PERIYAR UNIVERSITY

(NAAC 'A++' Grade with CGPA 3.61 (Cycle - 3)
State University - NIRF Rank 56 - State Public University Rank 25
SALEM - 636 011

CENTRE FOR DISTANCE AND ONLINE EDUCATION
(CDOE)

MASTER OF COMPUTER APPLICATIONS
SEMESTER - I

CORE II : PYTHON PROGRAMMING

(Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

M.C.A 2024 admission onwards

CORE – II
Python Programming

Prepared by:

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

Periyar University

Salem - 636011

SYLLABUS

PYTHON PROGRAMMING

Unit I: Introduction : Fundamental ideas of Computer Science - Strings, Assignment and Comments - Numeric Data types and Character sets - Expressions - Loops and Selection Statements: Definite iteration: the for Loop - selection: if and if-else statements - Conditional iteration: the while Loop
Unit – II: Strings and Text Files: Accessing Characters and substrings in strings - Data encryption-Strings and Number systems- String methods - Text - Lists and Dictionaries: Lists - Dictionaries - Design with Functions: A Quick review - Problem Solving with top-Down Design - Design with recursive Functions - Managing a Program's namespace - Higher-Order Functions
Unit III: Design with Classes: Getting inside Objects and Classes - Data-Modeling Examples - Building a New Data Structure - The Two - Dimensional Grid - Structuring Classes with Inheritance and Polymorphism-Graphical User Interfaces-The Behavior of terminal-Based programs and GUI-Based programs - Coding Simple GUI-Based programs - Windows and Window Components - Command Buttons and responding to events
Unit IV: Working with Python Packages: NumPy Library-Ndarray- Basic Operations - Indexing, Slicing and Iteration - Array manipulation - Pandas - The Series - The DataFrame - The Index Objects - Data Vizualization with Matplotlib- The Matplotlib Architecture -Pyplot- The Plotting Window - Adding Elements to the Chart - Line Charts - Bar Charts - Pie charts

Unit V: Django: Installing Django- Building an Application - Project Creation - Designing the Data Schema - Creating an administration site for models - Working with QuerySets and Managers - Retrieving Objects - Building List and Detail Views

TABLE OF CONTENTS

UNIT	TOPICS	PAGE
1	Introduction, Loops and Selection Statements	1
2	Strings and Text Files , Lists and Dictionaries, Design with Functions	64
3	Design with Classes	77
4	Working with Python Packages	132
5	Django	188

PYTHON PROGRAMMING

UNIT 1 – INTRODUCTION
Introduction : Fundamental ideas of Computer Science - Strings, Assignment and Comments - Numeric Data types and Character sets - Expressions - Loops and Selection Statements: Definite iteration: the for Loop - selection: if and if-else statements - Conditional iteration: the while Loop

Introduction : Fundamental Ideas of Computer Science

Section	Topic	Page No.
	UNIT – I	
	Unit Objectives	
Section 1.1	**Introduction: Fundamental Ideas of Computer Science**	**1**
1.1.1	Strings, Assignment and Comments	7
1.1.2	Numeric Data Types and Character Sets	20
1.1.3	Expressions	25
	Let Us Sum Up	28
	Check Your Progress	29
Section 1.2	**Loops and Selection Statements**	**33**
1.2.1	Define Iteration	36
1.2.2	The For Loop	39
1.2.3	Selection : IF and IF – ELSE Statements	41
1.2.4	Conditional Iteration : The While Loop	44
1.2.5	Players in Financial Services Sector	45
	Let Us Sum Up	48
	Check Your Progress	48
1.8	Unit- Summary	55
1.9	Glossary	55
1.10	Self- Assessment Questions	56
1.11	Activities / Exercises / Case Studies	57
1.12	Answers for Check your Progress	59
1.13	References and Suggested Readings	61

UNIT OBJECTIVE

In this unit, students will gain a comprehensive understanding of Abstract Data Types (ADTs), focusing on their definition, implementation, and significance in data structure design. They will learn to work with the Date ADT and Bag ADT, implementing and performing basic operations on these structures using Python. The unit will cover arrays extensively, including Python lists, two-dimensional arrays, and the Matrix ADT, equipping students with the skills to manipulate these data structures effectively. Additionally, students will explore sets and maps, understanding their properties and performing operations using Python's set and dictionary data structures. The concept of multi-dimensional arrays will also be addressed, highlighting their applications and implementation. Finally, students will learn to use iterators for traversing custom data structures, enhancing their ability to handle complex data manipulations.

SECTION 1.1: INTRODUCTION

Python is a high-level, interpreted programming language known for its simplicity and readability, making it an excellent choice for beginners and experienced programmers alike. Created by Guido van Rossum and first released in 1991, Python emphasizes code readability and allows programmers to express concepts in fewer lines of code compared to languages like C++ or Java. Its versatile nature makes it suitable for a wide range of applications, including web development, data analysis, artificial intelligence, scientific computing, and automation.

Python's syntax is clean and easy to learn, with a strong emphasis on readability and the use of whitespace to define code blocks rather than braces or keywords. This design choice reduces the potential for errors and makes the code more visually appealing. The language supports multiple programming paradigms, including procedural, object-oriented, and functional programming, providing flexibility for developers to choose the best approach for their projects.

One of Python's significant strengths is its extensive standard library, which offers modules and packages for virtually every task, from file I/O and system calls to web browsing and XML parsing. Additionally, the Python Package Index (PyPI) hosts thousands of third-party packages that extend Python's capabilities, making it a powerful tool for tackling a variety of challenges.

Python's popularity is also bolstered by its active and supportive community, which contributes to a wealth of tutorials, documentation, and forums. This community-driven support network helps newcomers quickly learn and troubleshoot issues, fostering an environment of continuous learning and improvement. Python's simplicity, readability, and versatility make it an ideal programming language for beginners and professionals alike.

1.1.1 – STRINGS ,ASSIGNMENTS AND COMMENTS

A **string** in is a sequence of characters enclosed in quotes. treats anything inside single ('...') or double ("...") quotes as a string. Strings are one of the most commonly used data types in for working with textual data, and they come with a rich set of built-in functions to manipulate and access the data they hold.

