SGOU
Higher Education for All

PYTHON PROGRAMMING

COURSE CODE: M25CA03DC
Master 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.

Python Programming

Course Code: M25CA03DC

Semester - I

Discipline Core Course
Postgraduate Programme
Master of Computer Applications
Self Learning Material

SREENARAYANAGURU OPEN UNIVERSITY

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

SREENARAYANAGURU
OPEN UNIVERSITY

PYTHON PROGRAMMING

Course Code: M25CA03DC
Semester- I
Discipline Core Course
Master of Computer Applications

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

Edition
April 2026

ISBN 978-81-998406-6-9

9 788199 840669

www.sgou.ac.in

Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear Learner,

It gives me immense pleasure and a deep sense of pride to warmly welcome you to Sreenarayanaguru Open University—a vibrant and progressive institution committed to transforming lives through inclusive, flexible, and high-quality education.

Established in September 2020 as a forward-looking initiative of the Government of Kerala, the University stands as a beacon of opportunity for learners seeking to advance their academic and professional aspirations through the open and distance learning mode. Guided by our foundational principle that "access and quality define equity," we are steadfast in our mission to democratize education while upholding uncompromising academic standards.

Our university is inspired by the timeless vision and philosophy of Sree Narayana Guru, whose ideals of knowledge, equality, and social transformation continue to guide our academic journey. His enduring legacy instils in us the responsibility to create an educational environment that empowers individuals, nurtures critical thinking, and contributes meaningfully to society.

Understanding the dynamic needs of contemporary learners, we have adopted a robust and learner-centric blended learning model, seamlessly integrating Self-Learning Materials, Academic Counselling, and Advanced Digital Learning Platforms. This holistic approach ensures flexibility without sacrificing academic depth, enabling you to learn at your own pace while remaining meaningfully connected to a vibrant academic ecosystem.

The Master of Computer Applications (MCA) programme you are embarking upon is carefully designed to position you at the forefront of the digital revolution. It uniquely blends strong theoretical foundations with practical, industry-oriented competencies. The curriculum emphasizes algorithmic thinking, system design, programming, database management, networking, and emerging technologies such as artificial intelligence, data science, and cloud computing.
What sets this programme apart is its forward-thinking design, which offers:

- Opportunities for skill enhancement aligned with industry needs
- Multidisciplinary learning pathways for broader intellectual development
- Exposure to innovative and emerging technology domains
- Multiple specialization options tailored to evolving career landscapes
- A strong focus on employability, entrepreneurship, and global career readiness
-

Our Self-Learning Materials are meticulously developed by experts, enriched with contemporary case studies, real-world applications, and practical insights to ensure clarity, engagement, and relevance. We are committed to equipping you not just with knowledge, but with the confidence and competence to excel in a competitive global environment.

At Sreenarayanaguru Open University, you are never alone in your learning journey. Our dedicated learner support system is designed to provide continuous academic guidance, timely assistance, and effective grievance redressal. We encourage you to actively engage with us, share your concerns, and make the most of the resources and support available to you.

As you begin this important phase of your academic journey, I urge you to embrace learning with curiosity, discipline, and determination. The world of technology is ever-evolving, and your willingness to adapt, innovate, and grow will define your success.

Remember, this is not just a programme—it is a pathway to transforming your future.
I wish you a fulfilling learning experience and a successful, inspiring career ahead.

Warm regards,

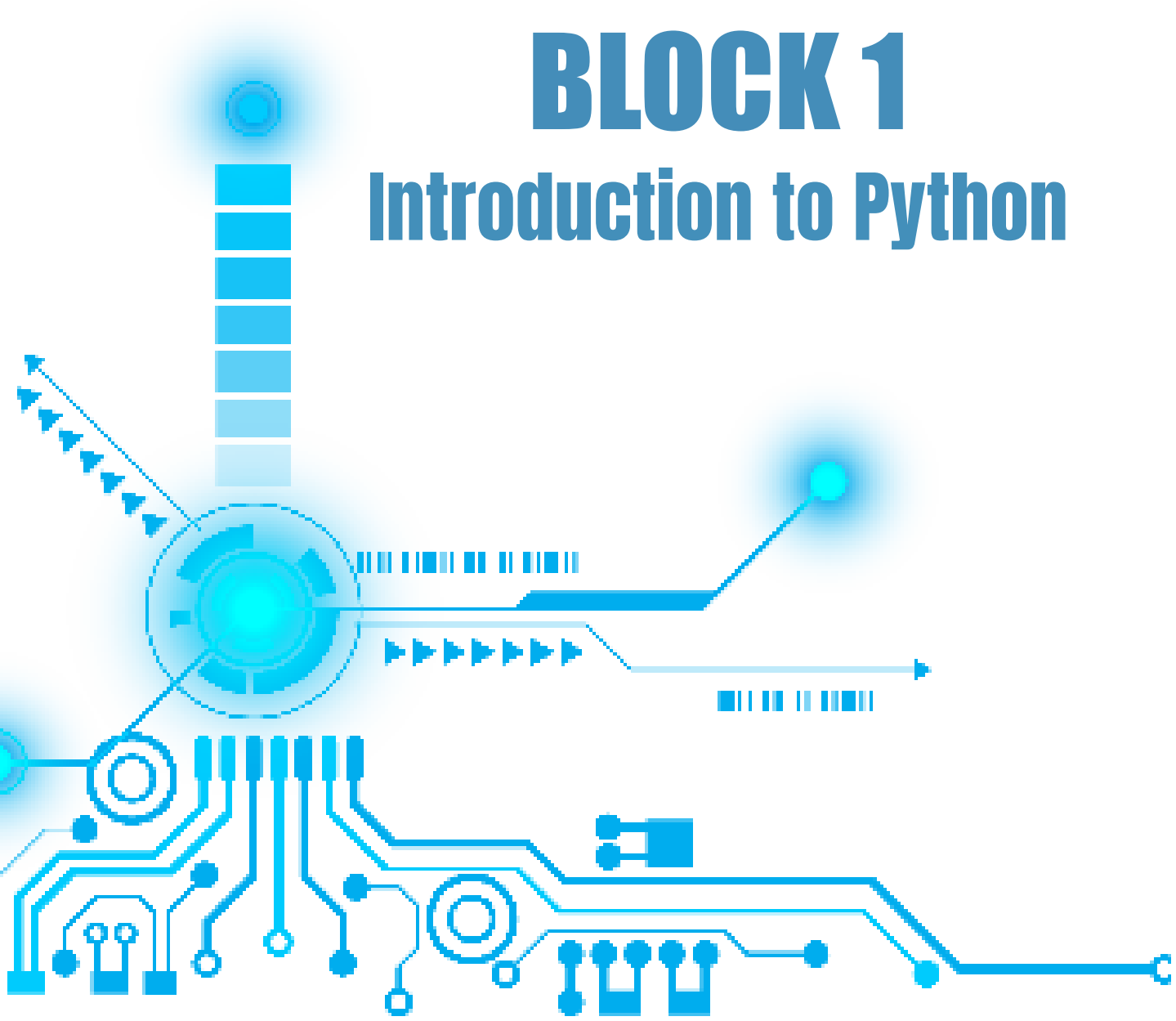
Prof. (Dr.) Jagathy Raj V. P.
Vice Chancellor

Contents

Block 01	**Introduction to Python**	**1**
Unit 1	Introduction to Python	2
Unit 2	Data Structures in Python and Built-in Methods of Data Structures	26
Unit 3	Functions	49
Unit 4	Introduction to Python Libraries	79
Block 02	**Object-Oriented Programming, File Handling, Error Handling and Database**	**92**
Unit 1	Object-Oriented Programming	93
Unit 2	File Handling	118
Unit 3	Exception Handling	146
Unit 4	Iterators	157
Block 03	**Libraries for Data and Web Applications**	**168**
Unit 1	Numpy	169
Unit 2	Pandas	190
Unit 3	Data Visualization	211
Unit 4	Web Programming with Flask	233
Block 04	**Python for Real-World**	**276**
Unit 1	Working with Databases	277
Unit 2	Introduction to Django	289
Unit 3	Working with APIs and JSON	299
Unit 4	Project Development	313
Model Question Paper Sets		**327**

BLOCK 1

Introduction to Python



1 UNIT Introduction to Python

Learning Outcomes

After completion of this unit, the learner will be able to:

- understand Python data types, variables, and constants
- use operators to perform arithmetic, comparison, and logical operations
- take input from the user and display output effectively
- apply decision-making and looping constructs in Python programs
- format output and run Python programs in interactive and script modes

Background

Python is one of the most popular programming languages in the world today, widely used in fields such as web development, data science, automation, and artificial intelligence. Its strength lies in being both powerful and easy to learn, making it suitable for beginners as well as professionals. Unlike many other languages, Python emphasizes simplicity and readability, allowing programmers to focus on solving problems rather than struggling with complex syntax.

The need for Python arises from the growing demand for a language that can be applied in diverse areas - whether it is developing websites, analyzing large datasets, creating intelligent applications, or automating repetitive tasks. Python's vast collection of libraries and strong community support make it a reliable choice for almost every domain of computing.

In this unit, you will begin your journey into Python programming by learning its essential building blocks: data types, variables, constants, operators, input

and output operations, decision-making statements, loops, basic formatting, and running Python programs. These topics form the foundation upon which more advanced concepts are built.

By the end of this unit, you will not only understand how Python works but also be able to write your own simple yet effective programs. Think of this as opening the door to a world where you can instruct the computer to perform tasks, solve problems, and even create intelligent systems - all through Python.

Keywords

Variable, constant, if, elif, for loop, while loop, IDE

Discussion

1.1.1 Data Types in Python

In Python, data types define the nature of the values a variable can store and determine the kinds of operations that can be performed on those values. They act as a set of rules that help the Python interpreter understand how to store the data in memory, how much memory to allocate, and how to process it efficiently. For example, integers and floating-point numbers require different storage formats and are processed differently when performing arithmetic operations.

One of Python's strengths is its dynamic typing capability, meaning you do not have to explicitly declare a variable's type before using it. The interpreter automatically assigns the most suitable data type based on the value you provide at runtime. For instance, assigning 25 makes the variable an integer (int), while assigning "Hello" makes it a string (str).

Python supports a wide range of built-in data types, including numeric types (integer, float, and complex), text type (string), sequence types (list, tuple, and range), mapping types (dictionary), set types (set, frozenset), and Boolean type (True or False). Each of these data types has specific operations and behaviors. For example, lists are mutable, allowing you to change their contents, whereas tuples are immutable and cannot be modified after creation.

Understanding data types is important not only for writing correct code but also for optimizing performance and preventing errors.

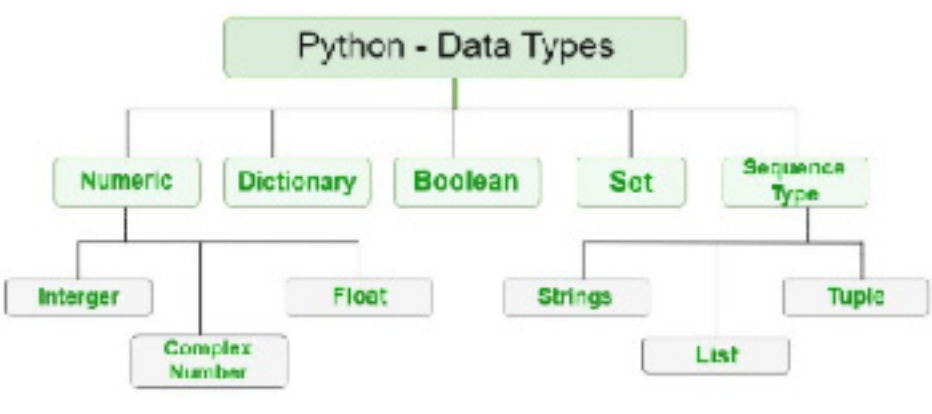


Fig 1.1.1 Python Data types

1. Numeric Data Types in Python

In Python, numeric data types are used to store numbers of different kinds, whether they are whole numbers, decimal numbers, or numbers with both real and imaginary parts. Python provides three main numeric types: int, float, and complex. Each of these is implemented as a built-in class in Python and supports a variety of mathematical operations.

Integers – Represented by the int class, integers are whole numbers that can be positive, negative, or zero, without any decimal or fractional part. For example, 10, -45, and 0 are integers. In Python, integers can be arbitrarily large because there is no fixed size limit, which makes them useful for handling very large numbers in scientific or financial calculations.

Floating-Point Numbers – Represented by the float class, floating-point numbers are real numbers that include a decimal point. They can also be written in scientific notation, where e or E is used to represent powers of ten. For example, 3.5, -0.75, and 4.2e3 (which is 4200.0) are floats. Floating-point numbers are commonly used in measurements, percentages, and other cases where appropriate precision is required.

Complex Numbers – Represented by the complex class, complex numbers consist of a real part and an imaginary part, written in the form (real) + (imaginary)j. For example, 4 + 2j is a complex number where 4 is the real part and 2j is the imaginary part. Complex numbers are mainly used in advanced mathematics, engineering, and scientific computing where calculations involve imaginary units.

2. Dictionary

A dictionary is a collection of data stored in key–value pairs. Each key is unique and is used to access its corresponding value.

(You will study dictionaries, List, tuple, and set in detail in the next unit.)

3. Set

A set is an unordered collection of unique elements. It does not allow duplicate values.

4. Boolean

The Boolean data type represents truth values. It can have only two values: True or False.

5. Sequence Types

Sequence types store multiple values in an ordered manner. The elements can be accessed using index positions.

Common sequence types in Python include:

- List
- Tuple
- String

String Data Type in Python

A string is a sequence of characters used to represent text in Python. Characters may include letters, numbers, symbols, or spaces.

Strings are written inside quotation marks. Python allows three types of quotes to create strings: single quotes (' '), double quotes (" "), and triple quotes (''' ''' or """ """).

Important Features of Strings

- Sequence type – A string is an ordered collection of characters.
- Indexing – Each character can be accessed using its position number (index).
- Immutable – Strings cannot be changed after they are created.
- Supports slicing – A part of the string can be extracted.

Common String Operations

- Concatenation – Joining two strings using +
- Repetition – Repeating a string using *
- Length – Finding number of characters using len( )

1.1.2 Variables and their Naming Rules

A **variable** is a fundamental concept in programming. It acts as a named storage location in the computer's memory, where data can be stored, retrieved, and manipulated during the execution of a program. In simple terms, a variable is like a container that holds information, and the name of the variable serves as a label to access the stored value whenever needed.

For example, if we want to store a student's age, instead of remembering the exact memory address in the computer, we assign a variable name such as age. This makes the program more readable, easier to understand, and simpler to maintain.

Rules for Naming Variables

When creating variables, programming languages impose certain rules to ensure consistency and avoid errors. These rules must be followed while naming variables:

1. Must start with a letter or an underscore (_)

A variable name cannot begin with a digit. Instead, it should always start with an alphabet (a–z, A–Z) or an underscore.

- Valid: age, _marks
- Invalid: 1student, 99value

2. Cannot start with a number

Variable names like 2score or 5data are not allowed. This restriction helps distinguish variable names from numerical values.

3. Can contain letters, numbers, and underscores (_)

After the first character, variable names may include alphabets, digits, or underscores. This allows descriptive naming while keeping identifiers flexible.

Examples: student_name, roll_no, marks2024

4. Case-sensitive

Variable names are case-sensitive, meaning name, Name, and NAME are treated as three different variables. This is an important point to remember, especially in languages like Python, Java, and C.

Example

```
age = 25
student_name = "Asha"
```

In the above code:

- The variable age stores the integer value 25.
- The variable student_name stores the string "Asha".

Here, the variable names are meaningful, which makes the program self-explanatory. Instead of writing something unclear like x = 25, using age = 25 immediately tells us what the data represents.

1.1.3 Constants

In programming, there are situations where certain values remain the same throughout the execution of a program. Such values are referred to as constants. A constant can be thought as a named value whose value is fixed and is not intended to change once it is defined. Constants make programs more reliable and easier to understand, as they clearly indicate values that are meant to remain unchanged.

For example, in mathematics, the value of π (pi) is always 3.1416, and in many applications, there may be limits such as the maximum number of users allowed in a system. These values are best represented as constants rather than ordinary variables.

Unlike some programming languages (such as C or Java) that provide dedicated keywords for declaring constants, Python does not have a built-in mechanism to enforce immutability of constants. However, by convention, programmers use variable names written in uppercase letters to represent constants. This signals to other developers (and to oneself) that these values are not supposed to be modified during program.

Although Python itself will not prevent reassignment of a constant, following this convention is an important programming practice that improves readability and helps maintain coding discipline.

Example

PI = 3.1416

MAX_USERS = 100

- Here, PI is a constant representing the mathematical value of π.
- MAX_USERS represents the maximum number of users permitted in a system.

Both values are written in uppercase letters to indicate that they are constants. While Python does not enforce immutability, changing their values in the program is considered bad practice.

Importance of Using Constants

1. Readability: Constants make the program easier to read and understand. For instance, writing PI * r * r is much clearer than using 3.1416 * r * r.

2. Maintainability: If a constant value needs to be updated (e.g., MAX_USERS from 100 to 200), it only has to be changed in one place, rather than throughout the entire program.

3. Error Reduction: Clearly distinguishing between values that change (variables) and those that do not (constants) reduces the chances of accidental modification and logical errors.

1.1.4 Operators

In any programming language, operators play a crucial role in performing operations on data. An operator is a special symbol that tells the interpreter to carry out a specific computation or action on one or more values (called operands). Operators allow us to perform tasks such as arithmetic calculations, comparisons, logical reasoning, and even checking membership within data structures.

For example, in the expression a + b, the symbol + is an arithmetic operator, and a and

b are the operands. The operator adds the two values and produces the result.

Types of Operators in Python

Python supports several categories of operators. Some of the commonly used ones are:

1. Arithmetic Operators

These are used to perform basic mathematical operations.

- Addition (+), Subtraction (-), Multiplication (*), Division (/)
- Floor Division (//), Modulus (%), Exponentiation (**)
- Example: 5 + 3 = 8, 7 % 3 = 1

2. Comparison Operators

Also called relational operators, these are used to compare two values. The result of a comparison operator is always a Boolean value: True or False.

- Equal to (==), Not equal to (!=)
- Greater than (>), Less than (<)
- Greater than or equal to (>=), Less than or equal to (<=)

3. Logical Operators

These are used to combine conditional statements and produce Boolean results.

- and (True if both conditions are true)
- or (True if at least one condition is true)
- not (Reverses the logical state)

4. Assignment Operators

These are used to assign values to variables. Python also provides compound assignment operators for performing an operation and assignment in a single step.

- = (assignment)
- +=, -=, *=, /= (shorthand assignment operations)

5. Membership Operators

These are used to test whether a value exists within a sequence such as a list, string, or tuple.

- in (returns True if the value is found)
- not in (returns True if the value is not found)

Example

a = 10

```
b = 3

print(a + b)    # 13  (Arithmetic Operator)

print(a % b)    # 1   (Modulus Operator)

print(a > b)    # True (Comparison Operator)
```

- a + b adds the two numbers and gives 13.
- a % b calculates the remainder when 10 is divided by 3, which is 1.
- a > b checks if 10 is greater than 3, which is True.

6. Identity Operators

Identity operators are used to check whether two variables refer to the same object in memory. Python provides two identity operators: is and is not.

is returns True if both variables refer to the same object.

is not returns True if the variables refer to different objects.

7. Bitwise Operators

Bitwise operators perform operations on the binary (bit-level) representation of integers. These operators are useful in low-level programming, data processing, and optimization tasks.

Common bitwise operators in Python include:

& (Bitwise AND) – Returns 1 if both bits are 1.

| (Bitwise OR) – Returns 1 if at least one bit is 1.

^ (Bitwise XOR) – Returns 1 if the bits are different.

~ (Bitwise NOT) – Inverts all bits.

<< (Left Shift) – Shifts bits to the left by a specified number of positions.

>> (Right Shift) – Shifts bits to the right.

Operators enhance the expressiveness of a programming language. They allow programmers to:

- Perform mathematical computations easily.
- Compare values and make decisions.
- Combine logical conditions for complex reasoning.
- Simplify code using assignment shortcuts.
- Work effectively with collections using membership checks.

1.1.5 Input and Output

Taking Input

When we write programs that interact with users, we often ask them to provide some data. In Python, this is commonly done using the input( ) function. For example:

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

At first glance, this looks simple. The user types something, and the program stores it. But here is an important detail: whatever the user types is always treated as a string (a sequence of characters), even if it looks like a number.

For example:

```
age = input("Enter your age: ")
print(age, type(age))
```

If the user enters 25, the output will be:

```
25 <class 'str'>
```

This shows that the entered value is not an integer, but a string containing the characters "2" and "5".

Why does Python behave this way?

The design choice makes input( ) flexible, since everything the user types like words, numbers, and symbols can first be captured as text. This way, the program does not need to guess whether the input is a number, a word, or something else. Everything is first collected as text, and then the programmer can decide how to use or convert it as needed. However, if our program requires numeric operations (like addition, multiplication, or comparison), the input string must be explicitly converted into the appropriate data type.

For example:

```
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
result = num1 + num2
print("Sum:", result)
```

Now, if the user types 10 and 20, Python converts them into integers before adding them, giving the correct result 30.

Similarly, if we want to accept decimal values, we use float( ) instead of int( ).

```
pi_value = float(input("Enter value of pi (approx): "))
```

Output

Just as input allows a program to receive information from the user, output enables it to communicate results back to the user. In Python, the most common way to display output is by using the print() function.

For example:

```
print("Hello, World!")
```

This statement displays the message Hello, World! on the screen.

The print( ) function is very flexible:

It can display text, numbers, variables, or even the result of an expression. It automatically adds a space between multiple items and moves to the next line after printing, unless told otherwise.

```
name = "Anita"
age = 22
print("Name:", name, "Age:", age)
```

Output:

```
Name: Anita Age: 22
```

Here, notice how Python separates each item with a space by default.

If we do not want the output to move to a new line, we can control this behavior using the **end** parameter:

```
print("Hello", end=" ")
print("World")
```

Output:

```
Hello World
```

The two messages appear on the same line because we replaced the default newline (\n) with a space.

1.1.6 Decision Making (if, elif, else)

In real life, we often make decisions based on conditions. For example:

If it rains, carry an umbrella.

If you have enough money, buy a ticket; otherwise, stay back.

In programming, we face similar scenarios. A program may need to:

- Approve or reject a user login based on a password.

- Display different messages based on age.
- Perform different calculations depending on the type of data entered.

That is, a program needs to choose between different paths of execution depending on the situation. This is called decision making or conditional execution.

In Python, decision making is handled using the if, elif, and else statements.

1.1.6.1 The if Statement

The simplest form is the if statement. It checks a condition. If the condition is true, a block of code is executed. If the condition is false, the block is skipped.

Syntax:

```
if condition:
    statement(s)
```

Example:

```
marks = 75
if marks >= 50:
    print("You passed the exam.")
```

Here, the message is printed only if the condition (marks >= 50) is true.

1.1.6.2 The if…else Statement

Sometimes, we want the program to do one thing if the condition is true, and another if it is false.

Syntax:

```
if condition:
   statement(s)  # executes if condition is true
else:
   statement(s)  # executes if condition is false
```

Example:

```
marks = 40
if marks >= 50:
   print("You passed the exam.")
else:
   print("You failed the exam.")
```

1.1.6.3 The if…elif…else Ladder

When there are multiple possible conditions, Python provides the **elif** keyword (short for else if).

Syntax:

```
if condition1:
    statement(s)
elif condition2:
    statement(s)
elif condition3:
    statement(s)
else:
    statement(s)
```

Example:

```
age = int(input("Enter your age: "))
if age >= 18:
    print("You are an adult.")
elif age >= 13:
    print("You are a teenager.")
else:
    print("You are a child.")
```

Here, the program checks each condition in order:

If age >= 18, it prints adult and skips the rest.

If not, but age >= 13, it prints a teenager.

Otherwise, it prints a child.

1.1.7 Loops

In everyday life, we often repeat tasks. For example:

A teacher marks each student's attendance one by one.

A cashier counts all the notes in a bundle.

We brush our teeth every morning in the same way.

Instead of writing the same set of instructions again and again, we use loops in programming. Loops allow a program to execute a block of code multiple times until a condition is satisfied.

Why Do We Need Loops?

Imagine you want to print numbers from 1 to 100. Without loops, you would need to write 100 print () statements. Loops make this task simple and efficient by repeating a block automatically.

Types of Loops in Python

Python mainly provides two types of loops:

for loop – Iterates over a sequence (like a list, string, or range).

while loop – Repeats as long as a condition is true.

1.1.7.1 The for Loop

The for loop is used when we know in advance how many times, we want to repeat something. It is often used with Python's range () function to generate a sequence of numbers.

Syntax:

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

Example: Print numbers from 1 to 5

```
for i in range(1, 6):
    print(i)
```

Output:

```
1
2
3
4
5
```

The loop starts with i = 1 and continues until i = 5.

1.1.7.2 The while Loop

The while loop is used when we do not know in advance how many times to repeat, but we know the condition to stop.

Syntax:

```
while condition:
    statement(s)
```

Example: Print numbers from 1 to 5

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Output:

```
1
2
3
4
5
```

Here, the loop continues as long as i <= 5.

1.1.7.3 Loop Control Statements

Sometimes we need more control over loops. Python provides:

break → Exits the loop immediately.

continue → Skips the current iteration and continues with the next.

pass → A placeholder that does nothing; used when a statement is required syntactically but no code needs to run.

Example with break and continue:

```
for i in range(1, 6):
    if i == 3:
        continue   # skip printing 3
    if i == 5:
        break      # stop the loop
    print(i)
```

Output:

```
1
```

2

4

1.1.7.4 Nested Loops

One loop can also be placed inside another loop. This is called a nested loop.

Example: Print a simple pattern

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

Output:

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

1.1.8 Basic Formatting

When we display results in a program, we usually want the output to be clear, readable, and well-structured. Simply printing values may sometimes look messy. Formatting allows us to control how variables, numbers, and text are displayed, without changing the actual data.

Why is formatting important?

Suppose you write a program to print a student's marks:

```
name = "Anita"
marks = 89.567
print(name, "scored", marks, "marks")
```

Output:

Anita scored 89.567 marks

This works, but it looks rough: the number has too many decimal places, and the structure is not neat. If we format the output:

print(f"{name} scored {marks:.1f} marks")

Output:

Anita scored 89.6 marks

Now the result is cleaner and more professional. This shows why formatting is essential.

1.1.8.1 Methods of Formatting in Python

Python provides different ways of formatting output:

(a) Using the format() Method

The format() method allows us to insert variables into a string by using curly braces {}.

Example 1:

name = "Anita"

age = 22

print("My name is {} and I am {} years old.".format(name, age))

Output:

My name is Anita and I am 22 years old.

The string contains {} placeholders. The format () method replaces the first {} with "Anita" and the second {} with 22.

Example 2 (Reordering):

print("Age: {1}, Name: {0}".format(name, age))

Output:

Age: 22, Name: Anita

{0} refers to the first argument "Anita". {1} refers to the second argument 22. By specifying numbers, we can control the order of substitution.

Example 3 (Decimal places):

pi = 3.14159

print("Value of pi: {:.2f}".format(pi))

Output:

Value of pi: 3.14

.2f means “format as floating-point with 2 decimal places”. Python rounds 3.14159 to 3.14.

(b) Using f-Strings (Formatted String Literals)

Starting from Python 3.6, a new feature called *f-strings* was introduced. They are considered the easiest and most powerful way to format strings. An f-string begins with the letter f before the quotation marks, and you can directly place variables or even expressions inside curly braces {}. Python will automatically replace the braces with the corresponding values.

Example 1:

```
name = "Rahul"

marks = 89.567

print(f"Student {name} scored {marks:.1f} marks.")
```

Output:

Student Rahul scored 89.6 marks.

The f before the string allows direct variable substitution. {name} is replaced by "Rahul". {marks:.1f} rounds the value to 1 decimal place, giving 89.6.

Example 2 (Expression inside f-string):

```
a, b = 5, 7

print(f"The sum of {a} and {b} is {a + b}.")
```

Output:

The sum of 5 and 7 is 12.

Variables a and b are inserted directly. The expression {a + b} is evaluated inside the f-string, producing 12.

(c) Using the % Operator (Old Style Formatting)

Before format () and f-strings, Python used the % operator. It still works, but is less preferred in modern code.

Example 1:

```
name = "Meena"

age = 25

print("Name: %s, Age: %d" % (name, age))
```

Output:

Name: Meena, Age: 25

%s is a placeholder for a string. %d is for integers. The values inside (name, age) are substituted in order.

Example 2 (Floating-point precision):

```
pi = 3.14159

print("Pi value: %.3f" % pi)
```

Output:

```
Pi value: 3.142
```

%.3f means "floating-point number with 3 decimal places." Python rounds 3.14159 to 3.142.

3. Alignment and Width in Formatting

Python allows us to align text and control column width.

Example:

```
print("{:<10} {:>5}".format("Name", "Age"))

print("{:<10} {:>5}".format("Anita", 22))

print("{:<10} {:>5}".format("Rahul", 21))
```

Output:

```
Name       Age
Anita       22
Rahul       21
```

:<10 → Align text to the left within 10 spaces.

:>5 → Align text to the right within 5 spaces.

This makes data appear in tabular form.

1.1.9 Running Python Programs

Before running Python programs, we must first install Python on our system. Once installed, programs can be executed in **interactive mode** or **script mode**.

1.1.9.1 Installing Python

Python is free and open-source. It can be installed on different operating systems like Windows, Linux, and macOS. Python is already installed on the Linux operating system. Python's infrastructure is extensively 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.

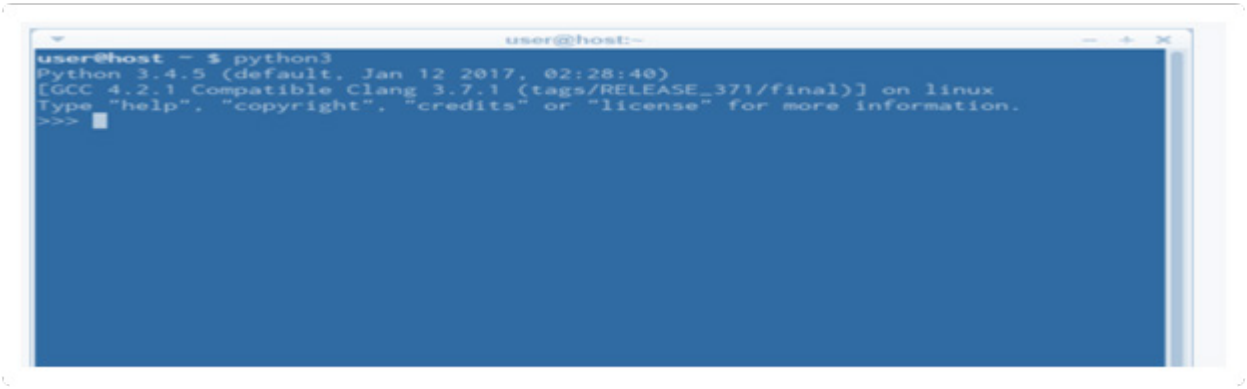


Fig 1.1.2 Python IDE

If you are a **Windows or macOS user**, download and install Python from https://www.python.org/

1.1.9.2 Python interactive mode

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

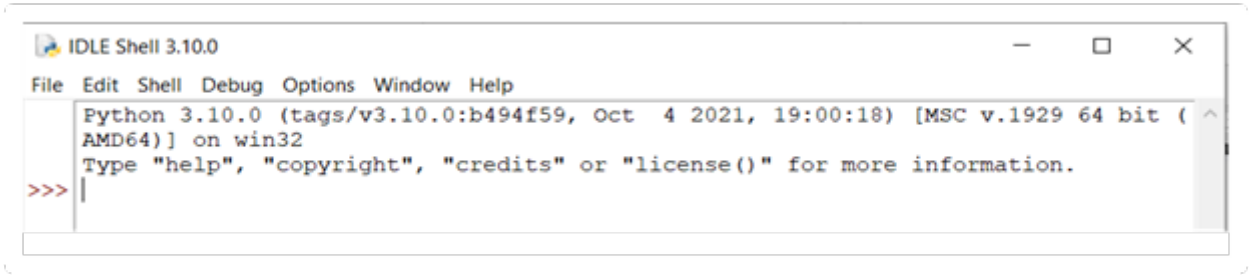


Fig 1.1.3 Python IDE

- When you type a statement in interactive mode, the interpreter executes it and displays the result, if any.
- 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.

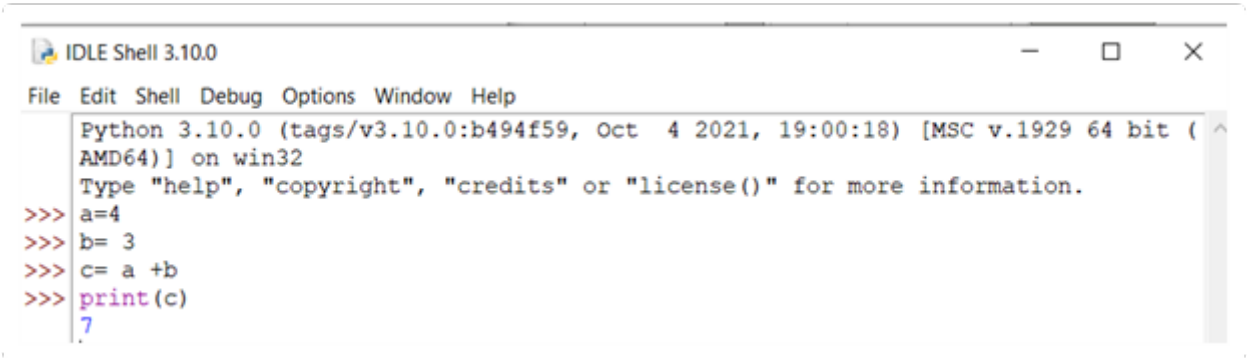


Fig 1.1.4 Python IDE

1.1.9.3 Python IDE

An IDE (Integrated Development Environment) helps in writing and managing source code better than a text editor. There are different 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:

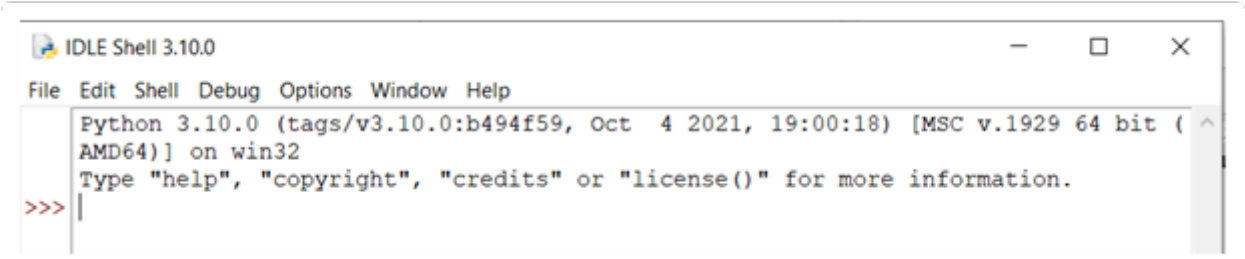


Fig 1.1.5 Python IDE

Create a new source file by clicking 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.

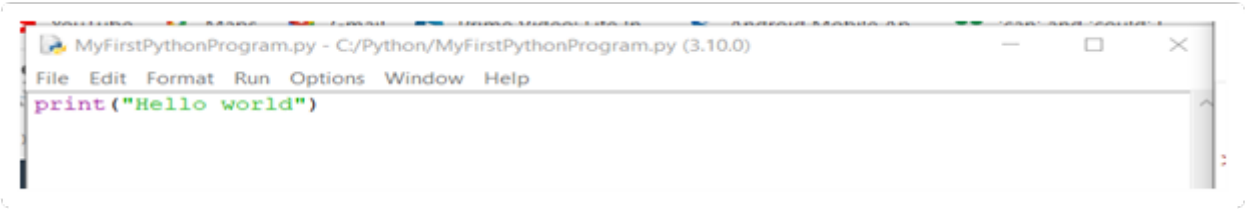


Fig 1.1.6 Python IDE

Save the file and click on Run in the IDLE menu. Do not 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 1.1.7 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.

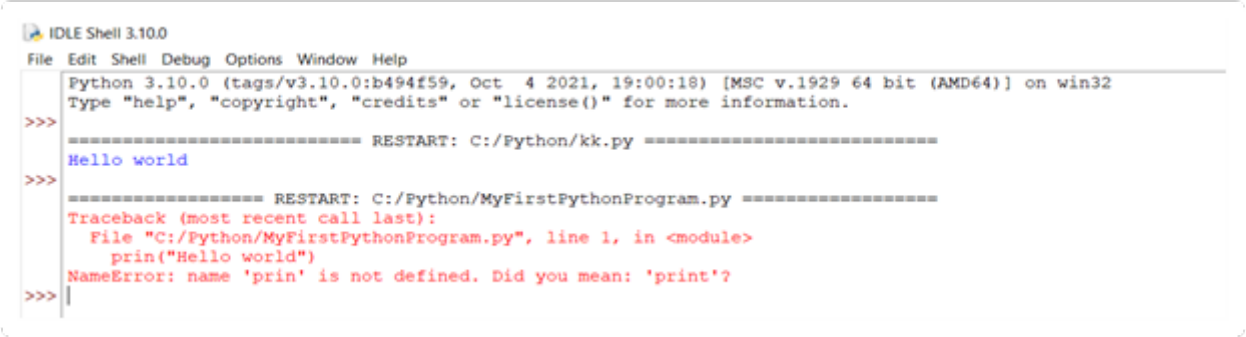


Fig 1.1.8 Python IDE

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.

```
print("Hello World"
```

```
  Input In [1]
    print("Hello World"
          ^
SyntaxError: '(' was never closed
```

Fig 1.1.9 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.

1. PyCharm:

PyCharm is an IDE for professional developers created by JetBrains

2. Spyder

Spyder is an open-source IDE written in Python, for scientific computing and data analysis.

Summarized Overview

Python is a high-level, interpreted programming language recognized for its simplicity, readability, and wide range of applications in areas such as web development, data science, automation, and artificial intelligence. Its clear syntax and versatile features make it an excellent starting point for beginners as well as a powerful tool for professional developers. In this unit, learners will be introduced to the core building blocks of Python, including data types, variables, constants, operators, and input and output operations, which are essential for handling and processing information. The unit also covers decision-making statements and loops that allow programs to perform logical operations and repetitive tasks, along with basic formatting techniques for improving output presentation. Finally, methods of running Python programs will be explored, providing a strong foundation for further programming practices.

Assignments

1. Explain the importance of Python as a programming language. Discuss at least three features that make it popular among developers and researchers.

2. Write a Python program to accept two numbers from the user, perform addition, subtraction, multiplication, and division, and display the results with proper formatting.
3. Differentiate between variables and constants in Python. Give suitable examples to illustrate how each is used in a program.
4. Using decision-making statements, write a Python program that accepts a student's marks and prints whether the student has passed or failed (assume the pass mark is 40).
5. Write a Python program using loops to display the multiplication table of a number entered by the user.

Reference

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
2. Matthes, E. (2019). *Python crash course: A hands-on, project-based introduction to programming* (2nd ed.). No Starch Press.
3. Sweigart, A. (2019). *Automate the boring stuff with Python: Practical programming for total beginners* (2nd ed.). No Starch Press.
4. Van Rossum, G., & Drake, F. L. (2009). The Python language reference manual. Network Theory Ltd.
5. Python Software Foundation. (2025). Python documentation. Retrieved from https://docs.python.org

Suggested Reading

1. Official Python Documentation (docs.python.org) for detailed explanations and examples.
2. Beginner-friendly tutorials on Real Python (realpython.com) and W3Schools Python Tutorial.
3. Books like Python Crash Course and Learning Python for structured learning.
4. Academic references and practical exercises from Automate the Boring Stuff with Python.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT

Data Structures in Python and Built-in Methods of Data Structures

Learning Outcomes

After completion of this unit, the learner will be able to:

- discuss the purpose of Python's core data structures (arrays, lists, tuples, sets, and dictionaries) and their role in efficient data handling
- differentiate between mutable and immutable data structures
- apply built-in operations and methods on arrays, lists, tuples, sets, and dictionaries to perform tasks such as insertion, deletion, searching, sorting, and traversal
- implement Python programs that integrate multiple data structures to solve computational problems and demonstrate efficiency in execution
- analyze real-world scenarios and determine the most suitable Python data structure to optimize performance, readability, and scalability of programs

Background

We deal with information every day, whether it is numbers in our mark sheet, names in a contact list, or files stored on a computer. All of this information is nothing but data. Handling a small amount of data is simple, but when the quantity grows larger, managing it becomes difficult. In your earlier learning of Python, you already know how to write simple programs using variables, operators, and loops. For example, you might have written a program to store the marks of three students in separate variables and then calculate their average. Such programs work well when the data is very small. But imagine if you need to store the marks of fifty students, or the

names of all the customers in a shop, or the list of items in a supermarket. Using separate variables for each value would not only be lengthy but also confusing and difficult to manage. This is where the need for organizing data in a systematic way arises. Python provides a variety of data structures such as lists, tuples, sets, and dictionaries that allow us to store collections of values efficiently, perform operations on them, and retrieve information whenever required. These structures transform our programs from handling only a few values into handling large amounts of information in a simple, flexible, and powerful manner. Understanding these data structures allows us to write programs that are not only simple but also powerful enough to solve real-world problems.

Keywords

List Set, Union, Intersection, Dictionary, Tuple, Mutable, Immutable, Search, Index. Array, Ndarray, Extend, Pop, Discard.

Discussion

In Python, a data structure is a way of organizing and storing data so that it can be accessed, modified, and processed efficiently within a program. Python provides a rich set of built-in data structures such as lists, tuples, sets, and dictionaries, each designed with unique properties and use cases. For example, lists are ordered and mutable, tuples are ordered but immutable, sets store unique and unordered elements, and dictionaries store data as key–value pairs for fast lookups. These built-in structures make it easy to manage collections of data without needing complex implementations. In addition, Python supports user-defined data structures like stacks, queues, linked lists, trees, and graphs, which can be implemented using classes and objects to solve more advanced computational problems. Choosing the right data structure in Python directly affects the efficiency of algorithms in terms of speed and memory usage. For instance, using a set for membership testing is much faster than using a list. Data structures in Python are widely applied in real-world scenarios such as managing databases, handling large datasets, implementing algorithms, and powering applications like web search engines, social media platforms, and recommendation systems. Thus, data structures in Python act as the building blocks of problem-solving, enabling programmers to write clean, optimized, and scalable code.

In Python, data structures are extremely important because they enable programmers to handle data efficiently and solve problems effectively. A good understanding of data structures allows us to write optimized, readable, and scalable programs. Python

provides a variety of built-in data structures such as lists, tuples, sets, and dictionaries, as well as powerful libraries like Collections and NumPy that extend data handling capabilities. Additionally, we can create user-defined data structures like stacks, queues, linked lists, trees, and graphs using classes and objects. The following section explains the important data structures in Python along with the key built-in functions associated with each one.

1.2.1 Array in Python

In Python, an array is a data structure that stores a collection of elements of the same data type in a contiguous block of memory, allowing efficient storage and faster access compared to general-purpose collections like lists. Unlike lists, which can hold elements of mixed data types, arrays are more restrictive but provide better performance for numerical and homogeneous data operations. Python does not have a built-in array data type like some other programming languages (such as C, Java), but it provides an array module that allows the creation of arrays with fixed-type elements, for example array('i', [1, 2, 3]) creates an integer array. Additionally, the NumPy library extends this functionality by introducing the ndarray, which supports powerful mathematical operations, multi-dimensional arrays, broadcasting, and efficient handling of large datasets. Arrays are indexed, ordered, and mutable, meaning elements can be accessed using indices, and can be modified, appended, or removed. They are widely used in tasks involving numerical computations, image processing, data analysis, and scientific computing because they allow vectorized operations that are much faster than iterating through Python lists. In essence, arrays in Python provide a specialized and efficient way of handling homogeneous data collections, making them an essential tool for performance-critical applications.

Syntax:

import array

arr = array.array(typecode, [initializers])

typecode → represents the type of elements (e.g., 'i' for integers, 'f' for floats).

initializers → list of initial values.

TypeCodes in array Module

- 'i' → integer
- 'f' → float
- 'd' → double
- 'u' → Unicode character

(Each type code specifies the type of elements stored in the array.)

Example:

```
import array
# Create an integer array
numbers = array.array('i', [10, 20, 30, 40, 50])
print(numbers)        # array('i', [10, 20, 30, 40, 50])
```

1.2.1.1 Array Operations in Python

Table 1.2.1 Array operations

Sl.No	Operations	Examples
1	**Traversal** (Accessing elements one by one)	arr = array.array('i', [10, 20, 30, 40, 50]) print("Original Array:", arr) print("Traversing the array:") for element in arr: print(element, end=" ") OUTPUT 10 20 30 40 50
2	Accessing (Indexing & Slicing)	print("\nFirst element:", arr[0]) print("Last element:", arr[-1]) print("Slice (index 1 to 3):", arr[1:4]) OUTPUT First element: 10 Last element: 50 Slice (index 1 to 3): array('i', [20, 30, 40])

3	Insertion	arr.insert(2, 25) # Insert 25 at index 2
		print("After Insertion:", arr)
		OUTPUT
		array('i', [10, 20, 25, 30, 40, 50])

4	Updation	arr[0] = 100 # Update first element
		print("After Updation:", arr)
		OUTPUT
		array('i', [100, 20, 25, 30, 40, 50])

5	Deletion	arr.remove(30) # Remove value 30
		print("After Removing 30:", arr)
		popped = arr.pop(2) # Remove element at index 2
		print("Popped Element:", popped)
		print("After Popping:", arr)
		OUTPUT
		After Removing 30: array('i', [100, 20, 25, 40, 50])
		Popped Element: 25
		After Popping: array('i', [100, 20, 40, 50])

5	Searching	index = arr.index(40) # Find index of element 40
		print("Index of 40:", index)
		OUTPUT
		Index of 40: 2

6	Reversing	arr.reverse() print("Reversed Array:", arr) OUTPUT array('i', [50, 40, 20, 100])

1.2.1.2 Built-in Methods in Python Arrays

Table 1.2.2 Built-in Methods in an array

S. No.	Methods	Examples

1	append(x)-Adds a new element x to the end of the array.	import array # Sample Array
		arr = array.array('i', [10, 20, 30, 40, 50])
		print("Original Array:", arr)
		arr.append(60)
		print("After append:", arr)
		OUTPUT
		array('i', [10, 20, 30, 40, 50, 60])

2	insert(i, x)-Inserts element x at position i (index).	arr.insert(2, 25) print("After insert:", arr) OUTPUT array('i', [10, 20, 25, 30, 40, 50, 60])
3	remove(x)-Removes the first occurrence of value x.	arr.remove(40) print("After remove:", arr) OUTPUT After remove: array('i', [10, 20, 25, 30, 50, 60])

4	pop([i])-Removes and returns the element at index i.
	If no index is given, it removes the last element.
		popped = arr.pop(2) # Removes element at index 2
		print("Popped Element:", popped)
		print("After pop:", arr)
		OUTPUT
		Popped Element: 25
		After pop: array('i', [10, 20, 30, 50, 60])

5	index(x)-Returns the first index where element x is found.	print("Index of 30:", arr.index(30)) OUTPUT Index of 30: 2
6	count(x)- Counts how many times x occurs in the array.	arr.append(30) print("Count of 30:", arr.count(30)) OUTPUT Count of 30: 2
7	extend(iterable) - Appends elements from another iterable (like list/array).	arr.extend([70, 80]) print("After extend:", arr) OUTPUT array('i', [30, 60, 50, 30, 20, 10, 70, 80])

8	fromlist(list)-Adds elements from a Python list into the array.	lst = [90, 100] arr.fromlist(lst) print("After fromlist:", arr) OUTPUT array('i', [30, 60, 50, 30, 20, 10, 70, 80, 90, 100])
9	tolist()-Converts array into a Python list.	print("Converted to list:", arr.tolist()) OUTPUT [30, 60, 50, 30, 20, 10, 70, 80, 90, 100]
10	itemsize-Returns the number of bytes each element occupies.	print("Itemsize:", arr.itemsize) OUTPUT Itemsize: 4

1.2.2 List in Python

In Python, a list is one of the most widely used built-in data structures that allows us to store an ordered collection of elements. A list is defined using square brackets [], with elements separated by commas, for example [10, 20, 30]. The elements of a list can be of different data types, such as integers, strings, floats, or even other lists (nested lists). Lists are mutable, which means their contents can be changed after creation—we can add, update, or remove elements using built-in methods like .append(), .insert(), .extend(), .remove(), and .pop(). Since lists are ordered and indexed, elements can be accessed using positive or negative indices, and they also support slicing to retrieve sub-lists. Lists allow duplicates, meaning the same value can appear multiple times, unlike sets. They are highly versatile and provide many useful operations such as sorting, reversing, counting occurrences, and list comprehensions for concise list creation. Because of their flexibility, lists are used in a wide range of real-world applications - for example, storing a sequence of student marks, a collection of names, or even complex datasets. Overall, a list is best understood as an ordered, mutable, and versatile collection in Python, making it an essential tool for problem-solving and programming.

```
#Creating  list
# empty list
L = []
# list literal
L = [1, 2, 3, "apple", 4.5]
```

```
# from iterable (tuple, range, string)
L2 = list((10, 20))        # [10, 20]
L3 = list(range(5))        # [0,1,2,3,4]
L4 = list("hello")         # ['h','e','l','l','o']
# nested list
matrix = [[1,2,3], [4,5,6], [7,8,9]]
```

1.2.2.1 List Operations in Python

Table 1.2.3: List operations

Sl.No	Operations	Examples
1	Indexing: Access individual elements using their index, which starts from 0. Negative indices access elements from the end of the list, with -1 being the last element.	a = [10,20,30,40,50] a[0] # 10 a[-1] # 50

2	Slicing- This is a powerful way to extract a sublist. The syntax is [start:stop:step]	my_list = [1, 2, 3, 4, 5, 6, 7, 8]
		print(my_list[2:5])
		# Output: [3, 4, 5]
		print(my_list[:3])
		# Output: [1, 2, 3] (from the beginning)
		print(my_list[4:]) # Output: [5, 6, 7, 8] (to the end)
		print(my_list[::2]) # Output: [1, 3, 5, 7] (every second element)
		print(my_list[::-1]) # Output: [8, 7, 6, 5, 4, 3, 2, 1] (reverses the list)

3	Traversal -Traversing a list means visiting or accessing each element of the list, one by one. This is a fundamental operation used to process, inspect, or modify the items within a list. The enumerate() function is a more elegant and efficient way to get both the index and the element at the same time. It returns pairs of (index, item).	#Using a for Loop
		my_list = ['apple', 'banana', 'cherry']
		for fruit in my_list:
		print(fruit)
		#Using a for Loop with range() and len()
		my_list = ['apple', 'banana', 'cherry']
		for i in range(len(my_list)):
		print(f"Index {i}: {my_list[i]}")
		# Using a for Loop with enumerate()
		my_list = ['apple', 'banana', 'cherry']
		for index, fruit in enumerate(my_list):
		print(f"Item at position {index} is {fruit}")
		OUTPUT
		Item at position 0 is apple
		Item at position 1 is banana
		Item at position 2 is cherry

4	Append - The append() method adds a single element to the end of a list. Extend -The extend() method appends all elements from an iterable (like another list, tuple, or string) to the end of the current list. Insert-The insert() method adds an element at a specified position in the list.	a.append(60) # add single element to end a.extend([70,80]) # append elements from iterable a.insert(2, 25) # insert 25 at index 2

5	Update - a list by assigning a new value to an element at a specific index	a[0] = 100 # replace element a[1:3] = [21,22] # replace slice (can change length)
6	Concatenation is the process of joining two or more lists together to create a new, single list. The + operator is used to perform this operation. Repetition is the process of creating a new list by repeating the elements of an existing list a specified number of times. The * operator is used for this purpose.	a = [10,20,30,40,50] c = a + [1,2,3] # concatenation -> new list d = a * 3 #Replication print (c) print (d) Output [10, 20, 30, 40, 50, 1, 2, 3] [10, 20, 30, 40, 50, 10, 20, 30, 40, 50, 10, 20, 30, 40, 50]

1.2.2.2 Built-in Methods in Python List

Table 1.2.4: Built-in Methods in List

Sl.No	Methods	Examples
1	Adding Elements append(element) → Adds a single element at the end of the list. extend(iterable) → Adds multiple elements (from list/ tuple/set, etc.) to the list. insert(index, element) → Inserts an element at a specific index.	fruits = ["apple", "banana"] fruits.append("cherry") # ['apple', 'banana', 'cherry'] fruits.extend(["orange", "kiwi"]) # ['apple', 'banana', 'cherry', 'orange', 'kiwi'] fruits.insert(1, "grapes") # ['apple', 'grapes', 'banana', 'cherry', 'orange', 'kiwi']

2	Removing Elements remove(element) → Removes the first occurrence of a value. pop([index]) → Removes element at given index (default: last element) and returns it. clear() → Removes all elements from the list.	fruits = ["apple", "banana", "cherry"] fruits.remove("banana") # ['apple', 'cherry'] print(fruits.pop()) # 'cherry', list becomes ['apple'] fruits.clear() # []
3	Searching & Counting index(element) → Returns index of the first occurrence of an element. count(element) → Returns the number of times an element appears.	numbers = [1, 2, 3, 2, 4, 2] print(numbers.index(3)) # 2 print(numbers.count(2)) # 3
4	Sorting & Reversing sort(reverse=False,key=None) → Sorts the list in ascending order (default). Can be customized with key functions. reverse() → Reverses the list elements in place.	nums = [3, 1, 4, 2] nums.sort() # [1, 2, 3, 4] nums.sort(reverse=True) # [4, 3, 2, 1] nums.reverse() # Reverses list
5	copy() → Returns a shallow copy of the list.	a = [1, 2, 3] b = a.copy() print(b) # [1, 2, 3]

6	len(list) → Returns number of elements in the list. max(list) / min(list) → Returns largest/smallest element. sum(list) → Returns sum of elements (numeric). any(list) → Returns True if any element is truthy. all(list) → Returns True if all elements are truthy.	nums = [10, 20, 30] print(len(nums)) # 3 print(max(nums)) # 30 print(min(nums)) # 10 print(sum(nums)) # 60 marks = [80, 65, 90, 75] # Check if all marks are greater than 50 result = all(m > 50 for m in marks) print(result) # True, because all marks > 50 # Another example values = [True, True, False] print(all(values)) # False (because one value is False) marks = [40, 55, 30, 70] # Check if any student passed with marks > 50 result = any(m > 50 for m in marks) print(result) # True (because 55 and 70 are > 50) # Another example values = [0, False, "", []] print(any(values)) # False (all are falsy values)

1.2.3 Frames in Python

In Python, a DataFrame is a two-dimensional, size-mutable, and heterogeneous data structure provided by the pandas library for handling structured data. It is similar to a table in a relational database or an Excel spreadsheet, where data is organized into rows and columns, each column having a label (column name) and each row having an index. A DataFrame can hold different types of data - integers, floats, strings, booleans, or even objects - within the same table, making it highly versatile for real-world datasets. DataFrames are built on top of NumPy arrays, which means they inherit efficient numerical operations, but they also add powerful features such as label-based indexing (.loc and .iloc), alignment of data, and handling of missing values. A DataFrame can be created from multiple sources like lists, dictionaries, NumPy arrays, CSV files, Excel

files, SQL databases, or even external APIs. Once created, it supports a wide range of operations including data selection, filtering, sorting, grouping, merging, reshaping, and aggregation, which makes it one of the most powerful tools for data analysis and manipulation. For example, analyzing sales reports, processing scientific datasets, or preparing machine learning inputs becomes straightforward with pandas DataFrames. In essence, a DataFrame in Python is a flexible and efficient container for structured data, widely used in data science, machine learning, and business analytics due to its ease of use and powerful capabilities.

