using the write() method?
18. What is data visualization?
19. Mention any two common code review methods.
20. Write any four variables naming rules in python.
21. What is array indexing in NumPy? Give an example.
22. Write any two set built in methods.
23. What is a package in Python? How is it different from a module?
24. List the different file access modes.
25. Define regular expressions.

### **Section C**

***Answer any 5 questions. Each carries four marks (5 x 4 = 20)***

26. Explain the role of configuration management and deployment tools in software deployment.
27. Explain about the below listed operators in python with examples.
  - a) Logical Operators
  - b) Bitwise Operators
  - c) Membership Operators
  - d) Identity Operators
28. Explain about numeric and Sequence Data Types.



29. Explain the stages of the Data Analysis Life Cycle in detail.
30. Explain any five important Matplotlib functions with their syntax and examples.
31. What is list comprehension in Python? Write a program to create a list of even numbers between 1 and 20 using list comprehension. Also explain how it differs from a traditional loop.
32. What is data abstraction? How does it differ from encapsulation?
33. Explain list indexing and slicing with examples.
34. Compare scatter graph and bar chart with suitable use cases.
35. What is data cleaning, and what are its benefits?

### Section D

***Answer any 2 questions. Each carries fifteen mark (2 x 15 = 30)***

36. Explain the working of the following decision-making statements in Python with examples:
  - i. if statement
  - ii. if-else statement
  - iii. elif statement
37. Explain the different data types in Python with examples.
38. Define inheritance and explain the types.
39. What are the data structures provided by the Pandas library in Python? Explain their advantages and show how to implement them with suitable examples.

സർവ്വകലാശാലാഗീതം

വിദ്യായാൽ സ്വതന്ത്രരാകണം  
വിശ്വപൗരരായി മാറണം  
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം  
ഗുരുപ്രകാശമേ നയിക്കണേ

കുതിരുട്ടിൽ നിന്നു ഞങ്ങളെ  
സൂര്യവീഥിയിൽ തെളിക്കണം  
സ്നേഹദീപ്തിയായ് വിളങ്ങണം  
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം  
ജാതിഭേദമാകെ മാറണം  
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ  
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരിപ്പുഴ ശ്രീകുമാർ

# SREENARAYANAGURU OPEN UNIVERSITY

## Regional Centres

### Kozhikode

Govt. Arts and Science College  
Meenchantha, Kozhikode,  
Kerala, Pin: 673002  
Ph: 04952920228  
email: rckdirector@sgou.ac.in

### Thalassery

Govt. Brennen College  
Dharmadam, Thalassery,  
Kannur, Pin: 670106  
Ph: 04902990494  
email: rctdirector@sgou.ac.in

### Tripunithura

Govt. College  
Tripunithura, Ernakulam,  
Kerala, Pin: 682301  
Ph: 04842927436  
email: rcedirector@sgou.ac.in

### Pattambi

Sree Neelakanta Govt. Sanskrit College  
Pattambi, Palakkad,  
Kerala, Pin: 679303  
Ph: 04662912009  
email: rcpdirector@sgou.ac.in

**DON'T LET IT  
BE TOO LATE**

**SAY  
NO  
TO  
DRUGS**

**LOVE YOURSELF  
AND ALWAYS BE  
HEALTHY**



**SREENARAYANAGURU OPEN UNIVERSITY**

The State University for Education, Training and Research in Blended Format, Kerala



# PROGRAMMING WITH PYTHON

COURSE CODE: B21CA07DC



Sreenarayanaguru Open University

Kollam, Kerala Pin- 691601, email: [info@sgou.ac.in](mailto:info@sgou.ac.in), [www.sgou.ac.in](http://www.sgou.ac.in) Ph: +91 474 2966841

ISBN 978-81-989642-7-4



9 788198 964274