Key Concepts of Strings in

1. Defining Strings

A string can be defined using single, double, or triple quotes:

- **Single quotes**: 'Hello'
- **Double quotes**: "Hello"
- **Triple quotes**: '''Hello''' or """Hello""" (useful for multi-line strings)

```
# Examples
string1 = 'Hello'
string2 = "World"
string3 = '''This is a multi-line string
which spans over multiple lines.'''
```

2. String Immutability

Strings in are **immutable**, meaning once a string is created, it cannot be changed. Any operation that alters a string results in the creation of a new string.

```
s = "Hello"
s[0] = 'h'  # This will raise an error as strings cannot be changed directly.
```

3. Accessing Characters in a String

Strings are treated as arrays of characters, where each character has an index. The index starts from 0 for the first character, and negative indexing can be used to access characters from the end of the string.

```
s = ""
print(s[0])    # Output: P
print(s[-1])   # Output: n (last character)
```

4. Slicing Strings

Extract a portion of a string using slicing. The slice operation is done using string[start:end], where start is inclusive, and end is exclusive.

```
s = "Programming"
print(s[0:6])    # Output:  (characters from index 0 to 5)
print(s[6:])     # Output: Programming (characters from index 6 to the end)
print(s[:6])     # Output:  (characters from the start to index 5)
```

5. Concatenation

Strings can be combined using the **+ operator**. This is called string concatenation.

```
s1 = "Hello"
s2 = "World"
result = s1 + " " + s2   # Output: Hello World
```

6. Repetition

Strings can be repeated using the * **operator**.

```
s = "Hello"
result = s * 3  # Output: HelloHelloHello
```

7. String Methods

Provides numerous built-in methods for working with strings. Here are a few commonly used string methods:

- **len()**: Returns the length of the string.

```
s = ""
print(len(s))  # Output: 6
```

- **lower()**: Converts all characters in the string to lowercase.

```
s = "Hello"
print(s.lower())  # Output: hello
```

- **upper()**: Converts all characters in the string to uppercase.

```
s = "Hello"
print(s.upper())  # Output: HELLO
```

- **replace(old, new)**: Replaces all occurrences of a substring with a new substring.

```
s = "Hello World"
print(s.replace("World", ""))  # Output: Hello
```

- **strip()**: Removes leading and trailing whitespace.

```
s = "  Hello  "
print(s.strip())  # Output: Hello
```

8. String Formatting

supports various ways to format strings, making it easy to insert variables or expressions into strings:

- **Using format()**:

```
name = "Alice"
age = 25
sentence = "My name is {} and I am {} years old.".format(name, age)
print(sentence)  # Output: My name is Alice and I am 25 years old.
```

- **Using f-strings** (introduced in Python 3.6+):

```
name = "Alice"
age = 25
```

```
sentence = f"My name is {name} and I am {age} years old."
print(sentence)  # Output: My name is Alice and I am 25 years old.
```

9. Escape Characters

To include special characters like quotes or newline characters within a string. Escape characters are used for this purpose:

- \n: Newline
- \t: Tab
- \\: Backslash
- \': Single quote
- \": Double quote

```
s = "This is a \"\" tutorial\nIt is very helpful!"
print(s)
# Output:
# This is a "" tutorial
# It is very helpful!
```

10. Iterating Through a String

Iterate through each character in a string using a for loop.

```
s = ""
for char in s:
    print(char)
```

String Operations

String allows various operations to be performed:

1. Membership Operators

- **in**: Checks if a substring exists within the string.
- **not in**: Checks if a substring does not exist within the string.

```
s = " Programming"
print("" in s)      # Output: True
print("Java" not in s)   # Output: True
```

2. Comparison Operators

Strings can be compared using comparison operators such as ==, !=, <, >, etc.

```
s1 = "abc"
s2 = "def"
print(s1 == s2)   # Output: False
print(s1 < s2)    # Output: True (lexicographically 'abc' < 'def')
```

3. Slicing and Indexing

Strings can be sliced using indices. Negative indices can also be used to refer to the position from the end.

```
s = ""
print(s[1:4])    # Output: yth (substring from index 1 to 3)
print(s[-3:])    # Output: hon (last three characters)
```

4. Reversing a String

Reverse a string by using slicing with a step value of -1.

```
s = ""
print(s[::-1])  # Output: nohtyP
```

STRING OPERATIONS

- ✓ **Indexing:** Accessing individual characters in the string using [].
- ✓ **Slicing:** Extracting parts of the string using : in the subscript.
- ✓ **Concatenation:** Combining strings using the + operator.
- ✓ **Length:** Getting the length of the string using len().
- ✓ **Case Conversion:** Converting the string to uppercase, lowercase, and title case.
- ✓ **Substring Checking:** Using "substring" in string to check if a substring exists.
- ✓ **Splitting:** Splitting the string into a list of words.
- ✓ **Joining:** Joining a list back into a single string using " ".join().
- ✓ **Replacement:** Replacing a substring with another using replace().
- ✓ **Stripping:** Removing leading and trailing whitespaces using strip().

PROGRAM

```
# Defining a string
text = "Hello, Python Programming!"
# 1. Accessing individual characters using indexing
first_char = text[0]        # First character
fifth_char = text[4]        # Fifth character
last_char = text[-1]        # Last character
# 2. Slicing to get substrings
substring1 = text[7:13]     # Extracting "Python"
substring2 = text[:5]       # Extracting "Hello"
substring3 = text[7:]       # Extracting from "Python" to end
substring4 = text[-12:-1]   # Extracting "Programmin"
# 3. String concatenation
new_text = "Learning " + text[:6] + "is fun!"  # "Learning Hello is fun!"
# 4. String length
text_length = len(text)
# 5. Case conversion
uppercase_text = text.upper()   # Convert to uppercase
lowercase_text = text.lower()   # Convert to lowercase
titlecase_text = text.title()   # Convert to title case
# 6. Checking for substrings
contains_python = "Python" in text  # Check if "Python" is in text
# 7. Splitting the string
split_text = text.split()  # Splits the string into words
# 8. Joining a list into a string
joined_text = " ".join(split_text)  # Joins the list back to string
# 9. String replacement
replaced_text = text.replace("Python", "Java")  # Replaces "Python" with "Java"
# 10. Stripping whitespace
whitespace_str = "   Hello, World!   "
stripped_text = whitespace_str.strip()   # Removes leading and trailing
whitespaces
print("Original text:", text)
print("First character:", first_char)
print("Fifth character:", fifth_char)
print("Last character:", last_char)
print("Substring 1 (Python):", substring1)
print("Substring 2 (Hello):", substring2)
print("Substring 3 (from Python onwards):", substring3)
print("Substring 4 (Programmin):", substring4)
print("New concatenated text:", new_text)
print("Length of text:", text_length)
print("Uppercase text:", uppercase_text)
print("Lowercase text:", lowercase_text)
print("Titlecase text:", titlecase_text)
print("Contains 'Python':", contains_python)
print("Split text:", split_text)
print("Joined text:", joined_text)
print("Replaced text (Python -> Java):", replaced_text)
print("Stripped text:", stripped_text)
```