1.2.4 Tuples in Python

In Python, a tuple is an ordered collection of elements that is very similar to a list but with one important difference - it is immutable. This means that once a tuple is created, its elements cannot be modified, added, or removed. Tuples are defined using parentheses (), with items separated by commas, and they can store different types of data such as numbers, strings, or even other tuples. For example, (10, "apple", 3.5) is a tuple containing an integer, a string, and a float. A tuple with a single element must include a trailing comma like (5,); otherwise, Python will treat it as just a number. Since tuples preserve the order of elements, they support indexing and slicing just like lists, allowing access using both positive and negative indices. The immutability of tuples makes them useful for storing fixed collections of data such as coordinates, dates, or database records. Tuples are also more memory efficient and faster than lists, and because they are immutable, they can be used as dictionary keys and stored in sets, unlike lists. Python provides common operations on tuples like concatenation, repetition, and functions such as len(), count(), and index(). Another powerful feature is packing and unpacking, where multiple values can be grouped into a tuple and then extracted into separate variables. In practice, tuples are widely used for representing data that should not change, such as a student's record ("Alice", 101, "MCA") or fixed positions like (x, y) coordinates. In short, a tuple is an ordered, immutable, and efficient data structure that ensures data integrity while offering speed and flexibility in Python programs.

Table 1.2.5: Tuple operations in Python

Sl.No	Operation	Example
1	Traversal	t = ("apple", "banana", "cherry") for item in t: print(item)
2	Indexing	t = (10, 20, 30, 40) print(t[0]) # Output: 10 print(t[2]) # Output: 30

3	Slicing	t = (1, 2, 3, 4, 5, 6) print(t[1:4]) # Output: (2, 3, 4) print(t[::-1]) # Output: (6, 5, 4, 3, 2, 1) → reversed tuple
4	Concatenation	t1 = (1, 2, 3) t2 = (4, 5, 6) print(t1 + t2) # Output: (1, 2, 3, 4, 5, 6)
5	Repetition	t = (7, 8) print(t * 3) # Output: (7, 8, 7, 8, 7, 8)
6	Membership - Check if an element exists in a tuple using in and not in.	t = ("apple", "banana", "cherry") for item in t: print(item)

1.2.4.1 Built-in Methods in Python Tuple

Tuples in Python come with a set of built-in functions that make it easier to perform operations like counting, searching, finding length, and calculating values. Since tuples are immutable, the built-in functions focus on accessing and analyzing data, not modifying it.

Table 1.2.6: Built-in Methods in Tuple

Sl.No	Methods	Examples
1	len()-Returns the **number of elements** in a tuple.	t = (10, 20, 30, 40) print(len(t)) # Output: 4
2	max()-Returns the **largest element** in the tuple.	t = (5, 12, 3, 19) print(max(t)) # Output: 19
3	min()-Returns the **smallest element** in the tuple.	t = (5, 12, 3, 19) print(min(t)) # Output: 3
4	sum()-Returns the **sum of all numeric elements** in the tuple.	t = (10, 20, 30) print(sum(t)) # Output: 60
5	sorted()-Returns a **new sorted list** of tuple elements .	t = (40, 10, 30, 20) print(sorted(t)) # Output: [10, 20, 30, 40]

6	tuple()-Converts an iterable (list, string, set, etc.) into a **tuple**.	l = [1, 2, 3] t = tuple(l) print(t) # Output: (1, 2, 3)
7	any()-Returns **True** if at least one element in the tuple is true (non-zero / non-empty).	t = (0, False, 5, "") print(any(t)) # Output: True

8	all()-Returns **True** if all elements are true	t = (1, 2, 3, 4)
		print(all(t)) # Output: True
		t2 = (1, 0, 3)
		print(all(t2)) # Output: False

9	enumerate()-Returns an **enumerate object** with index-value pairs from a tuple.	t = ("apple", "banana", "cherry") for index, value in enumerate(t): print(index, value) # Output: # 0 apple # 1 banana # 2 cherry
10	count()-Returns the **number of times an element occurs** in a tuple.	t = (1, 2, 2, 3, 2, 4) print(t.count(2)) # Output: 3
11.	index()-Returns the **index of the first occurrence** of an element.	t = (10, 20, 30, 20, 40) print(t.index(20)) # Output: 1

1.2.5 Dictionaries in Python

In Python, a **dictionary** is a powerful built-in data structure that stores information in the form of key – value pairs, where each key acts as a unique identifier for its associated value. Unlike sequences such as lists and tuples that are indexed by positions, dictionaries are indexed by keys, making them extremely efficient for lookups and data retrieval. A dictionary is defined using curly braces {}, with keys and values separated by a colon - for example, {"name": "Alice", "roll": 101, "course": "MCA"}. Keys must be unique and immutable (such as strings, numbers, or tuples), while values can be of any data type, including lists or even other dictionaries. Dictionaries are **mutable**, meaning elements can be added, updated, or removed after creation, and as of Python 3.7, they preserve the order of insertion. They provide several useful methods such as .keys(), .values(), and .items() to access their contents, and support iteration over keys

or key–value pairs. A dictionary can also be nested, allowing complex data structures to be represented easily, and it is commonly used to store structured data, such as a student record, a phonebook, or JSON objects from web applications. Since dictionaries provide **fast lookups, flexibility, and clarity**, they are widely used in real-world programming tasks where data must be organized and retrieved efficiently.

1.2.5.1 Dictionary Operations in Python

Table 1.2.7: Dictionary operations in Python

Sl.No	Operations	Examples

1	Creating a Dictionary - A dictionary is created using curly braces {} or the dict() function.	# Using {}
		student = {"name": "John", "age": 21, "course": "BCA"}
		# Using dict()
		student2 = dict(name="Alice", age=22, course="MCA")
		print(student)

2	Accessing Elements - Using .get() method (safer, avoids errors if key is missing)	student = {"name": "John", "age": 21, "course": "BCA"}
		print(student["name"]) # Output: John
		print(student.get("age")) # Output: 21
		print(student.get("marks", "Not Found"))
		# Output: Not Found

3	Adding and Updating	student["grade"] = "A"
		# Adding new key
		student["age"] = 22
		# Updating existing key
		print(student)

4	Traversing a Dictionary	student = {"name": "John", "age": 21, "course": "BCA"}
		for key in student:
		print(key, ":", student[key])
		for key, value in student.items():
		print(key, "=>", value)

5	Nested Dictionaries	students = {
		1: {"name": "John", "age": 21},
		2: {"name": "Alice", "age": 22}
		}
		print(students[1]["name"]) # Output: John

1.2.5.2 Built-in Methods in Python Dictionary

Table 1.2.8: Built-in Methods in Dictionary

Sl.No	Methods	Examples
1	clear()-Removes all items from dictionary	student = {"name": "John", "age": 21} student.clear() print(student) # {}
2	copy()- Returns a shallow copy	student = {"name": "John", "age": 21} copy_student = student.copy() print(copy_student) # {'name': 'John', 'age': 21}
3	fromkeys()-Creates dictionary from keys with a default value	keys = ["a", "b", "c"] new_dict = dict.fromkeys(keys, 0) print(new_dict) # {'a': 0, 'b': 0, 'c': 0}
4	get()-Returns value for a key, or default if not found	student = {"name": "John", "age": 21} print(student.get("age")) # 21 print(student.get("grade", "N/A")) # N/A
5	items()-Returns key–value pairs	student = {"name": "John", "age": 21} print(student.items()) # dict_items([('name', 'John'), ('age', 21)])
6	keys()-Returns all keys	student = {"name": "John", "age": 21} print(student.keys()) # dict_keys(['name', 'age'])

7	values()-Returns all values	student = {"name": "John", "age": 21} print(student.values()) # dict_values(['John', 21])
8	pop()-Removes key and returns value	student = {"name": "John", "age": 21} print(student.pop("age")) # 21 print(student) # {'name': 'John'}
9	popitem()-Removes and returns last inserted pair	student = {"name": "John", "age": 21} print(student.popitem()) # ('age', 21)
10	update()-Updates dictionary with another dictionary	student = {"name": "John"} student.update({"age": 21, "course": "BCA"}) print(student) # {'name': 'John', 'age': 21, 'course': 'BCA'}

1.2.6 Set in Python

In Python, a set is a built-in data structure that represents an unordered collection of unique elements, very similar to the concept of sets in mathematics. Unlike lists and tuples, sets do not allow duplicate values, and if duplicates are added, Python automatically removes them, ensuring that every element in the set is distinct. Sets are defined using curly braces {} or by using the set() constructor, for example, {1, 2, 3} or set([1, 2, 3]). An empty set must be created with set() because {} creates an empty dictionary by default. Since sets are unordered and non-indexed, their elements cannot be accessed using indexing or slicing like lists, but they can be iterated over in a loop. Sets are mutable, meaning elements can be added or removed using methods like .add(), .update(), .remove(), .discard(), and .clear(). One of the most powerful features of sets is their ability to perform mathematical operations such as union (|), intersection (&), difference (-), and symmetric difference (^), which makes them extremely useful for problems involving relationships between collections. For cases where immutability is required, Python provides frozenset, which are immutable sets that cannot be changed after creation and can even be used as dictionary keys. In real-world applications, sets are commonly used to remove duplicates from data, perform membership testing, and model mathematical concepts such as finding common elements or differences between groups - for example, finding common friends between two users in a social network. Overall, a set in Python is an efficient, flexible, and mathematically inspired data structure that ensures uniqueness while supporting powerful operations for handling collections of data.

1.2.6.1 Set operations in Python

Table 1.2.9 : Set operations in Python

Sl.No	Operations	Examples
1	Union (| or union())-Combines all unique elements from both sets.	A = {1, 2, 3} B = {3, 4, 5} print(A | B) # {1, 2, 3, 4, 5} print(A.union(B)) # {1, 2, 3, 4, 5}
2	Intersection (& or intersection())- Returns elements common to both sets.	A = {1, 2, 3} B = {2, 3, 4} print(A & B) # {2, 3} print(A.intersection(B)) # {2, 3}
3	Difference (- or difference()) -Returns elements in one set but not in the other.	A = {1, 2, 3, 4} B = {3, 4, 5} print(A - B) # {1, 2} print(A.difference(B)) # {1, 2}
4	Symmetric Difference (^ or symmetric_difference())-Returns elements in either set but not in both.	A = {1, 2, 3} B = {3, 4, 5} print(A ^ B) # {1, 2, 4, 5} print(A.symmetric_difference(B)) # {1, 2, 4, 5}
5	Subset (<= or issubset())-Checks if one set is contained within another.	A = {1, 2} B = {1, 2, 3, 4} print(A <= B) # True print(A.issubset(B)) # True
6	Superset (>= or issuperset())-Checks if one set contains another set.	A = {1, 2, 3} B = {1, 2} print(A >= B) # True print(A.issuperset(B)) # True

7	Disjoint (isdisjoint())-Checks if two sets have no elements in common.	A = {1, 2} B = {3, 4} print(A.isdisjoint(B)) # True

1.2.6.2 Built-in Methods in Python Set

Table 1.2.10: Built-in Methods in Set

Sl.No	Methods	Examples
1	Adding Elements- add(element) → Adds a single element to the set. update(iterable) → Adds multiple elements (list, tuple, set, etc.).	s = {1, 2, 3} s.add(4) # {1, 2, 3, 4} s.update([5, 6]) # {1, 2, 3, 4, 5, 6}

2	Removing Elements
	remove(element) → Removes the element, raises KeyError if not found.
	discard(element) → Removes the element, but no error if not found.
	pop() → Removes and returns a random element.
	clear() → Removes all elements, leaving an empty set.
		s = {1, 2, 3, 4}
		s.remove(2) # {1, 3, 4}
		s.discard(10) # No error
		s.pop() # Removes random element
		s.clear() # set()

3	copy() → Returns a shallow copy of the set.	A = {1, 2, 3} B = A.copy() print(B) # {1, 2, 3}

Summarized Overview

Python's data structures are essential for efficient data organization and manipulation, with different types suited for various tasks. Lists are ordered and mutable, accommodating diverse data types, making them highly versatile for general-purpose collections. Tuples are similar but are immutable, meaning their contents cannot be changed after creation, which makes them faster and ideal for fixed data like coordinates. Dictionaries store data as unique key-value pairs, providing extremely fast lookups and are perfect for structured data. Sets are unordered collections of unique elements, useful for removing duplicates and performing mathematical operations like unions and intersections. Additionally, while Python's built-in list is a general collection, the array module and the NumPy library provide more specialized, memory-efficient arrays for handling homogeneous numerical data. Each of these data structures has a specific purpose and a set of built-in operations that enable programmers to write optimized, readable, and scalable code for a wide range of applications, from data analysis to scientific computing.

Assignments

1. Why is an array from the array module more suitable for numerical operations than a standard Python list?
2. What is the main difference between a list and a tuple in Python?
3. What happens if you try to add a duplicate element to a set?
4. Explain the purpose of the pop() and remove() methods when working with lists.
5. How would you access the value associated with a specific key in a dictionary?
6. Discuss different data structures in Python?
7. Explain list and set operations in Python with suitable examples?
8. Discuss built-in methods in the dictionary?

Reference

1. **Thareja, R. (2025).** *Python Programming Using Problem-solving Approach* (3rd ed.). Oxford University Press.
2. **Mishra, A.** (2024). *Python Programming: From Basic to Advanced Concepts*
3. ***Machado, P. B.*** *(2024). Machine Learning with Python: Principles and practical techniques. Cambridge University Press.*
4. Matthes, E. (2022). *Python crash course: A hands-on, project-based introduction to programming* (3rd ed.). No Starch Press.
5. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

Suggested Reading

1. Python Software Foundation. (2025). *Data structures—Python 3.13.7 documentation*. Python.org.
2. GeeksforGeeks. (2025, July 23). *Differences and applications of list, tuple, set and dictionary in Python*.
3. Prasad, N. (2024, July 30). *Comprehensive guide to Python built-in data structures*. Retrieved from Analytics Vidhya website.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3 UNIT Functions

Learning Outcomes

After completion of this unit, the learner will be able to:

- define and use user-defined functions with various types of arguments
- recognize and use comprehensions to generate and manipulate collections concisely and efficiently
- use functions as inputs (as arguments), outputs (return functions from other functions) and built-in higher-order functions like map(), filter(), and reduce() to process data efficiently
- explain closures, use decorators to change how functions behave and write small inline functions with lambda
- describe iterators and generators to save memory, process data step by step, and handle very large or infinite sequences

Background

In earlier units, learners already learned the foundations of Python such as variables, data types, operators, and control structures like loops and conditionals, along with the use of built-in functions for common tasks. Building on this knowledge, the present unit delves deeper into the concept of functions, which are essential for writing modular and reusable programs. It introduces user-defined functions that allow programmers to encapsulate logic and improve code organization, while comprehensions offer concise ways to construct and manipulate data structures such as lists, sets, and dictionaries. The unit also expands into more advanced functional programming ideas like higher-order functions, which accept other functions as arguments or return them as results and closures, which allow functions to retain

values from their enclosing scope. In addition, learners will explore decorators, a powerful feature for extending or modifying function behavior without altering the source code. Concepts such as lambda expressions simplify short, anonymous operations, whereas iterators and generators support efficient and memory-friendly handling of sequences, including large or infinite sequences. Together, these topics extend the basic programming skills learned earlier into more powerful techniques that enable students to write cleaner, more efficient, and scalable Python programs.

Keywords

User-defined, Comprehensions, Higher-order, Closures, decorators, Lambdas, Generators, Iterators

Discussion

Functions are one of the most important features of Python, as they allow programmers to break down complex problems into smaller, reusable blocks of code. This unit focuses on both the basic and advanced aspects of functions, starting with user-defined functions that help structure programs in a logical and efficient way. It also explores comprehensions for creating data structures in a concise manner, higher-order functions that enable functional programming techniques, and advanced concepts like closures and decorators for extending functionality. Additionally, learners will study lambda functions for quick operations and the use of generators and iterators for efficient data handling. Together, these concepts provide the foundation for writing clean, flexible, and powerful Python programs.

1.3.1 Fundamentals of Functions

A function in Python is a self-contained block of reusable code that is designed to carry out a particular task or operation. It allows programmers to group a set of instructions under a single name, so that the same logic can be executed whenever needed, simply by calling the function. This makes programs easier to write, read, and maintain. Functions play an important role in breaking down large and complex programs into smaller, manageable units, which not only improves clarity but also reduces errors. They also enhance reusability, as once a function is defined, it can be used multiple times in different parts of a program without rewriting the same code. This leads to modularity, where a program can be divided into independent, logical sections, each responsible for a specific functionality. By using functions, developers save time, avoid redundancy, and create cleaner, more efficient, and more organized code.

Benefits of using Functions

1. **Code Reusability**: Once a function is defined, it can be reused multiple times throughout the program, reducing duplication of code.
2. **Modularity**: Functions break a program into smaller, logical parts, making it easier to understand, develop, and debug.
3. **Improved Readability**: By giving meaningful names to functions, the purpose of the code becomes clearer and more structured.
4. **Easier Maintenance**: If a change is needed, it can be made in the function definition instead of modifying the same code in multiple places.
5. **Reduced Complexity**: Large and complex problems can be solved by dividing them into smaller, manageable functions.
6. **Flexibility with Parameters**: Functions can take arguments and return values, making them adaptable to a wide variety of situations.
7. **Encapsulation**: Functions encapsulate logic and hide implementation details, allowing users to focus on how to use them rather than how they work internally.
8. **Testing and Debugging**: Functions can be tested individually, making it easier to identify and fix errors.

1.3.2 Types of Functions in Python

Functions in Python are essential building blocks that help organize code, promote reusability, and improve readability. They allow programmers to break down complex problems into smaller, manageable parts. Understanding the different types of functions is crucial for writing efficient, modular, and well-structured Python programs. In Python, functions can be broadly classified into the following types:

1. Built-in Functions: These are the functions that come predefined with Python. They are ready to use and simplify common tasks without needing extra code.

Example: *print(), len(), max(), min(), type(), range()*, etc.

2. User-defined Functions: These are functions created by programmers using the def keyword. They are used when we want to perform a specific task multiple times in a program.

Example:

```
def add(a, b):

return a + b
```

1.3.2.1 Built-in Functions

Built-in functions are the functions that come predefined with Python and are available for immediate use without requiring any import or additional code. They make programming easier by providing ready-made solutions for common operations such as

input/output, mathematical calculations, type conversions, and data manipulation. For example, print() is used to display output, len() returns the length of a sequence, type() gives the data type of a value, and sum() adds elements of an iterable (Table 1.3.1). Since these functions are already part of Python's standard library, programmers can use them directly, which saves time and improves productivity.

Table 1.3.1: Built-in Functions in Python

Function	Description	Example	Output
print()	Displays output on the screen	print("Hello")	Hello
len()	Returns the length of a sequence	len([1,2,3,4])	4
type()	Returns the data type of an object	type(10)	<class 'int'>
max()	Returns the largest item	max([5, 8, 2])	8
min()	Returns the smallest item	min([5, 8, 2])	2
sum()	Returns the sum of elements	sum([1,2,3])	6
abs()	Returns the absolute value	abs(-7)	7
round()	Rounds a number to a given number of decimal places	round(3.141, 2)	3.14
sorted()	Returns a sorted list	sorted([3,1,2])	[1, 2, 3]
input()	Takes input from the user	input("Enter name: ")	User enters a value

Built-in functions are an essential part of Python programming as they reduce the need to write basic logic repeatedly and allow developers to focus on solving higher-level problems.

Program 1 :

Source code:

```
numbers = [10, 20, 30, 40, 50]
print("Numbers:", numbers)                    # Using print()
print("Length of list:", len(numbers))              # Using len()
print("Maximum value:", max(numbers))         # Using max()
print("Minimum value:", min(numbers))         # Using min()
```

```
print("Sum of numbers:", sum(numbers))          # Using sum()
print("Average:", sum(numbers) / len(numbers))  # Combining functions
```

Output:

```
Numbers: [10, 20, 30, 40, 50]
Length of list: 5
Maximum value: 50
Minimum value: 10
Sum of numbers: 150
Average: 30.0
```

1.3.2.2 User defined functions

User-defined functions in Python are functions created by programmers to perform specific tasks according to the requirements of a program. Unlike built-in functions, which are already available in Python, user-defined functions allow greater flexibility by enabling developers to define their own logic using the *def* keyword. These functions can take inputs, called parameters, and can also return values after processing. By writing a function once and calling it multiple times, programmers can avoid code repetition, improve readability, and make their programs more modular and efficient. For example, instead of rewriting the same logic to calculate an average in different parts of a program, a user-defined function can be created and reused whenever needed. Thus, user-defined functions are an essential feature of Python that support reusability, modularity, and better program design.

Creating a Function in Python

We can define a function in Python, using the def keyword. We can add any type of functionality and properties to it as required.

Syntax:

```
def function_name(parameters):
    # function body
    return value                # optional
```

where,

def: Starts the function definition.

function_name: Name of the function.

parameters: Inputs passed to the function (inside ()), optional.

: : Indicates the start of the function body.

Indented code: The function body that runs when the function is called.

Example:

```
def fun():
    print("Welcome")
```

Calling a Function in Python

After creating a function in Python we can call it by using the name of the function in Python followed by parenthesis containing parameters of that particular function. Below is the example for calling a user-defined function in Python.

Example:

```
def fun():
    print("Welcome")
# Driver code to call a function
fun()
```

Output:

Welcome

1.3.3 Python Function Arguments

Arguments are the values passed inside the parentheses of the function. A function can have any number of arguments separated by a comma.

Syntax :

```
def function_name(parameter: data_type) -> return_type:
"""Docstring"""
# body of the function
return expression
```

data_type and return_type are optional in function declaration, meaning the same function can also be written as:

```
def function_name(parameter) :
"""Docstring"""
# body of the function
return expression
```

Let's understand this with an example. A simple function is created in Python to check whether the number passed as an argument to the function is even or odd.

Program 2 : Simple program to check whether the number passed as an argument to the function is even or odd.

Source Code :

```
def evenOdd(x: int) ->str:
    if (x % 2 == 0):
        return "Even"
    else:
        return "Odd"
print(evenOdd(16))
print(evenOdd(7))
```

Output:

```
Even
Odd
```

*# The above function can also be declared without **type_hints**, like this.*

Source Code:

```
def evenOdd(x):
    if (x % 2 == 0):
        return "Even"
    else:
        return "Odd"
print(evenOdd(16))
print(evenOdd(7))
```

Output:

```
Even
Odd
```

1.3.3.1 Types of Python Function Arguments

In Python, functions are designed to accept inputs called **arguments** (or parameters). Arguments allow values to be passed into functions, making them flexible and reusable. Python supports several types of function arguments, which provide different ways to pass values during a function call. The major types are **default arguments, keyword arguments, positional arguments, and arbitrary arguments.**

1. Default Arguments

A default argument is a parameter that assumes a default value if no value is provided during the function call. It makes functions more flexible and reduces the need for function overloading.

Example:

```
def display(name="Guest"):
    print("Hello,", name)
display()          # No argument passed
display("Alice")   # Argument passed
```

Output:

```
Hello, Guest
Hello, Alice
```

If no argument is passed, name takes the default value "Guest". When "Alice" is passed, it overrides the default.

2. Keyword Arguments (Named Arguments)

In keyword arguments, values are passed by specifying the parameter names. This makes the function call more descriptive and avoids confusion about the order of parameters.

Example:

```
def student(name, age):
    print("Name:", name)
    print("Age:", age)
student(age=20, name="John")
```

Output:

```
Name: John
Age: 20
```

3. Positional Arguments

In positional arguments, the values are assigned to parameters in the order they are provided. The number of values must match the number of parameters, otherwise an error occurs.

Example:

```
def add(a, b):
    print("Sum:", a + b)
add(10, 20)
```

Output:

Sum: 30

he first value 10 is assigned to a, and the second value 20 is assigned to b. The order is very important in this type.

4. Arbitrary Arguments

Sometimes, the number of arguments is not known in advance. Python allows functions to handle such cases with arbitrary arguments. These arguments can be classified as follows.

(a) *args: Arbitrary Positional Arguments

Using *args, a function can accept any number of positional arguments, which are collected into a tuple.

Example:

```
def display(*numbers):
    print("Numbers:", numbers)
display(1, 2, 3, 4, 5)
```

Output:

Numbers: (1, 2, 3, 4, 5)

All arguments are packed into a tuple (1, 2, 3, 4, 5) and can be accessed inside the function.

(b) **kwargs: Arbitrary Keyword Arguments

Using **kwargs, a function can accept any number of keyword arguments, which are collected into a dictionary.

Example:

```
def details(**info):
```

```
    print("Details:", info)

details(name="Alice", age=22, city="London")
```

Output:

```
Details: {'name': 'Alice', 'age': 22, 'city': 'London'}
```

All keyword arguments are stored as dictionary key-value pairs, making it useful when parameter names and values are dynamic. With these types of arguments, Python functions become highly flexible, allowing developers to design functions that handle fixed, optional, or even unlimited numbers of inputs.

1.3.4 Python Function within Functions

In Python, a function can be defined inside another function. This is called a nested function. Some basic key points are:

- The outer function contains one or more inner functions.
- Nested functions are useful for encapsulation and data hiding, as the inner function is only accessible within the outer function.
- They are also the foundation for advanced concepts like closures and decorators.

Program 3: Simple Nested Function

Source Code:

```
def outer_function(msg):
    def inner_function():
        print("Message from inner function:", msg)
    inner_function()
outer_function("Hello Python!")
```

Output:

```
Message from inner function: Hello Python!
```

outer_function() defines an inner function *inner_function()*. The inner function can access variables *(msg)* from the enclosing outer function.

Program 4: Returning Inner Function

Source Code:

```
def outer_function(a):
    def inner_function(b):
```

```
        return a + b
    return inner_function
# outer_function returns inner_function
result = outer_function(10)
print(result(5))
```

Output:

15

The *outer_function(10)* returns the inner_function. When result(5) is called, it adds 10 + 5. This is the idea behind **closures**, where the inner function "remembers" the variable a.

1.3.5 Pass by Value and Pass by Reference

When working with functions, one of the most important concepts is how arguments are passed. Different programming languages use different approaches such as **Pass by Value** or **Pass by Reference**. In Python, however, the situation is slightly unique, it uses a model known as **Pass by Object Reference** or **Call by Sharing**. Understanding how Python handles argument passing is essential for writing efficient and error-free programs.

a. Pass by Value

In pass by value, a copy of the actual data is passed to the function. Changes made inside the function do not affect the original variable outside the function. In Python, immutable objects (like integers, floats, strings, and tuples) behave like pass by value because modifications create a new object rather than changing the original.

Example:

```
def modify(x):
    x = x + 5
    print("Inside function:", x)
num = 10
modify(num)
print("Outside function:", num)
```

Output:

Inside function: 15

Outside function: 10

Here, the integer num remains unchanged because integers are immutable.

b. Pass by Reference

In pass by reference, the function receives the reference (memory address) of the variable. Changes made inside the function affect the original variable. In Python, mutable objects (like lists, dictionaries, and sets) behave like pass by reference because modifications inside the function reflect in the original object.

Example:

```
def modify_list(my_list):
    my_list.append(50)
    print("Inside function:", my_list)
numbers = [10, 20, 30]
modify_list(numbers)
print("Outside function:", numbers)
```

Output:

Inside function: [10, 20, 30, 50]

Outside function: [10, 20, 30, 50]

1.3.6 Comprehensions

Comprehensions in Python are a concise and elegant way to create new sequences such as lists, sets, or dictionaries by transforming or filtering elements from existing iterables. Instead of using lengthy loops and multiple lines of code, comprehensions allow us to achieve the same task in a single, readable line. They improve code readability, reduce complexity, and are widely used in modern Python programming.

Why do we need Comprehensions?

In Python, comprehensions are needed because they provide a concise and readable way to create and manipulate collections like lists, sets, and dictionaries. Instead of writing long loops with multiple lines of code, comprehensions allow you to express the same logic in a single, compact statement. They make the code shorter, easier to understand, and often more efficient. For example, list comprehensions can quickly generate lists based on conditions or transformations, helping programmers write clean and Python code. Some key points are:

- **Encourages Modular Thinking:** Promotes writing logic in smaller, reusable expressions.
- **Widely Used in Real-World Code:** Found in data science, web development, and automation scripts.

- **Easier Debugging & Testing:** Fewer lines reduce the surface area for bugs.
- **Seamless with Other Python Features:** Integrates well with functions like *zip(), enumerate(),* and *lambda.*

1.3.6.1 Types of Comprehensions in Python

Python offers different types of comprehensions to simplify the creation of data structures in a clean, readable way. Various types of comprehensions in Python are defined below.

1. List Comprehensions

List comprehensions allow the creation of lists in a single line, improving efficiency and readability. They follow a specific pattern to transform or filter data from an existing iterable.

Syntax:

```
[expression for item in iterable if condition]
```

where,

- **expression**: Operation applied to each item.
- **item**: Variable representing the element from the iterable.
- **iterable**: The source collection.
- **condition (optional)**: A filter to include only specific items.

Program 6: Generating a list of even numbers

Source Code:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
res = [num for num in a if num % 2 == 0]
print(res)
```

Output:

```
[2, 4, 6, 8]
```

This creates a list of even numbers by filtering elements from the list *a* that are divisible by 2.

2. Dictionary comprehension

Dictionary comprehensions are used to construct dictionaries in a compact form, making it easy to generate key-value pairs dynamically based on an iterable.

Syntax:

```
{key_expression: value_expression for item in iterable if condition}
```

where,

- **key_expression**: Determines the dictionary key.
- **value_expression**: Computes the value.
- **iterable**: The source collection.
- **condition (optional)**: Filters elements before adding them.

Program 7: Creating a dictionary of numbers and their cubes

Source Code:

```
res = {num: num**3 for num in range(1, 6)}
print(res)
```

Output:

```
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
```

This creates a dictionary where keys are numbers from 1 to 5 and values are their cubes.

3. Set comprehensions

Set comprehensions are similar to list comprehensions but result in sets, automatically eliminating duplicate values while maintaining a concise syntax.

Syntax:

```
{expression for item in iterable if condition}
```

where,

- **expression**: The operation applied to each item.
- **iterable**: The source collection.
- **condition (optional)**: Filters elements before adding them.

Program 8: Extracting unique even numbers

Source Code:

```
a = [1, 2, 2, 3, 4, 4, 5, 6, 6, 7]
res = {num for num in a if num % 2 == 0}
print(res)
```

Output:

```
{2, 4, 6}
```

This creates a set of even numbers from a, automatically removing duplicates.

4. Generator comprehensions

Generator comprehensions create iterators that generate values lazily, making them memory-efficient as elements are computed only when accessed.

Syntax:

(expression for item in iterable if condition)

where,

- **expression**: Operation applied to each item.
- **iterable**: The source collection.
- **condition (optional)**: Filters elements before including them.

Program 9: Generating even numbers using a generator

Source Code:

```
res = (num for num in range(10) if num % 2 == 0)
print(list(res))
```

Output:

```
[0, 2, 4, 6, 8]
```

This generator produces even numbers from 0 to 9, but values are only computed when accessed.

1.3.7 Higher-order functions

Functions are considered first-class objects in Python, which means they can be assigned to variables, passed as arguments, and even returned from other functions. A Higher-order function (HOF) is any function that either:

1. Takes one or more functions as arguments, or
2. Returns a function as its result.

Higher-order functions allow programmers to write cleaner, reusable, and more abstract code. They form the foundation for functional programming in Python and are widely used in operations like mapping, filtering, and reducing data.

A *Higher-order function* is a function that works with other functions as data.

- If a function accepts another function as a parameter, it is a higher-order function.
- If a function returns another function, it is a higher-order function.
- Many built-in functions in Python (like *map(), filter(), reduce(),* and *sorted()*) are higher-order functions.

Examples of Higher-order Functions

Program 10: Passing a Function as an Argument

Source Code:

```
def square(x):
    return x * x
def apply_function(func, value):
    return func(value)
print(apply_function(square, 5))
```

Output:

25

Here, apply_function() is a higher-order function because it takes another function (square ()) as an argument.

Program 11: Returning a Function

Source Code:

```
def outer_function(n):
    def inner_function(x):
        return x ** n
    return inner_function
power_of_2 = outer_function(2)
print(power_of_2(5))   # 5^2 = 25
```

Output:

25

The outer function returns the inner function, making it a higher-order function.

1.3.7.1 Built-in Higher-order Functions in Python

In Python, higher-order functions are functions that can take other functions as arguments or return them as results. They enable a powerful and flexible style of programming where functions can be passed around just like variables. Python provides several **built-in higher-order functions**, such *as map(), filter(), and reduce(),* which are commonly used to process and transform data in an efficient and expressive way. These functions

help simplify complex operations on sequences by combining functional programming concepts with Python's ease of use, making code both concise and readable.

1. map(): Applies a given function to each element of an iterable.

Example:

```
nums = [1, 2, 3, 4]

squares = list(map(lambda x: x**2, nums))

print(squares)
```

Output:

```
[1, 4, 9, 16]
```

2. filter(): Filters elements of an iterable based on a condition.

Example:

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

evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens)
```

Output:

```
[2, 4, 6]
```

3. reduce() (from functools): Applies a rolling computation to reduce a sequence into a single value.

Example:

```
from functools import reduce

nums = [1, 2, 3, 4]

product = reduce(lambda x, y: x * y, nums)

print(product)
```

Output:

```
24
```

4. sorted() with key: The sorted() function accepts another function as its key parameter.

Example:

```
words = ["apple", "banana", "kiwi", "cherry"]

sorted_words = sorted(words, key=len)
```

```
print(sorted_words)
```

Output:

```
['kiwi', 'apple', 'banana', 'cherry']
```

Advantages of Higher-order Functions

Higher-order functions in Python offer several advantages that make programming more efficient and expressive. Some of them are:

1. **Code Reusability**: Common patterns like mapping, filtering, and reducing can be reused.
2. **Abstraction**: Focus on *what* to do, not *how* to do it.
3. **Cleaner Syntax**: Reduces boilerplate code with concise one-liners.
4. **Foundation for Functional Programming**: Enables a declarative coding style.

Higher-order functions in Python allow functions to be treated as data, making code more abstract, flexible, and powerful. By using higher-order functions like map(), filter(), and reduce(), developers can write cleaner and more efficient programs. They are central to functional programming and are frequently combined with lambda functions, closures, and decorators for advanced usage.

1.3.8 Closures in Python

A **closure** in Python is a function object that remembers values from its enclosing scope, even if that scope has finished execution. A closure is created when:

- A nested function is defined inside another function.
- The inner function uses variables from the outer function.
- The outer function returns the inner function.

Closures are often used to encapsulate logic and maintain **state** across function calls.

Example:

```
def outer_function(msg):
    def inner_function():
        print("Message:", msg)
    return inner_function
my_closure = outer_function("Hello Closure")
my_closure()
```

Output:

```
Message: Hello Closure
```

Here, even though outer_function has finished execution, the inner function inner_function still remembers the value of msg.

Program 12: Maintaining State

Source Code:

```
def multiplier(n):
    def inner(x):
        return x * n
    return inner
times2 = multiplier(2)
times3 = multiplier(3)
print(times2(5))   # 10
print(times3(5))   # 15
```

Output:

```
10
15
```

times2 and *times3* are closures that remember the value of n used during their creation.

Advantages of Closures

- Encapsulate data without using classes.
- Provides a neat way to maintain state across calls.
- Serve as the foundation for advanced features like decorators.

1.3.9 Decorators in Python

A decorator is a special type of higher-order function that is used to modify or extend the behavior of other functions without permanently changing their code. Some basic points are:

- A decorator takes a function as input.
- It wraps additional functionality around it.
- It returns a new function with the added behavior.

In Python, decorators are usually applied using the *@decorator_name* syntax placed above a function definition.

Example:

```
def my_decorator(func):
    def wrapper():
```

```
        print("Before the function is called")
        func()
        print("After the function is called")
    return wrapper
@my_decorator
def say_hello():
    print("Hello World")
say_hello()
```

Output:

Before the function is called

Hello World

After the function is called

Here, *my_decorator* modifies the behavior of *say_hello* by adding extra code before and after it runs.

Program 13: Decorator with Arguments

Source Code:

```
def repeat_decorator(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator
@repeat_decorator(3)
def greet(name):
    print("Hello,", name)
greet("Python")
```

Output:

Hello, Python

Hello, Python

Hello, Python

The decorator repeat *decorator(3)* makes the function run 3 times.

Common Built-in Decorators

1. **@staticmethod:** Defines a static method inside a class.
2. **@classmethod:** Defines a method that receives the class as its first argument.
3. **@property:** Converts a method into a read-only property. (typically read-only unless a setter is defined).

Relationship between Closures and Decorators

- Closures provide the mechanism to remember variables from an enclosing scope.
- Decorators rely on closures to wrap additional behavior around functions.
- Without closures, decorators cannot function. Decorators typically use closures internally to maintain state.

Table 1.3.2: Differences between Closures And Decorators In Python

Aspect	Closures	Decorators
Definition	Function with access to its lexical scope	Higher-order function that modifies another function
Purpose	Retain access to variables in the enclosing scope	Add functionality to existing functions
Usage	Retain state after the outer function finishes	Modify behavior of functions or methods
Syntax	Nested function captures local variables	Applied with @ syntax above a function
Focus	Capturing state	Extending behavior

Closures and decorators are powerful features of Python that enable cleaner, more modular, and reusable code. Closures help retain state and encapsulate data. Decorators extend functionality without altering the original function. These concepts are heavily used in real-world Python applications such as web frameworks (e.g., Flask, Django), logging, authentication, and performance measurement. Closures and decorators are fundamental concepts in Python that enable more sophisticated and flexible coding patterns. Closures allow nested functions to capture and remember the state of their enclosing scope, while decorators provide a clean way to modify the behavior of functions. Mastering these concepts can greatly enhance your ability to write efficient, reusable, and maintainable code.

1.3.10 Lambdas

In Python, a lambda function is a small, anonymous function defined without using the standard def keyword. It is mainly used for short, throwaway operations where defining a full function would be unnecessary. Lambda functions are also called anonymous functions because they do not require a name.

The general syntax is:

lambda arguments: expression

where,

- **lambda:** keyword used to declare a lambda function.
- **arguments:** input values (like function parameters).
- **expression: a** single expression evaluated and returned.

Features of Lambda Functions

- Defined in a single line.
- Can take any number of arguments but can only have one expression.
- Automatically returns the evaluated value (no need for return).
- Used commonly with higher-order functions such as *map(), filter(),* and *reduce().*

Examples of Lambda Functions

Program 14: Simple Lambda

Source Code:

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

Output:

25

Program 15: Multiple Arguments

Source Code:

```
add = lambda a, b: a + b
print(add(10, 20))
```

Output:

30

Program 16: Using with map()

Source Code:

```
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, numbers))
print(squares)
```

Output:

```
[1, 4, 9, 16, 25]
```

Program 17: Using with filter()

Source Code:

```
numbers = [10, 21, 30, 41, 50]
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)
```

Output:

```
[21, 41]
```

Program 18: Using with reduce()

Source Code:

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

Output:

```
120
```

Advantages of Lambda Functions

- Concise and compact syntax.
- Useful for one-time use functions.
- Often used with functional programming tools (map, filter, reduce).
- Increases readability for small tasks.

Limitations of Lambda Functions

- Can only contain a single expression (no multiple statements).
- Not ideal for complex logic.
- Sometimes reduces readability if overused.

Lambda functions in Python are a powerful feature for writing short, inline, and functional-style code. They are especially useful when working with sequences, transformations, and conditions. While they are not a replacement for normal functions defined using *def,* they provide a quick and efficient way to perform small operations in fewer lines of code. In table 1.3.3 summarizes a comparison table of normal functions vs lambda functions in Python

Table 1.3.3: Comparison table of Normal Functions vs Lambda Functions in Python

Aspect	Normal Function (def)	Lambda Function (lambda)
Definition	Defined using the def keyword.	Defined using the lambda keyword.
Name	Usually has a name (can also be anonymous if assigned to a variable).	Always anonymous (but can be assigned to a variable).
Syntax	Can span multiple lines with multiple statements.	Single-line function with only one expression.
Return Statement	Requires explicit return keyword.	Returns the value automatically (no return needed).
Complexity	Suitable for large and complex operations.	Best suited for short, simple operations.
Readability	More readable for complex logic.	More concise for small, inline tasks.
Usage	General-purpose functions.	Commonly used with map(), filter(), reduce().
Example	def add(a, b): return a + b	add = lambda a, b: a + b

1.3.11 Generators and Iterators

Python provides Iterators and Generators as tools to handle sequences of data efficiently. Both are essential for working with large datasets, infinite sequences, or situations where we don't want to load everything into memory at once. They allow programmers to loop through elements one at a time, making programs more memory-efficient and fast. In table 1.3.4 defines the key differences between the iterators and generators.

1.3.11.1 Iterators in Python

An **iterator** is an object in Python that allows sequential traversal through elements of a collection (like lists, tuples, or strings). It implements two methods:

- **__iter__():** returns the iterator object itself.
- **__next__():** returns the next item in the sequence and raises StopIteration when items are exhausted.

Program 19: Using Iterator

Source Code:

```
numbers = [1, 2, 3]
it = iter(numbers)
print(next(it))   # Output: 1
print(next(it))   # Output: 2
print(next(it))   # Output: 3
# print(next(it)) → raises StopIteration
```

Output:

```
1
2
3
```

1.3.11.2 Generators in Python

A **generator** is a simpler way to create iterators. Instead of creating a class with __iter__() and __next__(), a generator uses the yield keyword inside a function to produce a sequence of values one at a time.

- Generators generate values lazily (on demand).
- They are memory-efficient, as they don't store the whole sequence at once.

Program 20: Generator Function

Source Code:

```
def my_generator():
    for i in range(1, 4):
        yield i
gen = my_generator()
```

```
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

Output:

1 2 3

1.3.11.3 Generator Expressions

Just like list comprehensions, Python supports generator expressions, which are written using parentheses () instead of square brackets [].

Example:

```
squares = (x*x for x in range(5))
for num in squares:
    print(num)
```

Output:

0

1

4

9

16

Table 1.3.4: Key Differences between Iterators and Generators

Aspect	Iterator	Generator
Definition	Object implementing __iter__() and __next__().	Function with yield or generator expression.
Creation	Created using classes and special methods.	Created using functions with yield.
Memory Usage	May store all elements in memory.	Generates values lazily (memory efficient)
Ease of Use	Requires more code (class + methods).	Very simple with yield or expressions.
Example	iter([1,2,3])	(x*x for x in range(5))

Advantages of Generators and Iterators

- Memory efficiency: processes items one at a time.
- Lazy evaluation: compute values only when needed.
- Useful for infinite sequences (e.g., Fibonacci numbers).
- Enhance performance when working with large datasets.

Program 21: Fibonacci Generator

Source Code:

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b
for num in fibonacci(6):
    print(num)
```

Output:

```
0
1
1
2
3
5
```

Iterators and Generators are powerful tools in Python for handling sequences efficiently. While iterators require more effort to implement, generators provide a simpler and more Pythonic approach using the yield keyword. Together, they help create scalable, efficient, and clean code when dealing with large or infinite data streams.

Summarized Overview

This unit explores advanced concepts of functions in Python, focusing on how they improve code structure, reusability, and efficiency. It begins with user-defined functions, where programmers can create their own reusable blocks of code for specific tasks. Different types of function arguments (positional, keyword, default, and arbitrary) provide flexibility in passing data. The unit also covers comprehensions (list, set, dictionary, and generator comprehensions), which offer a concise way to create collections in a single line of code. Moving further, higher - order functions like map(), filter(), and reduce() highlight the power of functions that can accept or return other functions. Advanced concepts such as closures (functions that capture variables from their enclosing scope) and decorators (functions that modify the behavior of other functions) demonstrate Python's functional programming style. The unit also introduces lambda functions, which are anonymous single-line functions useful for short tasks. Finally, generators and iterators are discussed as tools for handling data efficiently, enabling lazy evaluation and working with large or infinite sequences.

Assignments

1. Explain the concept of user-defined functions in Python. Discuss different types of function arguments (positional, keyword, default, and arbitrary) with suitable examples and outputs.
2. Write a detailed note on Python comprehensions. Differentiate between list, dictionary, and set comprehensions with examples. Also, explain the use of generator comprehensions and their advantages.
3. Define higher-order functions in Python. Illustrate the working of built-in higher-order functions like map(), filter(), and reduce() with practical examples. Compare them with the equivalent code using loops.
4. Explain the concept of closures with an example showing variable capturing. Define decorators and demonstrate how they can be used to modify the behavior of a function.
5. Discuss lambda functions and explain with examples where lambdas are useful.
6. Distinguish between iterators and generators in Python. Write an iterator class and a generator function to generate Fibonacci numbers up to n.

Reference

1. Matthes, E. (2023). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming* (2nd ed.). No Starch Press.
2. Barry, P. (2023). *Head First Python: A Brain-Friendly Guide* (2nd ed.). O'Reilly Media.
3. Sweigart, A. (2023). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners* (2nd ed.). No Starch Press.
4. McKinney, W. (2023). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (3rd ed.). O'Reilly Media.
5. Downey, A. B. (2023). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). Green Tea Press.

Suggested Reading

1. Python Official Documentation- https://docs.python.org/3/tutorial/controlflow.html
2. GeeksforGeeks- Python Programming Language https://www.geeksforgeeks.org/python-programming-language/
3. Real Python - https://realpython.com/
4. W3Schools- Python Tutorial https://www.w3schools.com/python/
5. Programiz - Learn Python Programming **https://www.programiz.com/python-programming**

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT

Introduction to Python Libraries

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the purpose of Python libraries and how they support code reuse and efficiency
- describe the role of pip and Pyhton package Index (PyPI) in Python module discovery and installation
- summarize the benefits of using virtual environments in Python project development
- identify different types of modules in Python, including built-in and user-defined modules

Background

Before learning about Python module discovery, installation, and library management, learners should have a foundational understanding of Python programming. This includes familiarity with basic syntax, such as variables, data types, functions, and control structures like loops and conditionals. They should also be comfortable writing and running Python scripts using a code editor or from the command line. Basic command-line skills are essential, as tasks like installing packages with pip and managing virtual environments often require using the terminal. Additionally, learners should understand how to navigate file systems and have Python properly installed and configured on their machines. A working knowledge of how to create, save, and execute .py files will further support effective learning of modules, libraries, and package management in Python.

Keywords

Pip, Python Package Index, Python libraries, Modules, Built-in modules

Discussion

A Python library is a group of related modules bundled together to provide reusable code. These libraries offer pre-written functionality that developers can integrate into multiple programs, saving time and effort by eliminating the need to write the same code repeatedly for different applications. Additionally, every Python library comes with its own source code. Python libraries are widely used across diverse domains such as machine learning, data visualization, and computer science.

A Python library allows us to avoid writing the same code over and over by providing a collection of modules or pre-written code blocks.

In a Microsoft Windows environment, library files typically have a .DLL extension, which stands for Dynamic Link Libraries. When a program is run after being linked with a library, the linker automatically searches for the library, accesses its functions, and helps interpret the program. As a result, we can easily use the functions and methods from the library within any Python program.

1.4.1 Standard Libraries in Python

The Python Standard Library includes the complete set of syntax rules, semantics, and tokens that define the Python language. It features built-in modules that offer essential system functionalities, such as input/output operations and other core features. Below are some widely used Python libraries:

1. NumPy

NumPy, short for Numerical Python, is one of the most widely used open-source libraries in Python. It provides powerful support for multi-dimensional arrays, a wide range of built-in mathematical functions, and efficient handling of large matrices - making it ideal for scientific and numerical computations. It serves as an N-dimensional container capable of storing various data types. With the NumPy array object, users can create arrays with multiple rows and columns. Additionally, libraries like TensorFlow rely on NumPy internally for numerous operations. NumPy can also be used to generate random numbers.

2. Matplotlib

Matplotlib is a Python library used for visualizing numerical data, playing a key role in data analysis. It allows users to create detailed and high-quality visual representations, such as line graphs, bar charts, pie charts, histograms, and scatter plots. As an open-

source tool, it is widely adopted for producing clear and informative plots in both research and industry settings.

3. PyGame

PyGame is a Python library that provides a user-friendly interface to SDL (Simple DirectMedia Layer), along with tools for handling audio, user input, and platform-independent graphics. It allows developers to utilize these resources to create video games in Python, making it a go-to choice for game development involving sound and visual elements.

4. Pandas

Pandas is a widely used open-source library in Python, licensed under the Berkeley Software Distribution (BSD). It is especially popular among data scientists due to its flexible data structures and powerful tools for data analysis. In the field of data science, Pandas is used for tasks such as data manipulation, cleaning, exploration, and inspection. It also supports operations like iteration, visualization, sorting, data conversion, re-indexing, aggregation, and combining datasets. With Pandas, users can perform complex data analysis and modeling within Python itself, without needing to switch to other programming environments.

5. PyBrain

PyBrain, short for Python-Based Reinforcement Learning, Artificial Intelligence, and Neural Networks, is an open-source library designed for those new to machine learning. It provides simple and accessible algorithms, making it a great starting point for beginners. Its user-friendly structure and adaptability make it ideal for researchers and developers beginning their journey in AI and machine learning.

6. Statsmodels

Statsmodels is a Python library used for estimating and analyzing statistical models. It delivers reliable and high-quality results for various statistical tests and analytical tasks, making it a valuable tool for statistical computing and data analysis.

7. TensorFlow

TensorFlow, developed by Google's Brain Team, is a powerful open-source library designed for complex computations. It is widely used for implementing machine learning and deep learning models, as well as solving advanced problems in mathematics and physics.

8. Seaborn

Seaborn is a Python package built on top of Matplotlib, used for visualizing statistical data. It simplifies the creation of informative and attractive graphs by offering various color palettes and themes. Seaborn makes it easy to generate complex plots and perform regression analysis through intuitive functions and built-in statistical visualization tools.

9. Scrapy

Scrapy is an open-source Python library designed for extracting data from websites.

It simplifies the process of web crawling and screen scraping. In addition to data extraction, Scrapy is also used for tasks such as automated testing and data mining.

10. PyTorch

PyTorch is one of the most widely used machine learning libraries in Python. It excels at optimizing tensor operations and handling neural network-related tasks. With its robust API and support for GPU acceleration, PyTorch enables efficient execution of complex tensor computations.

11. SciPy

SciPy, short for Scientific Python, is an extension of NumPy and works seamlessly with it. While NumPy focuses on efficient array operations such as sorting and indexing, SciPy builds on this by offering advanced tools for scientific computing and data processing. This open-source library includes a wide range of user-friendly functions and methods for high-level computations. SciPy also contains multiple specialized sub-packages, including io, optimize, cluster, interpolate, stats, spatial, integrate, special, sparse, and more

12. Keras

No list of Python libraries would be complete without mentioning Keras. This open-source library is specifically designed for building and experimenting with deep neural networks. With the growing importance of deep learning, Keras has become a preferred choice due to its user-friendly API built for ease of use by humans rather than machines. It is widely favored by both researchers and industry professionals over alternatives like Theano and raw TensorFlow. It's important to note that Keras requires TensorFlow as a backend, so TensorFlow must be installed beforehand.

13. Scikit-learn

Scikit-learn is a widely used open-source Python library designed to handle complex datasets. Built on top of SciPy and NumPy, it provides powerful tools for machine learning. It supports a variety of supervised and unsupervised learning algorithms such as classification, regression, and clustering, making it a go-to library for implementing standard machine learning techniques.

1.4.2 Modules

Python modules play a vital role in programming by promoting code reuse and simplifying the development and management of large-scale applications. They help separate implementation details from the main logic of a program, which leads to better code organization, easier readability, and improved maintainability. Using modules allows developers to write cleaner, more structured code that can be reused across multiple projects, thereby increasing productivity and efficiency.

A Python module can include a variety of elements such as functions, classes, variables, and other components. These can be imported into other programs using the import statement followed by the module name. Python comes with a number of built-in modules that are readily available, and developers also have the option to create custom modules by saving their code in files with a .py extension.

Python modules can be categorized into two main types:

1. Built-in Python Modules
2. User-defined Modules

1.4.3 Built-in Python Modules

Python comes equipped with a wide array of built-in modules that simplify many programming tasks and improve code clarity. These modules offer a broad range of functionality and can be used immediately without needing to install any additional packages. With these built-in tools, Python delivers a powerful programming environment right out of the box, enabling developers to perform common operations efficiently and with minimal setup.

Below are some commonly used built-in modules:

- **math:** Useful for performing advanced mathematical operations, including trigonometric and logarithmic functions.
- **datetime:** Provides tools for working with dates and times, including current timestamps and date arithmetic.
- **os:** Enables interaction with the operating system, such as accessing the file system, creating or removing directories, and running system commands.
- **sys:** Gives access to variables and functions related to the Python interpreter, including command-line arguments, interpreter version, and system paths.

```
import os
print(os.getcwd())
print(os.listdir())
# Example using the sys module
import sys
print(sys.version)
print(sys.argv)
# Example using the math module
import math
print(math.pi)
print(math.sin(math.pi / 2))
# Example using the datetime module
import datetime
```

```
now = datetime.datetime.now()
print(now)
print(now.year)
print(now.month)
print(now.day)
```

Output

```
C:\Users\Student
['file1.py', 'notes.txt', 'data.csv']
3.12.1 (tags/v3.12.1, Dec 5 2025, 10:00:00) [MSC v.1935 64 bit (AMD64)]
['example.py']
3.141592653589793
1.0
2026-02-18 10:30:45.123456
2026
2
18
```

1.4.4 User-Defined Modules in Python

User-defined modules are custom modules created by programmers to streamline their projects. These modules can include functions, classes, variables, and other reusable code components that can be shared and used across different Python scripts, making development more efficient and organized.

We can create a .py file that includes functions, variables, or classes, and then import that file into other Python programs to reuse its code.

Step 1: Create a module file (greetings.py)

```
# greetings.py
def say_hello(name):
return f"Hello, {name}!"
```

Step 2: Use the module in another file (main.py)

```
# main.py
import greetings
```

Output

Hello, Riya!

This example demonstrates how to define a basic function in a custom module and import it into a separate file to reuse the greeting functionality.

1.4.4.1 Creating a Module

To Create a Python module just we need to create a new file with a .py extension. In this file, you can define functions, classes, and variables that can be accessed and used in other Python programs. This method helps in organizing code efficiently and encourages code reuse across different projects.

Example:

```
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.")
```

Next, save the file with the name math_operations.py. This creates a module called math_operations, which includes four functions: add, subtract, multiply, and divide. These functions carry out basic arithmetic tasks and can be reused in other Python programs.

1.4.4.2 Import Statement

The import statement in Python is used to load modules or specific elements from modules into the current program's namespace. This enables developers to access and make use of the functions, classes, and variables defined within those modules.

Syntax:

```
import module_name
```

Example:

```
import math_operations
result = math_operations.add(5, 3)
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.

1.4.4.3 Naming a Module

Selecting an appropriate name for a Python module is important and should follow these key principles:

1. Be Descriptive: Choose a name that clearly reflects what the module does. This helps others quickly understand its function without needing to explore the code.
2. Keep It Short: Use a concise name to make it easier to type, remember, and use in your programs.
3. Use Lowercase Letters: Stick to lowercase naming for modules, following Python conventions. This differentiates them from class names and constants.
4. Use Underscores for Clarity: If the module name consists of multiple words, separate them with underscores (_) to enhance readability. For example, use "my_module" instead of "mymodule".
5. Avoid Naming Conflicts: Make sure your module name doesn't overlap with Python's built-in modules or keywords to prevent unexpected behavior and errors.
6. Make It Clear and Informative: Pick a name that clearly indicates what the module does. This helps both you and other developers understand its purpose without having to read through the entire code. For instance, a module dealing with string operations could be named "string_utils" or "str_helpers" to immediately convey its functionality.

1.4.4.4 Renaming a Module

To rename a module in Python, follow these steps:

1. Rename the Module File: Start by changing the name of the module file, keeping the .py extension intact. For example, if the file was originally named "old_module.py", rename it to "new_module.py".
2. Update Import Statements: Go through all your other code files and find any import statements that reference the old module name. Replace them with the new name. For instance, change "import old_module" to "import new_module".
3. Adjust Module References: Look for any parts of the code where the old module name is used (e.g., function or class calls) and update them accordingly. So, if the code includes something like "old_module.some_function()",itshould now be "new_module.some_function()".
4. Test the Code: Run your program to make sure everything works as expected after the changes. Check for any errors or issues, and fix them if needed.

1.4.4.5 Variables in Module

Variables defined in a Python module can be accessed and utilized by other programs that import that module. They act as storage units for data that can be shared across various parts of a program or even between multiple programs. This approach supports better data management and encourages modular, reusable code.

```
# my_module.py

my_variable = "Hello, World!"

def print_variable( ):

print(my_variable)
```

In this example, the module my_module defines a variable called my_variable, which is set to the string "Hello, World!". It also contains a function named print_variable() that outputs the value of my_variable.

1.4.5 Module Discovery and Installation

Python module management primarily involves the use of pip, the standard package installer, and the Python Package Index (PyPI) , the central repository for Python packages.

1.4.5.1 Module Discovery

Before installing a module, it is important to locate or discover it.

1. PyPI (Python Package Index)

A widely used approach for finding Python modules is by exploring PyPI, the central

online repository for Python packages. It contains thousands of third-party libraries that you can search or browse using relevant keywords, categories, or specific functionalities.

2. Online Communities and Resources

Platforms like Stack Overflow, Python-related Reddit communities, and various tech blogs often highlight or recommend useful Python libraries tailored to different programming needs.

3. Project Documentation

When dealing with existing projects, you can often find the required modules listed in their official documentation or in a requirements.txt file, which outlines all the dependencies needed to run the project.

1.4.5.2 Module Installation

Once a module is identified, we can install it using pip.

1. Installing a Package

```
pip install module_name
```

Replace module_name with the actual name (e.g., pip install requests)

2. Installing a Specific Version

```
pip install module_name==version_number
```

3. Installing from a Requirements File

To install multiple packages listed in a project's requirements.txt

```
pip install -r requirements.txt
```

Summarized Overview

The unit provides a comprehensive overview of Python modules and libraries, highlighting their role in promoting code reuse and simplifying development. Python libraries are collections of modules that offer pre-written functionality for various tasks, widely used in domains like data science, machine learning, and data visualization. Popular libraries such as NumPy, Pandas, Matplotlib, TensorFlow, and PyTorch are introduced along with their specific use cases. The concept of Python modules, individual .py files containing functions, variables, or classes, is also explained, including how to create, name, and import them into other programs. Built-in modules like math, os, and datetime, as well as user-defined modules, are discussed in detail. The unit also emphasizes the importance of managing dependencies using tools like pip and repositories like PyPI. The unit also guides on how to discover and install packages, manage versions, work with requirements.txt files, and use virtual environments to isolate project dependencies effectively.

Assignments

1. Explain the difference between a Python module and a Python library. Include examples of each and briefly describe how they support code reuse and modularity.
2. What is the role of pip and PyPI in Python? Discuss how packages are discovered and installed using these tools, and explain the syntax for installing specific versions.
3. Describe the purpose and benefits of using virtual environments in Python. Explain how they help in managing dependencies and avoiding conflicts between projects.
4. List and briefly describe any four widely used Python libraries. Explain their use cases and domains where they are most applicable (e.g., data science, game development, AI).
5. What are built-in and user-defined modules in Python? Compare both with examples and explain the advantages of creating custom modules.

Reference

1. Python Software Foundation. (2023). Python Standard Library. https://docs.python.org/3/library/
2. Python Packaging Authority. (2023). pip - The Python Package Installer. https://pip.pypa.io/
3. Python Packaging Authority. (2023). The Python Package Index (PyPI). https://pypi.org/
4. Oliphant, T. E. (2006). A guide to NumPy. Trelgol Publishing.
5. Chollet, F. (2017). Deep learning with Python. Manning Publications.

Suggested Reading

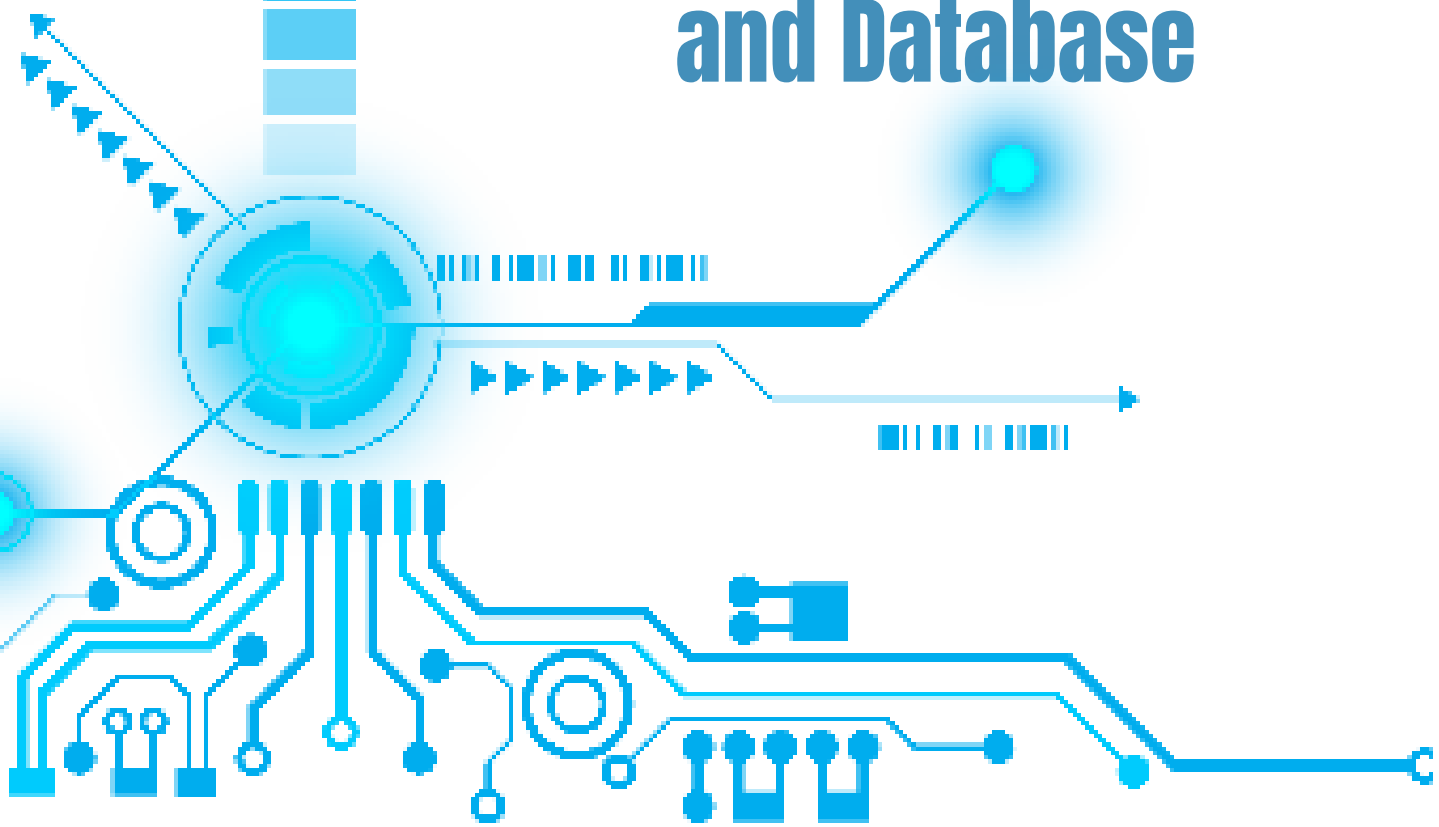
1. McKinney, W. (2017). *Python for data analysis: Data wrangling with pandas, NumPy, and IPython* (2nd ed.). O'Reilly Media.
2. VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. O'Reilly Media.
3. Sweigart, A. (2019). *Automate the boring stuff with Python: Practical programming for total beginners* (2nd ed.). No Starch Press.
4. Shaw, Z. A. (2017). *Learn Python 3 the hard way: A very simple introduction to the terrifyingly beautiful world of computers and code*. Addison-Wesley.
5. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

BLOCK 2

Object-Oriented Programming, File Handling, Error Handling and Database



1 UNIT Object-Oriented Programming

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the fundamental concepts of Object-Oriented Programming in Python such as classes, objects, constructors, inheritance, polymorphism, abstraction, and encapsulation
- demonstrate the use of constructors, methods, inheritance types, and access specifiers in creating structured Python programs
- implement polymorphism, magic methods, and abstraction using Python's abc module to design reusable and scalable programs
- differentiate between composition and inheritance, and analyze their use in solving real-world problems using Python classes and objects

Background

Object Oriented Programming (OOP) in Python evolved as a solution to the limitations of traditional procedural programming. In earlier approaches, programs were written as collections of functions that operated on shared data, often leading to complexity, redundancy, and reduced flexibility in large-scale systems. OOP addressed these issues by introducing the concept of organizing software around objects entities that bundle data and behavior together.

Python, being a high-level and versatile language, naturally supports OOP principles such as encapsulation, abstraction, inheritance, and polymorphism. These features allow developers to design modular, reusable, and maintainable code. Beyond theory, OOP concepts in Python are widely applied in building real-world systems, from graphical user interfaces and web applications to data analysis frameworks

and artificial intelligence models.

The study of OOP provides students with a deeper understanding of how software can be structured to reflect real life entities and their interactions. This not only simplifies problem solving but also prepares learners to design efficient solutions for practical applications such as file management, memory optimization, undo-redo operations, playlist handling, and implementing abstract models in simulations or games.

Keywords

Self-referential structures, Dynamic memory allocation, Access specifiers, Method overloading, Method overriding, Magic methods, Adjacency list

Discussion

Object Oriented Programming (OOP) in Python is a powerful programming paradigm that models real life entities using classes and objects. Instead of writing procedures or functions alone, OOP focuses on organizing code into objects, self-contained units that combine both data (attributes) and functions (methods) that operate on that data. This approach mirrors the way we think about the real world, making programs more intuitive, maintainable, scalable, and reusable.

OOP provides a structured way to manage complexity in large software projects by using several core concepts.

- Classes
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

2.1.1 Class

A class in Python is like a blueprint or template used to create objects. It defines the structure and behavior that the objects will have, including attributes (data) and methods (functions).

To understand the usefulness of classes, let's take a real-life example:

Imagine you are running a **dog shelter** and need to keep track of many dogs. Each dog has specific information like name, breed, age, and color.

If you try to manage this using lists or individual variables, it gets confusing very quickly. For example:

dog1 = ["Tommy", "Labrador", 5]

dog2 = ["Rocky", "Pug", 3]

You may forget what dog1[1] or dog2[2] refers to. Is it the breed? or age? This approach becomes messy and hard to maintain when dealing with many animals.

A **class** allows you to logically group related data and behavior. Instead of dealing with lists of mixed-up values, you can create a structured **Dog class** that clearly defines the properties (like name and age) and behaviors (like bark or eat) of each dog.

A **class** is a user-defined blueprint for creating objects that share the same attributes and methods. In Python, we use the class keyword to define a class.

- **Basic Syntax of a Class**

```
class ClassName:
    # Attributes and methods go here
    pass
```

The pass keyword is used as a placeholder when the class body is empty.

- **Creating an Empty Class**

```
class Dog:
    pass
```

This creates an empty Dog class. You can now use it to create Dog objects, even though it doesn't do anything yet. Later, you can add attributes and methods to it. Classes help you write cleaner and more organized programs by combining related data and functionality into reusable blocks.

2.1.2 Object

In Python, an **object** is an instance of a class that contains both **data (state)** and **functions (behavior)**. Everything in Python is actually treated as an object, even numbers, strings, lists, and dictionaries. For example, the number 12, the text "Hello, world", or a list like [1, 2, 3] are all objects because they store values and have built-in methods to perform actions.

Every object in Python has three important features:

- **State**: This refers to the data or attributes of the object. For example, the breed, age, and color of a dog.
- **Behavior**: This refers to the functions or methods an object can perform. A dog can bark, eat, or sleep. These are behaviors.
- **Identity**: This is the unique identity of each object, like a name that distinguishes one object from another.

Let's take a simple real-life example to understand this better. Imagine a class called Dog. When you create an object from this class, say dog1 = Dog(), you now have a specific dog. The **identity** of this dog can be its name, like "Bruno". Its **state** includes its breed ("Labrador"), age (5 years), and color ("brown"). Its **behavior** includes actions like barking or sleeping, which are defined by methods in the class. So, dog1.bark() would make the dog perform the bark action.

When you write obj = Dog(), you are creating an object called obj from the class Dog. This object will now hold its own values for the defined attributes and can use the methods available in the class. Objects help us bring real-world thinking into programming by allowing us to model things more naturally using state, behavior, and identity.

2.1.2.1 Methods in Python Classes

In Python, a **method** is a function that is defined inside a class and is used to define the **behavior** of an object. Just like attributes represent the state of an object, **methods represent actions** the object can perform. Methods always take *self* as their first parameter, which refers to the specific object that calls the method. This allows each object to behave according to its own data.

Let's continue with the Dog class. Suppose you want every dog to be able to bark or sleep. These actions can be defined as methods inside the class. For example:

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age
    def bark(self):
        print(f"{self.name} is barking!")
    def sleep(self):
        print(f"{self.name} is sleeping.")
```

Here, bark() and sleep() are **methods**. They define what a dog can do. When you create an object, say dog1 = Dog("Bruno", "Labrador", 5), you can call these methods using dot notation:

```
dog1.bark()    # Output: Bruno is barking!
dog1.sleep()   # Output: Bruno is sleeping.
```

These methods make it easy to control the behavior of each dog object. Even though all dog objects share the same methods, each object will respond using its own data. This is how methods help us implement real-world behaviors in programming.

2.1.3 Constructors

A **constructor** in Python is a special method used to **initialize objects** when they are created from a class. The constructor method in Python is called __init__(), and it is automatically executed as soon as a new object is created. The word init stands for "initialize", and it helps set up the initial state (attributes) of the object.

Every time you create a new object from a class, Python automatically calls the __init__() method. This method usually takes self as the first argument, followed by other parameters that you want to use for setting the object's attributes.

Let's see how this works in the context of our Dog class:

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age
```

In this example, the constructor __init__() accepts three values: name, breed, and age. It assigns these values to the object's attributes using the self keyword.

Now, when we create a dog object:

dog1 = Dog("Bruno", "Labrador", 5)

The constructor is automatically called, and it assigns:

- Bruno to self.name
- Labrador to self.breed
- 5 to self.age

This ensures that every dog object starts with the correct initial data. The constructor helps make object creation flexible and clean, especially when dealing with many attributes. Without it, you'd have to assign values one by one after creating the object, which would be inefficient and messy.

2.1.4 Inheritance

Inheritance is one of the core principles of object-oriented programming (OOP) in Python. It allows one class (called the child or derived class) to reuse the code and properties of another class (called the parent or base class). This makes it easier to organize and manage large programs without having to write the same code multiple times. Instead of rewriting the same variables or functions in every class, you can define them once in a parent class and let the child class use them. This feature not only reduces redundancy but also improves the structure and readability of code.

To understand inheritance more clearly, imagine a real-life example of a school. At a school, there are different types of people: students, teachers, and staff. All of them share some common characteristics like name, age, and address. Instead of creating these properties separately for each type of person, we can create a common class called Person that holds all the shared attributes. Then, we can create specific classes like Student, Teacher, and Staff that inherit the properties of the Person class and add their own unique features. For example, the Student class might include grade and roll number, while the Teacher class might have subject and salary.

In Python, this relationship is built using the concept of inheritance. We use the keyword class to define a class, and to inherit from another class, we simply place the parent class name inside parentheses after the child class name.

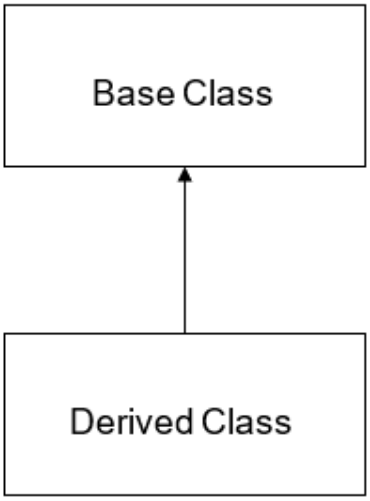


Fig 2.1.1 Inheritance

In Python, inheritance is implemented by passing the parent class name inside parentheses when defining the child class. The syntax of

```
class Parent:
    # Parent class code
class Child(Parent):
    # Child class code
```

This simple syntax allows the child class to inherit all accessible properties of the parent class.

For instance:

```
class Person:
    def show(self):
        print("This is a person")
class Student(Person):
    pass
s = Student()
s.show()
```

In this example, the Student class inherits the method show() from the Person class. Even though the Student class doesn't define its own method, it can still use the method from its parent.

2.1.4.1 Types of Inheritance in Python

Python supports **five main types of inheritance**, depending on the number of parent and child classes and the way they are connected.

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance

Single inheritance is the simplest form where a child class inherits from only one parent class. It is used when there is a clear one-to-one relationship between classes. As shown in figure, class B inherits from class A, which means that all the attributes and methods defined in class A become directly available to class B without redefining them.

For example, if we define a class Vehicle with a method start(), we don't need to rewrite this method in a class Car; we can simply make Car inherit from Vehicle. In Python, this is done using parentheses like class Car(Vehicle):.

```
class Vehicle:
    def start(self):
        print("Vehicle started")
```

```
class Car(Vehicle):
    pass
c = Car()
c.start()  # Output: Vehicle started
```

This example demonstrates **single inheritance**, where a child class inherits from just one parent class. The class Car doesn't define its own start() method but can still use it because it's inherited from Vehicle.

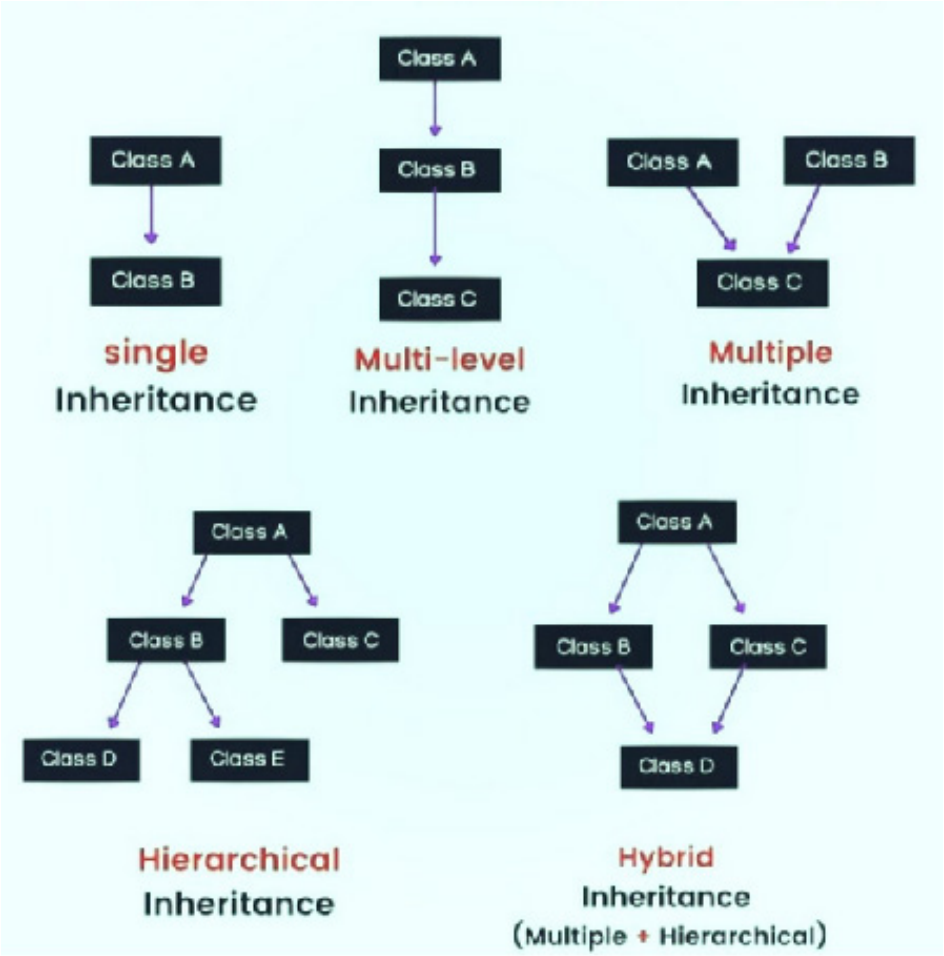


Fig 2.1.2 Inheritance types

2. Multiple Inheritance

In multiple inheritance, a child class inherits from more than one parent class. This is useful when a class needs to combine behaviors from different sources. As shown in Figure 2.1.3 class C inherits from both class A and class B, so it can use all the methods and attributes from both classes.

For instance, imagine a class Father with a method work(), and another class Mother with a method care(). A class Child can inherit from both, gaining the ability to use both methods.

```
class Father:
    def work(self):
        print("Father works")
class Mother:
    def care(self):
        print("Mother cares")
class Child(Father, Mother):
    pass
c = Child()
c.work()   # Output: Father works
c.care()   # Output: Mother cares
```

3. Multilevel Inheritance

Multilevel inheritance forms a chain where a class inherits from a parent class, which in turn inherits from another class. As shown in Figure 2.1.4, class B inherits from class A, and then class C inherits from class B. This means class C gets all the properties and behaviors of class B, as well as those of class A. The inheritance flows through the levels, allowing class C to access everything from both B and A.

A child class inherits from a parent, which in turn inherits from a grandparent. For example, class Grandparent has a method guide(), class Parent inherits from it and adds teach(), and class Child inherits from Parent. The Child class can now access both guide() and teach() methods.

```
class Grandparent:
    def guide(self):
        print("Grandparent guides")
class Parent(Grandparent):
    def teach(self):
        print("Parent teaches")
```

```
class Child(Parent):
    def learn(self):
        print("Child learns")
c = Child()
c.guide()   # Output: Grandparent guides
c.teach()   # Output: Parent teaches
c.learn()   # Output: Child learns
```

4. Hierarchical Inheritance

In hierarchical inheritance, multiple child classes inherit from the same parent class. This pattern models various types that share a common base.

As illustrated in Figure 2.1.5, class B, class C, and class D all inherit from a single parent class A. This implies that B, C, and D can each use the methods and attributes defined in A, but they are independent of one another. Such a structure promotes code reusability and allows multiple subclasses to maintain consistency by sharing a common base class.

Hierarchical inheritance occurs when multiple child classes inherit from the same parent class. Suppose Dog and Cat both inherit from a class Animal that has a method eat(). Both subclasses automatically gain access to the eat() method.

```
class Animal:
    def eat(self):
        print("Animal eats")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
class Cat(Animal):
    def meow(self):
        print("Cat meows")
d = Dog()
c = Cat()
d.eat()   # Output: Animal eats
c.eat()   # Output: Animal eats
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more inheritance types. It helps in modeling complex relationships.

```
# Base class
class Vehicle:
    def show_type(self):
        print("Vehicle: Used for transportation")
# First level child (single inheritance)
class Car(Vehicle):
    def car_features(self):
        print("Car: Has 4 wheels and AC")
# Another base class for multiple inheritance
class Electric:
    def battery_info(self):
        print("Electric: Runs on battery")
# Hybrid class (inherits from Car and Electric)
class ElectricCar(Car, Electric):
    def features(self):
        print("ElectricCar: Eco-friendly and silent")
# Create object of ElectricCar
ecar = ElectricCar()
ecar.show_type()       # Inherited from Vehicle
ecar.car_features()    # Inherited from Car
ecar.battery_info()    # Inherited from Electric
ecar.features()        # Defined in ElectricCar
#Output
Vehicle: Used for transportation
Car: Has 4 wheels and AC
Electric: Runs on battery
ElectricCar: Eco-friendly anInheritance
```

Hybrid inheritance in Python is a type of inheritance that combines two or more forms of inheritance, such as single, multiple, or multilevel inheritance, to model complex relationships between classes. It is useful when a single inheritance pattern is not enough to represent real-world scenarios.

In the given example, the base class Vehicle defines a common method show_type() that describes all vehicles in general. The class Car inherits from Vehicle through single inheritance and adds its own method car_features() to describe features specific to cars. Another independent base class, Electric, defines the method battery_info() to represent electric-related properties. The class ElectricCar demonstrates hybrid inheritance by inheriting from both Car and Electric. This means it indirectly inherits from Vehicle through Car and also directly from Electric, combining single and multiple inheritance patterns. When an object of ElectricCar is created, it can access methods from Vehicle, Car, Electric, and its own method features(), showcasing how hybrid inheritance allows reusing and combining functionality from multiple sources.

Python follows a specific search path called Method Resolution Order (MRO) to determine which method to call when there are multiple parent classes, ensuring consistent and predictable behavior. A real-life example of hybrid inheritance is a Tesla car, which is both a car (inheriting features like wheels and comfort) and electric (inheriting battery-related features), while also being a vehicle in general. This approach helps in building flexible and efficient class structures that reflect real-world relationships.

2.1.4.2 Access Specifiers in Different Types of Inheritance

In Python, access specifiers (or access modifiers) control how members of a class (variables or methods) are accessed in inheritance. There are three types:

Public (x): Can be accessed in all child classes regardless of inheritance type. Also accessible from outside the class. Inherited and fully accessible by child classes.

Protected (_x): Meant to be accessed inside the class and its subclasses only. Inherited and accessible in all types of inheritance, but not intended for outside use. Inherited by child classes and accessible within them, but not recommended to access from outside.

Private (__x): Not inherited directly. Not directly accessible in child classes. Python internally name-mangles private variables (e.g., _ClassName__x), so they are effectively hidden from inheritance.

Table 2.1.1 Accessibility of different classes

Access	Public	Protected	Private
Same class	Yes	Yes	Yes
Derived classes	Yes	Yes	No
Outside classes	Yes	No	No

2.1.5 Polymorphism

Polymorphism in Python refers to the ability of a single function or method to operate differently based on the object it is used with. The term polymorphism originates from Greek, meaning "many forms". In Python, this allows a method or function to adapt its behavior depending on the type of object invoking it.

For example, consider different animal classes where each animal produces a distinct sound. You can define a method named make_sound() that is used across all animal types, but each animal will produce its own specific sound. This means that the same method name can perform different actions depending on the object calling it.

```
class Dog:
def make_sound(self):
print("Bark")
class Cat:
def make_sound(self):
print("Meow")
dog = Dog()
cat = Cat()
dog.make_sound()
cat.make_sound()
#Output
Bark
Meow
```

Need of Polymorphism

- Ensures consistent interfaces across different classes.
- Allows objects to respond differently to the same method call.
- Promotes loose coupling by relying on shared behavior, not specific types.
- Enables writing flexible, reusable code that works across types.
- Simplifies testing and future extension of code.

2.1.5.1 Types of Polymorphism

Polymorphism in object-oriented programming (OOP) can be classified into two main types

1. Compile-time Polymorphism
2. Runtime Polymorphism

1. Compile-time Polymorphism (Static Polymorphism)

Compile-time polymorphism happens when the method to be called is determined at compile time, before the program is run. This type of polymorphism is achieved using-method overloading or operator overloading.

Method overloading is when multiple methods have the same name but differ in the number or type of their parameters. The correct method is chosen based on the arguments passed when calling the method.

```
class MathOperations:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c
math = MathOperations()
print(math.add(5, 10, 15))
#Output
30
```

2. Runtime Polymorphism (Dynamic Polymorphism)

Runtime polymorphism happens when the method to be called is determined at runtime, during the execution of the program. This type of polymorphism is achieved using method overriding.

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in the parent class. The version of the method that is called depends on the object that is used to invoke it.

```
class MathOperations:
    def calculate(self, a, b):
        return a + b
class AdvancedMath(MathOperations):
    def calculate(self, a, b):
        return a * b
```

```
basic = MathOperations()
advanced = AdvancedMath()
print(&amp;quot;Basic addition:&amp;quot;, basic.calculate(5, 3))
print(&amp;quot;Advanced multiplication:&amp;quot;, advanced.calculate(5, 3))
#Output:
8
15
```

2.1.6 Abstraction and Encapsulation

Object-Oriented Programming (OOP) is a programming paradigm that helps organize code using objects. Two of its most essential concepts are Abstraction and Encapsulation. These concepts allow developers to build systems that are easier to understand, maintain, and scale.

Abstraction means hiding the complex details and showing only the essential features of an object or system. It focuses on what an object does, rather than how it does it. Encapsulation means binding the data (variables) and the code (methods) that manipulate the data into a single unit a class. It also helps protect data from being directly accessed or modified.

2.1.6.1 Abstraction

Abstraction is the concept of hiding the internal implementation details and showing only the essential features of the object.

Purpose

- Focuses on what an object does rather than how it does it.l
- Helps in reducing complexity by suppressing lower-level details.

When you drive a car, you only need to know how to operate the steering wheel, accelerator, and brakes (what to do) , not how the engine or transmission system works (how it's done). Achieved using abstract classes and methods from the abc module (Abstract Base Class).

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
```

```
class Dog(Animal):
    def sound(self):
        return "Bark"
d = Dog()
print(d.sound())
# Output: Bark
```

2.1.6.2 Encapsulation

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the concept of hiding the internal details of how an object works and only exposing a controlled interface to the outside world. In Python, this is achieved by wrapping data (variables) and methods (functions) into a single unit, usually a class, and restricting direct access to some of the object's components. This protects the object's integrity by preventing unintended interference and misuse.

Encapsulation allows programmers to define access levels for class members using public, protected, or private access modifiers. By doing so, sensitive data can be kept hidden from direct access and only modified through well-defined interfaces like getter and setter methods. This not only enhances data security but also makes the code more modular, maintainable, and easier to debug.

1. Public Members

Public members are class attributes (variables) or methods (functions) that can be accessed from anywhere, both inside and outside the class. In Python, by default, all members of a class are public unless explicitly specified otherwise.

```
class Student:
def __init__(self, name, age):
self.name = name
self.age = age
s = Student(&amp;quot;Helen&amp;quot;, 23) # Create an instance of Student
print(&amp;quot;Name:&amp;quot;, s.name)
print(&amp;quot;Age:&amp;quot;, s.age)
#Output
Name: Helen
Age: 23
```

In the above example, name and age are public members of the Student class. They can be accessed and modified directly using the objects.

2. Protected Members

Protected members are variables or methods that can be accessed within the class and its subclasses, but should not be accessed directly from outside the class. In Python, protected members are defined by prefixing the name with a single underscore (e.g., _name).

```
class Student:
    def __init__(self, name, age):
        self._name = name
        self._age = age
s = Student(&amp;quot;Bob&amp;quot;, 21)
print(&amp;quot;Name:&amp;quot;, s._name)
print(&amp;quot;Age:&amp;quot;, s._age)
Output:
Name: Bob
Age:21
```

3. Private Members

Private members are variables or methods in a class that cannot be accessed directly from outside the class. In Python, we make something private by putting two underscores (__) in front of its name.

```
class Student:
    def __init__(self, name, marks):
        self.__name = name
        self.__marks = marks
    def display(self):
        print(&amp;quot;Name:&amp;quot;, self.__name)
        print(&amp;quot;Marks:&amp;quot;, self.__marks)
s = Student(&amp;quot;Helen&amp;quot;, 28)
s.display()
#Output:
Name: Alice
Marks: 28
```

2.1.7 Magic-dunder methods

Magic methods in Python, often called **dunder methods** (short for "double underscore"), are special predefined methods that begin and end with double underscores, such as __init__, __str__, __add__, and __ge__. They allow Python objects to interact seamlessly with built-in functions, operators, and language syntax. These methods are not usually called directly; instead, they are automatically invoked by Python when specific operations are performed on objects. For example, when you use the + operator on two numbers, Python internally calls the __add__() method to perform the addition. Similarly, when you print an object, Python calls its __str__() method to get a human-readable string representation. All built-in classes in Python define several magic methods, and you can check them by using the dir() function.

One of the key uses of magic methods is **operator overloading**, which allows you to redefine the behavior of operators for user-defined classes. For instance, by overriding __add__() in a custom class, you can define exactly what happens when two objects of that class are added. This is useful when working with custom data types, such as a Distance class, where + can be used to add feet and inches together with proper unit conversion. Similarly, comparison operators can be customized by overriding methods like __ge__() for >= or __lt__() for <, enabling you to compare complex objects meaningfully.

The __new__() method is another important magic method, called before __init__(). It is responsible for creating and returning a new instance of the class, after which __init__() initializes its attributes. This is similar to the new operator in languages like Java or C#. The __str__() method, on the other hand, is used to return a string representation of the object, which is especially useful for making printed output more readable.

By using magic methods, you can make your custom classes behave like Python's built-in types, giving them natural and intuitive behavior with operators, built-in functions, and type conversions. This not only improves code readability but also makes object-oriented designs more expressive and flexible. In short, magic methods are a powerful feature of Python that bridge the gap between user-defined objects and the core language features, enabling objects to integrate smoothly into Python's syntax and behavior.

Examples

1. __init__() – Object Initialization

This method is called **automatically** after an object is created to initialize its attributes.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
p = Person("Alice", 25)  # Calls __init__
print(p.name, p.age)
#Output :Alice 25
```

When you create p = Person("Alice", 25), Python first creates the object, then calls __init__() to set name and age.

2. __str__() – String Representation for Humans

Defines what gets displayed when you use print() on an object.

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages
    def __str__(self):
        return f"{self.title} ({self.pages} pages)"
b = Book("Python Basics", 300)
print(b)  # Calls __str__()
#Output
Python Basics (300 pages)
```

Without __str__(), printing an object would show something like <*__main__.Book object at 0x000*>

3. __add__() – Operator Overloading for +

Let's define how + works between objects.

```
class Distance:
    def __init__(self, feet, inches):
        self.feet = feet
        self.inches = inches
    def __add__(self, other):
```

```
        total_inches = self.inches + other.inches
        total_feet = self.feet + other.feet + total_inches // 12
        total_inches = total_inches % 12
        return Distance(total_feet, total_inches)
    def __str__(self):
        return f"{self.feet} ft {self.inches} in"
d1 = Distance(5, 8)
d2 = Distance(3, 7)
print(d1 + d2)  # Calls __add__()
#Output
9 ft 3 in
```

Now the + operator can combine distances meaningfully.

4. __ge__() – Greater Than or Equal To >=

Lets you compare custom objects using >=.

```
class Distance:
    def __init__(self, feet, inches):
        self.feet = feet
        self.inches = inches
    def __ge__(self, other):
        return (self.feet * 12 + self.inches) >= (other.feet * 12 + other.inches)
d1 = Distance(5, 8)
d2 = Distance(3, 7)
print(d1 >= d2)  # Calls __ge__()
#Output
True
```

Without overriding, Python wouldn't know how to compare two custom objects.

5. __new__() – Creating the Object

Called **before** __init__(), responsible for creating the object.

```
class Employee:
    def __new__(cls):
        print("__new__() is called")
        instance = super().__new__(cls)
        return instance
    def __init__(self):
        print("__init__() is called")
        self.name = "Satya"
emp = Employee()
#Output
__new__() is called
__init__() is called
```

You rarely override __new__(), but it's important when creating immutable objects or customizing creation.

2.1.8 Composition

Composition is an object-oriented programming (OOP) concept where one class contains an object of another class as a part of its state, instead of inheriting from it. In simple words, it means **"has-a"** relationship rather than **"is-a"** relationship.

For example, a Car **has an** Engine, but a car is not an engine. In composition, we create complex objects by combining simpler, reusable components. This approach makes the code more flexible, because we can change parts of the system without affecting the whole. If one component changes, other parts of the code don't break as easily, unlike in inheritance where changes in the parent class may affect all child classes. Composition is often preferred when you want to build functionality by combining multiple classes rather than relying on a single inheritance chain.

In composition, one class is made up of one or more objects from other classes, and it uses their functionality as needed. This approach offers looser coupling than inheritance, so changes in one class do not directly impact others. Composition encourages modular design, making it easier to maintain and extend the code.

Example of Composition:

```
class Engine:
    def start(self):
        print("Engine starts")
```

```
class Car:
    def __init__(self):
        self.engine = Engine()  # Car has an Engine
    def drive(self):
        self.engine.start()
        print("Car is moving")
my_car = Car()
my_car.drive()
#Output
Engine starts
Car is moving
```

Benefits of Composition

1. **Reusability** – You can reuse existing classes by including them as components, instead of rewriting code.
2. **Loose Coupling** – The composite and component classes are loosely connected. Changes to one have little effect on the other, making systems more maintainable.
3. **Easy to Modify** – Because of loose coupling, new features can be added or existing ones updated with minimal impact on other parts of the code.
4. **More Flexible than Inheritance** – Inheritance creates a fixed "is-a" hierarchy, but composition lets you freely swap or change components, which is especially useful when requirements change over time.

Table 2.1.2 Difference between Composition and Inheritance

Aspect	Composition	Inheritance
Relationship	"Has-a" (one class contains another)	"Is-a" (one class is a type of another)
Coupling	Loose coupling (components can be replaced easily)	Tight coupling (changes in parent affect children)
Reusability	Achieved by combining independent objects	Achieved by extending existing classes
Flexibility	High – components can be swapped or changed	Low – rigid structure due to hierarchy
Example	A Car **has an** Engine	A Dog **is an** Animal

Summarized Overview

Object-Oriented Programming (OOP) in Python is a programming style that models real-world entities as objects, each containing data (attributes) and operations (methods). Unlike procedural programming, which focuses on functions and step-by-step instructions, OOP emphasizes modularity, reusability, and easier maintenance of code. Python's built-in support for OOP features such as encapsulation, inheritance, abstraction, and polymorphism allows developers to build flexible and scalable applications. These principles make it possible to design software systems that closely mirror real-world problems for example, using classes and objects to represent files in operating systems, nodes in a linked list, or accounts in a banking system. In practice, OOP with Python is widely used in areas like memory management, implementation of data structures such as stacks and queues, graph representation, playlist management in media players, and undo-redo functionality in text editors. A solid understanding of OOP in Python equips learners to design structured, efficient, and adaptable solutions across a variety of domains.

Assignments

1. Define Object-Oriented Programming (OOP). How does it differ from procedural programming? Give an example in Python.
2. Explain the concept of classes and objects in Python with a real-life example.
3. Write a Python program to create a class Student with attributes name and marks. Create at least two objects and display their details.
4. Describe the process of traversing a singly linked list. Why is a temporary pointer required?
5. Discuss at least three real-life applications of linked lists and explain why linked lists are suitable for these cases.
6. Differentiate between inheritance and composition with the help of Python code examples.
7. What are the advantages of implementing stacks and queues using linked lists instead of arrays?

8. Write a short note on how linked lists are used in: a) Undo/Redo operations b) Music or image playlists
9. Explain how dynamic memory allocation is achieved through linked lists. Give a suitable example.
10. Create a Python class Polynomial to represent a polynomial using a linked list. Implement methods for addition and display.

Reference

1. Guttag, J. V. (2021). *Introduction to Computation and Programming Using Python: With Application to Computational Modeling and Understanding Data* (3rd ed.). MIT Press.
2. Reddy, Y. V. K. (2022). *Data Structures and Algorithms with Python*. McGraw Hill.
3. Tenenbaum, A. M., Langsam, Y., & Augenstein, M. J. (2019). *Data Structures Using C and Python*. Pearson.
4. Liang, Y. D. (2020). *Introduction to Programming Using Python* (2nd ed.). Pearson.

Suggested Reading

1. Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). Green Tea Press.
2. Severance, C. R. (2016). *Python for Everybody: Exploring Data in Python 3*. CreateSpace Independent Publishing Platform.
3. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
4. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT File Handling

Learning Outcomes

After completion of this unit, the learner will be able to:

- recall the basic file I/O operations such as reading, writing, and appending in Python
- explain the use of context managers in file handling and differentiate between handling CSV, JSON, and XML files
- describe the functions of the os and shutil modules for file and directory manipulation
- implement programs to perform file I/O operations and manipulate directories using CSV, JSON, and XML files

Background

Studying File Handling is essential because files are the primary way to store, access, and manage data in real-world applications. It allows programs to read, write, and append information, ensuring that data can be preserved and reused beyond program execution. The topic includes basic file I/O operations, the use of context managers for efficient resource handling, and working with structured data formats like CSV, JSON, and XML. It also covers file and directory manipulation using modules such as os and shutil, which are vital for organizing and managing files in an operating system. Learning these concepts helps in handling data securely, avoiding memory leaks, and ensuring data integrity. A major benefit is the ability to process large datasets, such as logs, reports, or configuration files, in an automated and efficient manner. File handling skills are also widely applicable in areas like data analysis, machine learning, and web development, where structured

file formats are commonly used. Additionally, understanding file and directory operations prepares students to build practical applications such as file managers, data converters, and backup tools. Mastering these techniques improves coding efficiency and resource management. Overall, studying file handling equips learners with practical programming skills that are essential for both academic projects and professional software development.

Keywords

Binary mode, context manager, shutil, mkdir, makedirs, rmdir, listdir, getcwd, chdir, copytree

Discussion

In practical programming scenarios, data often needs to be saved and accessed later for various purposes, such as storing user details, configuration files, or system logs. Python's file handling capabilities offer a straightforward and efficient way to work with files on a computer. It enables developers to create, open, read, write, and update files through built-in functions. With minimal code, Python can manage files in different modes, append new content, or replace existing data. This makes it a highly effective tool for maintaining data in a structured and permanent form. For any Python programmer, mastering file handling is crucial, particularly when dealing with automation, reporting, or data processing. Once the significance of file handling is clear, the next step is learning how to carry out core operations such as opening files, reading their contents, writing new information, and updating stored data.

2.2.1 Opening a file in Python

To work with files in Python, the first step is to open them using the built-in **open( )** function. This function gives the program access to a file stored on the system and enables different operations such as reading, writing, or appending. A file can be opened in various modes depending on the task, with each mode designed for a specific purpose. Once the file is opened, Python provides a clear and simple syntax to carry out the required actions effectively.

Syntax:

open("filename", "mode")

Here, **"filename"** refers to the file you want to work with, and **"mode"** defines how

you want to interact with it - whether by reading, writing, or appending. File modes are passed as the second argument to the **open( )** function and determine the way data is handled. The commonly used modes are:

"x" – Create (creates a new file and throws an error if it already exists)

"r" – Read (default mode)

"w" – Write (creates a new file or overwrites an existing one)

"a" – Append (adds data at the end of the file without overwriting existing content)

2.2.2 Creating a New File

To create a file that does not already exist, you can use the "x" mode with the open() function.

Example:

```
f = open("newfile.txt", "x")
```

If a file named *newfile.txt* is already present in the directory, Python will generate an error instead of overwriting it.

2.2.3 Reading from a File

Consider a file named *example.txt* that contains the following text:

Hello, Python learners!

Welcome to file handling.

Example:

```
with open("example.txt", "r") as f:
    data = f.read()
    print(data)
```

Output:

Hello, Python learners!

Welcome to file handling.

Here, the with statement ensures that the file is automatically closed after the operation is completed.

2.2.4 Writing to a File

To add content to a file, the **"w" mode** is used. If the file already exists, its previous content will be replaced with the new data.

Example:

```
with open("example.txt", "w") as f:
    f.write("This is a new line of text.")
```

After this operation, reading the file will show only the newly written text, as the old content is erased.

2.2.5 Appending to a File

If you want to add new content to a file without removing its existing data, use the **"a" mode**.

Example:

```
with open("example.txt", "a") as f:
    f.write("\nThis line is added to the file.")
```

Output after appending:

```
This is a new line of text.
This line is added to the file.
```

2.2.6 Binary Mode

Binary mode is applied when dealing with non-text files such as images, audio, executables, or other files that store raw binary data. Unlike text mode, opening a file in binary mode treats the file as a sequence of bytes rather than characters. No encoding or decoding is performed—data is read or written exactly as it exists in the file.

Binary mode is typically combined with other modes:

'rb' → Read a binary file

'wb' → Write a binary file (overwrite existing content)

'ab' → Append binary data

Example:

```
with open("photo.jpg", "rb") as file:
    content = file.read()
```

In this case, the image is loaded as raw byte data, which can later be used for tasks such as processing, transmission, or copying.

2.2.7 Context Managers

In programming, resources such as file operations or database connections are frequently used but are also limited in availability. The key challenge is to ensure these resources are released once they are no longer needed. Failing to do so can result in resource leaks, which may slow down the system or even cause it to crash. Python addresses this issue through context managers, which offer a clean and reliable way to allocate and

release resources automatically, ensuring proper management even when unexpected errors occur.

Why Context Managers are Needed

Context managers in Python are essential because they simplify and secure resource management. They provide automatic cleanup, ensuring that resources such as files or database connections are released without requiring manual intervention. This prevents resource leaks, guaranteeing proper release even when unexpected errors occur. By using the with statement, code becomes cleaner and more readable, eliminating the need for lengthy try-finally blocks. Context managers are also exception-safe, meaning cleanup is always performed regardless of whether an error is raised. Additionally, they offer custom control through the __enter__ and __exit__ methods, allowing developers to manage a wide range of resources effectively.

Risks of Not Closing Resources

If resources such as files are not properly closed, it can lead to critical problems like exhausting the system's available file descriptors. As a result, the program—and sometimes even the entire system—may slow down or crash.

Example: The following code opens a file repeatedly without closing it, which eventually causes resource exhaustion and triggers a system error:

```
file_descriptors = []

for x in range(100000):

    file_descriptors.append(open('test.txt', 'w'))
```

This will produce an error:

```
Traceback (most recent call last):

  File "context.py", line 3, in

OSError: [Errno 24] Too many open files: 'test.txt'
```

This error occurs because too many files remain open at once. Using **context managers** prevents such issues by ensuring resources are automatically closed after use.

2.2.7.1 Built-in Context Manager for File Handling

In Python, file handling is one of the most common scenarios where proper resource management is essential. The with statement acts as a built-in context manager that guarantees a file is closed automatically after its use, even if an error interrupts the process.

Example: The code below shows how a file can be opened and read safely using a context manager:

```
with open("test.txt") as f:

    data = f.read()
```

By using with, there is no need to explicitly call close(), and it also prevents resource leaks by handling unexpected failures gracefully.

2.2.7.2 Creating a Custom Context Manager Class

When building a context manager using a class, two key methods must be defined:

__enter__(): sets up the resource and returns it.

__exit__(): releases or cleans up the resource (e.g., closing a file).

Example:

The following program demonstrates how Python initializes the object, enters the context, executes the block, and then exits while performing cleanup:

```
class ContextManager:
    def __init__(self):
        print('init method called')
    def __enter__(self):
        print('enter method called')
        return self
    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('exit method called')
with ContextManager() as manager:
    print('with statement block')
```

Output:

init method called

enter method called

with statement block

exit method called

Explanation:

__init__() initializes the object.

__enter__() is triggered when the with block begins and returns the object.

The block of code inside with executes.

__exit__() runs at the end of the block to perform cleanup tasks.

2.2.7.3 File Handling with a Custom Context Manager

By applying the idea of custom context managers, we can design a class to manage files efficiently. The following FileManager class demonstrates how to open a file, perform

read/write operations, and automatically close it once the task is complete.

Example:

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None
    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file
    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.file.close()
with FileManager('test.txt', 'w') as f:
    f.write('Test')
print(f.closed)
```

Output:

True

Explanation:

__enter__() opens the file and returns the file object.

The file can be accessed inside the with block.

__exit__() ensures that the file is properly closed after the block.

print(f.closed) verifies that the file is indeed closed.

2.2.7.4 Database Connection Management with Context Manager

Similar to files, database connections are also limited resources. Forgetting to close a connection may lead to resource exhaustion. A context manager provides a safe and automatic way to handle such connections. The following example demonstrates a MongoDB connection manager:

Example:

```
from pymongo import MongoClient
class MongoDBConnectionManager:
    def __init__(self, hostname, port):
```

```
        self.hostname = hostname
        self.port = port
        self.connection = None
    def __enter__(self):
        self.connection = MongoClient(self.hostname, self.port)
        return self.connection
    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.connection.close()
with MongoDBConnectionManager('localhost', 27017) as mongo:
    collection = mongo.SampleDb.test
data = collection.find_one({'_id': 1})
    print(data.get('name'))
```

Explanation:

__enter__() establishes the MongoDB connection.

The mongo object inside the with block represents the active client connection.

Database operations can be performed safely within the block.

__exit__() guarantees the connection is closed automatically.

2.2.8 Handling CSV Files in Python

A CSV (Comma-Separated Values) file is a simple text format where each row represents a record, and individual fields are separated by commas. Because of its clarity and ease of use, CSV is widely applied in spreadsheets and database systems.

In Python, several useful operations can be performed when working with CSV files.