OUTPUT

```
Original text: Hello, Python Programming!
First character: H
Fifth character: o
Last character: !
Substring 1 (Python): Python
Substring 2 (Hello): Hello
Substring 3 (from Python onwards): Python Programming!
Substring 4 (Programmin): Programmin
New concatenated text: Learning Hello is fun!
Length of text: 26
Uppercase text: HELLO, PYTHON PROGRAMMING!
Lowercase text: hello, python programming!
Titlecase text: Hello, Python Programming!
Contains 'Python': True
Split text: ['Hello,', 'Python', 'Programming!']
Joined text: Hello, Python Programming!
Replaced text (Python -> Java): Hello, Java Programming!
Stripped text: Hello, World!
```

Python Operators

Operators are the first tools for processing variables and values. Python Operators can be grouped under these types:

- Arithmetic Operators
- Assignment Operators
- Membership Operators
- Logical Operators
- Relational Operators
- Identity Operators
- Bitwise Operators

Arithmetic Operators

Mathematical operations on numeric values can be performed by Arithmetic Operators.

+ - * / // % **

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- // (Floor Division): Returns the division outcome without the decimal part.
- % (Modulus): Returns the remainder from a division.
- ** (Exponent)

Operator Precedence

Control flow takes any expression left to right by default. However, there are predefined priority sequences for Python Operators, known as Operator Precedence. Operator priority sequence from highest to lowest is below:

- ()
- **
- * / // %
- + -

Note: If you encounter two operators from the same priority tier they are simply processed in Left to Right order

Example:

```
>>> 2+2*3
8
>>> (2+2)*3
12
```

ASSIGNMENT OPERATORS

Assignment Operators are used to assigning a variable reference for a value. The working of the assignment is from the right-hand side to the left-hand side.

L.H.S ← R.H.S

Example:

```
>>> x = 1
>>> x = y # Name Error
```

```
>>> y = x # Correct assignment
>>> y
1
```

Membership Operators

Membership Operators are used to checking the containment of an item in a collection. There are two membership operators for contains check: in and not in.

Example:

item in the collection

```
>>> 'h' in "Hello"
False
>>> 'h' in "hello
True
>>> 1 in [2, 40, 100]
False
```

not in gives us False wherever in gives True and vice-versa.

Example:

```
>>> 1 not in [2, 40, 100]
True
```

SAMPLE PROGRAM FOR OPERATORS

```
# Variables for demonstration
a = 15
b = 4
x = [1, 2, 3, 4, 5]
y = [1, 2, 3, 4, 5]
z = [6, 7, 8]
# --------------------- Arithmetic Operators ---------------------
print("Arithmetic Operators:")
print(f"a + b = {a + b}")  # Addition
print(f"a - b = {a - b}")  # Subtraction
print(f"a * b = {a * b}")  # Multiplication
print(f"a / b = {a / b}")  # Division
print(f"a % b = {a % b}")  # Modulus
print(f"a ** b = {a ** b}")  # Exponentiation
print(f"a // b = {a // b}")  # Floor Division
print("\n")
# --------------------- Assignment Operators ---------------------
print("Assignment Operators:")
```

```
c = 10
print(f"Initial value of c: {c}")
c += 2
print(f"c += 2 => {c}")
c -= 4
print(f"c -= 4 => {c}")
c *= 3
print(f"c *= 3 => {c}")
c /= 2
print(f"c /= 2 => {c}")
c %= 4
print(f"c %= 4 => {c}")
c **= 2
print(f"c **= 2 => {c}")
c //= 3
print(f"c //= 3 => {c}")
print("\n")
# --------------------- Membership Operators ---------------------
print("Membership Operators:")
print(f"3 in x: {3 in x}")  # True because 3 is in the list `x`
print(f"7 not in x: {7 not in x}")  # True because 7 is not in the list `x`
print("\n")
# --------------------- Logical Operators ---------------------
print("Logical Operators:")
print(f"(a > b) and (a > 10): {(a > b) and (a > 10)}")  # True if both conditions
are True
print(f"(a < b) or (a > 10): {(a < b) or (a > 10)}")  # True if any one condition is
True
print(f"not(a == b): {not(a == b)}")  # True if a is not equal to b
print("\n")
# --------------------- Relational (Comparison) Operators ---------------------
print("Relational (Comparison) Operators:")
print(f"a == b: {a == b}")  # False because 15 is not equal to 4
print(f"a != b: {a != b}")  # True because 15 is not equal to 4
print(f"a > b: {a > b}")  # True because 15 is greater than 4
print(f"a < b: {a < b}")  # False because 15 is not less than 4
print(f"a >= b: {a >= b}")  # True because 15 is greater than or equal to 4
print(f"a <= b: {a <= b}")  # False because 15 is not less than or equal to 4
print("\n")
# --------------------- Identity Operators ---------------------
print("Identity Operators:")
print(f"x is y: {x is y}")  # True because x and y are the same objects
print(f"x is not z: {x is not z}")  # True because x and z are different objects
print("\n")
```

```
# -------------------- Bitwise Operators --------------------
print("Bitwise Operators:")
print(f"a & b = {a & b}")  # Bitwise AND
print(f"a | b = {a | b}")  # Bitwise OR
print(f"a ^ b = {a ^ b}")  # Bitwise XOR
print(f"~a = {~a}")  # Bitwise NOT
print(f"a << 1 = {a << 1}")  # Bitwise left shift
print(f"a >> 1 = {a >> 1}")  # Bitwise right shift
```