2.2.8.1 Reading a CSV File

To read data from a CSV file in Python, the csv.reader object is used. A CSV file is first opened as a standard text file with Python's built-in open() function, which provides a file object. This file object is then passed to csv.reader for processing. In the example below, the file is opened in read mode, the header row is extracted, and all subsequent rows are stored for further operations.

```
import csv
filename = "aapl.csv"   # CSV file name
fields = []             # To store column headers
rows = []               # To store row data
```

```
with open(filename, 'r') as csvfile:
    csvreader = csv.reader(csvfile)   # Create reader object
    fields = next(csvreader)          # Extract header
    for row in csvreader:             # Read each row
        rows.append(row)
    print("Total number of rows: %d" % csvreader.line_num)  # Row count
print('Field names are: ' + ', '.join(fields))
print('\nFirst 5 rows are:\n')
for row in rows[:5]:
    for col in row:
        print("%10s" % col, end=" ")
    print('\n')
```

This script demonstrates how to load column names, count rows, and display the first few records of a CSV file efficiently.

Output:

```
Total no. of rows: 185
Field names are:Date, Open, High, Low, Close, Adj Close, Volume

First 5 rows are:

2014-09-29 100.589996 100.690002  98.040001  99.620003  93.514290  142718700

2014-10-06  99.949997 102.379997  98.309998 100.730003  94.556244  280258200

2014-10-13 101.330002 101.779999  95.180000  97.669998  91.683792  358539800

2014-10-20  98.320000 105.489998  98.220001 105.220001  98.771042  358532900

2014-10-27 104.849998 108.040001 104.699997 108.000000 101.380676  220230600
```

Explanation:

The with open(...) statement opens the CSV file in read mode securely using a context manager.

csv.reader(csvfile) converts the file into a CSV reader object.

The next(csvreader) function retrieves the first row, which contains the column headers.

By looping through the csvreader, each subsequent row is added to the rows list.

Finally, the program prints the total number of rows, the field names, and the first five data rows in a well-formatted manner.

2.2.8.2 Reading CSV Files into a Dictionary using csv

In Python, the csv module provides the DictReader class, which allows reading a CSV file directly into dictionaries. Each row in the file is converted into a dictionary where the headers act as keys.

For example, consider a file named employees.csv with the following content:

name,department,birthday_month

John Smith,HR,July

Alice Johnson,IT,October

Bob Williams,Finance,January

Example: The code below reads each row as a dictionary and stores it inside a list.

```
import csv
with open('employees.csv', mode='r') as file:
    csv_reader = csv.DictReader(file)  # Create DictReader
    data_list = []  # List to store dictionaries
    for row in csv_reader:
        data_list.append(row)
for data in data_list:
    print(data)
```

Output:

{'name': 'John Smith', 'department': 'HR', 'birthday_month': 'July'}

{'name': 'Alice Johnson', 'department': 'IT', 'birthday_month': 'October'}

{'name': 'Bob Williams', 'department': 'Finance', 'birthday_month': 'January'}

Explanation:

- with open(...) opens the file safely using a context manager.
- csv.DictReader(file) reads each row as a dictionary, mapping headers to values.
- data_list.append(row) collects all dictionaries into a list for further use.

Writing Data to a CSV File

To write information into a CSV file, the file must be opened in write mode ('w'). Once opened, the file object is passed to csv.writer, which creates a writer object to handle the writing process. The data can then be written row by row or all at once.

Example:

```
import csv

# Define header and data rows

fields = ['Name', 'Branch', 'Year', 'CGPA']

rows = [

    ['Nikhil', 'COE', '2', '9.0'],

    ['Sanchit', 'COE', '2', '9.1'],

    ['Aditya', 'IT', '2', '9.3'],

    ['Sagar', 'SE', '1', '9.5'],

    ['Prateek', 'MCE', '3', '7.8'],

    ['Sahil', 'EP', '2', '9.1']

]

filename = "university_records.csv"

with open(filename, 'w') as csvfile:

    csvwriter = csv.writer(csvfile)      # Initialize writer object

    csvwriter.writerow(fields)           # Write column headers

    csvwriter.writerows(rows)            # Write all rows at once
```

Explanation:

- fields stores the header names, while rows holds the actual data in a list of lists.
- with open(..., 'w') opens the CSV file in write mode, ensuring it is closed automatically afterward.
- csv.writer(csvfile) prepares the writer object to handle CSV operations.
- writerow(fields) writes the header as the first line.
- writerows(rows) adds all the data rows in one step.

2.2.8.3 Writing a Dictionary to a CSV File

When working with dictionary data in Python, the DictWriter class from the csv module is used to write key-value pairs directly into a CSV file. Each dictionary represents a row, with keys corresponding to column headers.

Example:

```
import csv
```

```
# Dictionary data to be written
mydict = [
    {'branch': 'COE', 'cgpa': '9.0', 'name': 'Nikhil', 'year': '2'},
    {'branch': 'COE', 'cgpa': '9.1', 'name': 'Sanchit', 'year': '2'},
    {'branch': 'IT',  'cgpa': '9.3', 'name': 'Aditya',  'year': '2'},
    {'branch': 'SE',  'cgpa': '9.5', 'name': 'Sagar',   'year': '1'},
    {'branch': 'MCE', 'cgpa': '7.8', 'name': 'Prateek', 'year': '3'},
    {'branch': 'EP',  'cgpa': '9.1', 'name': 'Sahil',   'year': '2'}
]
# Define column headers
fields = ['name', 'branch', 'year', 'cgpa']
# File name
filename = "university_records.csv"
# Writing dictionaries to CSV
with open(filename, 'w') as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames=fields)  # Create DictWriter object
    writer.writeheader()       # Write headers
    writer.writerows(mydict)   # Write all rows from list of dictionaries
```

Explanation:

- mydict contains multiple dictionaries, each representing a student's record.
- fields defines the order of columns in the CSV file.
- DictWriter is initialized with the file object and the defined field names.
- writeheader() adds the column headers as the first row.
- writerows(mydict) writes all dictionaries into the CSV file row by row.

2.2.9 Handling JSON files

JSON (JavaScript Object Notation) is a lightweight file format commonly used to store and exchange data, especially between a server and a web application. It is widely adopted for representing structured information in a simple and human-readable way. In this section, we will explore how to work with JSON data in Python. Python provides a built-in module named json, which makes it easy to parse, create, and manipulate JSON data.

Note: In Python, JSON data is typically handled as a string.

2.2.9.1 Importing JSON Module and Parsing JSON in Python

In Python, before working with any module, it must be imported using the import statement. To handle JSON data, we import the json module as shown below:

```
# Importing the JSON module

import json
```

Parsing JSON – Converting JSON to Python

The json module provides two main functions for parsing:

- load() → Parses JSON data from a file.
- loads() → Parses JSON data from a string.

Parsing a JSON String

The loads() function is commonly used to convert a JSON string into a Python dictionary.

Syntax:

```
json.loads(json_string)
```

Example: Converting JSON string into a Python dictionary

```
import json

# JSON string

employee = '{"name": "Nitin", "department": "Finance", "company": "GFG"}'

# Convert string to Python dictionary

employee_dict = json.loads(employee)

print("Data after conversion:")

print(employee_dict)

print(employee_dict['department'])

print("\nType of data:")

print(type(employee_dict))
```

Output:

```
Data after conversion:

{'name': 'Nitin', 'department': 'Finance', 'company': 'GFG'}

Finance

Type of data:

<class 'dict'>
```

Using loads(), JSON strings can be seamlessly transformed into Python dictionaries, enabling easier manipulation of data.

2.2.9.2 Reading a JSON File

In Python, the load() method is used to read data from a file that contains a JSON object. For example, if you have a file named student.json with student details, you can easily read and process its contents.

Syntax:

json.load(file_object)

Example: Reading a JSON file in Python

```
import json

# Open JSON file

f = open('data.json')

# Convert JSON object to a Python dictionary

data = json.load(f)

# Iterate through the JSON data

for i in data:

    print(i)

# Close the file

f.close()
```

Output:

The JSON data will be loaded and displayed as a list of Python dictionaries.

Note:

- When a JSON file is read in Python, its content is automatically converted into a list of dictionaries (or other corresponding Python data types).
- In this example, the file was opened using open() and closed using close(). However, it's generally better to use a context manager (with statement) for safer file handling.
- For detailed file handling operations in Python, refer to File Handling in Python.

2.2.9.3 Converting Python Objects to JSON

Python provides the dump() and dumps() methods from the json module to convert Python objects into JSON format.

The following Python data types can be easily converted into JSON strings:

- dict → object
- list, tuple → array
- str → string
- int, float → number
- True → true
- False → false
- None → null

2.2.9.4 Converting to JSON String

The dumps() method converts a Python object into a JSON string.

Syntax:

json.dumps(dictionary, indent)

- dictionary → Python object to be converted
- indent → spacing used for indentation in the JSON string

Example: Convert Python dictionary to JSON string

```
import json

# Data to be converted

dictionary = {

   "name": "sunil",

   "department": "HR",

   "Company": "GFG"

}

# Serializing to JSON string

json_object = json.dumps(dictionary)

print(json_object)
```

Output:

```
{"name": "sunil", "department": "HR", "Company": "GFG"}
```

2.2.9.5 Writing JSON to a File

The dump() method is used to write JSON data directly into a file.

Syntax:

json.dump(dictionary, file_pointer)

- dictionary → Python object to convert
- file_pointer → File opened in write ("w") or append ("a") mode

Example: Writing dictionary to a JSON file

```
import json

# Data to be written

dictionary = {

   "name": "Nisha",

   "rollno": 420,

   "cgpa": 10.10,

   "phonenumber": "1234567890"

}

# Writing JSON to file

with open("sample.json", "w") as outfile:

   json.dump(dictionary, outfile)
```

Output:

A new file sample.json will be created containing the JSON object.

2.2.10 Working with XML in Python

XML (Extensible Markup Language) is a markup language designed to be easily understood by both humans and machines. It provides a structured way to encode documents using a defined set of rules. In Python, we can read, parse, and write XML files efficiently using different libraries.

The process of reading an XML file and breaking it down into logical components is called parsing. Hence, when we say "reading XML," it usually refers to parsing the XML document.

For XML parsing in Python, two commonly used libraries are:

1. BeautifulSoup (used along with the lxml parser)
2. ElementTree

2.2.10.1 Using BeautifulSoup with lxml Parser

To read and write XML files, the BeautifulSoup library is often used. You can install it by running:

pip install beautifulsoup4

BeautifulSoup works with Python's built-in HTML parser, but it also supports third-party parsers like lxml, which is powerful for parsing XML/HTML documents. To install lxml, run:

pip install lxml

With these libraries, we can:

- Read and parse XML data
- Extract required information
- Create and write XML files with structured content

In the following sections, we'll first learn how to read and parse XML files, and later, how to create and write XML files in Python.

Reading Data from an XML File

Parsing an XML file generally involves two main steps:

1. Locating the tags – Identifying the elements or nodes within the XML structure.
2. Extracting data – Retrieving the content or attributes stored inside those tags.

Example:

XML file used:

<?xml version="1.0" encoding="utf-8"?>

<saranghe>

<child name="Frank" test="0">

FRANK likes EVERYONE

</child>

<unique>

Add a video URL in here

</unique>

<child name="Texas" test="1">

TEXAS is a PLACE

</child>

<child name="Frank" test="2">

Exclusively

</child>

<unique>

Add a workbook URL here

</unique>

<data>

Add the content of your article here

<family>

Add the font family of your text here

</family>

<size>

Add the font size of your text here

</size>

</data>

</saranghe>

```
from bs4 import BeautifulSoup

# Step 1: Read the XML file content into a variable

with open('dict.xml', 'r') as file:

    xml_data = file.read()

# Step 2: Parse the XML data using BeautifulSoup with the "xml" parser

soup = BeautifulSoup(xml_data, "xml")

# Step 3: Locate all occurrences of the <unique> tag

unique_tags = soup.find_all('unique')

print(unique_tags)

# Step 4: Use find() to retrieve the first <child> tag with attribute name="Frank"

child_tag = soup.find('child', {'name': 'Frank'})

print(child_tag)

# Step 5: Access the value of a specific attribute ('test') from the <child> tag

attribute_value = child_tag.get('test')

print(attribute_value)
```

- Explanation (rewritten):

The XML file is first opened and read into a variable.

BeautifulSoup with the "xml" parser is used to process the file content.

The program searches for all occurrences of the <unique> tag.

It then finds the first <child> tag that has an attribute name="Frank".

Finally, it extracts the value of the attribute test from that tag and prints it.

Output

```
[<unique>
  Add a video URL in here
 </unique>, <unique>
  Add a workbook URL here
 </unique>]
<child name="Frank" test="0">
  FRANK likes EVERYONE
 </child>
0
```

Writing an XML File

Creating or modifying an XML file is a straightforward process since XML is stored as plain text without any special encoding. However, updating certain parts of an XML document usually requires parsing it first to locate the relevant sections. Once parsed, you can make the necessary changes and then save or display the updated content.

Example:

```
from bs4 import BeautifulSoup

# Read the contents of the XML file

with open('dict.xml', 'r') as f:

    data = f.read()

# Parse the XML data using BeautifulSoup

bs_data = BeautifulSoup(data, 'xml')

# Replace the value of the attribute 'test' for a specific tag

# In this case, find all <child> tags with name="Frank"

for tag in bs_data.find_all('child', {'name': 'Frank'}):

    tag['test'] = "WHAT !!"

# Print the modified XML in a formatted structure

print(bs_data.prettify())
```

Output

```
<?xml version="1.0" encoding="utf-8"?>
<saranghe>
 <child name="Frank" test="WHAT !!">
  FRANK likes EVERYONE
 </child>
 <unique>
  Add a video URL in here
 </unique>
 <child name="Texas" test="1">
  TEXAS is a PLACE
 </child>
 <child name="Frank" test="WHAT !!">
  Exclusively
 </child>
 <unique>
  Add a workbook URL here
 </unique>
 <data>
  Add the content of your article here
  <family>
   Add the font family of your text here
  </family>
  <size>
   Add the font size of your text here
  </size>
 </data>
</saranghe>
```

2.2.10.2 Using Element Tree

The ElementTree module in Python offers a powerful set of tools for working with XML files. Its biggest advantage is that it is part of Python's standard library, so no external installation is required. Since XML data is inherently hierarchical, it can naturally be represented in the form of a tree structure, making it easier to navigate and manipulate. The ElementTree module allows us to represent an entire XML document as a single tree and provides methods to read, write, and modify its elements effectively.

In the following example, we demonstrate how to read data from an XML file using ElementTree. First, the ElementTree class from Python's xml module is imported with the alias ET. Then, the XML file is parsed using the ET.parse() method, and the root element (the parent tag) is retrieved with getroot(). From there, we can display the root tag, access attributes of its child tags, and extract the text content stored in specific elements.

Example:

```
# Importing ElementTree with alias ET

import xml.etree.ElementTree as ET
```

```
# Parsing the XML file

tree = ET.parse('dict.xml')

# Getting the root (parent) tag

root = tree.getroot()

# Printing the root tag and its memory reference

print(root)

# Printing the attributes of the first child tag

print(root[0].attrib)

# Printing text from the first sub-tag of the 5th child element

print(root[5][0].text)
```

Output

```
<Element 'saranghe' at 0x7fe97c2cee08>
{'name': 'Frank', 'test': '0'}

   Add the font family of your text here
```

Writing XML Files

Next, let's explore how to write data into an XML document. In this example, we will build an XML file from scratch using the ElementTree module.

We begin by creating a root (parent) tag named chess using ET.Element('chess'). All subsequent tags will be nested inside this root. After defining the root, we create a sub-element called Opening under the chess tag with the help of ET.SubElement(). Within Opening, we then add two more child elements named E4 and D4. Using the set() method, we assign attributes to these tags, while the text property allows us to insert text content inside them.

Once the structure is ready, we use ET.tostring() to convert the XML tree into a bytes object (despite its name, tostring() returns bytes instead of a string in some implementations). Finally, the XML data is written to a file named gameofsquares.xml by opening it in wb (write binary) mode. This ensures that the XML content is saved properly to disk.

Example (Writing XML File with ElementTree):

```
import xml.etree.ElementTree as ET

# Create the root (parent) tag named 'chess'

root = ET.Element('chess')

# Add a child element 'Opening' under the root
```

```
opening = ET.SubElement(root, 'Opening')

# Create two sub-elements inside 'Opening'

move1 = ET.SubElement(opening, 'E4')

move2 = ET.SubElement(opening, 'D4')

# Assign attributes to the sub-elements

move1.set('type', 'Accepted')

move2.set('type', 'Declined')

# Insert text content inside the sub-elements

move1.text = "King's Gambit Accepted"

move2.text = "Queen's Gambit Declined"

# Convert the XML structure into a bytes object

xml_data = ET.tostring(root)

# Save the XML content into a file in binary write mode

with open("GFG.xml", "wb") as file:

    file.write(xml_data)
```

In this rewritten example:

- The root element chess acts as the parent.
- Opening is added as a sub-element, and moves E4 and D4 are nested under it.
- Attributes (Accepted, Declined) and text descriptions are assigned to each move.
- Finally, the XML tree is serialized into bytes and stored in a file named GFG.xml.

2.2.11 Python Directory Management

Python Directory Management refers to working with and managing folders (directories) in a filesystem through Python. It involves operations such as creating, removing, navigating, renaming, and listing directory contents programmatically. Python simplifies these tasks using built-in modules like os, os.path, pathlib, and shutil.

Python directory management is essential because it simplifies common folder operations such as creating, deleting, renaming, and navigating directories. It helps organize files effectively, ensuring structured workflows within programs and applications. With built-in modules like os, os.path, and pathlib, Python provides cross-platform support, making directory handling seamless across different operating systems. Additionally, it

enables dynamic control over the current working directory while efficiently managing nested folder structures.

os and os.path Module

The os module in Python offers functions to interact with the operating system, allowing tasks like handling files, directories, processes, and environment variables. It is a cross-platform module, meaning it works seamlessly across different operating systems such as Windows, Linux, and macOS.

2.2.11.1 Working with Directories in Python

Python provides the os, os.path, and shutil modules to work with directories and files. These modules allow us to create, rename, move, delete, and check directories easily.

1. Creating a New Directory

- os.mkdir(path) → Creates a single folder. (Error if it already exists)
- os.makedirs(path) → Creates a folder along with all parent folders if needed.

```
import os

# create one folder

os.mkdir("my_directory")

# create nested folders

os.makedirs("parent/child")
```

2. Get Current Working Directory (CWD)

- os.getcwd() → Returns current folder path as a string.
- os.getcwdb() → Returns the same, but in bytes.

```
import os

print("String format:", os.getcwd())

print("Byte format:", os.getcwdb())
```

3. Renaming a Directory

- os.rename(old, new) → Renames a folder/file.
- os.renames() → Renames and creates missing directories if required.

```
import os

os.rename("my_directory", "renamed_directory")
```

4. Changing Current Directory

- os.chdir(path) → Switches to a new working directory.

```
import os
print("Before:", os.getcwd())
os.chdir('/home/user/Desktop/')
print("After:", os.getcwd())
```

5. Listing Files in a Directory

- os.listdir(path) → Lists all files/folders in a directory.

```
import os
print("Files:", os.listdir(os.getcwd()))
```

6. Removing a Directory

- os.rmdir(path) → Removes an empty folder.
- shutil.rmtree(path) → Removes a folder and all its contents.

```
import shutil
shutil.rmtree("my_directory")  # Deletes everything inside too
```

7. Check if Path is a Directory

- os.path.isdir(path) → Returns True if the path is a folder.

```
import os
print(os.path.isdir("/"))  # True
```

8. Get Size of Directory or File

- os.path.getsize(path) → Returns size in bytes.

```
import os
print(os.path.getsize(os.getcwd()))
```

9. Access & Modification Times

- os.path.getatime(path) → Last access time.
- os.path.getmtime(path) → Last modified time.

```
import os, time
print("Access:", time.ctime(os.path.getatime("/")))
print("Modified:", time.ctime(os.path.getmtime("/")))
```

10. shutil Module (Advanced File/Folder Handling)

- shutil.copytree(src, dst) → Copy a folder and all contents.
- shutil.rmtree(path) → Delete a folder and everything inside.
- shutil.move(src, dst) → Move/rename a folder or file.

```
import shutil

shutil.copytree("source", "destination")

shutil.move("source", "new_location")
```

- os → Basic folder/file handling (create, rename, list, remove).
- os.path → Check properties like size, type, timestamps.
- shutil → Advanced operations (copy, move, delete recursively).

Summarized Overview

File and directory handling in Python is an important concept that allows programmers to store, access, and organize data efficiently. Python makes it simple to perform operations such as creating, reading, writing, and appending files. It also supports working with binary files where data is handled as raw bytes. To ensure resources are managed properly, context managers using the with statement automatically open and close files, even if errors occur. This prevents resource leakage and keeps programs safe and efficient. Python also provides support for handling structured data formats like CSV, JSON, and XML. CSV files are useful for working with spreadsheets and tables, JSON is widely used for data exchange between applications, and XML is helpful for handling hierarchical data. Each of these formats can be read and written easily using Python libraries such as csv, json, and xml. For managing folders and system paths, Python provides the os and os.path modules. These allow developers to create new directories, rename them, change the current working directory, and list the contents of folders. The shutil module extends these features by allowing copying, moving, and removing entire directories and their contents. These modules work across different operating systems, making Python programs portable and flexible. By combining file handling with directory management, programmers can automate tasks such as logging, reporting, and data processing. These skills are useful in applications like database management, web development, and system automation. Overall, learning file and directory handling in Python helps programmers develop reliable and organized programs that can handle real world data effectively.

Assignments

1. Explain the importance of file handling in Python with real-world examples.
2. Describe different file opening modes in Python (r, w, a, x, rb, wb, ab) with examples.
3. What are context managers in Python, and how do they help in resource management?
4. Differentiate between writing and appending to a file in Python with suitable examples.
5. What is a CSV file, and how can it be read using csv.reader and csv.DictReader?
6. Explain the process of converting Python objects to JSON and JSON to Python objects with examples.
7. What is XML parsing in Python? Compare the use of BeautifulSoup and ElementTree for XML handling.
8. Discuss how Python's os, os.path, and shutil modules are used for directory management.

Reference

1. https://www.geeksforgeeks.org/python/python-exception-handling/
2. https://nptel.ac.in/courses/106106145

Suggested Reading

1. Matthes, Eric. Python crash course: A hands-on, project-based introduction to programming. no starch press, 2023.

2. Matthes, Eric. Python crash course: A hands-on, project-based introduction to programming. no starch press, 2023.
3. Ramalho, Luciano. Fluent Python: Clear, concise, and effective programming. " O'Reilly Media, Inc.", 2015.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3 UNIT Exception Handling

Learning Outcomes

After completion of this unit, the learner will be able to:

- identify different types of errors in Python such as syntax and runtime errors
- describe the purpose and structure of exception handling using try, except, and finally blocks
- recognize the need for custom exceptions and explain how they are defined and used
- list common debugging techniques including logging and traceback
- explain the importance of unit testing and compare basic features of unittest and pytest.

Background

Python has become one of the most widely used programming languages in both academic and professional environments due to its simplicity, readability, and versatility. As with any programming language, ensuring the correctness and reliability of Python code is a fundamental aspect of software development. However, during program execution, developers often encounter errors that disrupt the normal flow of a program. These errors, if not identified and handled properly, can lead to program crashes, incorrect outputs, or security vulnerabilities.

Understanding the types of errors such as syntax and runtime errors is essential for debugging and maintaining code. Python provides a built-in mechanism known as exception handling to deal with runtime errors in a controlled and structured

manner. Through the use of try, except, and finally blocks, developers can write robust code that gracefully handles unexpected situations without terminating the entire program.

Additionally, as programs grow in complexity, debugging becomes a critical process. Tools like the logging and traceback modules allow developers to trace the flow of execution and diagnose problems efficiently. To ensure individual units of code work as intended, unit testing frameworks like unittest and pytest are employed. These practices not only improve code quality but also contribute to software reliability, maintainability, and professional programming standards.

Keywords

Syntax Error, Exception Handling, Runtime Error, Debugging, Unit Testing

Discussion

2.3.1 Errors in Python

When developing a Python program, the primary objective is to ensure that it executes correctly and produces the expected results. However, during program execution, situations may arise where the program does not run as intended. These situations are caused by errors.

In Python, an error refers to any issue in the code that interrupts the normal flow of execution. Errors can occur due to a variety of reasons such as incorrect syntax, invalid operations, or unexpected inputs. Identifying and understanding these errors is crucial for effective program debugging and smooth program execution.

Python errors are broadly classified into the following categories:

1. Syntax Errors

A syntax error occurs when the rules of the Python language are violated. Python has a specific set of syntax rules, and if the code does not adhere to these rules, it cannot be executed.

The Python interpreter checks the code for correctness before running it. If a syntax error is found, the program will not start execution until the error is corrected.

Example:

```
print("Python Programming"
```

In the above code, the closing parenthesis) is missing. Since Python expects the parentheses to be properly matched, it raises a SyntaxError before execution begins.

2. Runtime Errors

A runtime error occurs while the program is running. Even though the syntax of the program is correct, an error is encountered during execution that causes the program to stop.

These errors are usually caused by invalid operations, such as dividing by zero, accessing a non-existent file, or using variables that have not been defined.

Example:

```
number = 10 / 0
    print(number)
```

Although the syntax is correct, dividing a number by zero is mathematically undefined. Therefore, Python raises a ZeroDivisionError when it encounters this operation during program execution.

2.3.2 Exception Handling in Python

In Python, errors that occur during program execution are known as exceptions. When an exception occurs, it disrupts the normal flow of the program, leading to abnormal termination if not handled properly. Exception handling is the mechanism provided by Python to detect, manage, and respond to such errors in a controlled manner, ensuring the program continues to operate or terminates gracefully. Python uses specific keywords like try, except, and finally to implement exception handling.

1. The try Block

The try block is used to write the code that might cause an error. If an error occurs inside the try block, Python immediately stops executing that block and moves to the except block.

Example:

```
try:
    number = int(input("Enter a number: "))
    print("You entered:", number)
except:
    print("Invalid input. Please enter a number.")
```

In this program, the try block contains code to take a number as input and display it. If the user enters a valid integer, the program runs without any issue. However, if the

user types something that cannot be converted into an integer (like text or symbols), Python raises an error. This error is caught by the except block, and a friendly message is displayed instead of crashing the program.

2. except Block

The except block contains the code that will be executed if an error occurs in the try block. You can also specify the type of exception to handle specific errors.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You cannot divide a number by zero.")
```

Here, the try block attempts to divide 10 by 0, which is not allowed in mathematics. This raises a ZeroDivisionError. The except ZeroDivisionError block specifically catches that error and displays a message explaining the problem. Without this block, the program would stop with an error message.

3. finally Block

The finally block is always executed, whether or not an error occurs. It is commonly used for tasks like closing files, releasing resources, or displaying final messages.

Example:

```
try:
    file = open("sample.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("The file was not found.")
finally:
    print("Execution completed.")
```

In this example, the try block tries to open and read a file named sample.txt. If the file does not exist, the FileNotFoundError is caught by the except block, and an error message is shown. Regardless of whether the file exists or not, the finally block executes and displays "Execution completed." This shows that it finally runs every time.

2.3.3 Custom Exception in Python

In Python, a custom exception is an error type created by the programmer to represent a specific problem in the program that is not already covered by the built-in exceptions. It allows developers to make error messages more meaningful and relevant to their application.

A custom exception is created by defining a new class that inherits from the built-in Exception class.

Example:

```
class NegativeNumberError(Exception):
    pass
def check_number(num):
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")
    else:
        print("The number is positive.")
try:
    number = int(input("Enter a number: "))
    check_number(number)
except NegativeNumberError as e:
    print("Error:", e)
```

In this program, a custom exception called NegativeNumberError is created by inheriting from the built-in Exception class. The function check_number() is used to verify if the entered number is positive. If the number is negative, the custom exception is raised with a specific error message. In the try block, the program takes input from the user and calls the check_number() function. If the entered number is negative, the NegativeNumberError is caught by the except block, and an appropriate error message is displayed. This example demonstrates how custom exceptions help handle specific error cases more clearly and make the program easier to maintain.

2.3.4 Debugging Techniques

When you write a Python program, sometimes it may not work the way you want. This could be because of mistakes in the code. Debugging is the process of finding and fixing these mistakes.

Here are some common and helpful techniques in Python for debugging:

1. Logging

Instead of using print() to see what your program is doing, you can use logging. It helps you keep track of what's happening in your code and also lets you save the messages.

Example:

```
import logging
logging.basicConfig(level=logging.INFO)
def divide(a, b):
    logging.info("Trying to divide %s by %s", a, b)
    try:
        return a / b
    except ZeroDivisionError:
        logging.error("Cannot divide by zero.")
        return None
divide(10, 2)
divide(5, 0)
```

In this program, we use the logging module instead of print() to display messages about what the code is doing. The line logging.basicConfig(level=logging.INFO) sets the logging level to show important messages like INFO and ERROR. When the divide() function is called, it logs a message saying what numbers are being divided. If the division is successful, the result is returned. But if the program tries to divide by zero, which is not allowed in math, it logs an error message using logging.error(). This is better than using print() because logging allows you to control the type of messages shown (like info or error), and you can easily save these messages to a file if needed. Logging is especially helpful in larger programs where you want to track the program's behavior or find out why something went wrong.

2. Traceback

In Python, when an error happens, the program usually shows a message called a traceback. A traceback tells you where the error happened in your code, which line caused the problem, and what type of error it is. You can also use the traceback module to print this information clearly in your own way.

Example:

```
import traceback
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Oops! Something went wrong.")
        traceback.print_exc()
divide(10, 0)
```

In the example above, the program tries to divide two numbers using a function. If the second number is zero, Python raises a ZeroDivisionError, and the program prints a message saying something went wrong. Then, it uses traceback.print_exc() to show the full error details. This is very helpful for finding out exactly where and why your program crashed. It's a useful tool when you are trying to fix errors because it shows the full path the program took before the error occurred.

2.3.5 Unit testing with unittest and pytest

Unit testing is the process of checking small parts of a program, such as individual functions or methods, to make sure they work correctly. This helps programmers find and fix mistakes early, before they become bigger problems. Python provides two main tools for writing unit tests: unittest, which is built into Python, and pytest, which is an external tool that you can install separately. Both tools help ensure that your program behaves as expected, and they are very useful for debugging and maintaining code.

2.3.5.1 Unit Testing with unittest

The unittest module comes built-in with Python, so you can use it without installing anything. It follows a class-based structure where you create a test class that inherits from unittest.TestCase. Inside this class, you write test methods that start with the word test_. These test methods use special functions like assertEqual() to compare the expected result with the actual result returned by your function.

Here is a simple example using unittest:

```
import unittest
def multiply(a, b):
    return a * b
class TestMath(unittest.TestCase):
    def test_multiply(self):
        self.assertEqual(multiply(3, 4), 12)
        self.assertEqual(multiply(0, 5), 0)
        self.assertEqual(multiply(-2, 3), -6)
if __name__ == '__main__':
    unittest.main()
```

In this example, we define a function called multiply() and a test class called TestMath. The method test_multiply() checks whether the multiplication results are correct using self.assertEqual(). When you run this program, unittest will test all the cases and tell you if any of them fail. This makes it easy to identify and fix errors in your code.

2.3.5.2 Unit Testing with pytest

pytest is another popular testing tool that many developers prefer because it is simpler

and more flexible. Unlike unittest, pytest does not require you to create classes. You can just write test functions directly using the assert keyword. However, pytest is not built into Python, so you need to install it first by running the command pip install pytest.

Here is a simple example using pytest:

```
def add(a, b):
    return a + b
def test_add():
    assert add(2, 3) == 5
    assert add(0, 0) == 0
    assert add(-1, 1) == 0
```

To run this test, save the code in a file named something like test_example.py, and then open your terminal or command prompt and run the command pytest test_example.py. If all the tests pass, pytest will show a success message. If a test fails, it will clearly show which test failed and what went wrong. This makes pytest a very beginner-friendly tool for testing Python code.

Summarized Overview

In Python programming, errors represent disruptions in the normal flow of execution caused by issues within the code. These errors are primarily classified into syntax errors and runtime errors. Syntax errors arise when the code violates the structural rules of the Python language, preventing the program from executing. In contrast, runtime errors occur during the execution of syntactically correct code and typically result from operations such as division by zero or accessing undefined variables. To manage such occurrences, Python provides a robust exception handling mechanism using constructs like try, except, and finally. Code that may raise an exception is enclosed within a try block, while the except block addresses specific exceptions if they occur. The finally block is executed regardless of whether an exception was raised, making it suitable for resource cleanup tasks.

Furthermore, Python supports the creation of custom exceptions by allowing users to define new classes that inherit from the built-in Exception class. This facilitates more meaningful and application-specific error handling. Debugging techniques such as utilizing the logging and traceback modules assist developers in identifying and diagnosing errors efficiently. To ensure the correctness and reliability of individual code components, unit testing is employed. Python offers two primary testing frameworks: unittest, which follows a class-based approach, and pytest, which provides a simpler, function-based testing style. These tools contribute significantly to software quality assurance and maintainability.

Assignments

1. Differentiate between syntax errors and runtime errors in Python. Provide one code example for each type and explain the error raised.
2. Explain the use of try, except, and finally blocks in Python exception handling with a suitable program. Describe how each block behaves during execution.
3. Write a Python program that raises and handles a custom exception. Clearly define the custom exception class and demonstrate how it is used in the program.
4. Discuss the advantages of using the logging module over the print() function for debugging in Python. Write a sample program that demonstrates the use of logging.
5. What is a traceback in Python? Illustrate its use with an example using the traceback module. Explain how it helps in debugging.
6. Compare unittest and pytest in terms of structure, ease of use, and functionality. Write a simple test case using each framework to test a basic mathematical function.
7. Write a Python program that reads a number from the user and checks whether it is negative. Use appropriate exception handling to manage invalid input and raise a custom exception for negative numbers.

Reference

1. https://www.geeksforgeeks.org/python/python-exception-handling/
2. https://nptel.ac.in/courses/106106145

Suggested Reading

1. Matthes, Eric. *Python crash course: A hands-on, project-based introduction to programming*. no starch press, 2023.
2. Matthes, Eric. *Python crash course: A hands-on, project-based introduction to programming*. no starch press, 2023.
3. Ramalho, Luciano. *Fluent Python: Clear, concise, and effective programming*. " O'Reilly Media, Inc.", 2015.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT Iterators

Learning Outcomes

After completion of this unit, the learner will be able to:

- describe the concept of iterators in Python and explain how they follow the iterator protocol using the __iter__() and __next__() methods.
- differentiate between iterables and iterators, and demonstrate the use of built-in functions iter() and next() for sequential data access.
- construct custom iterators and generator functions using the yield keyword, and explain how they provide lazy evaluation for efficient data processing.
- apply generator expressions in Python programs to create memory-efficient iterators and integrate them with built-in functions for data handling.

Background

When working with data in Python, efficiency and memory management are very important. Often, we do not want to store entire datasets in memory at once, especially when dealing with large files, continuous data streams, or infinite sequences. Instead, it is better to process data one piece at a time. Python provides two powerful tools to achieve this: iterators and generators.

An **iterator** is a special object that allows sequential access to elements, producing one value at a time while remembering its state. This approach avoids loading all data at once and ensures smooth processing even for very large collections. A **generator**, on the other hand, is a simpler way of creating iterators in Python.

Using the yield keyword or generator expressions, generators provide values lazily, meaning they are produced only when required.

The study of iterators and generators is essential for writing efficient Python programs. By mastering these concepts, learners can process data streams, work with large files, and implement memory-friendly solutions. This background provides the foundation for exploring how iterators, the __iter__() and __next__() methods, generator functions, and generator expressions make Python programming more effective.

Keywords

Iterator ,StopIteration ,Generator, Yield, Return, Lazy Evaluation, Generator Function

Discussion

Imagine you're reading a big book. You don't read all the pages at once, you read one page at a time.

When working with large amounts of data (like reading millions of lines from a file), it's not always a good idea to store everything in memory at once.

Iterators and Generators help you access data one item at a time, saving memory and making your code more efficient.

This saves memory and makes programs faster and smoother, especially when working with large data.

2.4.1 Iterator

An iterator in Python is a special type of object that allows you to access elements of a sequence or collection one at a time without storing the entire sequence in memory at once. It follows a specific set of rules called the iterator protocol, which requires two methods: __iter__() and __next__(). The __iter__() method returns the iterator object itself, and the __next__() method returns the next value each time it is called. When there are no more values to return, __next__() must raise the StopIteration exception. Iterators are especially useful when working with large datasets, infinite sequences, or data streams where it's impractical to store everything in memory at once.

Iterable vs Iterator

An iterable is any Python object that can return its members one at a time. Examples

include lists, tuples, dictionaries, and strings. These objects return an iterator when passed to the iter() function. An iterator is the actual object that keeps track of the current position and returns the next element each time __next__() is called.

Example:

```
numbers = [10, 20, 30]
it = iter(numbers)   # get iterator
print(next(it))      # 10
print(next(it))      # 20
print(next(it))      # 30
```

Use of Iterators

We use iterators to process data in a memory efficient and on demand manner. Instead of creating an entire list of results before processing, an iterator generates each item only when it is requested. This approach is ideal for tasks like reading very large files, streaming data from the internet, or working with infinitely long sequences (such as generating prime numbers). Iterators are also at the heart of Python's looping constructs when you use a for loop, Python internally uses an iterator to get each item.

Using an iterator generally involves two built-in functions: iter() and next(). The iter() function takes any iterable object (like a list) and returns an iterator. The next() function retrieves the next element from the iterator. When there are no elements left, StopIteration is raised. Most of the time, you don't manually call next() because Python's for loop does it automatically behind the scenes.

Example:

```
numbers = [10, 20, 30]
it = iter(numbers)
print(next(it))  # 10
print(next(it))  # 20
print(next(it))  # 30
```

next(it) here would raise StopIteration

Advantages of Iterators

One major advantage of iterators is memory efficiency. They produce items one at a time, so they don't require storing all elements in memory. This makes them suitable for working with large datasets. Iterators also allow for lazy evaluation, meaning values are generated only when needed, which can save time when you don't need the entire dataset. Additionally, iterators can represent infinite sequences that are generated

continuously without ever completing, such as live sensor readings or streaming log data.

Disadvantages of Iterators

While iterators have many benefits, they also have some limitations. An iterator can be exhausted once all elements have been retrieved, it cannot be reset or reused without creating a new iterator. This means if you need to process the same data again, you must regenerate the iterator. Iterators are also one directional, you can only move forward through the data, not backward. Finally, since they don't store data, you can't directly check the total length of an iterator without consuming it.

2.4.2 __iter__() and __next__()

2.4.2.1 The __iter__() Method

The __iter__() method is one of the two special methods that define an iterator in Python. This method should return the iterator object itself. In custom iterator classes, it is usually defined with return self because the class itself acts as its own iterator. This method is automatically called when an iterator is passed to a loop or when iter() is used.

2.4.2.2 The __next__() Method

The __next__() method is responsible for returning the next value from the iterator. It also needs to keep track of the current position in the sequence so it knows which item to return next. If there are no more elements to return, it must raise a StopIteration exception to signal the end of iteration. This method is automatically called each time the next() function is used or when a loop asks for the next item.

Example of a Custom Iterator Using __iter__() and __next__()

```
class CountUpTo:
    def __init__(self, limit):
        self.limit = limit
        self.current = 1
    def __iter__(self):
        return self  # returns the iterator object
    def __next__(self):
        if self.current <= self.limit:
            num = self.current
            self.current += 1
            return num
```

```
else:
        raise StopIteration  # signals no more items
# Using the custom iterator
counter = CountUpTo(5)
for number in counter:
    print(number)
```

Output:

1

2

3

4

5

In this example, __iter__() makes the object iterable, while __next__() generates each value until the limit is reached. Once all values are generated, the StopIteration exception stops the loop.

This code defines a custom iterator class CountUpTo that counts from 1 up to a given limit. The __init__ method initializes the limit and sets the starting value current to 1. The __iter__ method returns the iterator object itself (self), allowing it to be used in a for loop. The __next__ method returns the next number in sequence each time it's called, incrementing current until it exceeds the limit, at which point it raises StopIteration to signal that there are no more values to iterate over. When we create an object counter = CountUpTo(5) and use it in a for loop, Python internally calls __iter__() once and then __next__() repeatedly, printing 1, 2, 3, 4, and 5 in order.

2.4.3 Generator Functions

Python's generator functions are used to create iterators (which can be traversed like a list or tuple) and return a traversal object. It helps to traverse all the items one at a time present in the iterator.

Generator functions are defined as the normal functions, but to identify the difference between the normal function and generator function is that in the normal function, we use the return keyword to return the values, and in the generator function, instead of using the return, we use yield to execute our iterator.

In Python, generator functions provide a way to produce a sequence of values lazily, meaning values are generated only when required, instead of computing and storing all at once. This is useful for handling large datasets or infinite sequences without consuming too much memory.

There are **two main ways to create a generator** in Python:

1. Using the **yield** statement.
2. Using a **Generator Expression**.

2.4.3.1 Generator Function with yield

The **yield** keyword allows a function to return values one at a time, while preserving the function's state between successive calls. Unlike the return statement, which terminates a function completely, yield pauses the function and resumes from where it left off when called again.

```
def my_generator():
    yield 1
    yield 2
    yield 3
Using the Generator
gen = my_generator()
for value in gen:
    print(value)
Output:
1
2
3
```

yield pauses the function and remembers where it left off. The next time the function is called, it resumes from that point.

Difference Between yield and return

Table 2.4.1 Difference Between yield and return

return	yield
Ends the function	Pauses the function
Returns one value only	Can return many values (one by one)
Function cannot be resumed	Function resumes from where it paused

Example: Generator for Even Numbers up to 10

```
def even_numbers():
    for i in range(1, 11):
        if i % 2 == 0:
            yield i
for num in even_numbers():
    print(num)
Output:
  2
  4
  6
  8
  10
```

2.4.3.2 Generator Expressions

In Python, a **Generator Expression** is a simple and memory-efficient way to create iterators. It looks very similar to a list comprehension, but instead of generating the entire list in memory at once, it produces values **lazily** (one by one, on demand). This makes generator expressions extremely useful when working with large datasets, streams, or files, where creating a complete list would be wasteful or even impossible due to memory constraints.

A generator expression follows the same form as a list comprehension but uses **parentheses ()** instead of square brackets [].

Syntax : expression for item in iterable if condition

- **expression** – the value that will be yielded each time.
- **item in iterable** – the loop that goes through the iterable.
- **if condition (optional)** – a filter to include only specific items.

Example:

```
squares = (x*x for x in range(1, 6))
for s in squares:
    print(s)
Output:
  1
  4
  9
  16
  25
```

Here are some key points about generator expressions:

1. Memory Efficiency

One of the most important advantages of generator expressions is their memory efficiency. Unlike lists, which store all elements at once, generator expressions generate values on the fly, producing each result only when it is requested. This lazy evaluation process makes them ideal for working with large datasets or potentially infinite sequences, where storing all values in memory would be impractical. In this sense, generator expressions operate similarly to iterators and generator functions created with the yield keyword, since they both rely on the principle of generating values "just in time."

2. Concise Syntax

Generator expressions also provide a compact and expressive syntax. They resemble list comprehensions in structure but return a generator object rather than a list. This difference means that generator expressions do not immediately evaluate all elements, but instead prepare an iterable that yields results when iterated over. Furthermore, in generator expressions, the first or outermost for clause is evaluated immediately, while the subsequent expressions are only executed during iteration. This lazy execution helps Python conserve resources and improve performance when handling large computations.

3. Usage with Functions

Another significant strength of generator expressions is their seamless integration with Python's built-in functions. They can be passed directly as arguments to any function that accepts an iterable, making them particularly convenient in situations where values need to be processed on the fly. Generator expressions are especially useful with reduction functions such as sum(), min(), max(), heapq.nlargest(), and heapq.nsmallest(). Using a generator expression directly within these functions eliminates the need for intermediate data structures, resulting in both cleaner code and improved efficiency.

Advantages of Generator Expressions

Generator expressions are particularly powerful because of their memory efficiency and lazy evaluation.

- Memory Efficiency: They do not store all values at once.
- Lazy Evaluation: Values are generated only when required.
- Reusability in pipelines: Can be chained with functions like sum(), max(), any(), all().
- Useful with large data: Works well with big files or streams.

Summarized Overview

In Python, iterators and generators provide a way to handle data efficiently by producing one value at a time instead of storing entire datasets in memory. An iterator is an object that follows the iterator protocol, using the __iter__() method to return itself and the __next__() method to return the next value until a StopIteration exception signals the end. Iterables such as lists and strings can be converted into iterators using the built-in iter() function, and elements can be accessed with next().

Python also allows the creation of custom iterators by defining classes with __iter__() and __next__() methods. This makes it possible to design objects that generate values step by step, such as counters or sequences. To simplify this process, Python provides generator functions, which use the yield keyword instead of return. A generator function pauses its execution at yield and resumes from the same point when called again, making it easier to write memory-efficient code.

Finally, generator expressions offer a more concise way of creating generators. They look similar to list comprehensions but use parentheses instead of square brackets. Generator expressions generate values lazily, making them especially useful with functions like sum(), min(), and max() for efficient, on-the-fly data processing.

Together, iterators and generators form the foundation for Python's efficient data handling, enabling programmers to work smoothly with large datasets, data streams, or even infinite sequences without exhausting system memory.

Assignments

1. Explain the difference between an **iterable** and an **iterator** with suitable Python examples.
2. Write a Python program to demonstrate how the built-in functions iter() and next() work on a list.
3. Create a custom iterator class that generates the first **n odd numbers** using __iter__() and __next__().
4. Define a generator function using yield that produces the Fibonacci sequence up to a given limit.

5. Write a generator expression to generate the squares of numbers from 1 to 10. Compare its memory usage with a list comprehension.
6. Differentiate between the yield statement and the return statement with examples.
7. What happens when next() is called on an iterator after all its elements are exhausted? Illustrate with an example.
8. Discuss the advantages and disadvantages of using iterators in Python.
9. Explain how Python's **for loop** internally uses an iterator to traverse a sequence.
10. Write a Python program that reads lines from a text file using a generator expression, printing only the lines that start with the letter "A".

Reference

1. Python Software Foundation. (n.d.). *Python documentation*. Python.org. Retrieved August 26, 2025, from https://docs.python.org/3/
2. Van Rossum, G., & Drake, F. L. (2009). *The Python language reference manual*. Network Theory Ltd.

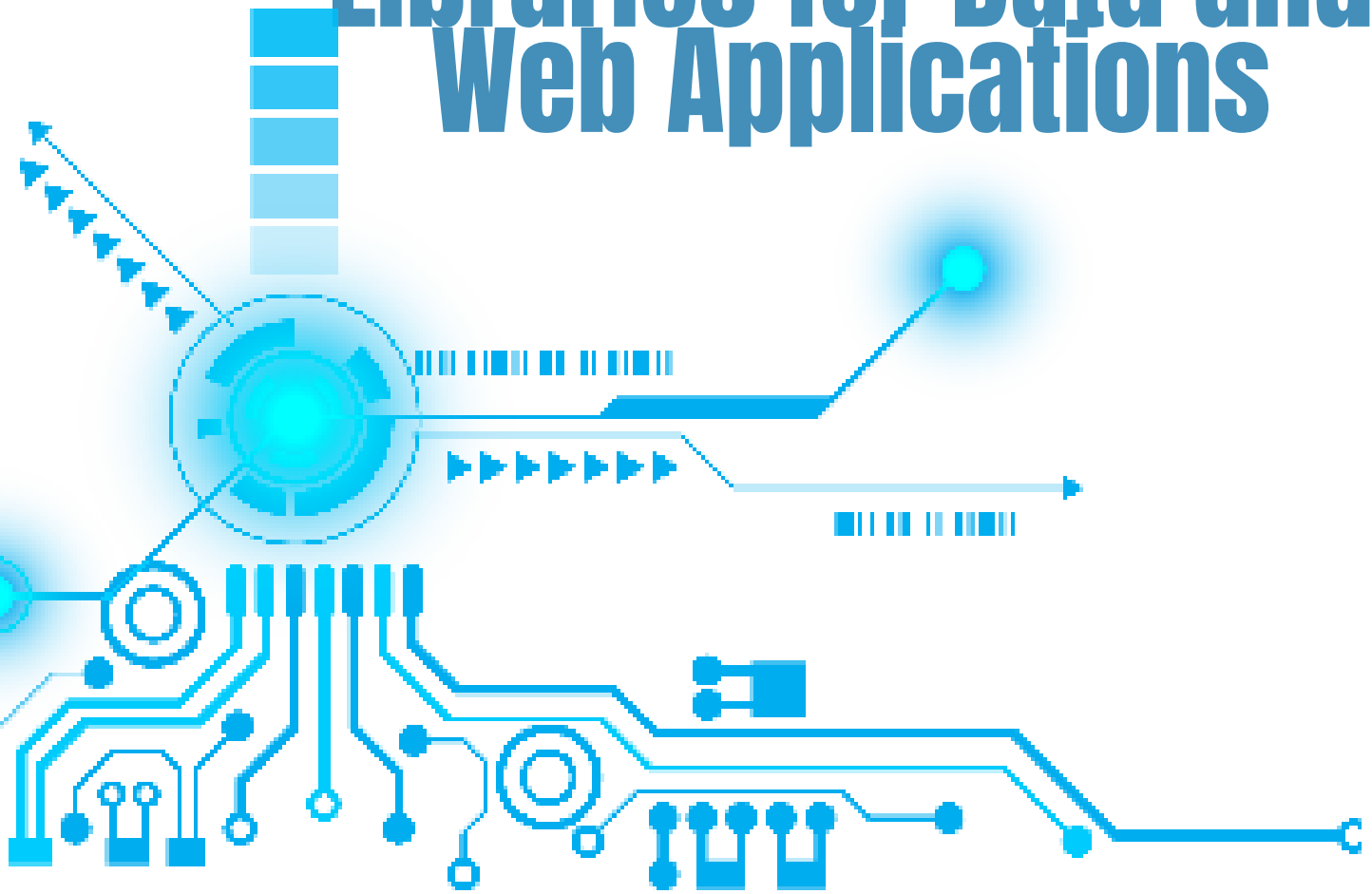
Suggested Reading

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
2. Beazley, D., & Jones, B. K. (2013). *Python cookbook* (3rd ed.). O'Reilly Media.
3. Downey, A. (2015). *Think Python: How to think like a computer scientist* (2nd ed.). O'Reilly Media.
4. Ramalho, L. (2015). *Fluent Python: Clear, concise, and effective programming*. O'Reilly Media.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

BLOCK 3
Libraries for Data and Web Applications



1 UNIT Numpy

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the concept of NumPy arrays and outline their role in efficient numerical computation compared to Python lists
- demonstrate how to perform array operations along with indexing and slicing techniques for accessing and modifying elements
- apply various array manipulation methods such as reshape, transpose, concatenate, split, insert, and delete to restructure data
- use NumPy's wide range of mathematical functions to carry out arithmetic, statistical, trigonometric, and exponential calculations
- examine the handling of arrays with unequal dimensions and determine the conditions under which operations are possible through automatic adjustment of shapes

Background

In the field of scientific computing and data-driven applications, efficiency and speed in handling large volumes of numerical data are of utmost importance. While Python provides flexible data structures such as lists and tuples, they are not optimized for heavy numerical computations, especially when dealing with multi-dimensional datasets or performing mathematical operations repeatedly. To overcome these limitations, the NumPy (Numerical Python) library was introduced as a high-performance, open-source tool designed specifically for numerical processing. At the heart of NumPy lies the ndarray (N-dimensional array), a homogeneous and memory-efficient data structure that enables fast element-wise

operations, matrix manipulations, and advanced indexing techniques. Beyond just array storage, NumPy integrates powerful features such as broadcasting, slicing, reshaping, and a vast collection of mathematical and statistical functions, which collectively eliminate the need for explicit loops and make code concise and efficient. It also serves as the computational foundation for higher-level libraries like Pandas, SciPy, and scikit-learn, thereby becoming an essential building block in modern data science, artificial intelligence, machine learning, and scientific research. This chapter introduces learners to the concepts of NumPy arrays, their operations, and associated tools, forming the basis for advanced numerical and analytical tasks in Python.

Keywords

N-dimensional array, Array Creation, Indexing, Slicing, Reshape, concatenation, Splitting, Mathematical functions, Broadcasting.

Discussion

3.1.1 NumPy

NumPy (Numerical Python) is one of the most important and fundamental packages for scientific and numerical computing in Python. It is a powerful open-source library that provides support for working with multidimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays efficiently. Unlike Python's built-in lists, which are flexible but slower when handling large volumes of numerical data, NumPy arrays are highly optimized for performance and allow fast computation. This makes NumPy especially useful for applications that require mathematical modeling, statistical analysis, data science, and machine learning.

NumPy introduces the concept of ndarray (N-dimensional array), which is its core data structure. The ndarray allows storage and manipulation of large datasets in a structured way, enabling vectorized operations without the need for explicit loops. It also supports advanced features such as broadcasting, masking, indexing, and slicing, making data handling both simpler and more efficient. In addition, NumPy provides tools for linear algebra, Fourier transforms, random number generation, and basic statistical operations, all of which are essential in numerical computation and data analysis.

In practice, NumPy is widely used in areas such as data science, machine learning, artificial intelligence, image processing, and scientific research, where speed and efficiency are crucial. It serves as the backbone for many other Python libraries such as

Pandas, SciPy, Matplotlib, and scikit-learn, making it a foundational tool in the Python ecosystem for scientific computing.

To begin using NumPy, it must first be installed. This can be done using the Python package manager pip with the following command:

Syntax:

pip install numpy

#To import NumPy:

import numpy as np

3.1.2 Creating NumPy Arrays

In NumPy, the fundamental building block is the ndarray (N-dimensional array).

NumPy provides multiple methods to create arrays depending on the requirement:

1. From existing Python sequences (lists, tuples)- The simplest way is to convert a list or tuple into a NumPy array using np.array().