OUTPUT

```
Arithmetic Operators:
a + b = 19
a - b = 11
a * b = 60
a / b = 3.75
a % b = 3
a ** b = 50625
a // b = 3
Assignment Operators:
Initial value of c: 10
c += 2 => 12
c -= 4 => 8
c *= 3 => 24
c /= 2 => 12.0
c %= 4 => 0.0
c **= 2 => 0.0
c //= 3 => 0.0
Membership Operators:
3 in x: True
7 not in x: True
Logical Operators:
(a > b) and (a > 10): True
(a < b) or (a > 10): True
not(a == b): True
Relational (Comparison) Operators:
a == b: False
a != b: True
a > b: True
a < b: False
a >= b: True
a <= b: False
Identity Operators:
x is y: True
x is not z: True
Bitwise Operators:
a & b = 4
a | b = 15
a ^ b = 11
~a = -16
a << 1 = 30
```

a >> 1 = 7

Python Comments

In Python, **comments** are lines of code that are not executed. They are used to explain and document the code, making it easier to understand for others or for future reference. Comments are essential for writing clean, readable, and maintainable code.

There are three types of comments in Python:

1. **Single-line comments**
2. **Multi-line comments**
3. **Docstrings**

1. Single-line Comments

A single-line comment starts with the hash symbol #. Everything on the line after the # is ignored by the Python interpreter.

Example:

```
# This is a single-line comment
x = 5  # This is also a comment after a statement
print(x)  # Output will be 5
```

- In the above example, the comment describes what the line of code does.

2. Multi-line Comments

Python does not have a built-in syntax for multi-line comments, but you can use multiple single-line comments or triple quotes (''' or """) to create block comments.

Option 1: Using multiple single-line comments

```
# This is a multi-line comment
# explaining that the following
# code prints a greeting message.
```

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

Option 2: Using triple quotes

Although technically not comments, you can use triple quotes for multi-line comments. Python treats text inside triple quotes as a string, but if not assigned to a variable, it gets ignored by the interpreter.

```
'''
This is a multi-line comment.
It can span multiple lines.
'''
print("Hello, world!")
```

3. Docstrings (Documentation Strings)

A **docstring** is a special kind of comment used to describe a function, class, or module. Docstrings are written using triple quotes (''' or """) and appear right after the definition of the function, class, or module.

- Unlike regular comments, **docstrings** are not ignored by the interpreter. They can be accessed using the __doc__ attribute.

Example:

```
def greet():
    """
    This function prints a greeting message.
    """
    print("Hello, world!")

# Accessing the docstring
print(greet.__doc__)
```

Output:

```
This function prints a greeting message.
```

- Docstrings are used to provide documentation for code, which can be displayed by built-in functions like help().

Example with help ():

```
def add(a, b):
    """
    This function returns the sum of two numbers.
     Parameters:
    a (int): First number
    b (int): Second number
    Returns:
    int: The sum of a and b
    """
    return a + b
help(add)
```

Output:

```
Help on function add in module __main__:
add(a, b)
    This function returns the sum of two numbers.
    Parameters:
    a (int): First number
    b (int): Second number
    Returns:
    int: The sum of a and b
```

1.1.2 – Numeric Datatypes and Character Sets

In Python, numeric data types are used to represent numbers. Python provides several built-in numeric types, each of which is used for different purposes. The primary numeric data types in Python are:

1. **int (Integer)**:

   ✓ Represents whole numbers, both positive and negative, without any decimal points.

   Example: -5, 0, 42

```
x = 10
y = -3
```

2. **float (Floating-point number)**:

   ✓ Represents real numbers that have a fractional part, indicated by a decimal point.

Example: 3.14, -0.001, 2.0

```
pi = 3.14159
negative float = -0.5
```

3. **complex (Complex number)**:

   ✓ Represents complex numbers, which have a real part and an imaginary part.

   ✓ The imaginary part is denoted by the letter j in Python.

Example: 1+2j, -3.5+4.2j

```
python
z = 2 + 3j
complex number = -1.5 + 2j
```

Converting Between Numeric Types

Python provides functions to convert between different numeric types:

- ✓ int () converts a number or a string to an integer.
- ✓ float () converts a number or a string to a floating-point number.
- ✓ complex () converts a number or a string to a complex number.

Python Code: Numeric Data Types

```
# --------------------- Numeric Data Types ---------------------
# Integer (int) - Whole numbers
integer_num = 42
print(f"Integer: {integer_num}")  # Output: Integer: 42
# Floating-point (float) - Numbers with decimal points
float_num = 3.14159
print(f"Float: {float_num}")  # Output: Float: 3.14159
# Complex number (complex) - Numbers with real and imaginary parts
complex_num = 2 + 3j
```

```
print(f"Complex Number: {complex_num}")  # Output: Complex Number:
(2+3j)
# -------------------- Arithmetic Operations -------------------
# Addition
add_result = integer_num + float_num
print(f"Addition of integer and float: {add_result}")  # Output: Addition of
integer and float: 45.14159
# Subtraction
sub_result = float_num - integer_num
print(f"Subtraction of integer from float: {sub_result}")  # Output: Subtraction of
integer from float: -38.85841
# Multiplication
mul_result = integer_num * float_num
print(f"Multiplication of integer and float: {mul_result}")  # Output: Multiplication
of integer and float: 131.8818
# Division
div_result = float_num / integer_num
print(f"Division of float by integer: {div_result}")  # Output: Division of float by
integer: 0.07478095238095238
# Complex Number Operations
# Real part
real_part = complex_num.real
print(f"Real part of complex number: {real_part}")  # Output: Real part of
complex number: 2.0
# Imaginary part
imaginary_part = complex_num.imag
print(f"Imaginary part of complex number: {imaginary_part}")  # Output:
Imaginary part of complex number: 3.0
# Complex number conjugate
complex_conjugate = complex_num.conjugate()
print(f"Complex conjugate: {complex_conjugate}")  # Output: Complex
conjugate: (2-3j)
# Complex number magnitude
complex_magnitude = abs(complex_num)
print(f"Magnitude of complex number: {complex_magnitude}")  # Output:
Magnitude of complex number: 3.605551275463989
# -------------------- Type Conversion --------------------
# Convert float to integer
float_to_int = int(float_num)
print(f"Float to integer conversion: {float_to_int}")  # Output: Float to integer
conversion: 3
# Convert integer to float
int_to_float = float(integer_num)
```