```
import numpy as np

# From list

arr1 = np.array([1, 2, 3, 4, 5])

print(arr1)

# From tuple

arr2 = np.array((10, 20, 30))

print(arr2)

# Multi-dimensional

arr3 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr3)

#3-D arrary

arr3 = np.array([[[1, 2, 3],
                  [4, 5, 6]],
                 [[7, 8, 9],
                  [10, 11, 12]]])

print("3D Array:\n", arr3)

print("Dimensions:", arr3.ndim)
```

```
print("Shape:", arr3.shape)
```

```
[1 2 3 4 5]
```

```
[10 20 30]
```

```
[[1 2 3]
 [4 5 6]]
```

```
3D Array:
 [[[ 1  2  3]
  [ 4  5  6]]
 [[ 7  8  9]
  [10 11 12]]]
```

```
Dimensions: 3
```

```
Shape: (2, 2, 3)
```

2. Using array creation functions (zeros, ones, empty, full)- NumPy has functions to create arrays filled with zeros, ones, or a constant value.

```
#Creates an array filled with 0s.
np.zeros((3, 4))   # 3x4 array of zeros
#Ones Array
np.ones((2, 5))   # 2x5 array of ones
#Empty Array
np.empty((3, 3))
#Full Array
np.full((2, 3), 7)
```

3. Using numerical ranges (arange, linspace, logspace).

```
arange()
np.arange(1, 10, 2)   # [1 3 5 7 9]
linspace()
np.linspace(0, 1, 5)
logspace()
np.logspace(1, 3, 5)   # from 10^1 to 10^3
```

4. Identity and diagonal arrays (eye, identity).

#Identity Matrix (eye and identity)

np.eye(4) # 4x4 identity matrix

np.identity(3) # same as eye(3)

Diagonal Array

np.diag([1, 2, 3, 4])

5. Random number arrays (random module).- The numpy.random module helps in creating arrays with random values.

np.random.rand(2, 3) # random floats [0,1)

np.random.randn(2, 3) # random samples from normal distribution

np.random.randint(1, 10, (3, 3)) # random integers between 1 and 9

3.1.3 NumPy Array Operations

NumPy arrays are powerful data structures because they support a wide range of operations. The array operations are broadly divided into the following categories:

1. Arithmetic Operations

Arithmetic operations are performed element by element. This is called vectorization in NumPy.

Syntax:

result = array1 <operator> array2

where <operator> can be +, -, *, /, **, %.

Example:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40])
arr2 = np.array([1, 2, 3, 4])
print("Addition:", arr1 + arr2)
print("Subtraction:", arr1 - arr2)
print("Multiplication:", arr1 * arr2)
print("Division:", arr1 / arr2)
print("Power:", arr1 ** arr2)
```

Output

Addition: [11 22 33 44]

Subtraction: [9 18 27 36]

Multiplication: [10 40 90 160]

Division: [10. 10. 10. 10.]

Power: [10   400   27000 2560000]

2. Relational (Comparison) Operations

Comparison operators work element-wise and return a Boolean array of True/False.

Syntax:

result = array1 <comparison_operator> array2

Operators: >, <, >=, <=, ==, !=

Example:

```
arr = np.array([5, 10, 15, 20])
print(arr > 10)    # Greater than
print(arr == 15)   # Equal to
print(arr != 20)   # Not equal
```

3. Logical Operations

Logical operations combine Boolean arrays or conditions.

Syntax:

np.logical_and(x, y) → True if both x and y are True.

np.logical_or(x, y) → True if at least one is True.

np.logical_not(x) → Negates the condition.

Example:

```
a = np.array([True, False, True])
b = np.array([False, False, True])
print("AND:", np.logical_and(a, b))
print("OR:", np.logical_or(a, b))
print("NOT (a):", np.logical_not(a))
```

4. Aggregate Functions (Statistical Operations)

Aggregate functions compute values across the entire array (like sum, mean, median, etc.).

Syntax:

np.sum(arr) → Sum of elements

np.mean(arr) → Average

np.median(arr) → Median

np.std(arr) → Standard Deviation

np.max(arr) / np.min(arr) → Maximum/Minimum

np.argmax(arr) / np.argmin(arr) → Index of maximum/minimum

Example:

```
arr = np.array([1, 2, 3, 4, 5])
print("Sum:", np.sum(arr))
print("Mean:", np.mean(arr))
print("Median:", np.median(arr))
print("Standard Deviation:", np.std(arr))
print("Maximum:", np.max(arr))
print("Index of Max:", np.argmax(arr))
```

5. Matrix Operations (Linear Algebra)

When working with 2D arrays (matrices), NumPy provides efficient matrix operations.

Matrix Multiplication

- **np.dot(A, B) or A @ B**

Transpose of a Matrix

```
print("Transpose:\n", A.T)
```

Determinant & Inverse

```
Print("Determinant:", np.linalg.det(A))
print("Inverse:\n", np.linalg.inv(A))
```

Eigenvalues and Eigenvectors

```
values, vectors = np.linalg.eig(A)
print("Eigenvalues:", values)
print("Eigenvectors:\n", vectors)
```

6. Set operations

#**Union** -np.union1d()

```
import numpy as np
A = np.array([1, 3, 5])
B = np.array([0, 2, 3])
# union of two arrays
```

```
result = np.union1d(A, B)
print(result)
# Output: [0 1 2 3 5]
#Intersection-np.intersect1d()
import numpy as np
A = np.array([1, 3, 5])
B = np.array([0, 2, 3])
# intersection of two arrays
result = np.intersect1d(A, B)
print(result)
# Output: [3]
#Set difference-np.setdiff1d()
import numpy as np
A = np.array([1, 3, 5])
B = np.array([0, 2, 3])
# difference of two arrays
result = np.setdiff1d(A, B)
print(result)
# Output: [1 5]
#Symmetric difference-np.setxor1d()
import numpy as np
A = np.array([1, 3, 5])
B = np.array([0, 2, 3])
# symmetric difference of two arrays
result = np.setxor1d(A, B)
print(result)
# Output: [0 1 2 5]
```

7. Other array operations

Reshape (change shape of array):

```
arr = np.arange(12)
print(arr.reshape(3, 4))
```

Concatenation:

```
a = np.array([1, 2])
b = np.array([3, 4])
print(np.concatenate((a, b)))
```

3.1.4 Indexing and Slicing

In NumPy, indexing is the process of accessing specific elements of an array using their position (index), which starts from 0 for the first element, just like in Python lists. Indexing is extremely powerful in NumPy because it not only allows you to retrieve individual elements but also supports advanced operations like negative indexing (to access elements from the end), multi-dimensional indexing (to directly pick elements from 2D or higher-dimensional arrays using row and column indices), fancy indexing (where you can pass a list or array of indices to access multiple values at once), and boolean indexing (where conditions are used to filter out elements that satisfy certain criteria). For example, arr[2] gives the third element, arr[-1] gives the last element, arr[1,2] accesses the element from the 2nd row and 3rd column in a 2D array, arr[[0,2,4]] selects multiple elements at once, and arr[arr > 10] selects only elements greater than 10. This makes NumPy indexing not just a way of retrieving data, but also a tool for data selection, filtering, and manipulation, which is why it is widely used in scientific computing and data analysis.

Table 3.1.1: Array indexing in NumPy

One-dimensional array	import numpy as np arr = np.array([10, 20, 30, 40, 50]) print(arr[0]) # Output: 10 print(arr[3]) # Output: 40

Two-dimensional array	arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
	print(arr2d[0, 1]) # Output: 2 (element in 1st row, 2nd column)
	print(arr2d[2, 2]) # Output: 9 (element in 3rd row, 3rd column)

Negative Indexing	arr = np.array([10, 20, 30, 40, 50]) print(arr[-1]) # Output: 50 print(arr[-3]) # Output: 30
Boolean Indexing	arr = np.array([10, 20, 30, 40, 50]) print(arr[arr > 25]) # Output: [30 40 50]

Finding and Printing Index in NumPy Use np.where() function.

Example:

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Find index of element 30

result = np.where(arr == 30)

print(result)        # Output: (array([2], dtype=int64),)

# Printing only index

print(result[0][0])  # Output: 2
```

Example : 2D array

```
arr2d = np.array([[1, 2, 3],
                  [4, 5, 2],
                  [7, 8, 9]])

result = np.where(arr2d == 2)

print(result)  # Output: (array([0, 1], dtype=int64), array([1, 2], dtype=int64))

# Combine row and column indices

indices = list(zip(result[0], result[1]))

print(indices)  # Output: [(0, 1), (1, 2)]
```

In NumPy, slicing refers to extracting a continuous portion (subarray) of an array using index ranges, and it follows the general syntax array[start:end:step], where start is the index of the first element to include (default is 0), end is the index where slicing stops but is not included (default is the array length), and step specifies the interval between elements (default is 1). Slicing is more powerful in NumPy compared to normal Python lists because it works seamlessly with multi-dimensional arrays, allowing row-wise, column-wise, or block-based extraction. For example, in a 1D array, arr[1:4] extracts elements from index 1 to 3, arr[:3] selects the first three elements, arr[::2] selects every alternate element, and arr[::-1] reverses the array. In 2D arrays, slicing can be done across rows and columns simultaneously, such as arr2d[0:2, 1:3] which selects a submatrix of the first two rows and the 2nd and 3rd columns, or arr2d[:, 0] which selects the entire first column. Negative indices can also be used, such as arr[-3:] to get the last three elements. Unlike indexing, which retrieves individual elements, slicing creates views (subsections that reference the original array), meaning changes made to the slice affect the original array unless explicitly copied. This makes slicing not only a convenient way to access and analyze subsets of data, but also an efficient tool for manipulating large datasets without unnecessary memory duplication.

Syntax of Slicing:

array[start:end:step]

- start → starting index (inclusive, default = 0)
- end → ending index (exclusive, default = length of array)
- step → interval between indices (default = 1)

Example:

```
arr = np.array([10, 20, 30, 40, 50, 60])
print(arr[1:4])     # Output: [20 30 40]
print(arr[:3])      # Output: [10 20 30] (from beginning)
print(arr[3:])      # Output: [40 50 60] (till end)
print(arr[::2])     # Output: [10 30 50] (step = 2)
```

Example 2 Dimensional array

```
arr2d = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9,10,11,12]])
# Extracting first two rows and first three columns
print(arr2d[0:2, 0:3])
# Output:
# [[1 2 3]
#  [5 6 7]]
# Extracting all rows but only 2nd column
print(arr2d[:, 1])   # Output: [ 2  6 10]
# Extracting last row
print(arr2d[-1, :])  # Output: [ 9 10 11 12]
```

Example Negative slicing:

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[-4:-1])   # Output: [20 30 40]
print(arr[::-1])    # Output: [50 40 30 20 10] (reversed array)
```

3.1.5 Array Manipulation in NumPy

NumPy provides a wide range of functions and operations to manipulate arrays efficiently, enabling reshaping, joining, splitting, and modifying data without the need for explicit loops. Array manipulation is crucial in scientific computing and data analysis because it allows one to reorganize data, combine datasets, or change array structures according to the requirements of computation or algorithms. Important techniques for manipulating arrays are described below:

Table 3.1.2: Array manipulation functions in NumPy

Sl.No	Operations	Example

1	Reshaping Arrays-Use reshape() to create a new view with a different shape.	import numpy as np
		arr = np.arange(12) # array([0, 1, 2, ..., 11])
		reshaped = arr.reshape(3, 4) # 3 rows, 4 columns
		print(reshaped)

2	Transposing Arrays-arr.T returns the transposed array.	arr2d = np.array([[1, 2, 3], [4, 5, 6]]) print(arr2d.T)

3	Joining Arrays-np.concatenate() → joins along an existing axis.
	np.vstack() → vertical stacking (adds as new rows)
	np.hstack() → horizontal stacking (adds as new columns)
		a = np.array([1, 2, 3])
		b = np.array([4, 5, 6])
		print(np.concatenate([a, b])) # Output: [1 2 3 4 5 6]
		A = np.array([[1,2],[3,4]])
		B = np.array([[5,6],[7,8]])
		print(np.vstack([A,B]))

4	Splitting-np.split(array, indices_or_sections) np.hsplit(array, sections) → horizontal split np.vsplit(array, sections) → vertical split	arr = np.arange(8) print(np.split(arr, 4)) # Output: [array([0,1]), array([2,3]), array([4,5]), array([6,7])]

5	Adding or Removing Elements-np. append(array, values) → add elements to the end np.insert(array, index, values) → insert at specific index np.delete(array, index) → remove elements	arr = np.array([1,2,3]) arr = np.append(arr, [4,5]) arr = np.insert(arr, 1, 10) arr = np.delete(arr, 0) print(arr) # Output: [10 2 3 4 5]

3.1.6 Mathematical Functions

NumPy provides a wide range of mathematical functions that allow efficient computations on arrays without using explicit loops. These functions are vectorized, meaning they operate element-wise on entire arrays, which is much faster than performing operations in standard Python lists. NumPy's mathematical functions can be classified into arithmetic, statistical, trigonometric, exponential/logarithmic, and rounding functions.

Table 3.1.3: Mathematical functions in NumPy

Sl.No	Functions	Examples
Arithmetic Functions:	These functions perform element-wise arithmetic operations on arrays. np.add(x, y) → element-wise addition np.subtract(x, y) → element-wise subtraction np.multiply(x, y) → element-wise multiplication np.divide(x, y) → element-wise division np.power(x, y) → element-wise exponentiation np.mod(x, y) → element-wise modulo division	import numpy as np a = np.array([1, 2, 3]) b = np.array([4, 5, 6]) print(np.add(a, b)) # [5 7 9] print(np.subtract(b, a)) # [3 3 3] print(np.multiply(a, b)) # [4 10 18] print(np.divide(b, a)) # [4. 2.5 2.] print(np.power(a, 2)) # [1 4 9] print(np.mod(b, 3)) # [1 2 0]

Statistical Functions	NumPy provides functions to calculate mean, median, standard deviation, sum, and cumulative operations: np.sum(arr) → sum of all elements np.mean(arr) → mean of elements np.median(arr) → median value np.std(arr) → standard deviation np.var(arr) → variance np.cumsum(arr) → cumulative sum np.cumprod(arr) → cumulative product	arr = np.array([1, 2, 3, 4, 5]) print(np.sum(arr)) # 15 print(np.mean(arr)) # 3.0 print(np.median(arr)) # 3.0 print(np.std(arr)) # 1.4142135623730951 print(np.var(arr)) # 2.0 print(np.cumsum(arr)) # [1 3 6 10 15] print(np.cumprod(arr)) # [1 2 6 24 120]
Trigonometric Functions	NumPy provides trigonometric functions for angle-based calculations. Angles are in radians by default. np.sin(x) → sine np.cos(x) → cosine np.tan(x) → tangent np.arcsin(x) → inverse sine np.arccos(x) → inverse cosine np.arctan(x) → inverse tangent	angles = np.array([0, np.pi/2, np.pi]) print(np.sin(angles)) # [0. 1. 0.] print(np.cos(angles)) # [1. 0. -1.] print(np.tan(angles)) # [0. inf 0.]
Exponential and Logarithmic Functions	NumPy provides functions for exponentials, powers, and logarithms: np.exp(x) → e^x (exponential) np.exp2(x) → 2^x np.log(x) → natural logarithm (ln) np.log2(x) → base-2 logarithm np.log10(x) → base-10 logarithm	arr = np.array([1, 2, 4]) print(np.exp(arr)) # [2.71828183 7.3890561 54.59815003] print(np.exp2(arr)) # [2. 4. 16.] print(np.log(arr)) # [0. 0.69314718 1.38629436] print(np.log2(arr)) # [0. 1. 2.] print(np.log10(arr)) # [0. 0.30103 0.60206]

Rounding and Absolute Functions	np.abs()-Returns element-wise absolute value of x. np.floor()-Returns element-wise floored value. np.ceil()-The ceiling function rounds up to the nearest integer greater than or equal to each element. np.round()-The np.round() function rounds numbers to the nearest integer or to a specified number of decimal places.	import numpy as np arr = np.array([-5, -2, 0, 3, 7]) abs_arr = np.abs(arr) print(abs_arr) # Output: [5 2 0 3 7] …… arr = np.array([1.7, 2.9, -3.4, -1.2]) print(np.floor(arr)) # Output: [1. 2. -4. -2.] …….. arr = np.array([1.7, 2.1, -3.4, -1.8]) print(np.ceil(arr)) # Output: [2. 3. -3. -1.] …….. arr = np.array([1.7, 2.5, -3.2, -1.8]) print(np.round(arr)) # Output: [2. 2. -3. -2.] # Rounding to 1 decimal place arr2 = np.array([1.754, 2.349, -3.276]) print(np.round(arr2, 1)) # Output: [1.8 2.3 -3.3]

6. Advanced mathematical functions	np.sqrt()-Computes the square root of each element in an array. np.exp()-Computes e^x for each element. np.sign()-Returns the sign of each element: 1 if positive 0 if zero -1 if negative np.maximum(a, b)-Element-wise maximum np.minimum(a, b)-Element-wise minimum. np.power(x, y) → computes x^y element-wise np.reciprocal()-Computes 1/x element-wise.	import numpy as np arr = np.array([4, 9, 16, 25]) sqrt_arr = np.sqrt(arr) print(sqrt_arr) # [2. 3. 4. 5.] …….. arr = np.array([1, 2, 3]) print(np.exp(arr)) # [2.71828183 7.3890561 20.08553692] …… arr = np.array([-5, 0, 3]) print(np.sign(arr)) # [-1 0 1] …… a = np.array([1, 4, 3]) b = np.array([2, 2, 5]) print(np.maximum(a, b)) # [2 4 5] print(np.minimum(a, b)) # [1 2 3] …… arr = np.array([2, 3, 4]) print(np.power(arr, 3)) # [8 27 64] …… arr = np.array([2, 4, 5]) print(np.reciprocal(arr)) # [0.5 0.25 0.2]

3.1.7 Broadcasting in Python

In Python, particularly with the NumPy library, **broadcasting** refers to the automatic process of performing arithmetic operations on arrays of different shapes without explicitly reshaping or copying data. Normally, mathematical operations require

arrays to have the same dimensions, but broadcasting allows NumPy to intelligently "stretch" the smaller array so it matches the shape of the larger one, all while avoiding unnecessary memory usage. The broadcasting rules work by comparing array shapes from right to left: two dimensions are compatible if they are equal or if one of them is 1, in which case the dimension with size 1 is virtually replicated to match the other. This mechanism enables operations like adding a scalar to a vector, combining vectors with matrices, or scaling higher-dimensional data efficiently. For example, adding a 1D array [10, 20, 30] to a 2D array of shape (2, 3) automatically expands the smaller array across rows to match the larger one, producing a clean element-wise sum. Broadcasting is widely used in scientific computing and data analysis because it makes code concise, improves readability, saves memory, and speeds up execution, eliminating the need for explicit loops.

The broadcasting mechanism follows **three key rules**:

1. Compare shapes from right to left (trailing dimensions).

NumPy checks dimensions one by one starting from the last.

2. Two dimensions are compatible if:

They are equal, OR

One of them is 1.

3. If dimensions are not compatible, broadcasting fails.

When one of the dimensions is 1, NumPy virtually replicates the elements along that dimension to match the other array's size.

Table 3.1.4: Examples of Broadcasting

Broadcasting Compatibility in NumPy	Examples
Scalar with Array-A scalar can be broadcasted to any array.	import numpy as np a = np.array([1, 2, 3, 4]) b = 5 result = a + b print(result) OUTPUT [6 7 8 9]

1D Array with 2D Array	a = np.array([[1, 2, 3], [4, 5, 6]]) b = np.array([10, 20, 30]) result = a + b print(result) OUTPUT [[11 22 33] [14 25 36]]

Row Vector with Column Vector	a = np.array([[1],
	[2],
	[3]]) # Shape (3,1)
	b = np.array([10, 20, 30]) # Shape (3,)
	result = a * b
	print(result)
	OUTPUT
	[[10 20 30]
	[20 40 60]
	[30 60 90]]

Incompatible Shape	a = np.array([[1, 2], [3, 4]]) b = np.array([10, 20, 30]) result = a + b OUTPUT ValueError: operands could not be broadcast together with shapes (2,2) (3,) Here, (2,2) and (3,) cannot align → Broadcasting fails.

In short, broadcasting in Python (NumPy) is a mechanism that makes operations between arrays of different shapes possible by virtually expanding the smaller array to match the larger one, without copying data, leading to faster, more efficient computations.

Summarized Overview

NumPy, short for *Numerical Python*, is a fundamental Python library for scientific and numerical computing that provides the highly efficient **ndarray** (N-dimensional array) data structure, which supports fast, vectorized operations and advanced functionalities such as broadcasting, indexing, and slicing. Arrays in NumPy can be created from Python sequences, array creation functions (zeros, ones, full, empty), numerical ranges (arange, linspace, logspace), identity and diagonal matrices, and random number generators. Once created, these arrays support a wide range of operations including arithmetic, relational, logical, statistical, and matrix computations (like multiplication, transpose, determinant, inverse, eigenvalues), as well as set operations (union, intersection, difference). Indexing and slicing in NumPy go far beyond basic element access, offering negative, multi-dimensional, fancy, and boolean indexing for powerful data selection and filtering, while slicing provides efficient subarray views. Array manipulation functions such as reshaping, transposing, joining, splitting, and adding/removing elements enable flexible data structuring, and NumPy's extensive library of mathematical functions covers arithmetic, statistical, trigonometric, exponential, logarithmic, rounding, and advanced operations like square root, reciprocal, and maximum/minimum. A core feature, **broadcasting**, allows arrays of different shapes to be combined in operations by virtually expanding smaller arrays to match larger ones without copying data, making computations concise, memory-efficient, and fast. Together, these features make NumPy the backbone of Python's data science ecosystem, powering libraries like Pandas, SciPy, Matplotlib, and scikit-learn, and serving as an indispensable tool in fields ranging from machine learning to scientific research.

Assignments

1. Explain the concept of NumPy arrays (ndarray) in detail. How are they different from Python lists in terms of storage, efficiency, and operations? Provide suitable examples.
2. Discuss various array creation methods in NumPy with examples.

3. Describe the different types of array operations supported in NumPy. Explain with examples the arithmetic, relational, logical, aggregate, matrix, and set operations.
4. What is indexing in NumPy? Explain the different types of indexing with proper syntax and examples.
5. Define slicing in NumPy. Explain its syntax, features, and advantages. Illustrate slicing operations in both 1D and 2D arrays with examples.
6. Explain array manipulation techniques in NumPy. with examples.
7. Define and explain the concept of broadcasting in NumPy. What are the rules of broadcasting?

Reference

1. Dixit, R. (2025). *Hands-on NumPy for numerical analysis*. Apple Books.
2. Johansson, R. (2024). *Numerical Python: Scientific computing and data science applications with NumPy, SciPy, and Matplotlib*. Apress.
3. Kumar, R. (2023). *Mastering data analysis with Python: A comprehensive guide to NumPy, Pandas, and Matplotlib*.
4. McKinney, W. (2017). *Python for Data Analysis* (2nd ed.). O'Reilly Media.
5. VanderPlas, J. (2016). *Python Data Science Handbook*. O'Reilly Media.
6. Oliphant, T. E. (2015). *A Guide to NumPy*. USA: Travis Oliphant.

Suggested Reading

1. NumPy Developers. (2025). *NumPy Documentation*. Retrieved from https://numpy.org/doc/
2. https://www.w3schools.com/python/numpy/default.asp
3. McKinney, W. (2017). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython* (2nd ed.). O'Reilly Media.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT

Pandas

Learning Outcomes

After completion of this unit, the learner will be able to:

- understand the structure and functionality of Pandas Series and DataFrames for efficient data representation
- perform essential data cleaning, filtering, and grouping operations to prepare datasets for analysis
- apply merging and reshaping techniques to integrate and reorganize data for meaningful insights
- handle missing data effectively using Pandas functions to maintain data quality and reliability

Background

Pandas is a powerful open-source Python library designed for data analysis and manipulation. It provides two primary data structures—Series and DataFrames—which allow users to store, organize, and analyze data efficiently. In real-world datasets, issues such as missing values, duplicates, and inconsistent formats are common, and Pandas offers robust tools for data cleaning and transformation. Beyond cleaning, it supports operations like filtering, grouping, merging, and reshaping, enabling analysts to explore patterns and extract insights. With its ability to handle structured data seamlessly, Pandas has become a fundamental tool in data science, machine learning, and business analytics.

Keywords

Series, DataFrame, Data Cleaning, Grouping, Reshaping

Discussion

3.2.1 Pandas

In today's digital world, we are surrounded by data everywhere - from social media feeds, online transactions, mobile apps, research experiments and business records. But having data alone is not useful unless we can organize it, clean it, and analyze it to extract meaningful insights. This is where Pandas, one of the most powerful Python libraries, comes into play.

The name Pandas is derived from "panel data", a term used in economics and statistics for structured datasets. Just like a panda bear that is loved worldwide for its simplicity and charm, the Pandas library has become a favorite among programmers and data scientists because it makes working with data simple, flexible, and efficient.

Why use Pandas?

Pandas is an essential tool in Python because it makes handling data both simple and efficient. It is capable of processing large datasets without difficulty, allowing analysts and programmers to work with vast amounts of information smoothly. One of its most important strengths is in data cleaning and preparation. Since real-world data often contains missing values, duplicates, or inconsistencies, Pandas provides convenient methods to detect and correct such problems, making the data ready for analysis.

Beyond cleaning, Pandas supports a wide range of operations including filtering, grouping, reshaping, and merging, which enable users to organize and analyze data in flexible ways. These operations help in extracting useful insights, combining multiple data sources, and transforming data into different formats for further study. Another major advantage is Pandas' seamless integration with other Python libraries such as NumPy for numerical computations, Matplotlib for data visualization, and Scikit-learn for machine learning.

Together, these features make Pandas a powerful and reliable foundation for data analysis, ensuring that Python can be used effectively to turn raw information into meaningful insights.

3.2.2 Installing Pandas

Before working with Pandas, it must first be installed in the Python environment. Since Pandas is not included in the standard Python library, it is usually installed using the

Python Package Installer (pip). Pip is a command-line tool that allows users to download and install external Python packages from the Python Package Index (PyPI).

To install Pandas, open a command prompt or terminal window and type the following command:

```
pip install pandas
```

When this command is executed, pip automatically downloads the latest stable version of Pandas along with any required dependencies, such as NumPy, and installs them in the Python environment. Once the installation is complete, Pandas can be imported into any Python program using the statement:

```
import pandas as pd
```

Here, the library is given an alias pd for convenience, which is a widely accepted convention in the Python community. After installation, students can immediately begin working with Pandas to perform data analysis tasks.

3.2.3 Pandas Data Structures

Pandas is one of the most powerful and widely used Python libraries for data analysis and manipulation. At its core, Pandas provides two primary data structures that serve as the foundation for handling data: Series and DataFrame. These data structures are built on top of NumPy arrays, which means they inherit high performance and efficiency, but add more flexibility by introducing labels and advanced functionality.

3.2.3.1 Pandas Series

A Series is a one-dimensional array-like structure designed for handling and manipulating data. What sets it apart is its powerful and flexible index, which allows for efficient data access and labeling.

A Series consists of two main components as shown in Fig 3.2.1.

1. Data – the array containing the actual values.
2. Index – an array of labels corresponding to the data values.

Index	Data
0	Mark
1	Justin
2	John
3	Vicky

Fig 3.2.1 Components of Series

In essence, a Series is a labeled, one-dimensional array that can store any data type.

Creating a Pandas Series

A pandas Series can be created using the following constructor

class pandas.Series(data, index, dtype, name, copy)

Parameters of Constructor are

1. data-The input can be in different formats such as an ndarray, a list, or a single constant value.
2. index-The index must contain unique, hashable values and must match the length of the data. If not provided, it defaults to np.arange(n).
3. dtype-Refers to the data type. If not specified, pandas will automatically detect it.
4. copy-Indicates whether to create a copy of the input data. The default setting is False.

Create an Empty Series

An empty Series object can be created by calling the pandas.Series( ) constructor without passing any data.

```
import pandas as pd                    #import the pandas library and aliasing as pd

s = pd.Series( )

print('Resultant Empty Series:\n',s)          # Display the result
```

Output:

```
Resultant Empty Series:

Series([], dtype: float64)
```

Create a Series from Python Dictionary

You can create a Series by passing a dictionary to the pd.Series() constructor. If no index is provided, the keys of the dictionary will be sorted and used as the Series index. If an index is specified, only the values that match the given labels will be extracted from the dictionary.

```
import pandas as pd

import numpy as np

data = {'a' : 0., 'b' : 1., 'c' : 2.}

s = pd.Series(data)

print(s)
```

Output:

```
a    0.0
```

b 1.0

c 2.0

dtype: float64

Key Characteristics of a Pandas Series

A Pandas Series is more than just a one-dimensional array. It comes with special properties that make it highly useful for data manipulation. Some of its key characteristics are:

1. Data is Mutable

The values stored inside a Series can be modified after its creation. This means you can change the content of a Series without recreating it.

Example:

```
import pandas as pd

s = pd.Series([10, 20, 30])

s[1] = 25  # modifying the second element

print(s)
```

Output:

0 10

1 25

2 30

dtype: int64

Here, the second value 20 has been changed to 25.

2. Size is Immutable

While the data can be changed, the number of elements in a Series cannot be altered once it is created. You cannot directly add or remove items from a Series as you would with a Python list. If you need a different size, you must create a new Series.

Example:

```
s = pd.Series([1, 2, 3])

# Trying to add a new element directly is not allowed

# s[3] = 4  # This will raise an error
```

However, you can use methods like append( ) or concat( ) to create a new Series with additional elements, but the original Series remains unchanged in size.

3. Composed of Two Arrays (Data and Index)

A Series can be thought of as a combination of two parallel arrays:

- One array stores the actual data values.
- The other array stores the labels (index values).

This structure makes it easy to map each piece of data to a specific label, just like a dictionary in Python.

Example:

```
s = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
print("Data:", s.values)
print("Index:", s.index)
```

Output:

```
Data: [100 200 300]
Index: Index(['a', 'b', 'c'], dtype='object')
```

4. Index as Labels

The labels used to identify elements in a Series are called the Index. The index acts as a reference to access data values conveniently, either by position or by label.

Example:

```
s = pd.Series([10, 20, 30], index=['x', 'y', 'z'])
print(s['y'])   # Accessing by label
print(s[2])     # Accessing by position
```

Output:

```
20
30
```

Here, 'y' is an index label, and 2 is a positional index. Both can be used to retrieve values.

3.2.3.2 DataFrame

A DataFrame is a two-dimensional structure as shown in Fig 6.2.2 ideal for organizing data in rows and columns, much like a spreadsheet or SQL table. It is the most widely used object in pandas. After loading data into a DataFrame, a variety of operations can be performed to explore, analyze, and interpret the data effectively.

Properties of DataFrames

1. A DataFrame has two axes: the row axis (axis=0) and the column axis (axis=1).

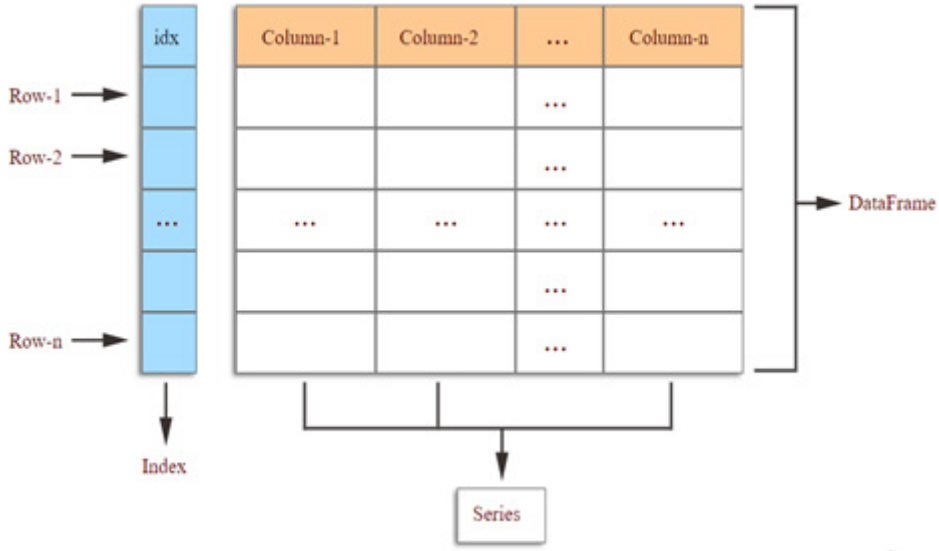


Fig 3.2.2 DataFrames

2. It resembles a spreadsheet, where the row identifiers are called indices and the column identifiers are called column names.
3. It can store heterogeneous data, meaning different data types in different columns.
4. The size of a DataFrame is mutable, meaning rows and columns can be added or removed.
5. The data within a DataFrame is also mutable, so values can be changed after creation.

Creating a Pandas DataFrame

1. Create an Empty DataFrame

An empty DataFrame by calling the DataFrame constructor without passing any arguments.

```
import pandas as pd

df = pd.DataFrame()

print(df)
```

2. Create a DataFrame from Lists

A DataFrame can be constructed from a single list or from multiple lists organized within a list (i.e., a list of lists).

```
import pandas as pd

data = [1,2,3,4,5]

df = pd.DataFrame(data)
```

```
print(df)
```

Output:

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

3. Creating DataFrame from dict of ndarray/lists

A DataFrame can be constructed using a dictionary, where each key represents a column name and the corresponding value is a list or array of data.

- All lists or arrays used must be of equal length.
- If you specify an index, its length must also match the length of the data.
- If no index is given, Pandas will automatically assign a default integer index starting from 0.

```
import pandas as pd
data = {'Name':['Tom', 'nick', 'krish', 'jack'],
        'Age':[20, 21, 19, 18]}
df = pd.DataFrame(data)
print(df)
```

Output:

```
    Name  Age
0    Tom   20
1   nick   21
2  krish   19
3   jack   18
```

DataFrame Operations

1. Selection

Selecting specific rows or columns from a DataFrame using labels (.loc[]) or index positions (.iloc[]).

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['NY', 'LA', 'Chicago']}
df = pd.DataFrame(data)
print(df['Name'])                    # Selecting a single column
print(df[['Name', 'City']])          # Selecting multiple columns
print(df.loc[1])  # Bob's row        # Selecting a row by index
```

2. Filtering

Extracting rows that meet specific conditions using boolean expressions.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'Salary': [50000, 60000, 70000]}
df = pd.DataFrame(data)
filtered_df = df[df['Age'] > 28]          # Filtering rows where Age > 28
print(filtered_df)
```

3. Sorting

Arranging rows in ascending or descending order based on column values.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)
print(df.sort_values(by='Age'))                    # Sorting by Age (ascending)
print(df.sort_values(by='Age', ascending=False))   # Sorting by Age (descending)
```

4. Merging

Combining two DataFrames using a common column or key.

```
import pandas as pd
df1 = pd.DataFrame({
```

```
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})
df2 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Salary': [50000, 60000, 70000]
})
merged_df = pd.merge(df1, df2, on='ID')        # Merging both DataFrames on ID
print(merged_df)
```

Mini Exercise:

1. Create a Series of your 5 favorite numbers.
2. Create a DataFrame with 3 columns: City, Population, Area.

3.2.4 Data Cleaning in Pandas

Data cleaning is a crucial step in preparing datasets for analysis, as raw data collected from real-world sources often contains missing values, duplicate records, or inconsistent formats. Pandas provides a wide range of functions to handle these issues efficiently, making it easier to transform messy data into a structured and reliable form. With tools for handling null values, correcting data types, removing duplicates, and standardizing formats, Pandas ensures that the dataset is accurate, consistent, and ready for meaningful analysis.

3.2.4.1 Renaming Columns

In many cases, the column names in a dataset may not be descriptive or may not follow a consistent naming style. Pandas allows us to rename one or more columns using the rename() method. The columns parameter takes a dictionary where the keys represent the existing column names and the values represent the new names. The inplace=True option ensures that the changes are applied directly to the DataFrame without creating a new copy.

Example:

```
df.rename(columns={'Marks': 'Score'}, inplace=True)
```

In this example, the column Marks is renamed to Score, making the dataset more meaningful.

3.2.4.2 Changing Data Types

Sometimes the data type of a column may not be suitable for analysis. For instance, numerical values may be stored as strings or integers when they are required as floating-point numbers. Pandas provides the astype() function to change the data type of a column.

Example:

df['Age'] = df['Age'].astype(float)

Here, the column Age is converted into the float data type, allowing more accurate calculations and statistical operations.

3.2.4.3 Removing Duplicates

Datasets often contain duplicate rows due to repeated entries or merging of multiple data sources. These duplicates can affect analysis results by introducing bias. Pandas provides the drop_duplicates() method to remove such rows.

Example:

df.drop_duplicates(inplace=True)

This command removes duplicate rows and keeps only the first occurrence, ensuring that the dataset contains unique records.

3.2.4.4 Removing Unwanted Columns

Some datasets may contain columns that are unnecessary for analysis, such as identifiers, metadata, or irrelevant information. Such columns can be removed using the drop() method with the columns parameter.

Example:

df.drop(columns=['Area'], inplace=True)

Here, the column Area is removed from the DataFrame, making the dataset more concise and focused on relevant features.

3.2.5 Filtering Data

Filtering is the operation of selecting a subset of rows (and sometimes columns) from a DataFrame that satisfy one or more conditions. In Pandas, filtering is fast and expressive because conditions are evaluated vectorially (all rows at once) rather than in loops. Effective filtering is essential to exploratory analysis, reporting, model building, and data quality checks.

3.2.5.1 Boolean Indexing: The Core Idea

Every condition evaluated on a column returns a Boolean Series—a sequence of True/False values aligned to the DataFrame's index. Passing this Boolean Series inside square brackets selects rows where the value is True.

Example

```
# Sample data

import pandas as pd

df = pd.DataFrame({
```

```
    'Student': ['Asha', 'Biju', 'Chitra', 'Deepak', 'Ekta', 'Farhan'],
    'Dept':    ['MCA', 'MCA', 'MBA', 'MCA', 'MBA', 'MCA'],
    'Age':     [21, 22, 24, 22, 23, 21],
    'Score':   [78, 85, 67, 92, 74, 85]
})
# Filter rows where Score > 80
df[df['Score'] > 80]
```

3.2.5.2 Combining Multiple Conditions

Use bitwise operators with parentheses:

- & for logical AND
- | for logical OR
- ~ for logical NOT

Important: Do not use Python's and/or with Series; they work only with single booleans.

Example

```
# MCA students with Score >= 80
df[(df['Dept'] == 'MCA') & (df['Score'] >= 80)]
# Non-MBA students OR Age < 22
df[(df['Dept'] != 'MBA') | (df['Age'] < 22)]
# NOT condition: students who are not 21 years old
df[~(df['Age'] == 21)]
```

3.2.5.3 Row vs. Column Selection with .loc

.loc allows you to filter rows and select columns in a single step.

Example

```
# Only Student and Score for MCA students
df.loc[df['Dept'] == 'MCA', ['Student', 'Score']]
```

Use .iloc when you want to filter by position rather than labels.

3.2.5.4 Membership and Range Tests: isin, between, query

- Membership (isin) keeps rows whose values appear in a list/Series.
- Range (between) is convenient for inclusive numeric ranges.

- Query (query) lets you write conditions as strings (useful for long expressions and large frames).

Examples

```
# Students from selected departments

df[df['Dept'].isin(['MCA', 'MSc'])]

# Scores between 70 and 90 inclusive

df[df['Score'].between(70, 90)]

# Using query (note: column names used directly)

df.query("Dept == 'MCA' and Score >= 80")
```

3.2.5.5 String-Based Filtering

Use .str accessor for vectorized string operations. Handle missing values with na=False where relevant.

Examples

```
# Names starting with 'A'

df[df['Student'].str.startswith('A')]

# Dept contains 'M' (case-sensitive)

df[df['Dept'].str.contains('M', na=False)]

# Case-insensitive match for 'mca'

df[df['Dept'].str.contains('^mca$', case=False, na=False, regex=True)]
```

3.2.5.6 Date/Time Filtering

Convert to datetime first (pd.to_datetime) and then use .dt for components or index-based slicing.

Examples

```
sales = pd.DataFrame({

   'Date': ['2025-08-01', '2025-08-05', '2025-08-05', '2025-08-10'],

   'City': ['Kochi', 'Kottayam', 'Kollam', 'Kochi'],

   'Amount': [1200, 800, 950, 2000]

})

sales['Date'] = pd.to_datetime(sales['Date'])

# Filter for a specific date
```

```
sales[sales['Date'] == '2025-08-05']

# Filter for August 2025

sales[(sales['Date'].dt.year == 2025) & (sales['Date'].dt.month == 8)]

# If Date is the index, label-based slice is very fast:

# sales = sales.set_index('Date')

# sales['2025-08-01':'2025-08-05']
```

3.2.5.7 Handling Missing Values in Filters

Missing values (NaN) can make comparisons return False or NaN. Use isna()/notna() to filter explicitly, or fill/ignore them appropriately.

Examples

```
df[df['Score'].notna()]          # keep rows with Score present

df[df['Score'].fillna(0) > 80]   # treat missing as 0 for comparison
```

For string searches, prefer na=False:

```
df[df['Student'].str.contains('a', case=False, na=False)]
```

3.2.5.8 Filtering vs. Conditional Replacement: where and mask

- where keeps values that meet a condition; others become NaN.
- mask does the opposite (replaces values where condition is True).

Examples

```
# Keep only high scores; others become NaN

df['HighScore'] = df['Score'].where(df['Score'] >= 80)

# Mask sensitive values

df['ScoreMasked'] = df['Score'].mask(df['Score'] < 80)
```

These are useful when you want to **preserve shape** (same number of rows) but mark unwanted values rather than drop rows.

3.2.5.9 Group-Aware Filtering

Sometimes a row should be kept or removed based on a condition **within its group** (e.g., department averages). Use groupby().transform to create an aligned Series for comparison, or groupby().filter to keep/discard entire groups.

Examples

```
# Keep rows with Score above the average Score of their own Dept
```

```
dept_mean = df.groupby('Dept')['Score'].transform('mean')

df_above_avg = df[df['Score'] > dept_mean]

# Keep only departments with at least 3 students

df_min3 = df.groupby('Dept').filter(lambda g: len(g) >= 3)
```

3.2.5.10 Column Filtering (Feature Selection)

Filtering also applies to columns:

```
# Keep only numeric columns

df_numeric = df.select_dtypes(include='number')

# Keep columns by name pattern

df_score_cols = df.filter(like='Score')     # name contains 'Score'

df_prefix = df.filter(regex=r'^(Stu|Dept)') # names start with 'Stu' or 'Dept'
```

3.2.6 Grouping Data

Grouping helps to **aggregate data** based on a column.

Example – Grouping and Mean

```
data = {
   'Department': ['CS', 'CS', 'Math', 'Math'],
   'Marks': [85, 90, 78, 88]
}

df = pd.DataFrame(data)

print(df.groupby('Department')['Marks'].mean())
```

Output:

```
Department

CS      87.5

Math    83.0

Name: Marks, dtype: float64
```

3.2.7 Merging DataFrames

In real-world applications, information is often stored across multiple tables or files. For example, one table may contain student details, while another table contains their exam scores. To perform meaningful analysis, these tables must be combined into a single, unified dataset. Pandas provides the merge() function, which works in a manner

very similar to SQL joins, allowing multiple DataFrames to be combined on the basis of common columns or indexes.

The merge() function is highly flexible. By specifying the key column(s) with the parameter on, Pandas aligns rows from both DataFrames where the values match. This avoids manual concatenation and ensures that related information is correctly linked together.

Example: Simple Merge

```
import pandas as pd

# First DataFrame: Student IDs with Names

df1 = pd.DataFrame({'ID': [1, 2], 'Name': ['A', 'B']})

# Second DataFrame: Student IDs with Scores

df2 = pd.DataFrame({'ID': [1, 2], 'Score': [85, 90]})

# Merge on the common column 'ID'

merged = pd.merge(df1, df2, on='ID')

print(merged)
```

Output

```
ID Name  Score

0  1   A    85

1  2   B    90
```

Explanation

- The two DataFrames (df1 and df2) both have a column named ID, which acts as the common key.
- The merge() function combines the rows where the **ID** values match.
- The resulting DataFrame merged contains three columns: **ID**, **Name**, and **Score**.

This process mirrors the concept of a **JOIN operation in SQL**, where related records from different tables are combined into one table.

Types of Joins in Pandas

Like SQL, Pandas supports different kinds of joins through the how parameter:

1. **Inner Join (default)** – keeps only rows with matching keys in both DataFrames.
2. **Left Join** – keeps all rows from the left DataFrame, adding matches from the right.

3. **Right Join** – keeps all rows from the right DataFrame, adding matches from the left.
4. **Outer Join** – keeps all rows from both DataFrames, filling missing values with NaN.

Example:

```
# Left join

merged_left = pd.merge(df1, df2, on='ID', how='left')
```

3.2.8 Reshaping Data

In data analysis, the same information may need to be represented in different structures depending on the task. For instance, while raw data is often stored in a **long format** (each row representing a single observation), certain analyses or visualizations are more effective in a **wide format** (values spread across multiple columns). Pandas provides powerful reshaping operations such as **pivoting** and **melting**, which allow us to convert data between these formats easily.

Reshaping makes data more flexible and helps analysts prepare it for reporting, visualization, or advanced computation.

3.2.8.1 Pivoting Data

The **pivot** operation reshapes data from a long format into a wide format by spreading values across columns. It requires three arguments:

- index: the column to use as row labels,
- columns: the column whose unique values will form new column headers,
- values: the column providing the values to fill the table.

Example: Pivot Table

```
import pandas as pd
data = {
   'Name': ['A', 'A', 'B', 'B'],
   'Subject': ['Math', 'CS', 'Math', 'CS'],
   'Marks': [90, 85, 88, 92]
}
df = pd.DataFrame(data)
pivot = df.pivot(index='Name', columns='Subject', values='Marks')
print(pivot)
```

Output

```
Subject  CS  Math
```

Name

A 85 90

B 92 88

Explanation:

- The column **Name** becomes the index.
- The unique values of **Subject** become new column headers.
- The **Marks** values are filled in the corresponding cells.
- As a result, each student now has a single row with separate columns for **Math** and **CS** marks.

This wide-format structure is useful for quick comparisons across categories.

3.2.8.2 Melting Data

The **melt** operation is the reverse of pivoting. It converts data from a wide format back into a long format, where all variable names are stored in a single column, and their values are stored in another column. This is particularly useful for plotting and statistical analysis where long format is preferred.

Example: Melt

```
melted = pd.melt(pivot, ignore_index=False)

print(melted)
```

Output

```
  variable  value
Name
A       CS     85
A     Math     90
B       CS     92
B     Math     88
```

Explanation:

- The wide-format pivot table is transformed back into a long format.
- The column headers (**CS**, **Math**) become row values under the column **variable**.
- Their corresponding values appear under the column **value**.
- The original index (**Name**) is retained because ignore_index=False is specified.

Summarized Overview

The unit covers essential functionalities of the Pandas library for data handling. It begins with an introduction to Series and DataFrame, the core data structures in Pandas. It then addresses data cleaning techniques, including handling missing values and removing inconsistencies. The next sections focus on filtering data based on conditions, performing grouping operations for aggregation and summarization, and using merging to combine multiple DataFrames. Finally, the syllabus includes reshaping operations such as pivoting and melting, along with systematic methods for dealing with missing data. Together, these topics provide a comprehensive foundation for effective data manipulation and preparation in Python.

Assignments

1. Explain the difference between Series and DataFrame in Pandas. Give suitable examples to demonstrate their creation and basic operations.
2. Discuss the importance of data cleaning in Pandas. Write Python code to rename a column, change a column's data type, remove duplicates, and drop unwanted columns in a DataFrame.
3. What is filtering in Pandas? Explain different ways of filtering data using conditions, logical operators, and string-based filtering with suitable examples.
4. Describe the role of grouping and aggregation in data analysis. Write a program to group student data by department and calculate the average score for each department.
5. Explain the concepts of merging and reshaping DataFrames. Demonstrate with Python code how to merge two DataFrames using a common key and then reshape the merged data using pivot and melt functions.

Reference

1. McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media.
2. VanderPlas, J. (2016). *Python Data Science Handbook*. O'Reilly Media.
3. Wes McKinney et al. (2023). *Pandas Documentation*. https://pandas.pydata.org/docs
4. Reitz, K., & Schlusser, T. (2021). *The Hitchhiker's Guide to Python*. O'Reilly Media.
5. Raschka, S., & Mirjalili, V. (2019). *Python Machine Learning*. Packt Publishing.

Suggested Reading

1. Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist*. O'Reilly Media.
2. Grus, J. (2019). *Data Science from Scratch: First Principles with Python*. O'Reilly Media.
3. Albon, C. (2018). *Machine Learning with Python Cookbook*. O'Reilly Media.
4. Seabold, S., & Perktold, J. (2010). *Statsmodels: Econometric and Statistical Modeling with Python*. Proceedings of the 9th Python in Science Conference.
5. Kaggle Learn Micro-Courses: *Pandas* and *Data Cleaning*. https://www.kaggle.com/learn

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3 UNIT Data Visualization

Learning Outcomes

After completion of this unit, the learner will be able to:

- identify the basic functions of Matplotlib and Seaborn used for creating line plots, bar charts, histograms, and scatter plots
- explain the differences between various types of plots and their applications in data visualization
- construct different types of visualizations such as line plots, bar charts, histograms, and scatter plots using Matplotlib and Seaborn
- demonstrate how to customize plots (titles, labels, colors, legends, and styles) to make data more interpretable and visually appealing

Background

Studying Data Visualization with Matplotlib and Seaborn is essential. Data, when represented visually, becomes easier to analyze, interpret, and communicate. Line plots, bar charts, histograms, and scatter plots help in identifying trends, patterns, and outliers that might not be obvious in raw data. These visualizations provide meaningful insights that support decision-making in research, business, and technology. Learning to customize graphs enhances clarity and makes them more suitable for presentations and reports. The benefit of mastering these tools lies in developing the ability to transform complex datasets into clear, impactful visuals that improve understanding, storytelling, and data-driven problem solving.

Keywords

Line Chart, Bar Chart, Histogram, Scatter Plot, Pie Chart, Box Plot, Customization, Color Palette, Pyplot

Discussion

3.3.1 Data Visualization with Matplotlib in Python

Matplotlib is a popular Python library for creating data visualizations, whether static, animated, or interactive. Built on top of NumPy, it efficiently manages large datasets and provides tools to generate different types of plots, including line graphs, bar charts, scatter plots, and more.

3.3.1.1 Visualizing Data with Pyplot in Matplotlib

The **Pyplot** module in Matplotlib offers an easy-to-use interface for plotting data. With just a few lines of code, it enables the creation of different types of graphs such as line plots, bar charts, and histograms. It simplifies the process of data visualization, making it easier to represent information clearly. Let's look at a few simple examples to see how it works in practice.

1. Line Chart

A line chart is one of the simplest types of plots, created using the plot() function. It is mainly used to show the relationship or trend between two sets of values, usually X and Y, plotted on different axes.

Syntax:

matplotlib.pyplot.plot(x, y)

Parameters:

- x, y → Coordinates of the data points.

See Fig. 3.3.1 Line Chart.

Example: The following code draws a basic line chart with labeled axes and a title:

```
import matplotlib.pyplot as plt

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

plt.plot(x, y)
```

```
plt.title("Line Chart")
plt.xlabel("X-Axis")
plt.ylabel("Y-Axis")
plt.show()
```

Output

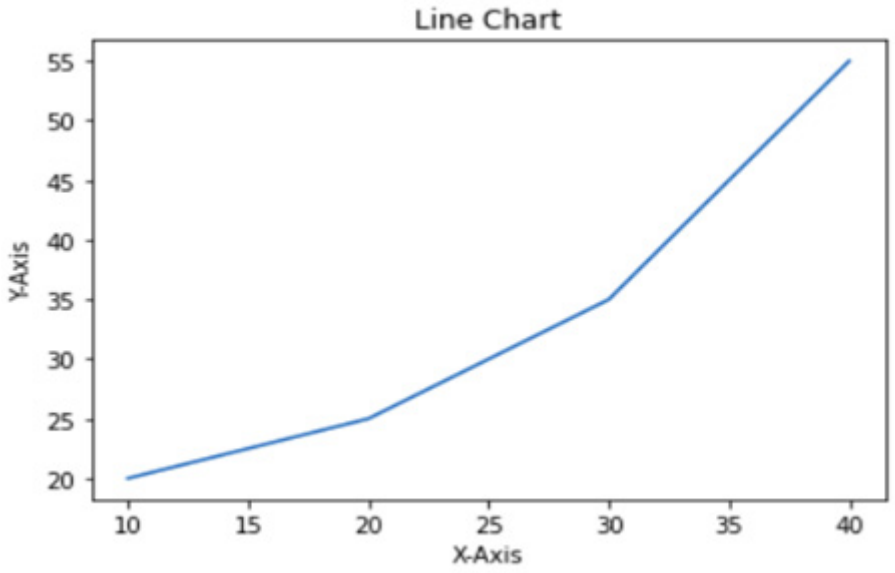


Fig. 3.3.1 Line Chart

2. Bar Chart

A bar chart is used to represent categorical data with rectangular bars, where the length of each bar corresponds to the value it represents. Bars can be displayed vertically or horizontally, making it easy to compare different categories.

Syntax:

matplotlib.pyplot.bar(x, height)

Parameters:

- x → Categories or positions along the X-axis.
- height → Values or heights of the bars along the Y-axis.

Example: The following code generates a bar chart showing total bills for different days. The X-axis shows the days, and the Y-axis represents the total bill amounts.