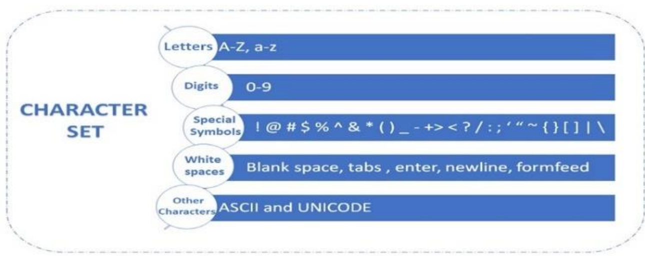
```
print(f"Integer to float conversion: {int_to_float}")  # Output: Integer to float conversion: 42.0
# Convert complex to real part (only the real part can be used)
complex_to_real = complex_num.real
print(f"Complex to real part conversion: {complex_to_real}")  # Output: Complex to real part conversion: 2.0
```

OUTPUT

```
Integer: 42
Float: 3.14159
Complex Number: (2+3j)
Addition of integer and float: 45.14159
Subtraction of integer from float: -38.85841
Multiplication of integer and float: 131.8818
Division of float by integer: 0.07478095238095238
Real part of complex number: 2.0
Imaginary part of complex number: 3.0
Complex conjugate: (2-3j)
Magnitude of complex number: 3.605551275463989
Float to integer conversion: 3
Integer to float conversion: 42.0
Complex to real part conversion: 2.0
```

CHARACTER SET

Python Character Set Character set is the set of valid characters that a language can recognize. A character represents any letter, digit or any other symbol. Python has the following character sets: Letters – A to Z, a to z Digits – 0 to 9 Special Symbols - + - * / etc. Whitespaces – Blank Space, tab, carriage return, newline, form feed Other characters – Python can process all ASCII and Unicode characters as part of data or literals.



Python Code: Character Set Handling

```
# -------------------- Character Set Handling --------------------
# Define a string with various characters
original_string = "Hello, World! 😊"
```

```
print("Original String:")
print(original_string)  # Output: Hello, World! 😊
# Encode the string to bytes using UTF-8 encoding
encoded_bytes = original_string.encode('utf-8')
print("\nEncoded Bytes (UTF-8):")
print(encoded_bytes)  # Output: b'Hello, World! \xf0\x9f\x98\x8a'
# Decode the bytes back to a string
decoded_string = encoded_bytes.decode('utf-8')
print("\nDecoded String:")
print(decoded_string)  # Output: Hello, World! 😊
# Encode the string to bytes using ASCII encoding (will fail for non-ASCII
characters)
try:
    ascii_encoded_bytes = original_string.encode('ascii')
except UnicodeEncodeError as e:
    print("\nEncoding Error (ASCII):")
    print(e)  # Output: 'ascii' codec can't encode character...
# Encode a string with only ASCII characters
ascii_string = "Hello, World!"
ascii_encoded_bytes = ascii_string.encode('ascii')
print("\nASCII Encoded Bytes:")
print(ascii_encoded_bytes)  # Output: b'Hello, World!'
# Decode the ASCII encoded bytes back to a string
ascii_decoded_string = ascii_encoded_bytes.decode('ascii')
print("\nASCII Decoded String:")
print(ascii_decoded_string)  # Output: Hello, World!
# Handling character sets in different languages
# Example: Japanese characters
japanese_string = "こんにちは世界"  # "Hello, World" in Japanese
print("\nJapanese String:")
print(japanese_string)  # Output: こんにちは世界
# Encode the Japanese string to bytes using UTF-8
japanese_encoded_bytes = japanese_string.encode('utf-8')
print("\nJapanese Encoded Bytes (UTF-8):")
print(japanese_encoded_bytes)  # Output:
b'\xe3\x81\x93\xe0\xa4\xbf\xe0\xa4\xa8\xe3\x81\xaf\xe4\xb8\x96\xe7\x95\x8c'
# Decode the Japanese bytes back to a string
japanese_decoded_string = japanese_encoded_bytes.decode('utf-8')
print("\nJapanese Decoded String:")
print(japanese_decoded_string)  # Output: こんにちは世界
```

OUTPUT

```
Original String:
Hello, World! 😊
```

```
Encoded Bytes (UTF-8):
b'Hello, World! \xf0\x9f\x98\x8a'
Decoded String:
Hello, World! 😊
Encoding Error (ASCII):
'ascii' codec can't encode character ...
ASCII Encoded Bytes:
b'Hello, World!'
ASCII Decoded String:
Hello, World!
Japanese String:
こんにちは世界
Japanese Encoded Bytes (UTF-8):
b'\xe3\x81\x93\xe0\xa4\xbf\xe0\xa4\xa8\xe3\x81\xaf\xe4\xb8\x96\xe7\x95\x8c'
Japanese Decoded String:
こんにちは世界
```

1.1.3 EXPRESSION

An expression is a combination of values, variables, operators, and function calls that evaluates to a single value. Expressions are fundamental in Python, and they can be used to perform calculations, comparisons, or logical operations.

Types of Expressions:

1. **Arithmetic Expressions**: Involves mathematical operations like addition, subtraction, multiplication, division, etc.
2. **Relational Expressions**: Compares values and returns a Boolean (True or False).
3. **Logical Expressions**: Combines multiple relational expressions using logical operators (and, or, not).
4. **Bitwise Expressions**: Operate at the bit level, such as AND, OR, XOR, shifts, etc.
5. **Lambda Expressions**: Used to create small anonymous functions.
6. **Compound Expressions**: Combine multiple types of expressions.

Python Program Demonstrating Different Expressions:

```
# 1. Arithmetic Expressions
```

```
a = 10
b = 5
c = 3
sum_result = a + b  # Addition
difference = a - b  # Subtraction
product = a * b  # Multiplication
quotient = a / b  # Division
modulus = a % b  # Modulo
exponent = a ** c  # Exponentiation
print("Arithmetic Expressions:")
print("Sum:", sum_result)          # Output: 15
print("Difference:", difference)    # Output: 5
print("Product:", product)         # Output: 50
print("Quotient:", quotient)       # Output: 2.0
print("Modulus:", modulus)         # Output: 0
print("Exponentiation:", exponent)  # Output: 1000
# 2. Relational Expressions
greater = a > b  # Greater than
less = a < c     # Less than
equal = a == 10  # Equal to
not_equal = b != c  # Not equal to
print("\nRelational Expressions:")
print("a > b:", greater)           # Output: True
print("a < c:", less)             # Output: False
print("a == 10:", equal)           # Output: True
print("b != c:", not_equal)        # Output: True
# 3. Logical Expressions
logical_and = (a > b) and (b > c)  # Both conditions are True
logical_or = (a > b) or (b < c)    # One condition is True
logical_not = not (a == b)         # Negates the result
print("\nLogical Expressions:")
print("a > b and b > c:", logical_and)  # Output: True
print("a > b or b < c:", logical_or)    # Output: True
print("not (a == b):", logical_not)     # Output: True
```

```
# 4. Bitwise Expressions
bitwise_and = a & b  # Bitwise AND
bitwise_or = a | b   # Bitwise OR
bitwise_xor = a ^ b  # Bitwise XOR
bitwise_shift_left = a << 1  # Shift left
bitwise_shift_right = a >> 1 # Shift right
print("\nBitwise Expressions:")
print("Bitwise AND (a & b):", bitwise_and)  # Output: 0
print("Bitwise OR (a | b):", bitwise_or)    # Output: 15
print("Bitwise XOR (a ^ b):", bitwise_xor)  # Output: 15
print("Shift left (a << 1):", bitwise_shift_left)  # Output: 20
print("Shift right (a >> 1):", bitwise_shift_right)  # Output: 5
# 5. Lambda Expressions
square = lambda x: x * x
double = lambda x: x * 2
print("\nLambda Expressions:")
print("Square of 5:", square(5))  # Output: 25
print("Double of 5:", double(5))  # Output: 10
# 6. Compound Expressions
compound_expression = (a + b) * (b - c) / (a % c)
print("\nCompound Expression:")
print("Result of compound expression:", compound_expression)  # Output: 10.0
```

Output:

```
Arithmetic Expressions:
Sum: 15
Difference: 5
Product: 50
Quotient: 2.0
Modulus: 0
Exponentiation: 1000
Relational Expressions:
a > b: True
a < c: False
```

```
a == 10: True
b != c: True

Logical Expressions:
a > b and b > c: True
a > b or b < c: True
not (a == b): True
Bitwise Expressions:
Bitwise AND (a & b): 0
Bitwise OR (a | b): 15
Bitwise XOR (a ^ b): 15
Shift left (a << 1): 20
Shift right (a >> 1): 5
Lambda Expressions:
Square of 5: 25
Double of 5: 10
Compound Expression:
Result of compound expression: 10.0
```

Let Us Sum Up

In this introductory unit, students will explore the fundamental concepts of computer science, starting with an understanding of strings, assignment statements, and comments. They will delve into the various numeric data types and character sets that form the foundation of data representation in computing. The unit will cover the construction and evaluation of expressions, providing students with essential skills for performing calculations and data manipulation. By the end of this unit, students will have a solid grasp of these basic yet crucial elements, forming a strong base for more advanced topics in computer science.

Check Your Progress

1. What is the primary purpose of using comments in a program?

   A) To increase the execution speed of the program.

   B) To provide explanations or annotations in the code for human readers.

C) To define variables.

D) To execute a block of code conditionally.

2. Which of the following is a valid string in Python?

A) 'Hello, World!'

B) 12345

C) True

D) 3.14

3. Which of the following data types is used to store decimal numbers in Python?

A) int

B) str

C) float

D) bool

4. In Python, what character is used to begin a single-line comment?

A) //

B) *

C) #

D) <!--

5. What is the purpose of an assignment statement in Python?

A) To compare two values.

B) To print values to the console.

C) To assign a value to a variable.

D) To iterate through a sequence.

6. Which of the following is a numeric data type in Python?

A) list

B) tuple

C) int

D) dict

7. What does the following Python code do? x = 5

A) It declares a function named x.

B) It assigns the value 5 to the variable x.

C) It prints the value 5.

D) It compares x with 5.

8. Which of the following character sets is used to represent text in Python?

A) Unicode

B) Binary

C) Hexadecimal

D) Octal

9. Which of the following is an example of an expression in Python?

A) x = 10

B) print("Hello, World!")

C) a + b * c

D) for i in range(5):

10. Which numeric data type would you use to represent the number of students in a class?

A) int

B) float

C) str

D) bool

11. Which function is used to convert a string into an integer in Python?

A) str()

B) int()

C) float()

D) bool()

12. What will be the result of the following expression: 3 + 2 * 5?

A) 25

B) 13

C) 10

D) 15

13. How do you denote a string in Python?

A) Using single or double quotes.

B) Using parentheses.

C) Using square brackets.

D) Using curly braces.

14. Which of the following is a correct variable name in Python?

A) 1st_var

B) var-1

C) _var1

D) var@1

15. What does the len() function do?

A) Calculates the length of a string.

B) Converts a string to lowercase.

C) Replaces characters in a string.

D) Joins two strings together.