```
import matplotlib.pyplot as plt
x = ['Thur', 'Fri', 'Sat', 'Sun']
y = [170, 120, 250, 190]
```

```
plt.bar(x, y)
plt.title("Bar Chart")
plt.xlabel("Day")
plt.ylabel("Total Bill")
plt.show()
```

See Fig. 3.3.2 Bar Chart

Output

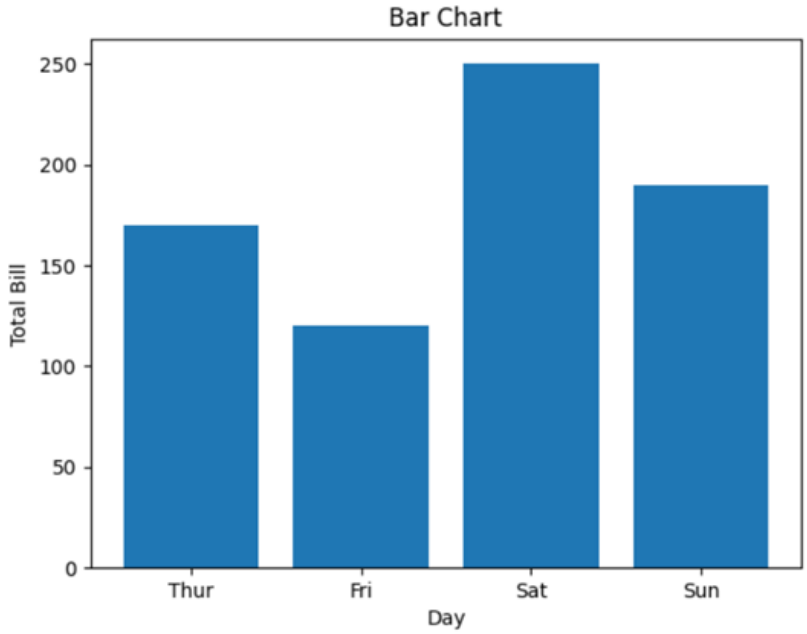


Fig 3.3.2 Bar Chart

3. Histogram

A histogram is used to visualize the distribution of a dataset by dividing the values into intervals, called bins, and showing how many data points fall into each bin. The hist() function in Matplotlib is used to create histograms. The X-axis represents the bins, while the Y-axis shows the frequency of data within each bin.

Syntax:

matplotlib.pyplot.hist(x, bins=None)

Parameters:

- x → The dataset or values to be plotted.
- bins → Number of intervals to divide the data into.

Example: The following example creates a histogram to display the frequency

distribution of total bill amounts. It divides the data into 10 bins and includes axis labels and a title for clarity.

```
import matplotlib.pyplot as plt
x = [7, 8, 9, 10, 10, 12, 12, 12, 13, 14, 14, 15, 16, 16, 17, 18, 18, 19, 20, 20,
    21, 22, 23, 24, 25, 25, 26, 28, 30, 32, 35, 36, 38, 40, 42, 44, 48, 50]
plt.hist(x, bins=10, color='steelblue')
plt.title("Histogram")
plt.xlabel("Total Bill")
plt.ylabel("Frequency")
plt.show()
```

See Fig. 3.3.3 Histogram

Output

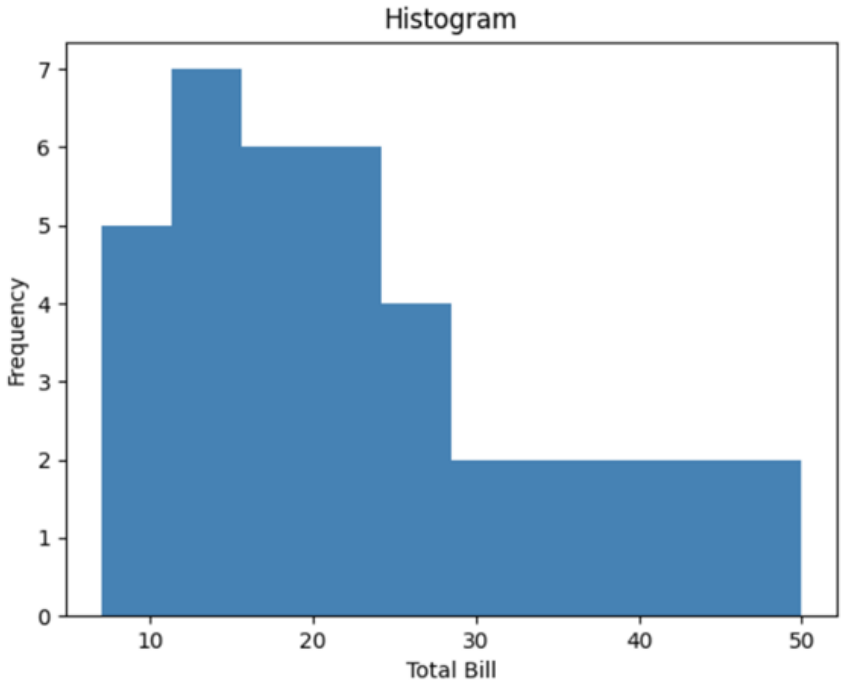


Fig 3.3.3 Histogram

4. Scatter Plot

Scatter plots are used to examine the relationship or correlation between two variables by plotting individual data points on a coordinate system. In Matplotlib, the scatter() function is used to generate scatter plots.

Syntax:

matplotlib.pyplot.scatter(x, y)

Parameters:

- x → X-axis coordinates of the data points.
- y → Y-axis coordinates of the data points.

Example: The following code creates a scatter plot to visualize the relationship between days and total bill amounts.

```
import matplotlib.pyplot as plt
x = ['Thur', 'Fri', 'Sat', 'Sun', 'Thur', 'Fri', 'Sat', 'Sun']
y = [170, 120, 250, 190, 160, 130, 240, 200]
plt.scatter(x, y, color='green')
plt.title("Scatter Plot")
plt.xlabel("Day")
plt.ylabel("Total Bill")
plt.show()
```

See Fig. 3.3.4 Scatter Plot

Output

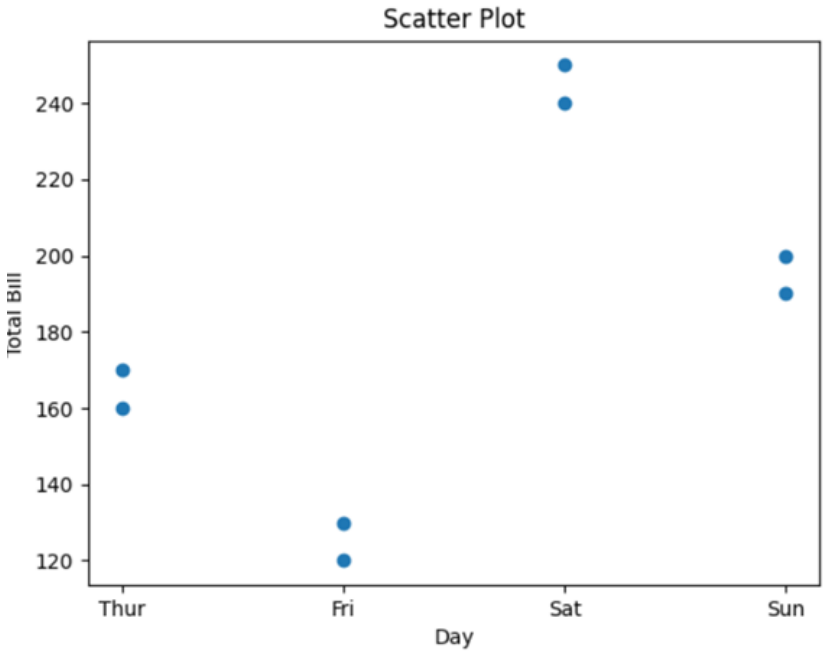


Fig 3.3.4 Scatter plot

5. Pie Chart

A pie chart is a circular visualization that represents data as proportions or percentages of a whole. Each segment (or wedge) corresponds to a specific category. In Matplotlib, the pie() function is used to create pie charts.

Syntax:

matplotlib.pyplot.pie(x, labels=None, autopct=None)

Parameters:

- x → Values for each slice of the pie.
- labels → Names or categories for each slice.

- autopct → String format to display percentages on slices (e.g., '%1.1f%%').

Example: The following code generates a pie chart to show the distribution of cars by brand. Each wedge represents the proportion of cars for that brand.

```
import matplotlib.pyplot as plt

cars = ['AUDI', 'BMW', 'FORD', 'TESLA', 'JAGUAR']

data = [23, 10, 35, 15, 12]

plt.pie(data, labels=cars, autopct='%1.1f%%', startangle=90)

plt.title("Car Brand Distribution")

plt.show()
```

See Fig. 3.3.5 Piechat

Output

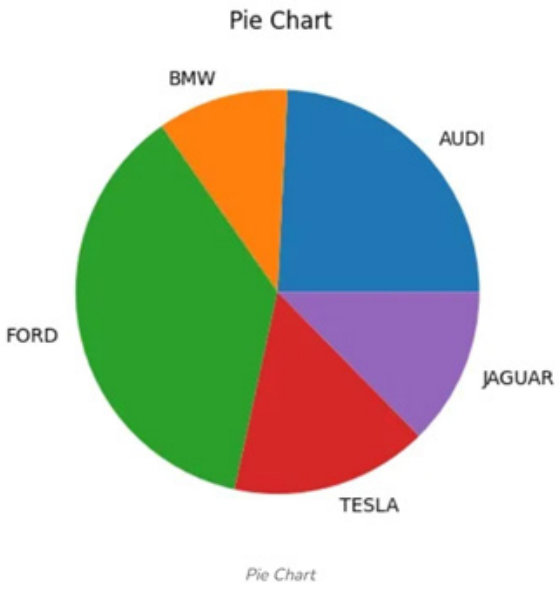


Fig 3.3.5 Pie Chart

6. Box Plot

A box plot (or whisker plot) is a graphical representation used to display the distribution of a dataset. It highlights key statistics such as minimum, maximum, median, and quartiles, and is also useful for identifying outliers.

Syntax:

matplotlib.pyplot.boxplot(x, notch=False, vert=True)

Parameters:

- x → The dataset to be plotted (list, array, or multiple groups).
- notch → If set to True, displays a notch around the median representing the confidence interval.
- vert → If True, draws vertical boxes; if False, draws horizontal boxes.

Example: The following code creates a box plot to compare the distribution of three different groups of data.

```
import matplotlib.pyplot as plt

data = [
    [10, 12, 14, 15, 18, 20, 22],
    [8, 9, 11, 13, 17, 19, 21],
    [14, 16, 18, 20, 23, 25, 27]
]

plt.boxplot(data)

plt.xlabel("Groups")

plt.ylabel("Values")

plt.title("Box Plot Example")

plt.show()
```

See Fig. 3.3.6 Box Plot

Output

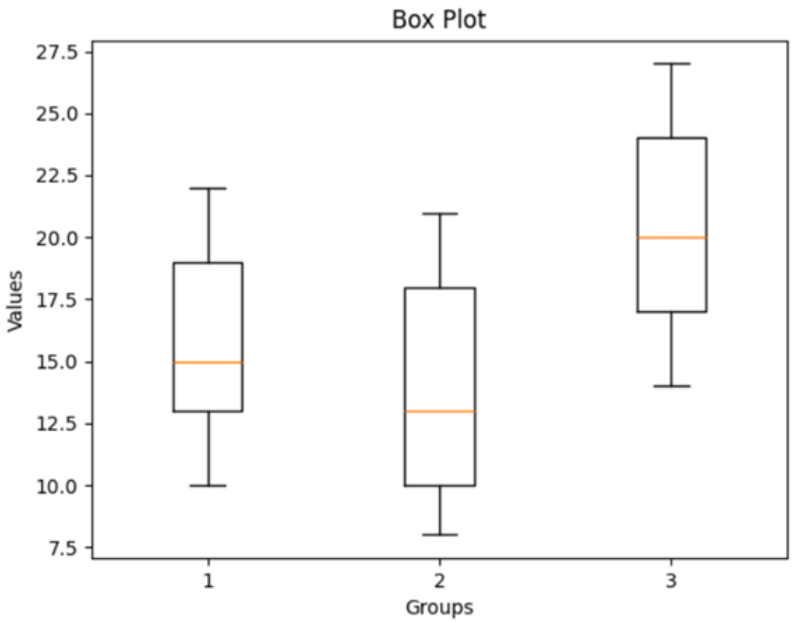


Fig. 3.3.6 Box Plot

3.3.1.2 Customizing Matplotlib Visualizations

Matplotlib provides various options to enhance the appearance and clarity of your plots. By modifying elements such as colors, line styles, markers, labels, titles, and gridlines, you can make your visualizations more informative and visually attractive.

Customization helps communicate data insights more effectively. Below are some common ways to personalize your plots:

1. Customizing a Line Chart

Line charts can be enhanced and styled by adjusting several attributes:

- Color → Change the line's color.
- Linewidth → Set the thickness of the line.
- Marker → Define the shape of data points on the line.
- Markersize → Adjust the size of the markers.
- Linestyle → Choose the line style (solid, dashed, dotted, etc.).

Example: The code below demonstrates a line chart with a green dashed line, thicker width, larger circular markers, and properly labeled axes and title.

```
import matplotlib.pyplot as plt
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]
plt.plot(x, y, color='green', linewidth=3, marker='o', markersize=15, linestyle='--')
plt.title("Customized Line Chart")
plt.xlabel("X-Axis")
plt.ylabel("Y-Axis")
plt.show()
```

See Fig. 3.3.6 Customizing Line chart.

Output

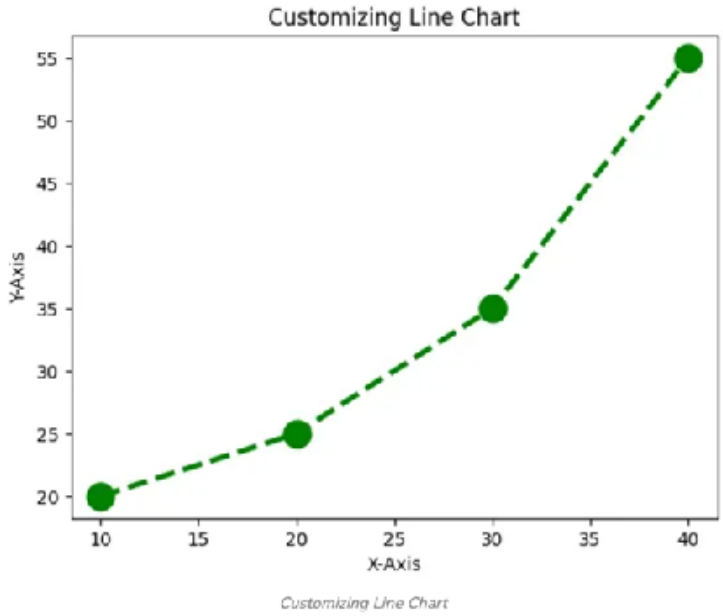


Fig 3.3.7 Customizing Line Chart

2. Customizing a Bar Chart

Bar charts can be styled to enhance readability and aesthetics by adjusting these properties:

- Color → Sets the fill color of the bars.
- Edgecolor → Defines the color of the bar borders.
- Linewidth → Controls the thickness of the bar edges.
- Width → Adjusts the width of each bar.

Example: The following code demonstrates a bar chart with green bars, black edges, thicker borders, and properly labeled axes and title.

```
import matplotlib.pyplot as plt
x = ['Thur', 'Fri', 'Sat', 'Sun']
y = [170, 120, 250, 190]
plt.bar(x, y, color='green', edgecolor='black', linewidth=2)
plt.title("Customized Bar Chart")
plt.xlabel("Day")
plt.ylabel("Total Bill")
plt.show()
```

See Fig. 3.3.8 Customizing Bar Chart.

Output

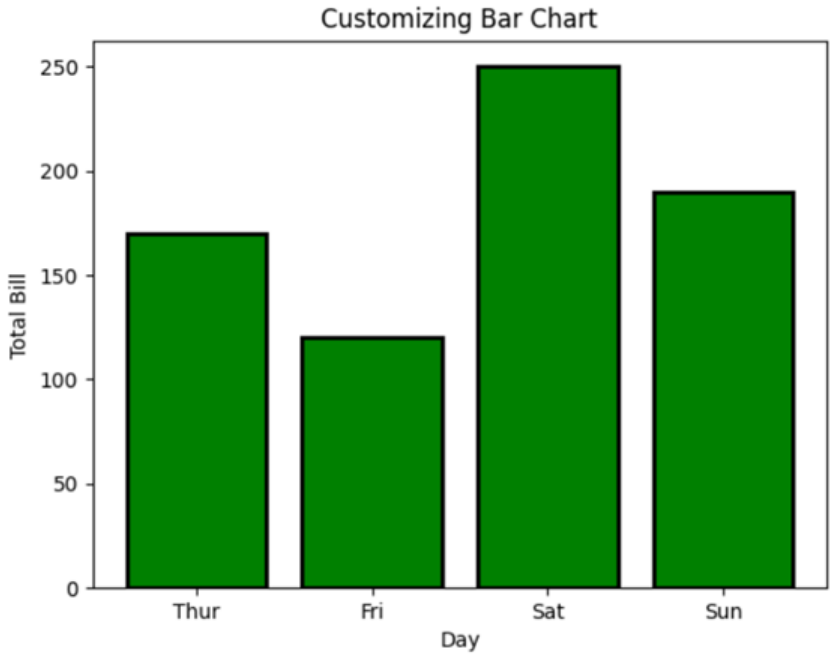


Fig. 3.3.8 Customizing Bar Chart

3. Customizing a Histogram

Histograms can be enhanced for better visualization and clarity by applying several customizations:

- Bins → Sets the number of intervals to group the data.
- Color → Fills the bars with a specific color.
- Edgecolor → Sets the color of the bar borders.
- Linestyle → Determines the style of the bar edges (solid, dashed, etc.).
- Alpha → Adjusts the transparency of the bars (0 = fully transparent, 1 = fully opaque).

Example: The following code produces a histogram with green bars, blue dashed edges, semi-transparent fill, and labeled axes and title.

```
import matplotlib.pyplot as plt
x = [7, 8, 9, 10, 10, 12, 12, 12, 13, 14, 14, 15, 16, 16, 17,
    18, 18, 19, 20, 20, 21, 22, 23, 24, 25, 25, 26, 28, 30,
    32, 35, 36, 38, 40, 42, 44, 48, 50]
plt.hist(x, bins=10, color='green', edgecolor='blue', linestyle='--', alpha=0.5)
plt.title("Customized Histogram")
plt.xlabel("Total Bill")
plt.ylabel("Frequency")
plt.show()
```

See Fig. 3.3.9 Customizing Histogram.

Output

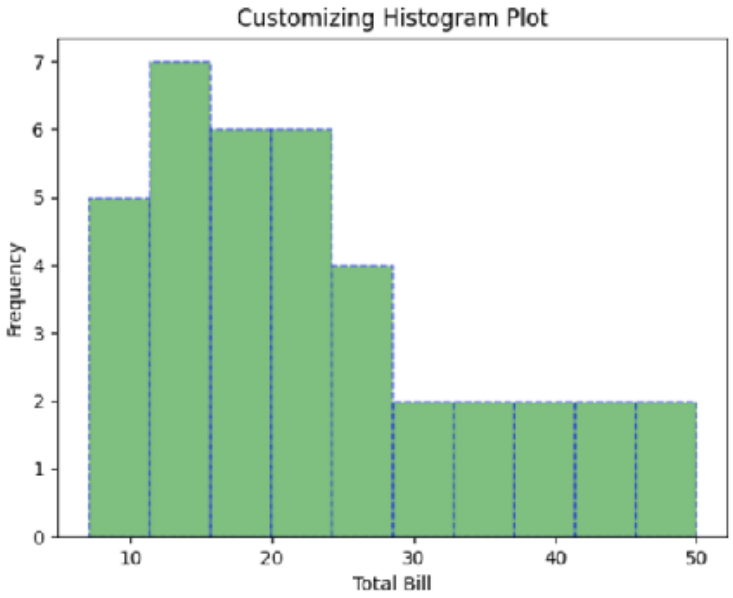


Fig. 3.3.9 Customizing Histogram Plot

4. Customizing a Scatter Plot

Scatter plots can be made more informative and visually appealing by adjusting several properties:

- s → Sets the size of the markers (can be a single value or a list).
- c → Determines the color of markers (single color or sequence of colors).
- marker → Defines the shape of the markers (circle, diamond, etc.).
- linewidths → Sets the thickness of marker edges.
- edgecolor → Changes the color of the marker borders.
- alpha → Controls transparency of the markers (0 = fully transparent, 1 = fully opaque).

Example: The following code generates a customized scatter plot with diamond-shaped markers. Marker color reflects size, marker size represents total bill amounts, and partial transparency improves visibility. The plot also includes labeled axes and a title.

```
import matplotlib.pyplot as plt
x = ['Thur', 'Fri', 'Sat', 'Sun', 'Thur', 'Fri', 'Sat', 'Sun']
y = [170, 120, 250, 190, 180, 130, 260, 200]
size = [2, 3, 4, 2, 3, 2, 4, 3]
bill = [170, 120, 250, 190, 180, 130, 260, 200]
plt.scatter(x, y, c=size, s=bill, marker='D', alpha=0.5)
plt.title("Customized Scatter Plot")
plt.xlabel("Day")
plt.ylabel("Total Bill")
plt.show()
```

See Fig. 3.3.10 Customizing Scatter Plot.

Output

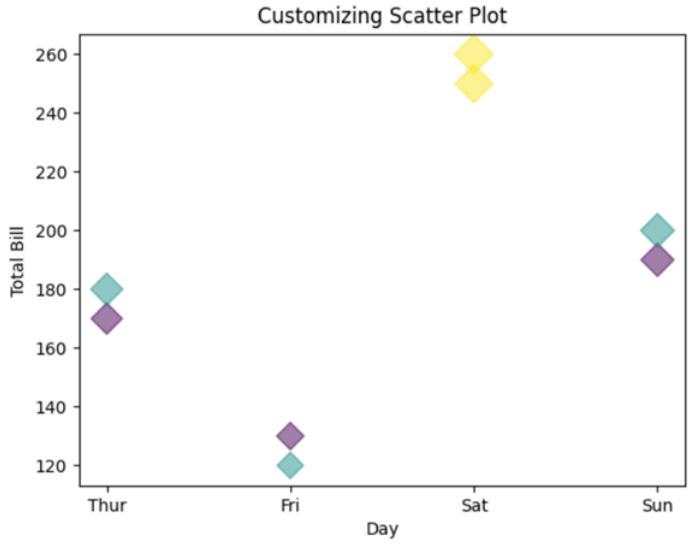


Fig 3.3.10 Customizing Scatter Plot

3.3.2 Data Visualization with Seaborn – Python

Seaborn is a widely used Python library for producing visually appealing statistical graphics. It is built on top of Matplotlib and works seamlessly with Pandas, allowing users to create advanced plots, such as line charts, heatmaps, and violin plots easily and with minimal code.

3.3.2.1 Creating Plots with Seaborn

Seaborn simplifies the process of generating clear and informative statistical visualizations using just a few lines of code. It provides built-in themes, color palettes, and specialized functions for different types of data, making plotting more intuitive and visually appealing.

Below are examples of common plot types with simple code to demonstrate how to use Seaborn effectively.

1. Line Plot

A line plot is used to display the relationship between two numeric variables, often across time. It can also show comparisons between multiple groups using separate lines.

Syntax:

sns.lineplot(x=None, y=None, data=None)

Parameters:

- x, y: Numeric variables, which can be lists, arrays, or column names from a DataFrame.
- data: The DataFrame containing the data.

Example:

```
import pandas as pd

import matplotlib.pyplot as plt

data = {'Name': ['ANSH', 'SAHIL', 'JAYAN', 'ANURAG'], 'Age': [21, 23, 20, 24]}

df = pd.DataFrame(data)

plt.plot(df.index, df['Age'])

plt.xlabel('Index')

plt.ylabel('Age')

plt.title('Age Line Plot')

plt.show()
```

See Fig. 3.3.11 Age Line Plot.

Output

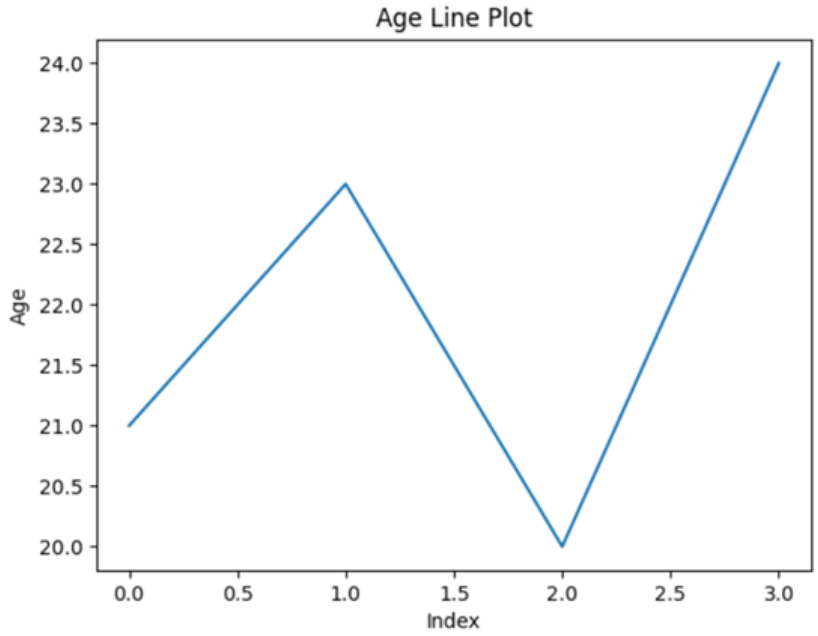


Fig 3.3.11 Age Line Plot

2. Scatter Plot

Scatter plots are useful for visualizing the relationship between two numerical variables. They help reveal correlations, trends, or patterns by plotting data points on a two-dimensional graph.

Syntax:

sns.scatterplot(x=None, y=None, data=None)

Parameters:

- x, y: Numeric variables to be plotted on the x-axis and y-axis.
- data (optional): The dataset (e.g., a DataFrame) containing the variables.

Returns:

An Axes object displaying the scatter plot.

Example:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
data = {'Name': ['ANSH', 'SAHIL', 'JAYAN', 'ANURAG'], 'Age': [21, 23, 20, 24]}
df = pd.DataFrame(data)
sns.scatterplot(x=df.index, y='Age', data=df)
```

plt.show()

See Fig. 3.3.12 Scatter plot.

Output

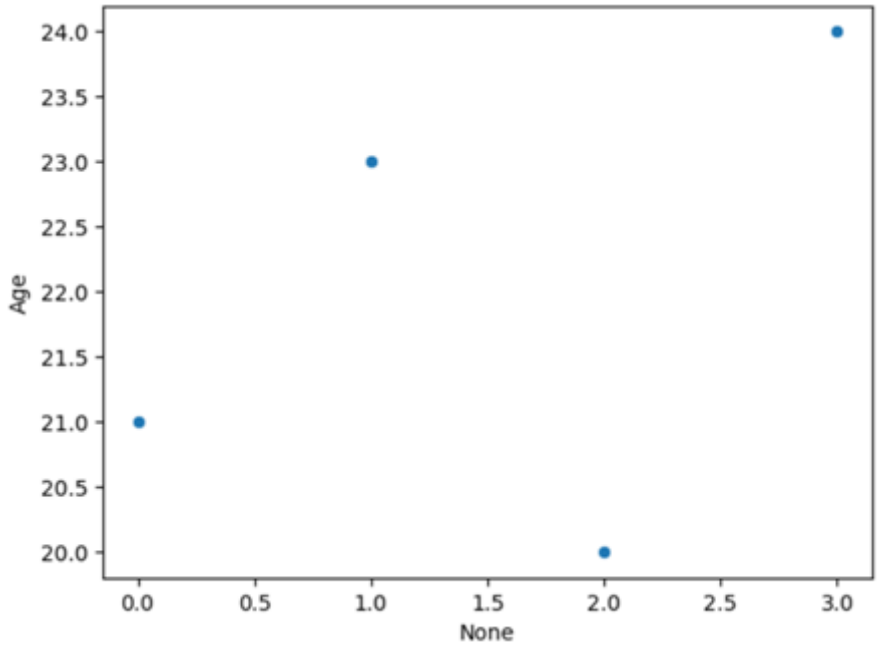


Fig 3.3.12 Scatter plot

3.3.2.2 Customizing Seaborn Plots in Python

Enhancing Seaborn plots through customization improves both their clarity and visual impact, making it easier to interpret data insights. The following techniques can be used to tailor Seaborn plots to your needs:

1. Adding Titles and Axis Labels

Including meaningful titles and axis labels makes plots easier to read and interpret. In Seaborn, you can use Matplotlib functions plt.title(), plt.xlabel(), and plt.ylabel() to add this information.

```
import seaborn as sns

import matplotlib.pyplot as plt

# Load dataset

iris = sns.load_dataset('iris')

# Create scatter plot

sns.scatterplot(x='sepal_length', y='sepal_width', data=iris)

# Customize with title and axis labels

plt.title('Sepal Length vs Sepal Width')
```

plt.xlabel('Sepal Length (cm)')

plt.ylabel('Sepal Width (cm)')

plt.show()

See Fig. 3.3.13 Adding Titles and Labels.

Output

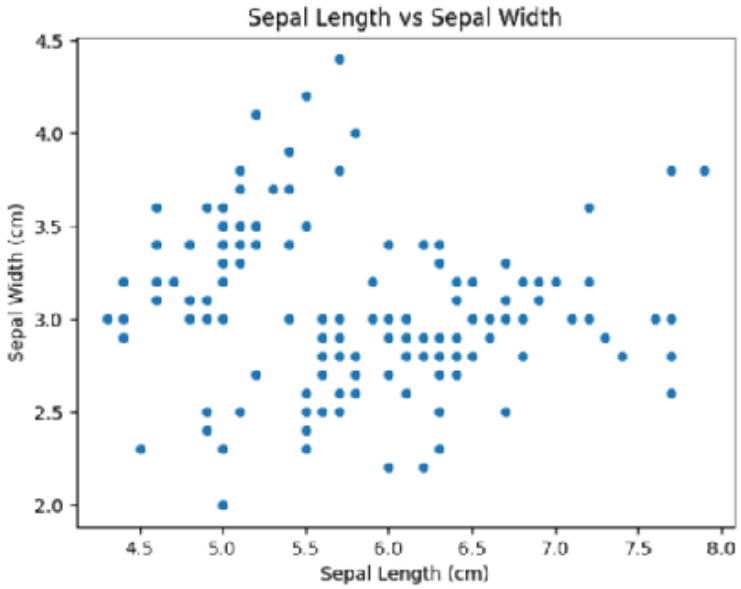


Fig. 3.3.13 Adding Titles and Labels

2. Using Built-in Styles and Grids in Seaborn

Seaborn offers several built-in styles to customize the background and grid of your plots. These styles enhance clarity and readability, allowing you to select the one that best fits your presentation or analysis.

Common Styles:

- darkgrid – Dark background with light gridlines; provides strong contrast.
- whitegrid – Light background with subtle gridlines; ideal for statistical plots.
- dark – Dark background without gridlines; gives a sleek, modern appearance.
- white – Plain white background without gridlines; simple and minimalistic.
- ticks – White background with sharply styled axis ticks; suitable for publication-ready visuals.

import seaborn as sns

import matplotlib.pyplot as plt

Set Seaborn style

```
sns.set_style("whitegrid")

# Create a box plot

sns.boxplot(x='species', y='petal_length', data=sns.load_dataset('iris'))

plt.title('Petal Length Distribution by Species')

plt.show()
```

See Fig. 3.3.14 Representation of Whitegrid in Boxplot

Output

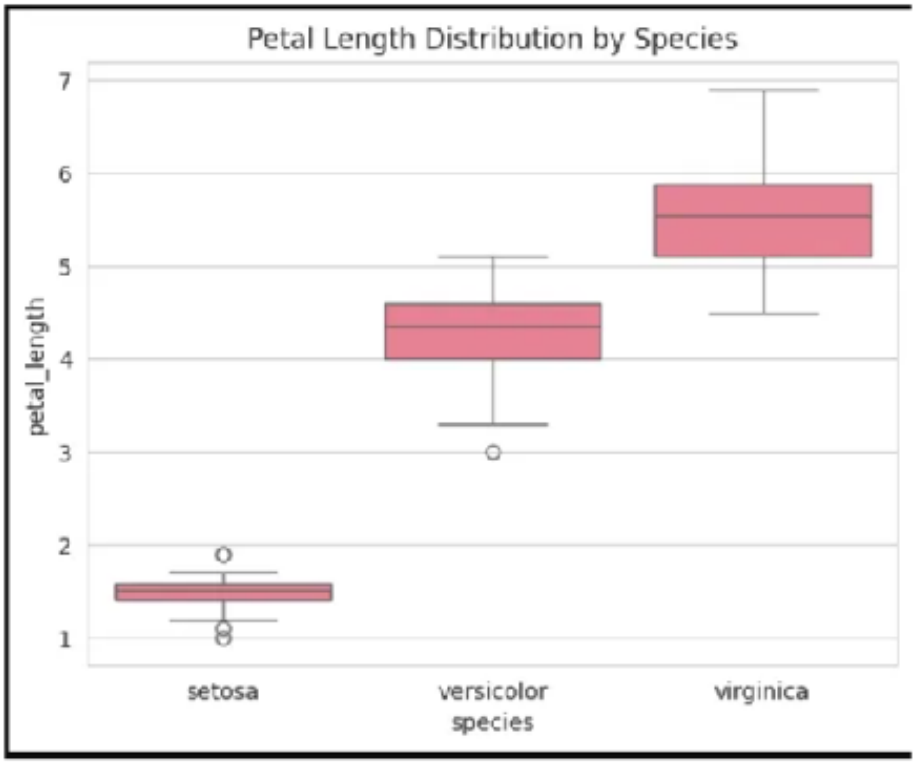


Fig. 3.3.14 Representation of Whitegrid in Boxplot

3. Customizing Color Palettes

Seaborn allows you to enhance your plots by applying color palettes, which can make your visualizations more appealing and easier to interpret. You can select from pre-defined palettes such as "deep", "muted", or "bright", or create your own custom palette using sns.color_palette(). Applying the right colors can improve clarity and help your plots align with the theme or purpose of your data.

Example – Using a Built-in Palette:

```
import seaborn as sns

import matplotlib.pyplot as plt
```

```
# Apply a built-in pastel palette
sns.set_palette("pastel")
# Create a violin plot
sns.violinplot(x='species', y='petal_length', data=sns.load_dataset('iris'))
plt.title('Petal Length Distribution by Species')
plt.show()
```

See Fig.3.3.15 Using Built-in Palette.

Output

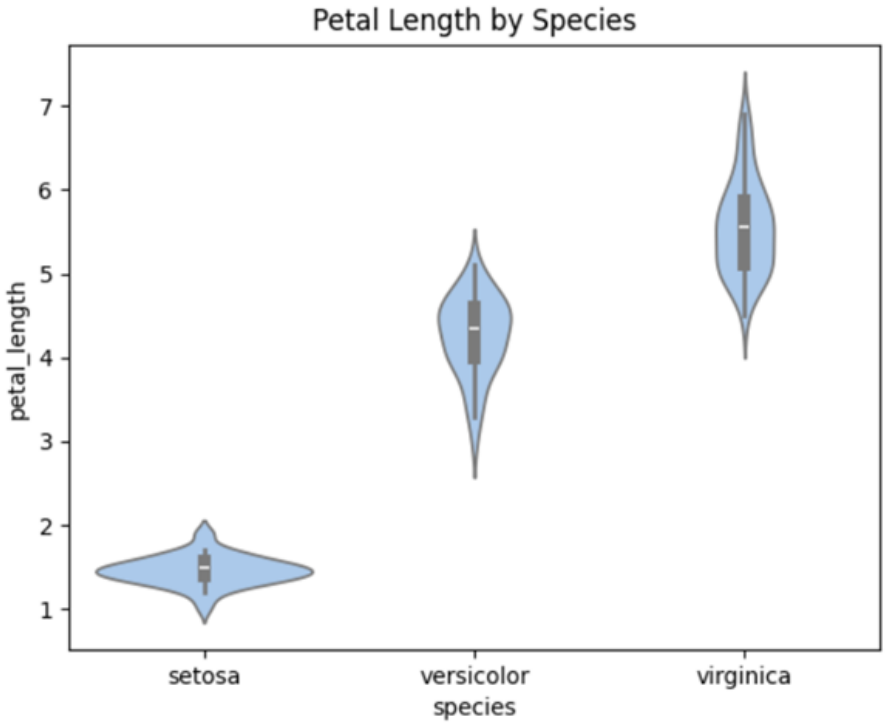


Fig.3.3.15 Using Built in Palette

b) Applying a Custom Color Palette

You can also define your own color palette to give your plots a unique look. This allows you to assign specific colors to different categories for better visual distinction.

```
import seaborn as sns
import matplotlib.pyplot as plt
# Define a custom color palette
custom_colors = ['#FF5733', '#33FFBD', '#335BFF']
sns.set_palette(custom_colors)
```

```
# Create a violin plot using the custom colors
sns.violinplot(x='species', y='petal_length', data=sns.load_dataset('iris'))
plt.title('Petal Length Distribution with Custom Colors')
plt.show()
```

See Fig.3.3.16 Using Custom Palette.

Output

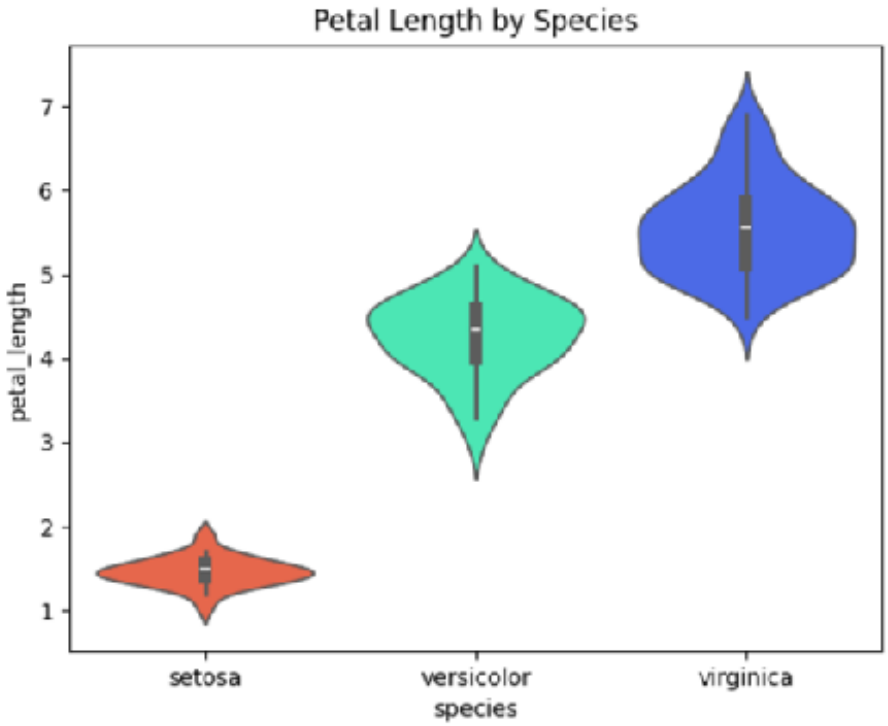


Fig 3.3.16 Using Custom Palette

Summarized Overview

Matplotlib is a widely used Python library for creating static, animated, and interactive visualizations. Built on NumPy, it efficiently handles large datasets and supports a variety of plot types, including line charts, bar charts, scatter plots, histograms, pie charts, and box plots. The Pyplot module in Matplotlib provides a simple interface to generate these plots with minimal code. Line charts display the relationship or trend between two variables using the plot() function. Bar charts visually compare categorical data with rectangular bars, while histograms show the frequency distribution of data by grouping it into bins. Scatter plots reveal correlations or patterns between two numerical variables, and pie charts depict data as proportions of a whole. Box plots summarize the distribution of data, highlighting the median, quartiles, and outliers. Matplotlib also allows extensive customization, such as adjusting colors, line styles, marker types, widths, and transparency to enhance clarity and aesthetics. Similarly, Seaborn is built on top of Matplotlib and integrates with Pandas to simplify the creation of statistical plots. It offers built-in themes, color palettes, and functions for line plots, scatter plots, violin plots, and more. Users can further customize Seaborn plots by adding titles, axis labels, selecting grid styles, or applying built-in and custom color palettes. Customization improves the readability, visual appeal, and interpretability of data insights. Together, Matplotlib and Seaborn provide powerful tools for data visualization in Python, allowing users to create both basic and advanced visualizations effectively. By combining flexibility with simplicity, these libraries help communicate data insights clearly and professionally.

Assignments

1. Explain the concept of a line chart in Matplotlib and write a Python program to plot a line chart showing the trend of sales over four consecutive days.
2. Describe the bar chart and its uses in data visualization. Create a bar chart using Matplotlib to compare the total bill amounts for different weekdays.
3. What is a histogram, and how does it help in understanding data distribution? Write a Python program to plot a histogram for a given dataset of total bills.
4. Define a scatter plot and discuss its significance in analyzing relationships between two variables. Create a scatter plot using Matplotlib for total bills versus days of the week.

5. Explain the purpose of a pie chart and how it represents data as proportions. Using Matplotlib, generate a pie chart showing the percentage distribution of cars by brand.
6. Discuss the box plot and its role in visualizing data distribution and detecting outliers. Write a Python program to create a box plot comparing three groups of numerical data.
7. Explain at least three ways to customize Matplotlib plots (e.g., line charts, bar charts, histograms) to improve readability and aesthetics. Provide example code for one customized plot.
8. Describe how Seaborn enhances data visualization over Matplotlib. Demonstrate with code how to create a scatter plot with titles, axis labels, and a custom color palette using Seaborn.

Reference

1. Idris, Ivan. Python data analysis cookbook. Packt Publishing Ltd, 2016.
2. Mukhiya, Suresh Kumar, and Usman Ahmed. Hands-On Exploratory Data Analysis with Python: Perform EDA techniques to understand, summarize, and investigate your data. Packt Publishing Ltd, 2020.
3. Waskom, Michael L. "Seaborn: statistical data visualization." Journal of open source software 6.60 (2021): 3021.

Suggested Reading

1. Pooja, Dr. Data Visualization with Python: Exploring Matplotlib, Seaborn, and Bokeh for Interactive Visualizations (English Edition). BPB Publications, 2023.
2. Dougherty, Jack, and Ilya Ilyankou. Hands-on data visualization. " O'Reilly Media, Inc.", 2021.
3. Gupta, Pramod, and Anupam Bagchi. "Data visualization with Python." Essentials of Python for Artificial Intelligence and Machine Learning. Cham: Springer Nature Switzerland, 2024. 237-282

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT

Web Programming with Flask

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the purpose of Python libraries and how they support code reuse
- explain the purpose and basic components of the Flask framework
- describe the process of routing URLs to functions in a Flask application
- summarize how templates are used to render dynamic HTML content
- illustrate the difference between GET and POST requests for handling form data
- differentiate between the four basic CRUD operations (Create, Read, Update, Delete)

Background

Web programming has evolved significantly, moving from static HTML pages to dynamic, interactive applications. Modern web frameworks, like Flask, simplify this process by providing a structured environment and tools for developers. They abstract away the complexities of low-level networking and protocol handling, allowing programmers to focus on application logic. Python, a popular language for web development, offers a variety of frameworks, with Flask being a standout choice for its simplicity and minimalism. It's a micro-framework, meaning it provides the core components for a web application without forcing developers into a rigid structure or including a lot of unnecessary features. This "less is more" approach makes Flask ideal for beginners, small projects, and APIs where a lightweight solution is needed. It empowers developers to build and deploy web applications quickly and efficiently.

This unit focuses on the fundamentals of web development using Flask. You will start by understanding the basic structure of a Flask application, including routing, which maps URLs to specific functions that handle requests. A key part of any interactive web application is user input, which is managed through forms. You will learn how to create and handle forms to collect user data. The concept of templates is crucial for separating the presentation layer (HTML) from the application logic (Python code), allowing for a clean and maintainable codebase. You will also cover handling GET and POST requests, the two primary methods for communicating with a web server and learn how to use them for different purposes. Finally, you will be introduced to basic CRUD operations (Create, Read, Update, Delete), which are the building blocks of most data-driven applications. Mastering these concepts will provide a solid foundation for building more complex and feature-rich web applications with Flask.

Keywords

Routing, Forms, Templates, GET, POST, CRUD operations

Discussion

Web Programming with Flask is a powerful and flexible way to build dynamic web applications using Python. Flask is a lightweight micro-framework that provides essential tools and features for developing web projects with simplicity and efficiency. This unit introduces the fundamentals of Flask, covering the basics of application setup, routing mechanisms, handling forms, and working with templates to create interactive user interfaces. It also explores how to manage client requests through GET and POST methods and demonstrates how to implement basic CRUD (Create, Read, Update, Delete) operations, which are at the core of most web applications. By learning these concepts, you gain practical knowledge to design, develop, and manage functional web applications.

3.4.1 Introduction to Flask

Flask is a lightweight web framework in Python used to build web applications and APIs. It is called a micro-framework because it provides only the essential tools to get started, while giving developers the flexibility to add extensions as needed for features like databases, authentication, or form handling. Flask includes a built-in development server, routing system, and template engine (Jinja2) to create dynamic web pages. Its simplicity, flexibility, and ease of learning make it a popular choice for both beginners

and professionals to quickly develop scalable and efficient web applications. Unlike larger frameworks that come with many built-in features, Flask provides only the essentials such as routing, request handling, and templates allowing developers to add extensions only when needed. This makes it easy to learn, and well-suited for beginners as well as professionals. It is widely used for creating everything from small prototypes to scalable applications and RESTful APIs, with strong community support and plenty of available extensions.

It was developed by Armin Ronacher. Flask's framework is more explicit than Django's framework and is also easier to learn because it has less base code to implement a simple web application. Flask Python is based on the WSGI (Web Server Gateway Interface) toolkit and Jinja2 template engine.

There are many modules or frameworks which allow building your webpage using python like Bottle, Django, and Flask. But the real popular ones are Flask and Django. Django is easy to use as compared to Flask, but Flask provides you with the versatility to program with. Flask is a web application framework written in Python. To understand what Flask is you have to understand a few general terms.

1. **WSGI:** Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.
2. **Werkzeug:** It is a WSGI toolkit, which implements requests and response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.
3. **jinja2:** jinja2 is a popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

3.4.1.1 Advantages of Flask

Flask is simple, flexible, and powerful, making it a great choice for rapid development and scalable web applications. Some advantages are:

1. Lightweight and Simple

- Provides only the essential components, making it easy to learn and use.
- Developers can add features as required instead of dealing with unnecessary complexity.

2. Flexibility

- Does not enforce strict project structures or tools.
- Allows developers to choose libraries, databases, and extensions as per project needs.

3. Built-in Development Server and Debugger

- Comes with a powerful debugger and built-in server for quick testing.

- Helps identify and fix errors during development easily.

4. Jinja2 Template Engine

- Supports dynamic HTML generation.
- Makes it easier to separate design (frontend) from logic (backend).

5. RESTful Request Handling

- Supports HTTP methods like GET, POST, PUT, and DELETE.
- Ideal for building REST APIs and CRUD applications.

6. Large Community and Extensions

- Offers a wide range of extensions for database integration, authentication, and more.
- Extensive documentation and community support.

7. Scalability

- Suitable for both small projects and larger, more complex applications.
- Used by startups as well as enterprise-level systems.

3.4.1.2 Basic Flask Program

To install Flask, you first need to have Python installed on your system (preferably version 3.6 or above). Once Python is ready, you can use the package manager *pip* to install Flask by running the command pip install flask in your terminal or command prompt. After installation, you can verify it by typing python *-m* flask *--version*, which will show the Flask version along with its dependencies like Werkzeug and Jinja2. For better project management, it is recommended to create a virtual environment and install Flask inside it, ensuring that your project dependencies remain organized and separate from system-wide packages.

A basic Flask application requires only a few lines of code. It creates a web server, defines routes (URLs), and returns responses to the user.

Example:

```
from flask import Flask

# Create the Flask application

app = Flask(__name__)

# Define a route (URL) and its corresponding function

@app.route('/')

def home():
```

```
    return "Hello, Flask! Welcome to your first web app."

# Run the app

if __name__ == '__main__':

    app.run(debug=True)
```

where,

- *from flask import Flask* : Imports the Flask class.
- *app = Flask(__name__)* : Creates the Flask application object.
- *@app.route('/')* : Defines the home route (URL path /).
- *def home():* :Function that returns the response when a user visits the home page.
- *app.run(debug=True)* : Runs the app with debug mode enabled, which helps in automatic reloading and error tracking.

When you save this code as app.py and run it, Flask starts a local development server (usually at http://127.0.0.1:5000/). Opening this URL in a browser will display:

Hello, Flask! Welcome to your first web app.

3.4.2 Routing in Flask

In Flask, **routing** is the process of linking a specific URL (web address) to a function in the application so that when a user visits that URL, the corresponding function is executed, and a response is sent back to the browser. It is defined using the *@app.route()* decorator, which maps URLs to view functions. Routes can be **static**, where the URL is fixed (like */about*), or **dynamic**, where parts of the URL act as variables (like */user/<name>)*. Routing is essential in web applications because it allows users to navigate between different pages and access personalized or dynamic content.

Defining Routes

In Flask, routes are defined using the **@app.route() decorator**, which maps a URL to a Python function (called a **view function**). When a user visits the specified URL, Flask executes the function and returns its output as a response. A single application can have multiple routes, each handling different pages or functionalities.

Example:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def home():

    return "This is the Home Page"
```

```
@app.route('/about')
def about():
    return "This is the About Page"
if __name__ == '__main__':
    app.run(debug=True)
```

where,

- *@app.route('/')* → Defines the home page route.
- *home()* → Runs when the home URL *(/)* is accessed.
- *@app.route('/about')* → Defines another route for the About page.
- *about()* → Runs when */about* is visited.

Every route corresponds to a function, and the function's return value is displayed in the browser when the route is accessed.

3.4.2.1 Static Routing

In Flask, **static routing** refers to defining fixed URL paths that always return the same response whenever they are accessed. These routes do not accept any variables, meaning the content displayed is constant for all users. Static routing is typically used for pages like the Home page, About page, or Contact page, where the information remains the same regardless of who visits the site. It is the simplest form of routing and forms the foundation of navigation in a Flask application.

Syntax

```
@app.route('/fixed_path')
def function_name():
    return "Fixed response"
```

where,

- ***@app.route('/fixed_path')***
  - This is a decorator in Flask.
  - It tells the application that whenever the user visits the URL path /fixed_path, the function below it should be executed.
  - **Example:** If the path is /about, visiting http://127.0.0.1:5000/about will call the linked function.
- ***def function_name():***
  - This defines the view function that runs when the specified route is accessed.

- The function name can be anything (e.g., home, about, contact).

♦ ***return "Fixed response"***

- The function must return a value, usually a string, HTML content, or even a template.
- This return value is what the user sees in their web browser.

Whenever a user goes to */fixed_path* in the browser, Flask will call the *function_name()* function and display the message Fixed response on the screen.

Example:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def home():

    return "Welcome to the Home Page"

@app.route('/about')

def about():

    return "This is the About Page"

if __name__ == '__main__':

    app.run(debug=True)
```

Output:

Visiting http://127.0.0.1:5000/

Welcome to the Home Page

Visiting http://127.0.0.1:5000/about

This is the About Page

3.4.2.2 Dynamic Routing

In Flask, **dynamic routing** refers to defining routes that include variable parts in the URL, allowing the application to display content that changes based on user input or parameters passed in the URL. Instead of always returning the same response like static routes, dynamic routes can handle values such as usernames, IDs, or other data directly from the URL. For example, a route like */user/<name>* will accept different names and return personalized responses, making dynamic routing useful for building user profiles, product pages, and other interactive web applications.

Syntax

```
@app.route('/path/<variable>')
def function_name(variable):
    return f"Value is {variable}"
```

where,

- ***@app.route('/path/<variable>'):***
  - This defines a dynamic route.
  - The part inside < > (here, *<variable>)* acts as a placeholder in the URL.
  - Whatever value the user provides in place of *<variable>* will be captured and passed to the function as an argument.
  - **Example:** Visiting */path/hello* will capture "*hello*" as the variable.
- ***def function_name(variable):***
  - This is the view function that handles the request.
  - The parameter variable in the function receives the value from the URL.
  - The function name can be anything (like show_user, product, profile, etc.).
- ***return f"Value is {variable}":***
  - The function returns a response containing the value passed in the URL.
  - **Example:** If the user visits */path/123*, the output will be **"Value is 123"**.

When a user visits /path/something, Flask will send that *something* to the function as a variable, and the function will display it back in the response.