16. Which of the following is not a numeric data type in Python?

A) int

B) float

C) complex

D) string

17. Which symbol is used for exponentiation in Python?

A) ^

B) **

C) *

D) //

18. What will be the result of the following code: print("Hello" + "World")?

A) Hello World

B) HelloWorld

C) "Hello" "World"

D) Error

19. Which method is used to convert all characters of a string to uppercase?

A) upper()

B) capitalize()

C) title()

D) uppercase()

20. How can you include a newline character in a string in Python?

A) \n

B) \\n

C) /n

D) \newline

21. Which keyword is used to create a function in Python?

A) function

B) define

C) def

D) func

22. Which of the following is a valid Boolean value in Python?

A) TRUE

B) true

C) False

D) 1

23. What will be the output of print(2 ** 3)?

A) 6

B) 8

C) 9

D) 23

24. Which of the following is used to create a comment in Python?

A) # This is a comment

B) /* This is a comment */

C) // This is a comment

D) <!-- This is a comment -->

25. How do you create a multi-line string in Python?

A) Using double quotes ""

B) Using triple quotes """ or '''

C) Using parentheses ()

D) Using square brackets []

26. Which function can be used to get the ASCII value of a character in Python?

A) ord()

B) char()

C) ascii()

D) chr()

27. What is the correct syntax to output the type of a variable or object in Python?

A) print(typeOf(x))

B) print(typeof(x))

C) print(type(x))

D) print(class(x))

28. What will the following code output? print("10" + "5")

A) 15

B) 105

C) 10 5

D) Error

29. Which method would you use to remove whitespace from the beginning and end of a string?

A) strip()

B) trim()

C) rstrip()

D) ltrim()

30. Which of the following is used to concatenate two strings in Python?

A) &

B) +

C) *

D) –

SECTION 1.2 LOOPS AND SLECTION STATEMENTA

1.1.3 Loops and Selection Statements

In Python, expressions are combinations of values, variables, operators, and function calls that evaluate to a single value. Expressions are the building blocks of Python code and are used to perform computations, manipulate data, and make decisions. Here's a breakdown of different types of expressions in Python:

Numeric Expressions:

Numeric expressions involve arithmetic operations on numeric values such as integers, floats, and complex numbers.

```
# Addition
        result = 10 + 5
# Subtraction
        result = 10 - 5
# Multiplication
        result = 10 * 5
# Division
        result = 10 / 5
# Floor Division (returns the integer part of the division)
        result = 10 // 5
# Modulus (returns the remainder of the division)
                result = 10 % 3
# Exponentiation
        result = 10 ** 3
```

Boolean Expressions:

Boolean expressions evaluate to either True or False and are typically used in conditions and control flow statements.

```
# Comparison Operators
        result = 10 > 5
        result = 10 == 5
        result = 10 != 5
        result = 10 <= 5
        result = 10 >= 5
# Logical Operators
        result = (10 > 5) and (5 < 3)
        result = (10 > 5) or (5 < 3)
        result = not (10 > 5)
```

String Expressions:

String expressions involve operations on string values such as concatenation, slicing, and formatting.

```
# Concatenation
        result = "Hello" + " " + "World"
# String Repetition
        result = "Python" * 3
# String Length
        result = len("Python")
# String Slicing
        s = "Python"
        result = s [1:4]  # Returns "yth"
# String Formatting
        name = "Alice"
        age = 30
        result = f"My name is {name} and I am {age} years old."
```

Function Call Expressions:

Function call expressions involve calling functions with arguments and optionally capturing their return values.

```
# Built-in Functions
        result = max (10, 5, 8)   # Returns the maximum value among the
        arguments
        result = abs (-10)        # Returns the absolute value

# User-defined Functions
        def square(x):
          return x ** 2
result = square (5)
```

Attribute Access Expressions:

Attribute access expressions involve accessing attributes or methods of objects.

```
# Attribute Access
        result = "Python”. Upper ()   # Returns "PYTHON"
        result = [1, 2, 3]. append(4) # Modifies the list to [1, 2, 3, 4]
```

List and Dictionary Expressions:

List and dictionary expressions involve creating and manipulating lists and dictionaries using comprehensions or literals.

```
# List Comprehension
squares = [x ** 2 for x in range (5)]  # Generates [0, 1, 4, 9, 16]
# Dictionary Comprehension
squares_dict = {x: x ** 2 for x in range (5)}  # Generates {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Lambda Expressions:

Lambda expressions are anonymous functions that can have any number of arguments but only one expression.

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

These are some of the common types of expressions in Python.

1.2.1 – DEFINE ITERATION

Types of Loops:

1. **for Loop**: Iterates over a sequence (like a list, tuple, or string).
2. **while Loop**: Repeats as long as a given condition is True.

Selection Statements (Conditional Statements):

1. **if statement**: Executes a block of code if a condition is True.
2. **if-else statement**: Executes one block of code if the condition is True and another block if it's False.
3. **if-elif-else statement**: Checks multiple conditions in sequence and executes the first True block.

```
# Example 1: Using a for loop with an if-else statement

numbers = [1, 2, 3, 4, 5]

print("For loop with if-else:")

for num in numbers:

    if num % 2 == 0:
```

```
        print(f"{num} is even")
    else:
        print(f"{num} is odd")


# Example 2: Using a while loop with an if-elif-else statement
print("\nWhile loop with if-elif-else:")
counter = 0
while counter < 5:
    if counter == 0:
        print("Counter is zero")
    elif counter < 3:
        print(f"Counter is {counter}, less than 3")
    else:
        print(f"Counter is {counter}, greater than or equal to 3")
    counter += 1
# Example 3: Using a nested for loop with a break statement
print("\nNested for loop with break:")
for i in range(3):
    for j in range(3):
        if i == j:
            print(f"Breaking out of inner loop when i = {i} and j = {j}")
            break  # Exit the inner loop
        print(f"i = {i}, j = {j}")
# Example 4: Using a while loop with a continue statement
print("\nWhile loop with continue:")
```

```
count = 0
while count < 5:
    count += 1
    if count == 3:
        print("Skipping 3")
        continue  # Skip the rest of the loop for this iteration
    print(f"Count is {count}")
# Example 5: Iterating over a dictionary using a for loop
print("\nIterating over a dictionary:")
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

OUTPUT

```
For loop with if-else:
1 is odd
2 is even
3 is odd
4 is even
5 is odd
While loop with if-elif-else:
Counter is zero
Counter is 1, less than 3
Counter is 2, less than 3
Counter is 3, greater than or equal to 3
Counter is 4, greater than or equal to 4
```

Nested for loop with break:

i = 0, j = 1

i = 0, j = 2

Breaking out of inner loop when i = 1 and j = 1

i = 2, j = 0

Breaking out of inner loop when i = 2 and j = 2

While loop with continue:

Count is 1

Count is 2

Skipping 3

Count is 4

Count is 5

Iterating over a dictionary:

name: Alice

age: 25

city: New York

1.2.2 – FOR LOOP

A for loop is used to iterate over a sequence (such as a list, tuple, string, or range) or any other inerrable object. It allows you to execute a block of code repeatedly for each item in the sequence.