Example:

```
from flask import Flask
app = Flask(__name__)
@app.route('/user/<name>')
def user(name):
    return f"Hello, {name}!"
@app.route('/product/<int:id>')
def product(id):
    return f"Product ID: {id}"
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Output:

Visiting http://127.0.0.1:5000/user/John

Hello, John!

Visiting http://127.0.0.1:5000/product/101

Product ID: 101

Table 3.4.1: Static Routing vs Dynamic Routing in Flask

Feature	Static Routing	Dynamic Routing
Definition	URL path is fixed and always returns the same response.	URL path includes variables, so response changes based on user input.
URL Example	/about → always shows *About Page*.	/user/<name> → shows different content for different names.
Syntax	@app.route('/about')	@app.route('/user/<name>')
Function Example	python @app.route('/about') def about(): return "About Page" 	python @app.route('/user/<name>') def user(name): return f"Hello {name}"
Output Example	Visiting /about → **"About Page"**	Visiting /user/John → **"Hello John"**
Use Case	Static pages like Home, About, Contact.	Dynamic content like User Profiles, Product Pages, Dashboards.
Flexibility	Less flexible, content is always same.	Highly flexible, content changes based on URL parameters.

3.4.3 Forms

Forms in Flask are used to collect user input from the client side (web browser) and send it to the server for processing. They allow users to enter data such as text, numbers, passwords, or choices, which can then be used in applications for authentication, registration, data storage, searching, and more. In Flask, forms can be created using HTML form tags or with the help of libraries like Flask-WTF (which integrates WTForms for easier handling and validation).

3.4.3.1 Features of Forms in Flask

1. **User Input Collection**: Forms allow users to submit data through text fields, checkboxes, radio buttons, dropdowns, etc.
2. **Data Handling via HTTP Methods** : Supports GET and POST requests to send data to the server.

3. **Integration with Templates** : Forms can be embedded in Jinja2 templates for dynamic rendering.
4. **Form Validation** : Flask (with Flask-WTF) provides validation features to ensure submitted data is correct (e.g., email format, required fields).
5. **CSRF Protection** : Flask-WTF adds security against Cross-Site Request Forgery attacks by generating tokens.
6. **Error Handling** : Provides mechanisms to display error messages if validation fails.
7. **Flexibility** : Forms can be simple HTML or advanced with Flask-WTF, depending on the complexity needed.
8. **Seamless Integration with Backend** : Captured data can be processed, stored in databases, or used for application logic.

3.4.3.2 Installation for Forms in Flask

To work with advanced forms (validation, CSRF protection, easy handling), Flask usually uses Flask-WTF (a wrapper around WTForms).

1. Install Flask (if not already installed):

```
pip install flask
```

2. Install Flask-WTF:

```
pip install flask-wtf
```

3. In your Flask app, import it:

```
from flask_wtf import FlaskForm

from wtforms import StringField, PasswordField, SubmitField

from wtforms.validators import DataRequired
```

3.4.3.3 Common Field Types in Flask-WTF

Below table 3.4.2 are the frequently used form field types (from *wtforms*).

Table 3.4.2 : Common Field Types in Flask-WTF

Field Type	Description	Example
StringField	For short text input (like username, name, etc.)	username = StringField('Username')
PasswordField	For password input (masked text)	password = PasswordField('Password')
TextAreaField	For multi-line text input	about = TextAreaField('About Me')
EmailField	For email input with validation	email = EmailField('Email')

Field Type	Description	Example
IntegerField	For numeric input (integers only)	age = IntegerField('Age')
DecimalField	For decimal number input	price = DecimalField('Price')
BooleanField	Checkbox (True/False input)	remember = BooleanField('Remember Me')
RadioField	For selecting one option from multiple choices	gender = RadioField('Gender', choices=[('M','Male'),('F','Female')])
SelectField	Dropdown menu for single selection	country = SelectField('Country', choices=[('in','India'),('us','USA')])
SelectMultipleField	Dropdown allowing multiple selections	hobbies = SelectMultipleField('Hobbies', choices=[('cr','Cricket'),('mu','Music')])
FileField	For file upload	file = FileField('Upload File')
SubmitField	A submit button	submit = SubmitField('Login')

Program 1: Building a sample form using the above field types

Source Code:

```
# Importing Libraries..
from flask import Flask, render_template, request
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField
from wtforms import DecimalField, RadioField, SelectField, TextAreaField, FileField
from wtforms.validators import InputRequired
from werkzeug.security import generate_password_hash
app = Flask(__name__)
app.config['SECRET_KEY'] = 'secretkey'
class MyForm(FlaskForm):
    name = StringField('Name', validators=[InputRequired()])
    password = PasswordField('Password', validators=[InputRequired()])
    remember_me = BooleanField('Remember me')
```

```
    salary = DecimalField('Salary', validators=[InputRequired()])
    gender = RadioField('Gender', choices=[
                    ('male', 'Male'), ('female', 'Female')])
    country = SelectField('Country', choices=[('IN', 'India'), ('US', 'United States'),
                                    ('UK', 'United Kingdom')])
    message = TextAreaField('Message', validators=[InputRequired()])
    photo = FileField('Photo')
@app.route('/', methods=['GET', 'POST'])
def index():
    form = MyForm()
    if form.validate_on_submit():
        name = form.name.data
        password = form.password.data
        remember_me = form.remember_me.data
        salary = form.salary.data
        gender = form.gender.data
        country = form.country.data
        message = form.message.data
        photo = form.photo.data.filename
return f'Name: {name} <br> Password: {generate_password_hash(password)} <br>
Remember me: {remember_me} <br> Salary: {salary} <br> Gender: {gender} <br>
Country: {country} <br> Message: {message} <br> Photo: {photo}'
        <br > Remember me: {remember_me} < br > Salary: {salary} < br > Gender:
{gender}
    <br > Country: {country} < br > Message: {message} < br > Photo: {photo}'
    return render_template('index.html', form=form)
if __name__ == '__main__':
    app.run()
```

Html code

```
<!DOCTYPE html>
<html>
<head>
    <title>My Form</title>
</head>
<body>
    <h1>My Form</h1>
    <form method="post" action="/" enctype="multipart/form-data">
        {{ form.csrf_token }}
        <p>{{ form.name.label }} {{ form.name() }}</p>
        <p>{{ form.password.label }} {{ form.password() }}</p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.salary.label }} {{ form.salary() }}</p>
        <p>{{ form.gender.label }} {{ form.gender() }}</p>
        <p>{{ form.country.label }} {{ form.country() }}</p>
        <p>{{ form.message.label }} {{ form.message() }}</p>
        <p>{{ form.photo.label }} {{ form.photo() }}</p>
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

where,

- **Created a Form Class:** *MyForm* inherits from *FlaskForm* and includes various fields like *StringField, PasswordField, BooleanField*, etc., with appropriate validators.
- **Passed Form to Template:** In the *index* function, an instance of *MyForm* is created and sent to *index.html.*
- **Handled Form Submission:** The *validate_on_submit* method checks if the form is valid before processing data.
- **Retrieved and Displayed Data:** If valid, form data is extracted and shown in the browser.
- **Secured Passwords:** Used *generate_password_hash* to encrypt passwords for better security.

Output:

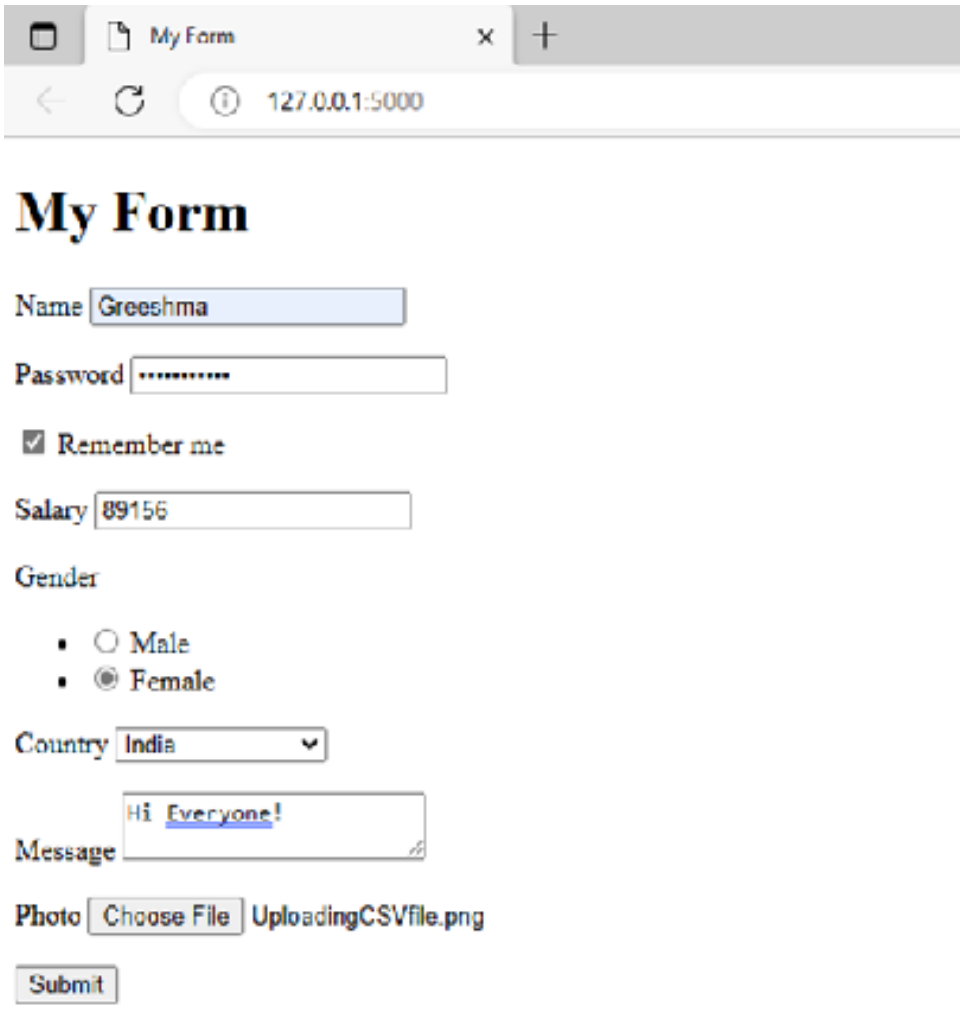


Fig 3.4.1 Form details

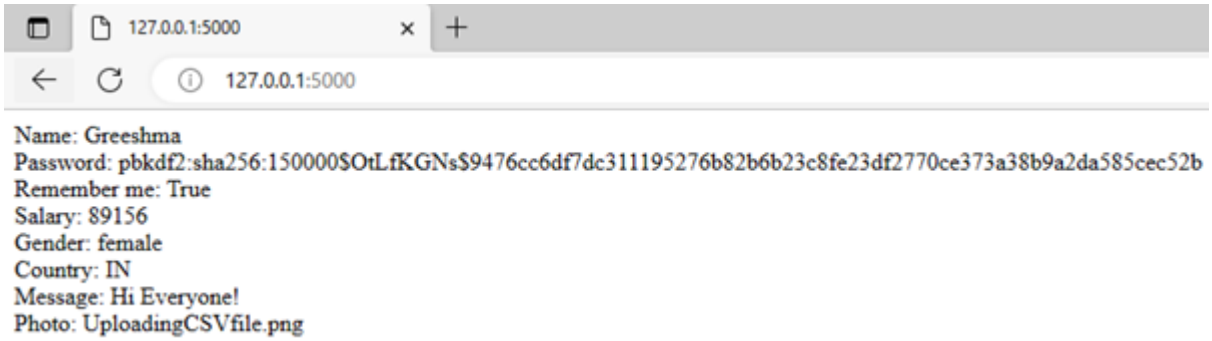


Fig 3.4.2 Details of the submitted form

Let's see another example of a simple Sign-up form using the Flask-WTF library where we use the field types mentioned above in our application.

Program 2: Sign-up form

Source Code:

```
from flask import Flask, render_template
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField ,SubmitField
from wtforms.validators import InputRequired, Length
app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret'
# Using StringField, PasswordField andSubmitField from Flask-WTForms library
# for username,password fields and submit button..
class LoginForm(FlaskForm):
  username = StringField('Username', validators=[InputRequired('Username required!'),
            Length(min=5, max=25, message='Username must be in 5 to 25 characters')])
          password = PasswordField('Password',validators=[InputRequired('Password required')])
    submit = SubmitField('Submit')
@app.route('/Signup', methods=['GET', 'POST'])
def form():
    form = LoginForm()
    if form.validate_on_submit():
      return '<h1>Hi {}!!. Your form is submitted successfully!!'.format(form.username.data)
    return render_template('Signup.html', form=form)
if __name__ == '__main__':
    app.run(debug=True)
```

Signup.html code

```
<!DOCTYPE html>
<html>
<head>
<title>Flask Form</title>
</head>
```

```
<body>
<h1>Signup </h1>
<form method="POST" action="{{ url_for('form') }}">
    {{ form.csrf_token }}
    {{ form.username.label }}
    {{ form.username }}
    <br>
    <br>
    {{ form.password.label }}
    {{ form.password }}
    <br>
    <br>
    {{ form.submit }}
</form>
</body>
</html>
```

where,

- **{{ form.csrf_token }}** adds a hidden input field to the form that contains a **CSRF** token. This is a security feature that helps prevent cross-site request forgery attacks.
- **{{ form.username.label }}** renders an HTML label for the username field of the form.
- **{{ form.username }}** renders an HTML input element for the username field of the form.
- **{{ form.password.label }}** renders an HTML label for the password field of the form.
- **{{ form.password }}** renders an HTML input element for the password field of the form.
- **{{ form. submit }}** renders an HTML input element for the form's submit button.

In our Flask route, we used *request.method* to check if the form was submitted (fig 3.4.3). The form is validated and processed only after submission (fig 3.4.4). Once submitted successfully, a message is displayed along with the username (fig 3.4.5).

Output:

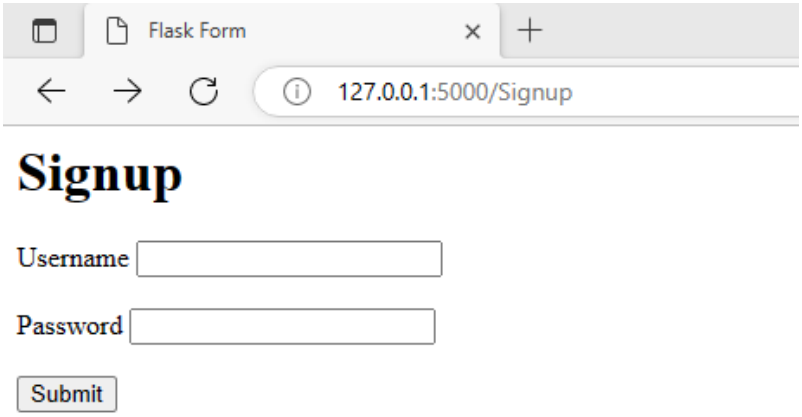


Fig 3.4.3 Signup form

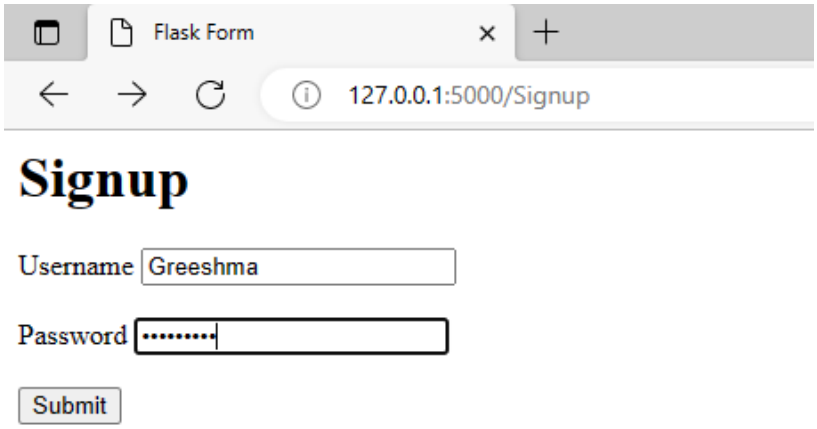


Fig 3.4.4 Details of the Signup form

Fig 3.4.5: Form submitted

3.4.4 Templates

In Flask, templates are HTML files managed using the Jinja2 template engine, which allow developers to insert dynamic content into web pages by using placeholders for variables and control structures like loops and conditions. They are stored in a special templates folder and rendered through the *render_template()* function. Templates are used because they help separate the design (frontend) from the application logic (backend), make code more organized and reusable, and enable web applications to display dynamic and personalized content instead of static pages.

3.4.4.1 Create a templates Folder

In Flask, all HTML files are stored inside a special folder named *templates*. Flask

automatically looks for templates in this folder when you use the *render_template()* function. Create a folder named templates in your project directory and create the index. html file.

Steps to Create a Templates Folder

1. Create a project directory (e.g., *myproject*).
2. Inside it, create a Python file (e.g., *app.py*).
3. Create a folder named templates in the same directory as *app.py.*
4. Save all your HTML files inside the *templates* folder.

Example

app.py

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def home():
    return render_template('index.html')   # loads from templates folder
@app.route('/about')
def about():
    return render_template('about.html')   # loads from templates folder
if __name__ == '__main__':
    app.run(debug=True)
```

templates/index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Home</title>
</head>
<body>
    <h1>Welcome to the Home Page!</h1>
</body>
</html>
```

Output:

Visiting / displays as:

Welcome to the Home Page!

Visiting */about* displays content from **about.html**

You must create a folder named *templates* (all lowercase) for Flask to detect and load HTML files automatically.

3.4.4.2 Templating with Jinja2 in Flask

Jinja2 is a powerful templating engine used by Flask to create dynamic web pages by combining static HTML with data from the backend. It allows developers to embed Python-like expressions directly inside HTML using special syntax, such as *{{ }}* for variables and *{% %}* for control statements like loops and conditions. Jinja2 makes it easier to separate the application logic from the presentation layer, supports template inheritance for reusability, and enables web applications to display personalized and dynamic content efficiently.

Jinja2 Syntax in Flask

Jinja2 provides special syntax to insert dynamic content, apply logic, and reuse HTML layouts.

1. Variables: Used to display dynamic values passed from Flask to the template.

Syntax: *{{ variable }}*

Example:

Hello, {{ name }}!

Output: If *name = "Alice",*

Hello, Alice!

2. Filters: Filters modify variables before displaying them (e.g., uppercase, lowercase, length).

Syntax: *{{ variable|filter }}*

Example:

{{ name|upper }}

Output: If *name = "Alice",*

ALICE

3. Conditions (if-else): Used to control what content is displayed.

Syntax:

{% if condition %}

...content...

{% elif another_condition %}

...content...

{% else %}

...content...

{% endif %}

Example:

```
{% if age >= 18 %}
  <p>You are an adult.</p>
{% else %}
  <p>You are a minor.</p>
{% endif %}
```

Output: If *age = 20*,

You are an adult.

4. Loops (for): Used to repeat content for each item in a list.

Syntax:

{% for item in list %}

{{ item }}

{% endfor %}

Example:

```
<ul>
{% for user in users %}
  <li>{{ user }}</li>
{% endfor %}
</ul>
```

Output: If *users = ["John", "Emma"],*

John

Emma

5. Comments: Used to add notes inside templates (not shown in output).

Syntax:

{# This is a comment #}

Example:

{# This will not appear on the webpage #}

6. Template Inheritance

Used to reuse layouts across multiple pages. *base.html* provides a structure, and child templates extend it.

Syntax:

{% extends "base.html" %}

{% block content %} {% endblock %}

Example:

base.html

```
<!DOCTYPE html>
<html>
<head><title>My Website</title></head>
<body>
  <h1>Header Section</h1>
  {% block content %} {% endblock %}
  <footer>Footer Section</footer>
</body>
</html>
```

home.html

```
{% extends "base.html" %}
{% block content %}
  <p>Welcome to the Home Page!</p>
{% endblock %}
```

Output:

Shows header and footer from *base.html* and inserts home page content in between.

3.4.5 Handling GET and POST

In Flask, handling **GET and POST** requests is an essential part of working with forms and user input. These two HTTP methods define how data is sent between the client (browser) and the server. The GET method is typically used to request or retrieve information from the server, while the POST method is used to submit or send data to the server for processing. By specifying these methods in routes and forms, Flask applications can control how data is transmitted, accessed, and processed, making them crucial for building interactive and dynamic web applications.

GET Method

In Flask, the **GET method** is used to request data from the server. When a form uses GET, the input values are appended to the URL as query parameters, which makes them visible in the address bar. This method is mainly used for retrieving or searching information rather than sending sensitive data. Since GET requests can be bookmarked and cached, they are suitable for actions like search forms, filtering data, or navigation where the request can be repeated without affecting the server state. However, because the data is exposed in the URL, GET is less secure and should not be used for sensitive operations like login.

Example: *http://localhost:5000/?username=Swathy&password=123*

POST Method

The **POST method** in Flask is used to send data securely from the client to the server. Unlike GET, the data is included in the request body instead of the URL, which hides it from the user's address bar and browser history. This makes POST more secure and appropriate for handling sensitive or confidential data such as passwords, payments, or file uploads. It also allows sending large amounts of data without the limitations of URL length. Since POST requests are not cached or bookmarked, they are generally used for operations that modify server-side data, such as submitting forms, creating accounts, or updating records.

Example: Used in login, signup, file uploads (secure).

Program 3: Handling GET and POST in Flask

Source Code:

app.py

```
from flask import Flask, request, render_template
app = Flask(__name__)
@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":   # If form submitted
        username = request.form.get("username")
        password = request.form.get("password")
```

```
        return f"<h2>POST Method Used</h2><p>Welcome, {username}!</p>"
    else:   # Default GET request
        return render_template("form.html")
if __name__ == "__main__":
    app.run(debug=True)
```

form.html (inside templates/ folder)

```
<!DOCTYPE html>
<html>
<head>
    <title>Form Example</title>
</head>
<body>
    <h1>Login Form</h1>
    <form method="POST">   <!-- You can change to GET to test -->
        <p>Username: <input type="text" name="username"></p>
        <p>Password: <input type="password" name="password"></p>
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

Output:

- When you open **http://127.0.0.1:5000/** (GET request), you'll see the login form**.**
- If you enter Username: Swathy, Password: 12345 and click Submit**.**
  - Since method is POST:
  - POST Method Used,

    Welcome, Swathy!
- If you change the form method in HTML to "GET" and submit:
  - The URL will show:

    http://127.0.0.1:5000/?username=Swathy&password=12345
  - Flask can read values from request.args instead of request.form.

Table 3.4.3: Comparison table of GET vs POST in Flask forms

Aspect	GET	POST
Where data is sent	Appended to the URL as query parameters	Sent in the request body
Visibility	Data is visible in the URL	Data is hidden (not shown in URL)
Security	Less secure, not good for passwords or sensitive info	More secure, suitable for login, signup, file uploads
Length limitation	Limited (depends on URL length, usually ~2048 characters)	Practically unlimited data
Use cases	Search forms, filters, navigation, bookmarking results	Login forms, registration, payments, uploading files
Flask access method	request.args.get("field")	request.form.get("field")
Can be bookmarked?	Yes, because data is in URL	No, because data is in request body
Browser caching	Cached in history	Not cached

3.4.6 Basic CRUD operations

Flask can be integrated with a database to store, retrieve, update, and delete data efficiently. By using extensions like **Flask-SQLAlchemy**, developers can define database tables as Python classes and perform operations without writing raw SQL. This allows Flask applications to manage data dynamically and support full CRUD functionality. This brings several advantages:

- Simplifies database management.
- Improves security.
- Supports multiple database systems like SQLite, MySQL and PostgreSQL.
- Easily integrates with Flask using the Flask - SQLAlchemy extension.

3.4.6.1 Installing Flask and Flask-SQLAlchemy

To build web applications in Python using Flask and manage databases efficiently with SQLAlchemy, the first step is to install the required packages. Flask provides the core framework for creating web applications, while Flask-SQLAlchemy acts as an ORM (Object Relational Mapper) to simplify database interactions. Installing these packages using pip allows developers to start building dynamic, database-driven web applications quickly and efficiently.

1. Using pip

Open your terminal or command prompt and run:

pip install flask

pip install flask_sqlalchemy

2. Verify Installation

You can check if Flask and SQLAlchemy are installed correctly:

python -m flask --version

python -c "import flask_sqlalchemy; print(flask_sqlalchemy.__version__)"

If both commands run without errors, the installation was successful.

3. Optional: Create a Virtual Environment

It's a good practice to use a virtual environment:

 # Create a virtual environment

python -m venv venv

 # Activate it (Windows)

venv\Scripts\activate

 # Activate it (Mac/Linux)

source venv/bin/activate

 # Install Flask and SQLAlchemy inside the environment

pip install flask flask_sqlalchemy

This keeps your project dependencies isolated and organized.

File Structure

Below is the snapshot of the file structure fig 3.4.6 of the project after its completed.

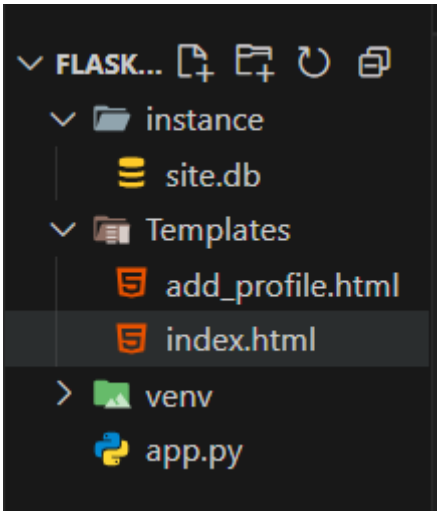


Fig 3.4.6 File structure

3.4.6.2 Creating app.py

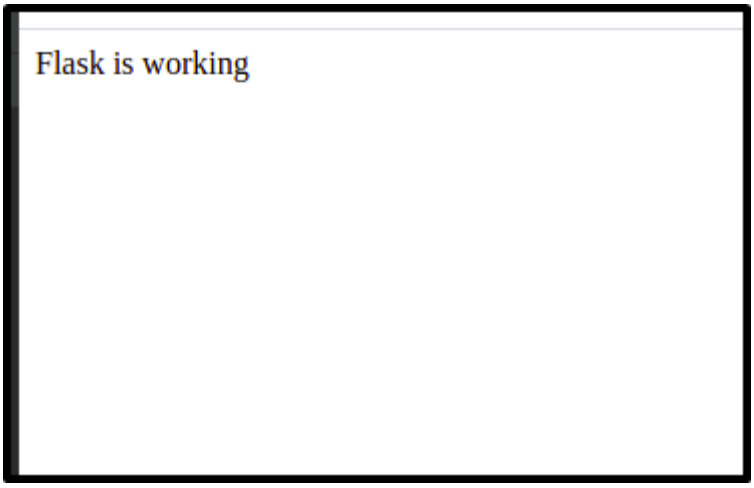
Once the installation is complete, we can create our main Flask application file, app.

py. To verify that everything is installed and running correctly, paste the basic app code below and start the application by running python app.py in the terminal (this code will be updated later).

Example :

```
from flask import Flask

app = Flask(__name__)

'''If everything works fine you will get a

message that Flask is working on the first

page of the application

'''

@app.route('/')

def check():

    return 'Flask is working'

if __name__ == '__main__':

    app.run()
```

Output:



3.4.6.3 Setting Up SQLAlchemy

To create a database we need to import **SQLAlchemy** in **app.py**, set up SQLite configuration, and create a database instance as shown below.

Example :

```
from flask import Flask, render_template, request, redirect
```

```
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

# Configure SQLite database

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False  # Avoids a warning

# Create SQLAlchemy instance

db = SQLAlchemy(app)

# Run the app and create database

if __name__ == '__main__':
    with app.app_context():  # Needed for DB operations
        db.create_all()      # Creates the database and tables
    app.run(debug=True)
```

We set up Flask, connect it to a SQLite database (site.db), and use db.create_all() to create the database when the app runs. The app_context() ensures SQLAlchemy works properly.

3.4.6.4 Creating Model

In SQLAlchemy we use classes to create our database structure. In our application, we will create a Profile table that will be responsible for holding the user's id, first name, last name, and age.

Example:

```
class Profile(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(20), unique=False, nullable=False)
    last_name = db.Column(db.String(20), unique=False, nullable=False)
    age = db.Column(db.Integer, nullable=False)
    # repr method represents how one object of this datatable
    # will look like
    def __repr__(self):
        return f"Name : {self.first_name}, Age: {self.age}"
```

The Table 3.4.2 below explains some of the keywords used in the model class.

Table 3.4.2: Keywords used in the model class

Keywords	Description
Column	used to create a new column in the database table
Integer	An integer data field
primary_key	If set to True for a field ensures that the field can be used to uniquely identify objects of the data table.
String	A string data field. String(<maximum length>)
unique	If set to True it ensures that every data in that field is unique.
nullable	If set to False it ensures that the data in the field cannot be null.
__repr__	Function used to represent objects of the data table.

3.4.6.5 Display data on Index Page

Create an "**index.html**" file in the Templates folder. This will be our root page, it will display all the saved profiles in the database. Jinja templating will dynamically render the data in the HTML file. The delete function will be added later.

Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Index Page</title>
  </head>
  <body>
    <h3>Profiles</h3>
    <a href="/add_data">ADD</a>
    <br>
    <table>
      <thead>
        <th>Id</th>
```

```
        <th>First Name</th>
        <th>Last Name</th>
        <th>Age</th>
        <th>#</th>
      </thead>
      {% for data in profiles %}
      <tbody>
        <td>{{data.id}}</td>
        <td>{{data.first_name}}</td>
        <td>{{data.last_name}}</td>
        <td>{{data.age}}</td>
        <td><a href="/delete/{{data.id}}" type="button">Delete</a></td>
      </tbody>
      {% endfor%}
    </table>
  </body>
</html>
```

We loop through every object in profiles that we pass down to our template in our index function and print all its data in a tabular form. The index function in our **app.py** is updated as follows.

```
@app.route('/add_data')
def add_data():
    return render_template('add_profile.html')
```

3.4.6.6 Creating add_profile.html

In the templates folder, create file "**add_profile.html**", it will render the form that takes user input for the profile details and the form will be linked to a "**/add**" route.

Program 3 : Creating add_profile.html

Source Code:

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <title>Add Profile</title>
  </head>
  <body>
    <h3>Profile form</h3>
    <form action="/add" method="POST">
      <label>First Name</label>
      <input type="text" name="first_name" placeholder="first name...">
      <label>Last Name</label>
      <input type="text" name= "last_name" placeholder="last name...">
      <label>Age</label>
      <input type="number" name="age" placeholder="age..">
      <button type="submit">Add</button>
    </form>
  </body>
</html>
```

3.4.6.7 Creating "/add" route in app.py

This route will receive the form data, create a user object and add it to the database.

Program 4 : Creating "/add" route in app.py

Source Code:

```
@app.route('/add', methods=["POST"])
def profile():
    first_name = request.form.get("first_name")
    last_name = request.form.get("last_name")
    age = request.form.get("age")
    if first_name != '' and last_name != '' and age is not None:
        p = Profile(first_name=first_name, last_name=last_name, age=age)
        db.session.add(p)
        db.session.commit()
        return redirect('/')
    else:
        return redirect('/')
```

To check whether the code is working fine or not, we can run the following command to start the local server.

```
python app.py
```

Now, visit **http://localhost:5000/add_data** and you will be able to see the form.

Output:

3.4.6.8 Function to add data to the database

Create a "**/add**" route in **app.py**. In this route will use request objects to get form data inside the function then create an object of the **Profile** class and store it in our database using database sessions.

Program 5 : Function to add profiles

Source Code:

```
@app.route('/add', methods=["POST"])
def profile():
    first_name = request.form.get("first_name")
    last_name = request.form.get("last_name")
    age = request.form.get("age")
    if first_name != '' and last_name != '' and age is not None:
        p = Profile(first_name=first_name, last_name=last_name, age=age)
        db.session.add(p)
        db.session.commit()
        return redirect('/')
    else:
        return redirect('/')
```

Once the function is executed it redirects us back to the index page of the application.

3.4.6.9 Deleting data from our database

To delete data we have already used an anchor tag in our table and now we will just be associating a function with it.

Program 6 : Deleting data from our database

Source Code:

```
@app.route('/delete/<int:id>')
def erase(id):
    # Deletes the data on the basis of unique id and
    # redirects to home page
    data = Profile.query.get(id)
    db.session.delete(data)
    db.session.commit()
    return redirect('/')
```

The function queries data on the basis of id and then deletes it from our database. The entire code for *app.py, index.html*, and *add-profile.html* is given below (Complete Final Code).

Program 7 : app.py

Source Code:

```
from flask import Flask, request, redirect
from flask.templating import render_template
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.debug = True
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False  # Avoids a warning
db = SQLAlchemy(app)
# Model
class Profile(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(20), unique=False, nullable=False)
```

```
    last_name = db.Column(db.String(20), unique=False, nullable=False)
    age = db.Column(db.Integer, nullable=False)
    # repr method represents how one object of this datatable
    # will look like
    def __repr__(self):
        return f"Name : {self.first_name}, Age: {self.age}"
from flask import Flask, request, redirect
from flask.templating import render_template
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.debug = True
# adding configuration for using a sqlite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
# Creating an SQLAlchemy instance
db = SQLAlchemy(app)
# Models
class Profile(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(20), unique=False, nullable=False)
    last_name = db.Column(db.String(20), unique=False, nullable=False)
    age = db.Column(db.Integer, nullable=False)
    def __repr__(self):
        return f"Name : {self.first_name}, Age: {self.age}"
# function to render index page
@app.route('/')
def index():
    profiles = Profile.query.all()
    return render_template('index.html', profiles=profiles)
@app.route('/add_data')
```

```
def add_data():
    return render_template('add_profile.html')
# function to add profiles
@app.route('/add', methods=["POST"])
def profile():
    first_name = request.form.get("first_name")
    last_name = request.form.get("last_name")
    age = request.form.get("age")
    if first_name != '' and last_name != '' and age is not None:
        p = Profile(first_name=first_name, last_name=last_name, age=age)
        db.session.add(p)
        db.session.commit()
        return redirect('/')
    else:
        return redirect('/')
@app.route('/delete/<int:id>')
def erase(id):
    data = Profile.query.get(id)
    db.session.delete(data)
    db.session.commit()
    return redirect('/')
if __name__ == '__main__':
    with app.app_context():  # Needed for DB operations outside a request
        db.create_all()      # Creates the database and tables
    app.run(debug=True)
```

index.html

```
<!DOCTYPE html>
<html>
  <head>
```

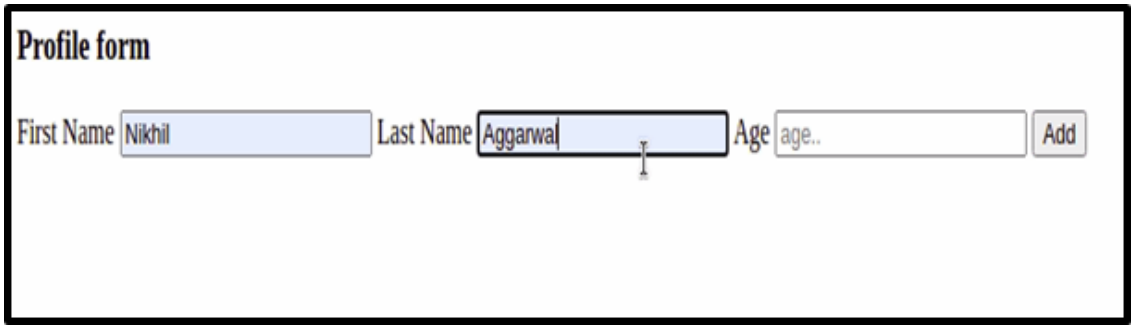
```
    <title>Index Page</title>
  </head>
  <body>
    <h3>Profiles</h3>
    <a href="/add_data">ADD</a>
    <br>
    <table>
      <thead>
        <th>Id</th>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Age</th>
        <th>#</th>
      </thead>
      {% for data in profiles %}
      <tbody>
        <td>{{data.id}}</td>
        <td>{{data.first_name}}</td>
        <td>{{data.last_name}}</td>
        <td>{{data.age}}</td>
        <td><a href="/delete/{{data.id}}" type="button">Delete</a></td>
      </tbody>
      {% endfor%}
    </table>
  </body>
</html>
```

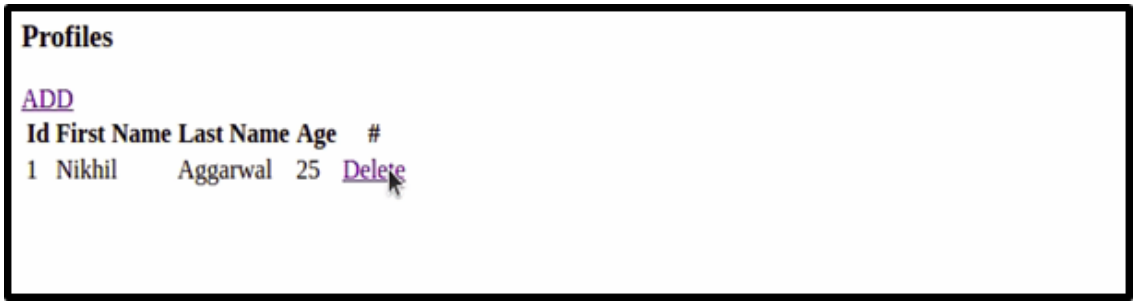
add_profile.html

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <title>Add Profile</title>
  </head>
  <body>
    <h3>Profile form</h3>
    <form action="/add" method="POST">
      <label>First Name</label>
      <input type="text" name="first_name" placeholder="first name...">
      <label>Last Name</label>
      <input type="text" name= "last_name" placeholder="last name...">
      <label>Age</label>
      <input type="number" name="age" placeholder="age..">
      <button type="submit">Add</button>
    </form>
  </body>
</html>
```

Output:





3.4.6.10 CRUD Operations

CRUD stands for Create, Read, Update, Delete. These are the four basic operations performed on data in a database. In Flask, CRUD operations are typically implemented using Flask routes, HTML forms, and Flask-SQLAlchemy for database interaction.

- **Create:** Add new records to the database.
- **Read:** Retrieve and display existing records.
- **Update:** Modify existing records.
- **Delete:** Remove records from the database.

1. CREATE Operation

The Create operation adds new records to the database. In Flask, this is usually done via a form submission, where the submitted data is used to create a new database object and save it using SQLAlchemy.

Syntax:

new_record = Model(field1=value1, field2=value2)

db.session.add(new_record)

db.session.commit()

where,

- *Model(...)* creates a new object with given field values.
- *db.session.add()* stages the object to be inserted.
- *db.session.commit()* saves it permanently to the database.

Example:

```
@app.route('/add', methods=['GET', 'POST'])
def add_user():
    if request.method == 'POST':
        name = request.form['name']
        email = request.form['email']
```

```
        user = User(name=name, email=email)
        db.session.add(user)
        db.session.commit()
        return redirect(url_for('index'))
    return render_template('add.html')
```

Output:

- Visiting /add shows a form.
- After submitting Name: "John Doe", Email: "john@example.com", a new record appears on the homepage list.

2. READ Operation

The Read operation fetches and displays records from the database. In Flask, this is done using queries like Model.query.all() or Model.query.get(id).

Syntax:

records = Model.query.all() ***# fetch all records***

record = Model.query.get(id) ***# fetch single record by ID***

where,

- *query.all()* returns a list of all objects.
- *query.get(id)* fetches a specific object by primary key.
- Data is usually passed to HTML templates using *render_template.*

Example:

```
@app.route('/')
def index():
    users = User.query.all()
    return render_template('index.html', users=users)
```

Output:

Users List:

- John Doe - john@example.com
- Alice Smith - alice@example.com

3. UPDATE Operation

The Update operation modifies existing records. In Flask, you first fetch the record, update its fields, and commit the changes.

Syntax:

record = Model.query.get(id)

record.field1 = new_value

db.session.commit()

where,

- Fetch the object to update using *query.get(id).*
- Change the fields directly.
- Commit the session to save changes in the database.

Example:

```
@app.route('/edit/<int:id>', methods=['GET', 'POST'])
def edit_user(id):
    user = User.query.get_or_404(id)
    if request.method == 'POST':
        user.name = request.form['name']
        user.email = request.form['email']
        db.session.commit()
        return redirect(url_for('index'))
    return render_template('edit.html', user=user)
```

Output:

Visiting */edit/1* shows form pre-filled with *John Doe.*

Changing name to *John D.* updates the homepage list.

4. DELETE Operation

The Delete operation removes records from the database. In Flask, you fetch the record and delete it using SQLAlchemy.

Syntax:

record = Model.query.get(id)

db.session.delete(record)

db.session.commit()

where,

- Fetch the object using its ID.

- Use *db.session.delete()* to stage it for deletion.
- Commit the session to remove it permanently.

Example:

```
@app.route('/delete/<int:id>')
def delete_user(id):
    user = User.query.get_or_404(id)
    db.session.delete(user)
    db.session.commit()
    return redirect(url_for('index'))
```

Output:

Clicking "Delete" next to *John D.* removes him from the users list immediately.

In summary, the unit Web Programming with Flask equips learners with the essential skills required to build dynamic and data-driven web applications. By combining Flask's core concepts such as routing, templates, forms, request handling, and CRUD operations, students gain both theoretical understanding and practical experience in developing interactive websites. This foundation not only prepares them to design scalable applications but also provides a strong base for exploring advanced topics in web development and full-stack programming.

Summarized Overview

The unit Web Programming with Flask provides a comprehensive introduction to developing web applications using Flask, a lightweight and flexible Python framework. It begins with an overview of Flask, its installation, and the structure of a basic application, enabling learners to understand how web servers handle requests and responses. A major focus is placed on *routing*, where students learn to create static routes for fixed pages and dynamic routes that can process variable URL parameters. The unit also introduces *forms*, demonstrating how to design user input fields and capture data. Through handling *GET and POST* requests, learners understand how data is transmitted between client and server, and how to process user input securely and effectively within Flask applications.

The unit then shifts to *templates and Jinja2*, which allow developers to create dynamic and reusable HTML pages. Students explore template inheritance, control struc-

tures, and variable substitution, making web pages more interactive and manageable. Finally, the unit covers *basic CRUD operations (Create, Read, Update, Delete)* with Flask and a database, providing hands-on experience in building applications that can store, retrieve, and modify data persistently. By the end of the unit, learners will not only understand the core features of Flask but also gain the ability to integrate routing, templates, forms, request handling, and database operations into a fully functional, data-driven web application.

Assignments

1. Explain the features of Flask that make it a lightweight web framework. Write the steps to install Flask and create a simple "Hello World" web application.
2. Differentiate between static routing and dynamic routing in Flask with suitable examples. Write the syntax and code to demonstrate dynamic URL routing with parameters.
3. Define forms in Flask and explain the process of handling user input. Write a Flask program to design a form and handle both GET and POST requests with output.
4. What are templates in Flask? Explain template inheritance using Jinja2 with an example. Show how a base template can be extended by a child template.
5. Define CRUD operations in the context of Flask applications. Write a Flask program using SQLAlchemy to demonstrate Create, Read, Update, and Delete operations on a simple database table.

Reference

1. Grinberg, M. (2018). *Flask web development: Developing web applications with Python* (2nd ed.). O'Reilly Media.
2. Shapiro, A. (2020). *Flask framework cookbook: Over 80 practical recipes to help you build web applications with Flask and solve common issues* (2nd ed.). Packt Publishing.

3. Esposito, D., & Ravindran, S. (2021). *Modern Web Development with Flask: Learn to develop powerful and production-ready web applications with Flask* (1st ed.). Packt Publishing.
4. Santos, J. (2023). *Web development with Python and Flask: Build dynamic websites and applications with ease*. Independently published.
5. Ronacher, A., & Pallets Team. (2022). *The Flask Mega-Tutorial: A comprehensive guide to web development with Flask*. Independently published.

Suggested Reading

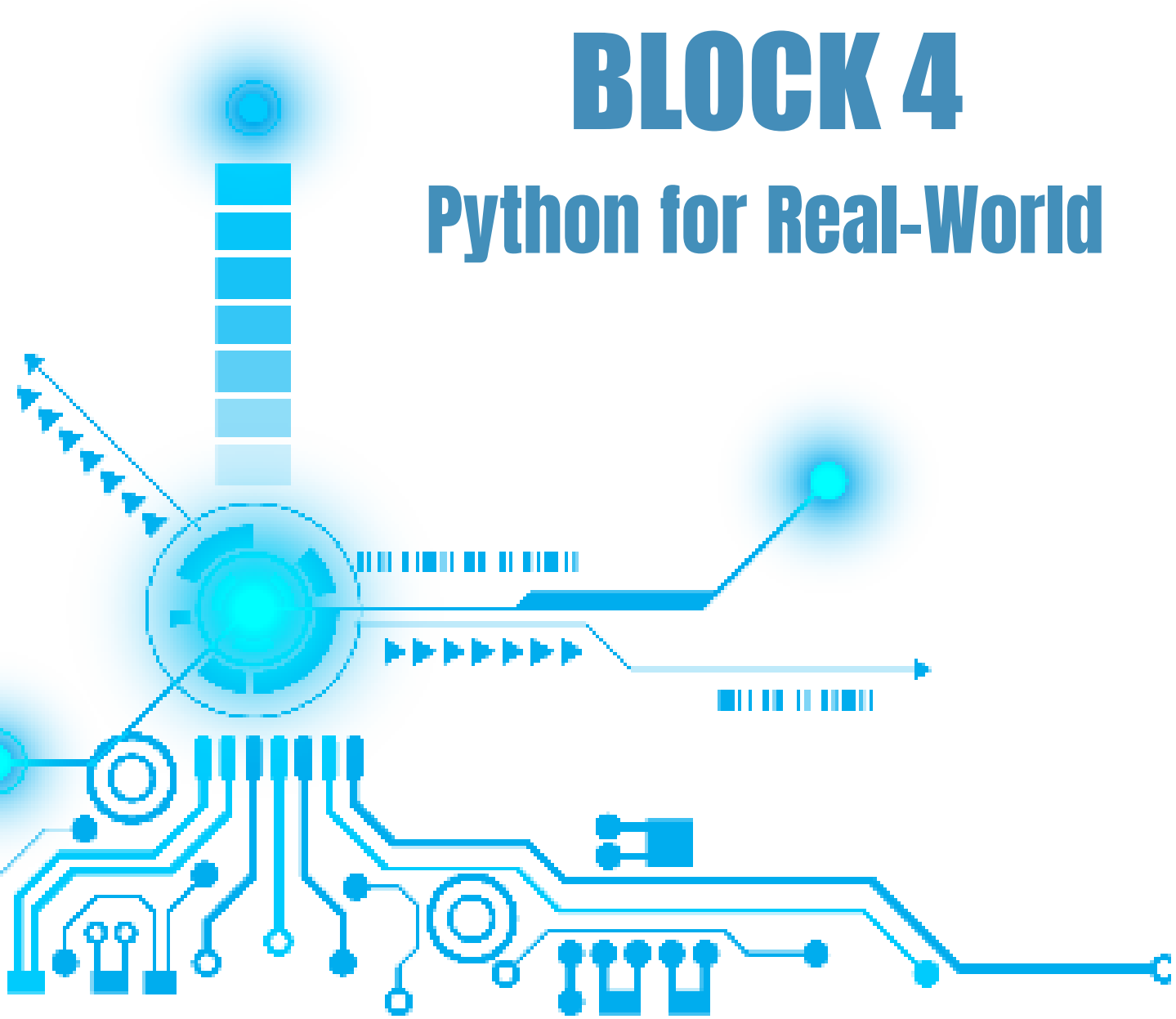
1. Flask Official Documentation – Quickstart & Tutorial https://flask.palletsprojects.com/en/stable/quickstart/
2. Dev.to – Build a Basic CRUD Application with Flask-PyMongo https://dev.to/mongodb/build-a-basic-crud-application-with-flask-pymongo-1kp3
3. Medium – Building RESTful APIs with Flask: CRUD Guide https://medium.com/@mathur.danduprolu/building-restful-apis-with-flask-a-beginners-guide-to-crud-operations-part-2-7-4ff74083a3ee
4. Python in Plain English https://python.plainenglish.io/flask-for-beginners-create-your-first-crud-application-9e2500495b27
5. ThePythonCode.com https://thepythoncode.com/article/front-end-of-crud-flask-app-using-jinja-and-bootstrap

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

BLOCK 4

Python for Real-World



1 UNIT

Working with Databases

Learning Outcomes

After completion of this unit, the learner will be able to:

- define the basic concepts of databases and identify the differences between SQLite and MySQL
- describe the purpose of sqlite3 and mysql.connector modules for database connectivity in Python
- explain the steps for performing CRUD (Create, Read, Update, Delete) operations using Python
- interpret how Pandas can be integrated with databases for data analysis and manipulation

Background

In our daily life, we often deal with data whether it is student details in a school, customer records in a shop, or even our online transactions. Managing such data manually is difficult and time-consuming. This is where databases become useful, as they allow us to store, organize, and retrieve information in an efficient way. In computer applications, we frequently use databases like SQLite and MySQL to handle data.

Python provides built-in and external modules such as sqlite3 and mysql.connector that help us connect to these databases and perform different tasks. These tasks are known as CRUD operations (Create, Read, Update, Delete), which form the foundation of data management. Along with this, modern data analysis tools like Pandas can be integrated with databases to view query results in a neat tabular format, making data easier to analyze.

Databases also support data consistency, security, and multi-user access, which makes them highly reliable in real-world applications. SQLite is lightweight and suitable for small-scale projects, while MySQL is powerful and widely used in enterprise-level applications. By learning how to integrate Python with these databases, learners gain practical skills for building applications that involve data storage, processing, and analysis. These skills are valuable in fields such as software development, data science, and business analytics.

Keywords

Databases, SQLite, MySQL, CRUD operations, Pandas

Discussion

4.1.1 Database Handling with SQLite & MySQL

Databases are used to store and manage information in an organized manner. They allow us to add new data, search for existing data, update values, or remove unwanted records. In Python, two commonly used libraries for working with databases are sqlite3 (for SQLite) and mysql.connector (for MySQL). The basic operations performed on databases are often called CRUD operations, which stand for Create, Read, Update, and Delete. Once we learn these operations, we can also connect databases with Pandas to make data analysis easier.

4.1.2 SQLite with sqlite3

SQLite is a lightweight, serverless, and self-contained database engine. It is widely used for applications that require a simple, fast, and low-overhead database. One of the key features of SQLite is that it is file-based, meaning it stores the entire database in a single file on disk. This makes it very easy to set up and use, as there is no need for a separate server process or complex configurations.

SQLite is a zero-configuration database, which means that developers do not need to worry about installing, configuring, or maintaining a database server. This makes SQLite particularly useful for embedded systems, mobile apps, desktop applications, and quick prototyping. Its lightweight nature also allows it to run in environments with limited resources, such as smartphones, IoT devices, or other systems where a full-fledged server-based database might be too resource-intensive.

Python's built-in sqlite3 module provides an easy-to-use interface for working with SQLite databases. It allows us to perform operations like connecting to a database,

executing SQL commands, and retrieving results directly from within Python. The fact that sqlite3 is a part of the standard Python library means that no installation is required, making it very convenient for small projects or learning purposes.

Example: Creating a Table in SQLite

```
import sqlite3

# Connect to SQLite database (creates the file if it doesn't exist)

conn = sqlite3.connect("student.db")

cursor = conn.cursor()

# Create a table if it doesn't already exist

cursor.execute("""

CREATE TABLE IF NOT EXISTS students (

    id INTEGER PRIMARY KEY,

    name TEXT,

    age INTEGER,

    grade TEXT

)

""")

conn.commit()  # Save the changes

conn.close()   # Close the connection
```

In this example, we begin by connecting to the SQLite database using sqlite3. connect("student.db"). This command creates a new database file named student.db if it does not already exist. The cursor object allows us to execute SQL commands. The CREATE TABLE SQL statement defines a new table called students with columns for id, name, age, and grade. The IF NOT EXISTS condition ensures that the table will only be created if it doesn't already exist. After executing the SQL command, we use conn.commit() to save the changes and conn.close() to safely close the connection.

4.1.2.1 Operations in SQLite using sqlite3

When working with databases, the core operations revolve around the CRUD operations: Create, Read, Update, and Delete. These operations allow us to manage and manipulate the data stored in the database. Below is an explanation of how each operation works in the context of SQLite using Python's sqlite3 module.

1. Create (Insert Data)

The Create operation involves adding new records into a table. In SQLite, we use the

INSERT INTO SQL statement to insert data into a table.

```
import sqlite3

# Connect to SQLite database (creates the file if it doesn't exist)

conn = sqlite3.connect("student.db")

cursor = conn.cursor()

# Insert new student data into the table

cursor.execute("INSERT INTO students (name, age, grade) VALUES (?, ?, ?)", ("Alice", 20, "A"))

# Commit the changes to the database

conn.commit()

# Close the connection

conn.close()
```

In this example, a new record for the student named "Alice" is added to the students table with the age 20 and grade "A". The INSERT INTO statement inserts this data into the table. After executing the command, conn.commit() saves the change to the database, and the connection is closed.

2. Read (Retrieve Data)

The Read operation allows us to fetch data from a database. We use the SELECT SQL statement to retrieve records from a table. The fetchall() function is used to fetch all rows returned by the query.

Example: Reading Data from a Table

```
import sqlite3

# Connect to the SQLite database

conn = sqlite3.connect("student.db")

cursor = conn.cursor()

# Retrieve all records from the students table

cursor.execute("SELECT * FROM students")

rows = cursor.fetchall()

# Display the data

for row in rows:

    print(row)
```

```
# Close the connection
conn.close()
```

The cursor.execute("SELECT * FROM students") query retrieves all rows from the students table, selecting all columns. The cursor.fetchall() function returns a list of tuples, each representing a row with values for id, name, age, and grade. The for loop iterates through the rows, and print(row) displays each row as a tuple.

3. Update (Modify Data)

The Update operation is used to modify existing records in a table. In SQLite, the UPDATE SQL statement is used to change the values in one or more columns of a table.

Example: Updating Data in a Table

```
import sqlite3

# Connect to the SQLite database

conn = sqlite3.connect("student.db")

cursor = conn.cursor()

# Update the grade of a student named 'Alice'

cursor.execute("UPDATE students SET grade = ? WHERE name = ?", ("A+", "Alice"))

# Commit the changes to the database

conn.commit()

# Close the connection

conn.close()
```

The cursor.execute("UPDATE students SET grade = ? WHERE name = ?", ("A+", "Alice")) query updates the grade to "A+" for the row where the name is "Alice". The WHERE clause ensures that only Alice's record is updated, preventing changes to all rows. Finally, conn.commit() saves the changes to the database permanently.

4. Delete (Remove Data)

The Delete operation allows us to remove records from a table. In SQLite, we use the DELETE FROM SQL statement to delete records.

Example: Deleting Data from a Table

```
import sqlite3

# Connect to the SQLite database

conn = sqlite3.connect("student.db")

cursor = conn.cursor()
```

```
# Delete the record for a student named 'Alice'
cursor.execute("DELETE FROM students WHERE name = ?", ("Alice",))
# Commit the changes to the database
conn.commit()
# Close the connection
conn.close()
```

The cursor.execute("DELETE FROM students WHERE name = ?", ("Alice",)) query deletes the record(s) where the name is "Alice". The WHERE clause ensures that only the specific record(s) matching the condition are deleted; without it, all rows in the table would be deleted. Finally, conn.commit() saves the changes, permanently removing the record from the database.

4.1.3 MySQL using mysql.connector

MySQL is one of the most popular relational database management systems used to store and manage data. To connect Python with MySQL, we can use the mysql.connector module, which allows Python programs to interact with MySQL databases. Through this connector, we can perform important database tasks like inserting new records, reading existing data, updating values, and deleting unwanted records. These actions are commonly called CRUD operations like Create, Read, Update, and Delete. Learning how to perform CRUD operations using Python and MySQL helps us build applications that can handle real-world data efficiently.

4.1.3.1 CRUD Operations in MySQL using mysql.connector

1. Connecting to MySQL

Before performing any operation, we must connect Python with MySQL. The mysql.connector module provides an easy way to connect using a username, password, and database name. After connection, a cursor is created to execute SQL commands.

Example:

```
import mysql.connector
# Connect to MySQL
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="your_password",
    database="testdb"
)
```

```
cursor = conn.cursor()
print("Connected to MySQL successfully")
Output:
Connected to MySQL successfully
```

2. Create Operation (Insert Data)

The Create step means adding new records into a table. In MySQL, this is done using the INSERT INTO statement. After inserting, we use commit() to permanently save the changes.

Example:

```
sql = "INSERT INTO students (name, age, grade) VALUES (%s, %s, %s)"
values = ("Alice", 21, "B")
cursor.execute(sql, values)
conn.commit()
print("Record inserted successfully")
Output:
Record inserted successfully
```

3. Read Operation (Fetch Data)

The Read step means retrieving data from the database. We use the SELECT statement to fetch records. The method fetchall() is used to collect all rows and display them.

Example:

```
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

3. Update Operation (Modify Data)

The Update step is used when we want to change the existing data in the table. The UPDATE statement is used along with the WHERE clause to ensure only specific rows are updated.

```
sql = "UPDATE students SET grade = %s WHERE name = %s"
values = ("A+", "Alice")
cursor.execute(sql, values)
```

```
conn.commit()
print("Record updated successfully")
```

4. Delete Operation (Remove Data)

The Delete step removes unwanted records from the table. The DELETE statement is used, and the WHERE clause ensures that only the intended record is deleted.

```
sql = "DELETE FROM students WHERE name = %s"
value = ("Alice",)
cursor.execute(sql, value)
conn.commit()
print("Record deleted successfully")
```

4.1.4 Integration of SQLite with Pandas

SQLite is a lightweight database that stores data in a single file, and Pandas is a Python library widely used for data analysis. By integrating both, we can easily transfer data between a SQLite database and a Pandas DataFrame. This makes it possible to run SQL queries and then analyze the results directly using Pandas functions.

Reading Data from SQLite into Pandas

We can connect to SQLite using the built-in sqlite3 module in Python. Once connected, we can write SQL queries and load their results into a Pandas DataFrame.

Example:

```
import sqlite3
import pandas as pd
# Connect to SQLite database (or create if it doesn't exist)
conn = sqlite3.connect("school.db")
# SQL query
query = "SELECT * FROM students"
# Load result into Pandas DataFrame
df = pd.read_sql(query, conn)
# Display DataFrame
print(df)
# Close connection
conn.close()
```

When we use sqlite3.connect("school.db"), it connects our Python program to the SQLite database file, creating it if it does not already exist. After establishing the connection, we can run SQL queries on the database. The command pd.read_sql(query, conn) is then used to execute the SQL query and directly load the results into a Pandas DataFrame. This DataFrame automatically organizes the data in a tabular format with proper column labels, which makes it much easier to view, analyze, and process compared to handling raw query results.

4.1.5 Integration of MySQL with Pandas

Working with databases becomes easier when we combine MySQL with Pandas in Python. Pandas allows us to load SQL query results directly into a DataFrame, making data analysis simple and efficient. With this integration, we can perform queries on MySQL tables and then use Pandas functions to analyze, clean, and visualize the data.

Reading Data from MySQL into Pandas

To read data, we first connect to the MySQL database using mysql.connector and then run an SQL query. Instead of just printing the results, we pass the query output into a Pandas DataFrame. This helps in handling tabular data more easily.

Example:

```
import mysql.connector
import pandas as pd
# Connect to database
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="password",
    database="school"
)
# Write query
query = "SELECT * FROM students"
# Load query result into a Pandas DataFrame
df = pd.read_sql(query, conn)
# Display DataFrame
print(df)
# Close connection
conn.close()
```

In this program, we use pd.read_sql() to directly execute the SQL query and load the result into a DataFrame called df. Each row of the table is displayed neatly in tabular format with column headings like id, name, age, and grade. This makes it easier to work with the data, as Pandas offers powerful functions for filtering, grouping, and analyzing it.

Summarized Overview

Databases are essential tools for storing and managing information in an organized way. They allow us to insert new data, retrieve existing records, update values, and delete unwanted entries. In Python, two popular libraries for working with databases are sqlite3 (for SQLite) and mysql.connector (for MySQL). These libraries make it possible to perform CRUD operations, which stand for Create, Read, Update, and Delete.

SQLite is a lightweight, file-based database engine that requires no server or configuration. It is widely used in mobile apps, embedded systems, and small projects. Using Python's built-in sqlite3 module, we can connect to an SQLite database, create tables, insert data, fetch results, update records, and delete entries with ease.

MySQL, on the other hand, is a server-based relational database management system suited for larger applications. By using the mysql.connector module in Python, we can connect to a MySQL server, execute queries, and perform CRUD operations in a structured manner.

Both SQLite and MySQL can be integrated with Pandas, a powerful Python library for data analysis. With Pandas, query results can be directly loaded into a DataFrame, making it easier to filter, analyze, and visualize data in tabular form.

Assignments

1. Explain the steps involved in connecting Python with an SQLite database and performing basic CRUD operations using the sqlite3 module.
2. Write a Python program using mysql.connector to create a table in a MySQL database and insert at least three student records. Explain each step.

3. Demonstrate with an example how the UPDATE and DELETE operations are performed in MySQL using mysql.connector.
4. Explain how Pandas can be integrated with SQLite to read data into a DataFrame. Write a simple program to show how query results are displayed in tabular format.
5. Describe the importance of committing changes (commit()) in database operations. What will happen if we forget to use commit() after INSERT or DELETE in SQLite or MySQL?

Reference

1. https://www.geeksforgeeks.org/python/python-sqlite-crud-operations/

Suggested Reading

1. Matthes, Eric. *Python crash course: A hands-on, project-based introduction to programming*. no starch press, 2023.
2. Matthes, Eric. *Python crash course: A hands-on, project-based introduction to programming*. no starch press, 2023.
3. Ramalho, Luciano. *Fluent Python: Clear, concise, and effective programming*. " O'Reilly Media, Inc.", 2015.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT Introduction to Django

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the role of Django's Model-View-Template (MVT) architecture
- describe the functions of Django components
- implement a basic Django application
- describe Django's ORM to perform database operations

Background

Before learning Django, learners should have a foundational understanding of Python programming, including concepts such as variables, control structures (like loops and conditionals), functions, and object-oriented programming (classes and objects). Familiarity with basic HTML is also important, as Django uses HTML templates to render content on web pages. Additionally, learners should have a general awareness of how web applications function, particularly the client-server model and the basics of HTTP requests and responses. This background knowledge ensures that learners can effectively grasp Django's architecture, use its built-in tools, and develop simple web applications with confidence.

Keywords

Web Framework, Model, View, Template, Object-Relational Mapping, Scalability

Discussion

Django is a Python-based web framework designed to simplify the process of building websites and web applications. It offers a range of built-in tools and functionalities that help developers create applications more efficiently and with less effort. By taking care of routine web development tasks such as managing databases, routing URLs, handling forms, and implementing user authentication, Django allows developers to concentrate on the unique aspects of their projects. It follows the "Don't Repeat Yourself" (DRY) principle, which encourages reusable code and enhances maintainability.

4.2.1 Features of Django

Django is a powerful web framework known for several standout features that make it highly suitable for web application development:

1. **High Speed:** Django enables rapid development, allowing ideas to be turned into fully functional applications in a short amount of time.
2. **Comprehensive Tools:** It comes equipped with a wide range of built-in modules and optional packages that support essential features like user authentication, permission control, and content management.
3. **Highly Flexible**: Django can be adapted to various types of projects, whether it's a content management system (CMS), an online store, or an on-demand service platform.
4. **Strong Security:** It includes built-in protection against common web vulnerabilities such as SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and clickjacking.
5. **Easily Scalable:** Applications built with Django can efficiently handle increasing user loads, making it suitable for projects expecting rapid growth or high traffic.

4.2.2 Django Architecture

Django is a high-level web framework built with Python that promotes fast development and a clean, practical coding style. Its core strength lies in a well-organized architecture that divides a web application into separate layers, making it more modular, easier to maintain, and scalable. Rather than strictly following the traditional MVC (Model-View-Controller) pattern, Django adopts its own variation called the MTV architecture, which stands for Model, Template, and View.

In the structure given in Fig 4.2.1, each component has a defined role: the Model is responsible for managing data and the underlying database schema, the Template controls how data is presented to users through the user interface, and the View contains the logic that processes requests and connects the model with the template. This separation of concerns streamlines development and enables developers to create sophisticated web applications efficiently and effectively.

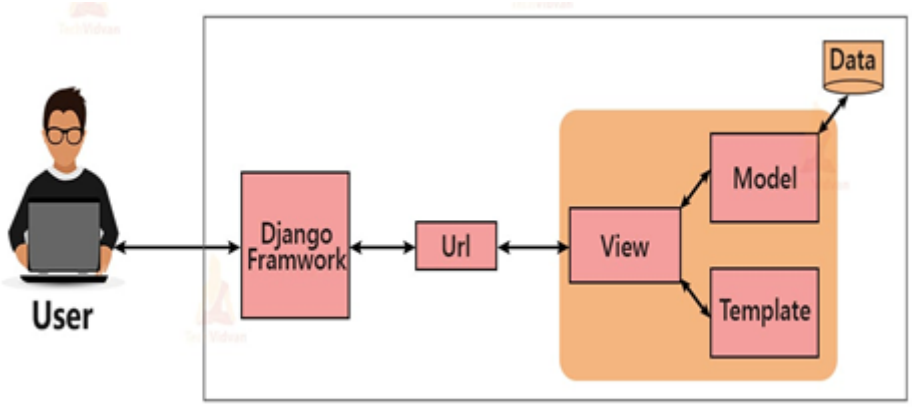


Fig 4.2.1 Django Architecture

4.2.2.1 View

In Django, a view is a function or method that processes incoming HTTP requests. It serves as the bridge between the user interface and the data. Views are responsible for retrieving the necessary information often by interacting with models and passing it to templates to generate the final response displayed to the user. These view functions are typically placed in a file named 'views.py' within a Django application.

A view can be connected to various data sources such as a database, an external API, or internal application logic. While it is commonly executed on the same server that receives the HTTP request, this may not always be the case especially in high-traffic environments. For instance, in a cloud-based web application, different components of the system may be distributed across multiple virtual machines or servers. This means the code that processes the request may run separately from the code that receives it, enhancing scalability and performance under heavy loads.

The Model, on the other hand, is defined as a Python class. Django uses the attributes defined in the model class to automatically create a corresponding database table structure. By using Django's Object-Relational Mapper (ORM), developers can perform database operations such as creating, retrieving, updating, and deleting records in a clean, object-oriented way without writing raw SQL queries.

In the overall request-response cycle, the View collects input data from the user, communicates with the Model to process or retrieve relevant data, and finally sends that data to the Template, which formats it for display to the user.

4.2.2.2 Model

In Django, a model defines the structure of data and outlines how that data is stored and accessed from the database. It essentially acts as the foundation for both data representation and database operations.

To simplify database interactions, Django uses a powerful technique called Object Relational Mapping (ORM). ORM allows developers to interact with the database using Python objects, instead of writing raw SQL queries. This abstraction makes working

with data more intuitive and reduces the need for in-depth knowledge of the underlying database structure.

While SQL is the traditional method for querying and managing databases, it often requires a solid understanding of database schema and relationships. Django's ORM solves this problem by letting developers perform operations like creating, retrieving, updating, and deleting records using Python code.

In a Django application, model definitions are typically placed in a file named models.py, where each model corresponds to a database table, and its attributes map to the table's columns.

4.2.2.3 Template

In Django, a template is a text file that outlines the structure and presentation of a web page or any other output format. Although templates are most commonly used to generate HTML files, they can also define the structure for other file types. The key feature of a template is its use of placeholders, which are dynamically replaced with actual data at runtime.

Django templates are typically written in standard HTML, but they also include special syntax called the Django Template Language (DTL). This allows developers to insert variables, apply logic, and render dynamic content using tags and filters.

For Example:

```
<h1>My Homepage</h1>

<p>My name is {{ firstname }}.</p>
```

In the above code, {{ firstname }} is a placeholder that will be replaced with actual data provided by the view.

When a view processes a request, it can gather or compute the necessary data, organize it into a context, and pass this context to a template. Django's template engine then renders the final HTML page by inserting the context data into the appropriate placeholder blocks. This process plays a crucial role in Django's Model-View-Template (MVT) architecture.

By default, Django templates are stored in a directory named templates within the application folder. This structure helps keep presentation files organized and separate from logic and data handling.

4.2.2.4 Benefits of Django Architecture

One of the key reasons for Django's growing popularity is its architecture, which enables smooth interaction between components without requiring complicated code. This structured approach brings several advantages:

1. **Rapid Development:** Django's architecture allows multiple developers to work concurrently on different parts of the application. Since the framework

clearly separates each component (Model, View, and Template), teams can efficiently collaborate and build features in parallel, speeding up the overall development process.

2. **Loosely Coupled Structure**: The components in Django are designed to be independent of each other. This modularity enhances security, as sensitive data like model definitions and logic remains securely on the server and is not exposed through the front end.
3. **Easy Maintenance and Flexibility:** One of the standout benefits of Django's architecture is its adaptability. Changes made to one component (such as the model or view) typically do not require modifications to others. This separation of concerns allows developers to update, expand, or improve features with minimal disruption, offering greater flexibility in customizing and scaling web applications compared to many other frameworks.

4.2.3 Creating simple Django projects

Step 1 : Download and Install Python

Verify installation:python --version

pip --version

Step 2 : Create a Virtual Environment

Step 3 : Install Django

pip install django

Verify installation: django-admin --version

Step 4 : Create a Django Project

django-admin startproject myproject

cd myproject

Project structure:myproject/

manage.py

myproject/

__init__.py

settings.py

urls.py

asgi.py

wsgi.py

Step 5 : Run the Development Server

python manage.py runserver

Step 6 : Create an Application

python manage.py startapp myapp

Structure : myapp/

admin.py

apps.py

models.py

tests.py

views.py

migrations/

__init__.py

Step 7 : Register the App

Open myproject/settings.py

Add 'myapp' to INSTALLED_APPS:

```
 INSTALLED_APPS = [
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'myapp',  # ← Add this line
        ]
```

Step 8 : Create a Basic View

Edit myproject/urls.py:

```
from django.contrib import admin
from django.urls import path
from myapp.views import home
urlpatterns = [
path('admin/', admin.site.urls),
path('', home),  # Home route
 ]
```

Step 9 : Run the Server Again

```
python manage.py runserver
```

step1: django-admin startproject myproject

```
cd myproject
python manage.py startapp myappnew
```

step2 : myappnew/views.py:

```
from django.http import HttpResponse
def hello(request):
return HttpResponse("Hello, MCA Students welcome to Django program!")
```

step3 : myproject/urls.py:

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', views.hello),
]
```

step 4: Run the server:

```
python manage.py runserver
```

Summarized Overview

The unit provides a comprehensive overview of Django, a Python-based web framework designed to streamline the development of websites and web applications. It highlights Django's key features, such as rapid development, built-in tools for authentication and content management, high flexibility, strong security, and scalability. The unit explains Django's Model-View-Template (MVT) architecture, where the Model manages data and database interactions through Django's ORM, the View handles logic and processes user requests, and the Template controls the presentation layer using Django's template language. Each component is clearly separated, promoting maintainability and modular development. It also outlines the step-by-step procedure for creating a simple Django project from installing Python and Django to setting up a project, creating an application, defining views, and running the development server. Overall, the unit emphasizes Django's efficiency, structural clarity, and a developer-friendly approach to building dynamic, secure, and scalable web applications.

Assignments

1. Explain the architecture of Django. How does the Model-View-Template (MVT) pattern differ from the traditional Model-View-Controller (MVC) pattern?
2. List and describe five key features of Django that make it suitable for modern web application development.
3. Write the step-by-step procedure to create a simple Django project including commands and file structure.
4. Create a basic Django web application that displays a welcome message on the homepage using views and templates. Provide code snippets for views.py, urls.py, and the template file.
5. Discuss the role of Django's ORM. How does it simplify database operations compared to traditional SQL queries?

Reference

1. https://docs.djangoproject.com
2. Holovaty, A., & Kaplan-Moss, J. (2009). *The definitive guide to Django: Web development done right* (2nd ed.). Apress.
3. Syntx, A. (2018). *Mastering Django: Core* (2nd ed.). Packt Publishing.

Suggested Reading

1. Vincent, W. S. (2020). *Django for beginners: Build websites with Python and Django* (3rd ed.). WelcomeToCode.
2. Greenfeld, A. R., & Greenfeld, D. R. (2021). *Two scoops of Django 3.x: Best practices for the Django web framework*. Two Scoops Press.
3. Forcier, J., Bissex, P., & Chun, W. J. (2008). *Python web development with Django*. Addison-Wesley Professional.
4. Django Software Foundation. (n.d.). *Django documentation*.
5. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3 UNIT Working with APIs and JSON

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the basic structure and working principles of REST APIs
- describe how the requests library is used to consume REST APIs in Python
- interpret JSON data returned from APIs and explain how it can be parsed in Python
- discuss the role of Flask in creating simple APIs for web applications
- summarize how operations can be performed through REST API endpoints

Background

Application Programming Interfaces (APIs) play a vital role in modern software development by enabling communication between different applications and systems. REST (Representational State Transfer) APIs, in particular, have become the most widely used standard due to their simplicity, scalability, and reliance on HTTP methods such as GET, POST, PUT, and DELETE. Since APIs commonly exchange data in JSON (JavaScript Object Notation) format, it is essential for developers to understand how to request, receive, and process JSON data efficiently. Python provides powerful libraries, such as the requests library, that make it straightforward to consume APIs and work with JSON responses, allowing developers to integrate external services and data sources into their applications.

In addition to consuming APIs, developers often need to create their own APIs for sharing data or enabling interaction between clients and servers. Flask, a lightweight and flexible Python web framework, is well-suited for building simple RESTful APIs. With Flask, developers can define routes, handle HTTP requests, and return JSON responses with minimal code, making it an ideal tool for prototyping and small-scale applications. This unit focuses on providing learners with a clear understanding of how to consume REST APIs, parse JSON data, and build simple APIs using Flask, laying a strong foundation for real-world projects in web and mobile application development.

Keywords

REST API, JSON, PATCH, GET, Parsing

Discussion

Modern applications rarely work in isolation. Instead, they constantly **communicate with** external systems to fetch, process, and share data. For example: A mobile banking app fetches account balances from the bank's servers. A food delivery app pulls restaurant menus and delivery times in real time. A ride-hailing app connects to GPS and mapping services to calculate routes and fares.

How do these different systems "talk" to each other, even though they are built in different programming languages or hosted on different platforms?

The answer is through **APIs (Application Programming Interfaces)**.

4.3.1 What is an API?

An API defines a set of rules and protocols that allow one software system to interact with another. Think of it as a contract:

If you send a request in a specific format, you will receive a response in a predictable structure.

Imagine a **restaurant**. You, the customer, don't enter the kitchen to cook your food. Instead, you place an order with a **waiter**. The waiter takes your request, communicates with the chef, and brings the dish back.

Here:

- You = Client (application or user)

- Waiter = API
- Kitchen = Server (system providing the data)

APIs make communication **standardized and efficient**, just like a waiter ensures that orders flow smoothly.

4.3.2 REST APIs – A Popular Style

Among various types of APIs, the most widely used today is the REST (Representational State Transfer) API. REST APIs use the HTTP protocol (the same one used by web browsers) for communication.

4.3.2.1 Characteristics of REST APIs

1. Stateless: Each request is independent; the server does not remember previous requests.
2. Client-Server model: Client requests, server responds.
3. Resource-based: Everything (user, product, post, order) is treated as a "resource" that can be identified by a URL.

4.3.2.2 Common HTTP Methods in REST APIs

REST APIs use **HTTP methods** to define the type of action the client wants to perform on the server's resources. Each method has a specific purpose, just like different tools in a toolbox. Let us explore the most common methods in detail.

1. GET – Retrieve Data

The GET method is used when you want to request information from a server without changing anything on it. It is like reading a book from a library – you borrow it, read it, but you do not modify the original book.

Example in real life:

- A weather app using a GET request to fetch today's temperature.
- A shopping site displaying product details when you click on a product.

Key points:

- Does **not** change or update data.
- Can be repeated multiple times safely (idempotent).
- Data may be cached to improve speed.

Python Example:

```
import requests

response = requests.get("https://jsonplaceholder.typicode.com/users/1")

print(response.json())
```

This retrieves user details with ID = 1.

2. POST – Create New Data

The POST method is used when you want to send new data to the server, usually to create a new record. Think of it like filling out a registration form and submitting it — the server receives your details and stores them as a new entry.

Example in real life:

Registering as a new user on a website.

Adding a new blog post or comment.

Uploading a new photo to your profile.

Key points:

Creates **new entries** on the server.

Each POST request may result in a **different outcome** (not idempotent).

Often used with request body (the data being sent).

Python Example:

```
import requests

new_post = {
   "title": "My First Post",
   "body": "This is the content of my post.",
   "userId": 1
}

response = requests.post("https://jsonplaceholder.typicode.com/posts", json=new_post)

print(response.json())
```

This creates a new post with the given title and body.

3. PUT / PATCH – Update Existing Data

The **PUT** and **PATCH** methods are used when you want to **modify existing data** on the server.

- **PUT** → Replaces the **entire record** with new data.
- **PATCH** → Updates **only part of the record** (like editing just one field).

Think of it like updating your profile:

- PUT = replacing your entire profile (name, email, picture, etc.) with new

data.

- PATCH = just updating your profile picture while leaving everything else untouched.

Example in real life:

- Updating your delivery address in an e-commerce app.
- Correcting the spelling of your name in your account.

Key points:

- Changes existing resources.
- PUT is **idempotent** (same request multiple times has the same effect).
- PATCH is more **efficient** when only small updates are needed.

Python Example:

```
import requests

updated_post = {
   "title": "Updated Post Title",
   "body": "Updated content here",
   "userId": 1
}

# Full update
response_put = requests.put("https://jsonplaceholder.typicode.com/posts/1", json=updated_post)
print("PUT Response:", response_put.json())

# Partial update
response_patch = requests.patch("https://jsonplaceholder.typicode.com/posts/1", json={"title": "Only Title Changed"})
print("PATCH Response:", response_patch.json())
```

4. DELETE – Remove Data

The **DELETE method** is used to **remove existing data** from the server. This is like **throwing away an old file** that you no longer need.

Example in real life:

- Deleting a post from social media.

- Removing an item from your shopping cart.
- Cancelling a booking.

Key points:

- Permanently removes the resource (if allowed).
- Idempotent: sending DELETE multiple times results in the same outcome — the resource is gone.
- Must be used carefully, as it changes server data.

Python Example:

```
import requests

response = requests.delete("https://jsonplaceholder.typicode.com/posts/1")

print("Status Code:", response.status_code)
```

If successful, the server confirms deletion with a **status code 200 (OK)** or **204 (No Content)**.

4.3.3 JSON – The Language of Data Exchange

When you fetch data from a REST API, the server usually responds with information in **JSON (JavaScript Object Notation)** format.

4.3.3.1 Why JSON?

- **Lightweight** → Small file size, fast transmission.
- **Human-readable** → Easy to understand for developers.
- **Language-independent** → Works in Python, Java, JavaScript, etc.
- **Hierarchical structure** → Can represent complex data with nesting.

4.3.3.2 Example of JSON

Suppose you request weather data:

```
{
 "city": "Kochi",
 "temperature": 29,
 "condition": "Cloudy",
 "forecast": [
  {"day": "Monday", "temp": 28},
  {"day": "Tuesday", "temp": 30}
```

]

}

This JSON:

- Has **key-value pairs** ("city": "Kochi")
- Can store **numbers** (29), **strings** ("Cloudy"), and even **arrays** (forecast).
- Can also have **nested objects**.

4.3.4 Consuming REST APIs in Python with requests

Python provides the **requests library**, which makes working with APIs easy and intuitive.

4.3.4.1 Sending a GET Request

```
import requests

# Fetch a random joke

response = requests.get("https://official-joke-api.appspot.com/random_joke")
```

Here:

- The **URL** points to the API endpoint.
- requests.get() sends an HTTP GET request.
- The server's reply is stored in the variable response.

4.3.4.2 Checking the Response

```
print(response.status_code)
```

- 200 → Success
- 404 → Resource not found
- 500 → Server error

4.3.5 Parsing JSON Responses

APIs generally send back data in **JSON (JavaScript Object Notation)** format because it is lightweight, easy to read, and supported across almost every modern programming language. When Python receives a response from an API, the raw data is typically in the form of a text string. Although you could treat it as plain text, that would be inconvenient, since you would need to manually extract values by searching through the string. To make this process simpler, Python's requests library provides the .json() method, which automatically converts the JSON text into a **Python dictionary**. This conversion is powerful because it allows developers to work with the response data just like they work with any other dictionary in Python: using keys to directly access values, updating fields, or even looping through nested structures. For example, if an API returns

user details in JSON, you can access data["name"] or data["email"] immediately after calling .json(). This makes API data handling seamless and natural in Python, bridging the gap between web-based data exchange and familiar Python data structures.

```
data = response.json()

print(data)
```

Sample output:

```
{
  "id": 42,
  "type": "general",
  "setup": "Why don't programmers like nature?",
  "punchline": "It has too many bugs."
}
```

Now, access the values just like a Python dictionary:

```
print("Joke ID:", data["id"])

print("Setup:", data["setup"])

print("Punchline:", data["punchline"])
```

4.3.5.1 Example: Real-Time Weather

Let's build a **mini weather app** using the OpenWeatherMap API.

```
import requests

url = "https://api.openweathermap.org/data/2.5/weather"

params = {
    "q": "Kochi",
    "appid": "your_api_key",
    "units": "metric"
}

response = requests.get(url, params=params)

weather_data = response.json()

print("City:", weather_data["name"])

print("Temperature:", weather_data["main"]["temp"], "°C")
```

```
print("Condition:", weather_data["weather"][0]["description"])
```

Here we pass **parameters** (q, appid, units) to the API.

The API responds with JSON containing weather details.

We parse the JSON and extract city name, temperature, and condition.

4.3.5.2 Handling Errors Gracefully

Not all API requests succeed. Some common issues include invalid URLs, missing API keys, or network problems. Good programs handle these gracefully:

```
if response.status_code == 200:
    data = response.json()
    print("Success:", data)
else:
    print("Error:", response.status_code)
```

This ensures your program does not crash unexpectedly.

4.3.5.3 Why APIs and JSON are Important

Interoperability: APIs allow different systems to communicate, regardless of language or platform.

Scalability: Developers can build small apps that rely on powerful external services (maps, payments, weather).

Efficiency: JSON makes data exchange fast and compact.

Career Relevance: Every modern software developer is expected to understand APIs and JSON.

4.3.6 Creating simple APIs with Flask

Creating simple APIs with Flask is an easy and efficient way to build web services that allow applications to communicate with each other. Flask, being a lightweight and flexible Python web framework, provides simple tools to define routes, handle requests, and return data in formats like JSON. With just a few lines of code, developers can create RESTful APIs that support operations such as retrieving, adding, updating, and deleting data. This makes Flask an ideal choice for beginners and professionals who want to quickly develop and test APIs for web and mobile applications.

Step 1: Install Flask

If Flask is not installed, run:

```
pip install flask
```

Step 2: Basic API with Flask

A simple Flask API usually consists of defining routes that respond with JSON.

```
from flask import Flask, jsonify, request
app = Flask(__name__)
# Sample data
users = [
    {"id": 1, "name": "Alice", "age": 25},
    {"id": 2, "name": "Bob", "age": 30}
]
# Home route
@app.route('/')
def home():
    return "Welcome to the Simple Flask API!"
# GET all users
@app.route('/users', methods=['GET'])
def get_users():
    return jsonify(users)
# GET a single user by ID
@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    return jsonify(user) if user else jsonify({"error": "User not found"}), 404
# POST - Add new user
@app.route('/users', methods=['POST'])
def add_user():
    new_user = request.get_json()
    users.append(new_user)
    return jsonify(new_user), 201
# PUT - Update a user
@app.route('/users/<int:user_id>', methods=['PUT'])
```

```
def update_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    if not user:
        return jsonify({"error": "User not found"}), 404
    data = request.get_json()
    user.update(data)
    return jsonify(user)
# DELETE - Remove a user
@app.route('/users/<int:user_id>', methods=['DELETE'])
def delete_user(user_id):
    global users
    users = [u for u in users if u["id"] != user_id]
    return jsonify({"message": "User deleted successfully"})
if __name__ == '__main__':
    app.run(debug=True)
```

Step 3: Run the App

Save the file as app.py and run:

python app.py

The API will run at http://127.0.0.1:5000/

The example API usage demonstrates how different HTTP methods interact with the */users* endpoint to perform various operations. A *GET /users* request retrieves all users, while *GET /users/1* fetches details of the user with ID 1. To add a new user, a *POST /users* request is sent along with a JSON body containing the user's information. Updating an existing user, such as the one with ID 2, is done using *PUT /users/2* with a JSON body that carries the updated details. Finally, the *DELETE /users/1* request removes the user with ID 1 from the list. Together, these operations cover the essential CRUD (Create, Read, Update, Delete) functionality of the API.

Summarized Overview

This unit focuses on developing a strong understanding of how APIs function as a bridge between different applications and services, with special emphasis on REST APIs, which are widely used in web and mobile development. Students will explore how to consume REST APIs using Python's requests library to send and receive data over HTTP. They will also learn how to work with JSON (JavaScript Object Notation), the most common data format used in APIs, by parsing responses, extracting meaningful information, and applying the data within Python programs. This practical knowledge equips learners with the ability to integrate third-party services, such as weather data, social media platforms, or financial information, into their own applications.

In addition to consuming external APIs, the unit introduces the process of creating simple RESTful APIs using Flask, a lightweight and flexible Python framework. Students will gain experience in defining routes, handling HTTP request methods such as GET, POST, PUT, and DELETE, and returning JSON-formatted responses. The unit also emphasizes the role of APIs in supporting CRUD (Create, Read, Update, Delete) operations, enabling efficient client-server communication. By the end of the unit, learners will be able to both integrate data from external services into Python applications and design basic APIs of their own, laying the foundation for more advanced web application development and real-world software projects.

Assignments

1. Write a Python program using the requests library to fetch data from a public REST API (e.g., OpenWeather, JSONPlaceholder). Display specific details from the JSON response, such as weather conditions or user information.
2. Given a JSON response containing a list of students with their names, grades, and subjects, write a Python script to parse the data and print only the names of students who scored above 80 in Mathematics.
3. Develop a simple Flask API to manage a collection of books with fields such as id, title, and author. Implement routes to perform CRUD operations: retrieve all books, retrieve a book by ID, add a new book, update a book, and delete a book.
4. Build a small Python application that consumes data from an external REST API (e.g., COVID-19 statistics, movie database) and provides it through your own Flask API after parsing and restructuring the JSON data.

Reference

1. Grinberg, M. (2018). *Flask web development: Developing web applications with Python* (2nd ed.). O'Reilly Media.
2. Relan, K. (2019). *Building REST APIs with Flask: Harness the power of Python microframeworks to build production-ready REST APIs* (1st ed.). Apress. https://doi.org/10.1007/978-1-4842-5022-8
3. Taneja, S., & Bhatt, H. (2021). *Mastering Python for web: Flask, Django, and Web development tools* (1st ed.). BPB Publications.
4. Guttag, J. V. (2021). *Introduction to computation and programming using Python: With application to computational modeling and understanding data* (3rd ed.). The MIT Press.

Suggested Reading

1. 1. Real Python – *Working with APIs in Python* https://realpython.com/api-integration-in-python
2. W3Schools – *Python JSON* https://www.w3schools.com/python/python_json.asp
3. Flask Official Documentation https://flask.palletsprojects.com
4. GeeksforGeeks – *Flask REST API* https://www.geeksforgeeks.org/flask-rest-api

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT

Project Development

Learning Outcomes

After completion of this unit, the learner will be able to:

- analyze complex Python project requirements and design an effective solution incorporating appropriate libraries, data handling techniques, and interface elements
- evaluate and select suitable GUI or web frameworks to implement a user-friendly and functional project interfaces
- create a fully functional Python mini-project that integrates data processing, visualization, and reporting features to solve a real-world problem
- critically assess the performance and usability of the developed project and refine it based on testing and user feedback to ensure robustness and reliability

Background

Project development in Python is essential for gaining hands-on experience and applying theoretical concepts to real-world problems. It helps learners understand the complete software development lifecycle, from planning and design to implementation and deployment. By using Python libraries, such as Pandas for data handling, Matplotlib for visualization, and Tkinter or Flask for GUI and web interfaces, students can efficiently perform complex tasks. Working on mini-projects enhances problem-solving skills as learners encounter and resolve coding challenges during development. Data handling and analysis become easier, enabling meaningful insights to be derived from datasets. Generating reports programmatically improves the ability to present results clearly and professionally.

Additionally, students learn modular programming, making code more readable, maintainable, and reusable. Building interactive applications through GUI or web interfaces provides practical experience relevant to user-oriented software. Completing mini-projects also strengthens portfolios, boosting confidence for internships, research, or job opportunities. Overall, project development in Python transforms theoretical knowledge into practical skills, preparing learners for larger, real-world projects and professional growth.

Keywords

Python Libraries, GUI, Web Interface, Data Handling, Report Generation, Visualization, Exception Handling, Testing

Discussion

4.4.1 Creating a Mini Project in Python

Python is a powerful and flexible programming language commonly used for a wide range of applications, including web development and data analysis. A great way to learn and strengthen your Python skills is by working on mini projects. These projects allow you to apply theoretical knowledge while gaining practical, hands-on experience in building real-world applications. In this guide, we will walk you through the step-by-step process of developing a Python mini project.

Selecting a Project Idea

The initial step in developing a mini project in Python is to select a concept that captures your interest. This could be anything from a basic game, a calculator, or a weather application. The most important aspect is to choose a project that matches your current skill level while offering opportunities to learn and expand your abilities.

After deciding on a project, divide it into smaller, manageable components or features. For instance, if you plan to develop a simple game, consider the individual elements like player controls, scoring mechanism, and game-over conditions. Breaking the project into smaller parts makes planning, development, and implementation much more organized and achievable.

Preparing Your Development Environment

Before starting your mini project, it is essential to configure your development

environment correctly. This involves installing Python on your system and selecting a suitable integrated development environment (IDE) or code editor.

To install Python, go to the official Python website and download the latest version that matches your operating system. Follow the step-by-step installation guide provided on the site to complete the setup.

Next, select an IDE or code editor to write your programs. Common options are PyCharm, Visual Studio Code, Sublime Text, and Atom. Install the one you prefer and take some time to explore its features.

Writing Your Code

Once you have finalized your project idea and set up your development environment, you can begin coding your mini project. Start by creating a new Python file and giving it a meaningful name.

First, import all the libraries or modules required for your project. For instance, if you are developing a game, you may need to import the pygame library to handle graphics and user interactions.

Then, implement your project's features step by step. Break each feature into smaller functions or classes to maintain clean, organized, and modular code.

Make sure to test your code regularly during development. Frequent testing helps you catch and fix errors or bugs early, ensuring smoother progress and more reliable results.

Testing and Improving Your Python Mini Project

After completing the code for your mini project, the next step is thorough testing. Evaluate each feature individually to ensure everything functions correctly. If any issues arise, apply debugging methods to locate and fix the root cause.

Once initial testing is done, seek feedback from others who can try your project. This could include friends, colleagues, or members of online Python communities. Pay attention to their suggestions and incorporate improvements where needed.

Refine your project further by adding new features or enhancing existing ones based on insights gained during development. This process not only improves your project but also deepens your understanding of Python concepts.

Developing a Python mini project is a great way for beginners to practice coding and gain practical experience. By following this process of selecting a project idea, setting up your environment, writing code, testing, and refining based on feedback, you can build a polished, functional project and enhance your programming skills. Start creating your Python mini project today and apply what you have learned.

4.4.2 Sample python programs

Sample Program 1: Python Program for Word Guessing Game

This tutorial demonstrates how to create a simple Python word guessing game, where

the player attempts to guess a randomly selected word within a fixed number of attempts.

Word Guessing Game Overview

The game allows the user to guess letters in a randomly chosen word. Feedback is provided after each guess, helping the player complete the word or lose the game depending on their inputs.

1. Importing the Random Module

import random

2. Asking for User Name and Greeting

The program requests the user's name using the input() function, storing it in the variable name. A greeting message follows.

```
name = input("What is your name? ")

print("Good Luck !", name)
```

3. Creating a List of Words and Selecting One Randomly

Define a list of possible words for the game. The program randomly selects one word from this list using random.choice().

```
words = ['rainbow', 'computer', 'science', 'programming',
        'python', 'mathematics', 'player', 'condition',
        'reverse', 'water', 'board', 'geeks']

word = random.choice(words)
```

4. Prompting the User to Guess

```
print("Guess the characters")
```

5. Initializing Guesses and Number of Attempts

guesses stores all characters guessed so far.

turns represents the number of attempts allowed, initially set to 12.

```
guesses = ''

turns = 12
```

6. Main Game Loop

The loop continues while the player has remaining turns. The user is prompted to guess characters in each iteration.

6.1 Checking Each Character in the Word

failed counts the number of letters not yet guessed.

A for loop iterates through the word, displaying the letter if guessed or an underscore if not.

```
failed = 0
for char in word:
    if char in guesses:
        print(char, end=" ")
    else:
        print("_")
        failed += 1
```

6.2 Checking for a Win

If failed equals 0, all letters are guessed correctly. The user wins, and the word is displayed. The loop ends.

```
if failed == 0:
    print("You Win")
    print("The word is:", word)
    break
```

6.3 Prompting for the Next Guess

The user is asked to guess a character, which is added to guesses.

```
guess = input("guess a character:")
guesses += guess
```

6.4 Handling Wrong Guesses

If the guessed character is not in the word, decrease turns by 1 and show a message with remaining attempts.

```
if guess not in word:
    turns -= 1
    print("Wrong")
    print("You have", turns, "more guesses")
```

6.5 Checking for a Loss

If turns reaches 0, the game ends with a loss message.

```
if turns == 0:
    print("You Lose")
```

7. Ending the Game

The game concludes either when the user correctly guesses all letters or runs out of attempts.

Sample Code

```
import random
name = input("What is your name? ")
print("Good Luck !", name)
words = ['rainbow', 'computer', 'science', 'programming',
        'python', 'mathematics', 'player', 'condition',
        'reverse', 'water', 'board', 'omega']
word = random.choice(words)
print("Guess the characters")
guesses = ''
turns = 12
while turns > 0:
    failed = 0
    for char in word:
        if char in guesses:
            print(char, end=" ")
        else:
            print("_")
            failed += 1
    if failed == 0:
        print("You Win")
        print("The word is:", word)
        break
    print()
    guess = input("guess a character:")
    guesses += guess
```

```
if guess not in word:
    turns -= 1
    print("Wrong")
    print("You have", turns, "more guesses")
    if turns == 0:
        print("You Lose")
```

Sample Output

```
What is your name? Victor
Good Luck! Victor
Guess the characters

_ _ _ _ _
guess a character: o
o _ _ _ _
guess a character: m
o m _ _ _
guess a character: e
o m e _ _
guess a character: g
o m e g _
guess a character: a
o m e g a
You Win
The word is: omega
```

Sample Program 2: Convert Emoji to Text in Python

In Python, you can convert emojis or emoticons into text using the demoji module. This module allows you to identify, remove, or replace emojis in text strings effectively. To install the demoji module, use the following command:

```
pip install demoji
```

The demoji module requires an initial download of emoji data from the Unicode Consortium's emoji repository, since the list of emojis is regularly updated. The following code snippet can be used to perform this download:

import demoji

demoji.download_codes()

Output

```
C:\Users\Raju>python
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import demoji
>>> demoji.download_codes()
Downloading emoji data ...
... OK (Got response in 0.87 seconds)
Writing emoji data to C:\Users\Raju\.demoji\codes.json ...
... OK
>>>
```

Here is a Python program that demonstrates how to convert emojis in a text to their corresponding descriptions using the demoji module:

```
# First, install demoji if not already installed

# pip install demoji

import demoji

# Download the latest emoji definitions

demoji.download_codes()

# Sample text with emojis

text = "I love Python 🐍 and programming 💻💖!"

# Replace emojis with their text descriptions

converted_text = demoji.replace(text, '')

print("Original Text: ", text)

print("Converted Text: ", converted_text)
```

Explanation:

demoji.download_codes() downloads the latest emoji definitions from the Unicode repository.

demoji.replace(text, '') removes or replaces emojis from the text. You can also replace them with their names by using demoji.findall(text) which gives a dictionary of emojis and their description.

Output

Original Text:

I love Python 🐍 and programming 💻💖!

Converted Text (emojis removed):

I love Python and programming !

If you use demoji.findall(text) instead to get emoji names, the output would look like this:

```
import demoji

demoji.download_codes()

text = "I love Python 🐍 and programming 💻💙!"

emoji_dict = demoji.findall(text)

print(emoji_dict)
```

Output:

```
{'🐍': 'snake', '💻': 'laptop', '💙': 'red heart'}
```

Sample Program 3:Creating a Countdown Timer in Python

In this tutorial, we will learn how to create a countdown timer using Python. The program will prompt the user to enter the duration of the countdown in seconds. The timer will then display the countdown in the format minutes:seconds on the screen. We will utilize Python's time module for this purpose.

Step-by-Step Instructions

This project uses the time module and its sleep() function. Follow these steps to implement a countdown timer:

1. Import the time module with import time.
2. Ask the user to input the countdown duration in seconds.
3. Convert the input to an integer since input() returns a string.
4. Define a function countdown(t) to handle the countdown logic.
5. Use a while loop to continue the countdown until t reaches zero.
6. Inside the loop:
   - ♦ Use divmod(t, 60) to separate the total seconds into minutes and seconds.
   - ♦ Format the time as a string using '{:02d}:{:02d}'.format(mins, secs).
   - ♦ Print the formatted time on the same line with end='\r' to overwrite the previous output.
   - ♦ Pause for 1 second using time.sleep(1).
   - ♦ Reduce t by 1 in each iteration.
7. When the loop ends, display "Stopped!!" to indicate that the countdown has finished.

Python Code: Countdown Timer

```
import time

def countdown(t):
    while t:
        mins, secs = divmod(t, 60)
        timer = '{:02d}:{:02d}'.format(mins, secs)
        print(timer, end='\r')  # Overwrites the previous line each second
        time.sleep(1)
        t -= 1
    print("Stopped!!")
t = input("Enter the time in seconds: ")
countdown(int(t))
```

This program will start a live countdown and alert the user when the timer reaches zero.

4.4.3 Sample python project ideas

1. Student Management System

This project involves creating a console or GUI-based application to manage student information, including names, roll numbers, grades, and attendance. Using object-oriented programming concepts like classes and inheritance, students can learn to structure code efficiently. File handling with CSV or JSON allows storing and retrieving data, while exception handling ensures smooth operation in case of invalid inputs. The project can be extended to generate reports or summaries of student performance, giving practical experience in data handling and report generation.

2. Expense Tracker Application

An expense tracker helps users log daily expenses, categorize them, and visualize spending trends. Using Python libraries like Pandas for data handling and Matplotlib or Seaborn for visualizations, students can analyze monthly or weekly spending patterns. The application can include CRUD operations to manage expenses, and a simple GUI using Tkinter or a web interface with Flask can make it interactive. This project develops skills in data management, visualization, and practical use of Python libraries.

3. Quiz or Trivia Game

A Python-based quiz application can allow users to answer multiple-choice questions fetched from files or APIs. Using file handling and JSON for question storage, students can implement scoring, timers, and difficulty levels. Object-oriented programming concepts like classes for questions and game logic help structure the project. This project emphasizes user interaction, exception handling, and working with structured

data while offering opportunities to generate summary reports for scores.

4. Data Analysis and Visualization Dashboard

This project focuses on analyzing a dataset, such as sales data, employee performance, or COVID-19 statistics. Students use Pandas for data cleaning, filtering, grouping, and aggregation. Matplotlib and Seaborn are used to generate visualizations like line charts, bar charts, histograms, and scatter plots. The project can also include exporting the analysis as a report in CSV, PDF, or Excel format. This hands-on project develops skills in real-world data analysis, visualization, and reporting using Python libraries

After gaining a solid foundation in Python programming, data handling, libraries, and web or GUI development, learners are well-equipped to generate their own project ideas and implement them independently. By applying the concepts, techniques, and skills acquired throughout the course, they can explore real-world problems, experiment with different approaches, and create innovative solutions. This process not only reinforces their understanding but also encourages continuous learning, problem-solving, and the development of practical expertise. By taking initiative and building projects beyond guided exercises, learners can further enhance their confidence, creativity, and readiness for professional applications in Python development.

Summarized Overview

Project development in Python allows learners to apply their programming knowledge to practical, real-world scenarios. It involves planning a mini-project by defining objectives, requirements, and the expected outcome. Python libraries such as NumPy, Pandas, Matplotlib, and Seaborn are often used for data manipulation, analysis, and visualization. Learners can design graphical user interfaces (GUI) using Tkinter or web interfaces using Flask or Django for better user interaction. File handling and database integration help in managing, storing, and retrieving data efficiently. Exception handling ensures the project runs smoothly without unexpected interruptions. The development process also emphasizes writing reusable and modular code through functions, classes, and packages. Reporting features can be implemented to summarize results in text, CSV, or graphical formats. Testing and debugging play a critical role in refining the project and improving functionality. Overall, this approach helps learners gain hands-on experience, fosters problem-solving skills, and prepares them to create independent Python-based projects.

Assignments

1. Explain the process of planning and implementing a mini-project in Python. How do libraries, GUI/web interfaces, data handling, and report generation contribute to effective project development?
2. Compare and contrast using a GUI interface versus a web interface in a Python project. Discuss the advantages and challenges of each approach in terms of user interaction and functionality.
3. Given a dataset, design a Python mini-project that includes data cleaning, visualization, and report generation. Outline the libraries you would use and justify your choices.
4. Describe the role of exception handling and testing in Python project development. How do these practices improve the reliability and maintainability of a project?

Reference

1. Kopec, David. Classic computer science problems in Python. Simon and Schuster, 2019.
2. Lutz, Mark. Programming Python: powerful object-oriented programming. " O'Reilly Media, Inc.", 2010.
3. Summerfield, Mark. Programming in Python 3: a complete introduction to the Python language. Addison-Wesley Professional, 2010.

Suggested Reading

1. Sweigart, Al. The Big Book of Small Python Projects: 81 Easy Practice Programs. No Starch Press, 2021.
2. Cassell, Laura, and Alan Gauld. Python projects. John Wiley & Sons, 2014.
3. Sweigart, Al. Invent your own computer games with python. No Starch Press, 2016.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: …… Reg. No : .....

Name : …….....

Model Question Paper- set-I

End Semester Examination

Discipline Core Course

M25CA03DC - Python Programming

(CBCS - UG)

2025-26 - Admission Onwards

Time: 3 Hours Max Marks: 70

Section A

Answer any ten of the following questions. Each answer should be in two or three valid points. Each question carries 2 marks. (10x2=20)

1. Differentiate between Series and DataFrame in Pandas.
2. What is pip in Python?
3. What is the purpose of the commit( ) method in database operations?
4. Name any two debugging methods.
5. What is an f-string? Give a simple example.
6. Define CRUD operations in the context of Flask applications
7. What is a constructor?
8. When developing a Python project, what is the primary purpose of using a library ?
9. Define a set in Python with an example.
10. What is meant by negative indexing? What does the syntax a[start:end] represent?
11. Write the syntax for opening a file.
12. Explain the role of the 'Template' and the 'View' in Django's MVT architecture.
13. What is a histogram? What type of data is represented using a histogram?

14. Write a Python program to draw a line chart using Matplotlib.

15. What is an iterator?

Section B

Answer any six of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 5 marks. (5X6=30)

16. Describe the concept of loops in Python with a suitable example.

17. What are context managers in Python? Explain their importance in file handling and demonstrate how to create a custom context manager using a class with an example.

18. Describe bar charts and explain how they are used to compare different categories of data using Matplotlib.

19. Write a Python program to create a countdown timer.

20. Explain the built-in methods available in a dictionary in Python.

21. Write a short note on Python directories.

22. A programmer needs to extract specific rows and columns from a large dataset without using loops. Describe how indexing and slicing in NumPy can solve this problem.

23. Explain the architecture of the Django framework with a neat diagram. Explain the role of Model, View, and Template in Django.

24. Describe built-in modules and user-defined modules with suitable examples.

Section C

Answer any two of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 10 marks. (10X2=20)

25. In a data streaming application, values are generated one at a time instead of storing them in memory. Explain how generator functions using ‘yield‘ can be used in such scenarios.

26. Explain filtering of data in Pandas with examples.

27. Explain different HTTP methods used in REST APIs (GET, POST, PUT, PATCH, DELETE) with examples and use cases.

28. Analyze the concept of user-defined modules in Python, explaining their structure, advantages, limitations, and role in modular programming with suitable examples.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: …… Reg. No : .....

Name : …….....

Model Question Paper- set-II

End Semester Examination

Discipline Core Course

M25CA03DC - Python Programming

(CBCS - UG)

2025-26 - Admission Onwards

Time: 3 Hours Max Marks: 70

Section A

Answer any ten of the following questions. Each answer should be in two or three valid points. Each question carries 2 marks. (10x2=20)

1. What is a DataFrame in Pandas? What does the .dtypes property indicate?
2. What is a module in Python?
3. What is the role of mysql.connector in Python?
4. What is Unit Testing with unittest?
5. What is string formatting? Mention any one method used in Python.
6. What is the use of GET and POST methods in Python forms?
7. What is abstraction?
8. What is JSON? Give an example of JSON data.
9. How do you create a list? Give an example.
10. What is the difference between reshape() and resize()?
11. What is a generator function?
12. What is the role of Templates in Django?

13. What is a scatter plot? What does a scatter plot show about two variables?

14. How will you plot a line chart in Python?

15. What is context manager in python?

Section B

Answer any six of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 5 marks. (5X6=30)

16. Explain the different types of operators in Python with examples.

17. Explain file handling in Python. Describe different file modes and illustrate how to open, read, write, and append data to a file with examples.

18. Explain the concepts of a scatter plot and a box plot, and illustrate each with a suitable example.

19. Write a Python program to implement a simple Word Guessing Game.

20. Given a dictionary of employee salaries, write a program to:

- Add a new employee with salary
- Update an existing employee's salary
- Remove an employee
- Display all employee names
- Print the final dictionary

21. Explain about the Unit testing with unittest and pytest.

22. Explain the concept of NumPy arrays in Python. Demonstrate how array operations, indexing and slicing, and broadcasting are performed with suitable examples.

23. Explain the steps to create a simple Django project. Describe the structure of a Django project with its main files.

24. What is a closure? Explain with an example.

Section C

Answer any two of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 10 marks. (10X2=20)

25. What is an iterator? Explain __iter__() and __next__() with an example.

26. Explain in detail the process of creating a Pandas DataFrame. Also discuss the various DataFrame operations with suitable examples.

27. Explain the concept of an API. Describe the characteristics of REST APIs and explain common HTTP methods (GET, POST, PUT/PATCH, DELETE) with examples in Python.

28. Discuss how Python supports code reusability and organization through its library system. Include the roles of modules and packages, methods of accessing them, and how external libraries are made available in a Python environment.

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ 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

SREENARAYANAGURU OPEN UNIVERSITY

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

Python programming

COURSE CODE: M25CA03DC

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-998406-6-9
9 788199 840669