The basic syntax of a for loop in Python is as follows:

```
for item in iterable:
    # Code to be executed for each item
```

- item is a variable that represents each individual element in the sequence during each iteration of the loop.
- inerrable is the sequence or inerrable object that the loop iterates over.

Examples:

1. Iterating over a List:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:
apple
banana
cherry

2. Iterating over a String:

```
for char in "Python":
    print(char)
```

Output:

P
y
t
h
o
n

3. Iterating over a Range:

```
for i in range (5):
    print(i)
```

Output:

0
1
2
3
4

4. Using enumerate to Access Index and Item:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print (f"Index: {index}, Fruit: {fruit}")
```

Output:

Index: 0, Fruit: apple
Index: 1, Fruit: banana

Index: 2, Fruit: cherry

*5. **Using zip to Iterate Over Multiple Lists:***

```
names = ["Alice", "Bob", "Charlie"]
ages = [30, 25, 35]
for name, age in zip(names, ages):
   print (f"Name: {name}, Age: {age}")
```

Output:

Name: Alice, Age: 30
Name: Bob, Age: 25
Name: Charlie, Age: 35

1.2.3 – SELECTION : OF AND IF-ELSE STATEMENTS

Selection statements, also known as conditional statements, are used to control the flow of the program based on certain conditions. The two primary selection statements in Python are:

1. if Statement

The if statement is used to execute a block of code only if a specified condition is true.

Syntax:

```
if condition:
   # Code to be executed if the condition is true
```

2. if...else Statement

The if...else statement is used to execute one block of code if the condition is true and another block of code if the condition is false.

Syntax:

```
if condition:
   # Code to be executed if the condition is true
else:
   # Code to be executed if the condition is false
```

3. if...elif...else Statement

The if...elif...else statement is used when there are multiple conditions to be checked.

Syntax:

```
if condition1:
   # Code to be executed if condition1 is true
elif condition2:
   # Code to be executed if condition1 is false and condition2 is true
else:
   # Code to be executed if both condition1 and condition2 are false
```

Nested if Statements

if statements inside other if statements, known as nested if statements.

```
# Example: if Statement
print("Example 1: if Statement")
num1 = 10
if num1 > 5:
   print(f"{num1} is greater than 5.")  # Output: 10 is greater than 5
print()
# Example: if...else Statement
print("Example 2: if...else Statement")
num2 = 3
if num2 > 5:
   print(f"{num2} is greater than 5.")
else:
   print(f"{num2} is less than or equal to 5.")  # Output: 3 is less than or equal
to 5
print()
# Example: if...elif...else Statement
print("Example 3: if...elif...else Statement")
num3 = 7
if num3 > 10:
   print(f"{num3} is greater than 10.")
elif num3 == 7:
   print(f"{num3} is exactly 7.")  # Output: 7 is exactly 7
else:
   print(f"{num3} is less than 7.")
print()
# Example: Nested if Statement
```

```
print("Example 4: Nested if Statements")
num4 = 15
if num4 > 10:
    print(f"{num4} is greater than 10.")  # Output: 15 is greater than 10
    if num4 % 2 == 0:
        print(f"{num4} is even.")
    else:
        print(f"{num4} is odd.")  # Output: 15 is odd
else:
    print(f"{num4} is 10 or less.")
```

OUTPUT

Example 1: if Statement

10 is greater than 5.

Example 2: if...else Statement

3 is less than or equal to 5.

Example 3: if...elif...else Statement

7 is exactly 7.

Example 4: Nested if Statements

15 is greater than 10.

15 is odd.

Loop Control Statements

1. break Statement:

The break statement is used to exit the loop prematurely.

2. continue Statement:

The continue statement is used to skip the current iteration and continue with the next iteration of the loop.

for loops are versatile and widely used in Python for iterating over collections, performing repetitive tasks, and implementing various algorithms.

OUTPUT

```
# Example: break statement
print("Example 1: break Statement")
for num in range(1, 11):
    if num == 6:
```

```
        print("Loop breaks at number 6.")
        break  # Exits the loop when num equals 6
    print(num)  # Prints numbers from 1 to 5
print()
# Example: continue statement
print("Example 2: continue Statement")
for num in range(1, 11):
    if num == 6:
        print("Skipping number 6.")
        continue  # Skips the rest of the loop when num equals 6
    print(num)  # Prints numbers except 6
```

1.2.4 – CONDITIONAL ITERATION : THE While Loop

A while loop is used to repeatedly execute a block of code as long as a specified condition is true. It continues to execute the block of code until the condition becomes false.

The basic syntax of a while loop in Python is as follows:

```
while condition:
    # Code to be executed as long as the condition is true
```

- ✓ condition is a Boolean expression that determines whether the loop should continue or not.
- ✓ The block of code under the while loop is executed repeatedly until the condition becomes false.

Example:

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

Output:

```
0
1
2
3
4
```

In this example, the loop continues to execute as long as the condition count < 5 is true. The value of count starts from 0, and it is incremented by 1 in each iteration. When count becomes 5, the condition becomes false, and the loop terminates.

Infinite Loop:

Be cautious while using while loops to avoid creating an infinite loop, where the condition never becomes false. An infinite loop can cause your program to hang or become unresponsive.

```
# Infinite loop
    while True:
        print ("This is an infinite loop!")
```

To break out of an infinite loop, you can use a loop control statement like break, as shown below:

```
while True:
        response = input ("Enter 'quit' to exit: ")
        if response == 'quit':
            break
        print ("You entered:", response)
```

In this example, the loop continues indefinitely until the user enters "quit". Once the user enters "quit", the break statement is executed, and the loop terminates.

While loops are useful when you want to execute a block of code repeatedly until a certain condition is met. However, be careful to ensure that the condition eventually becomes false to avoid infinite loops.

1.2.5– PLAYERS IN FINANCIAL SERVICES SECTOR

To incorporate **selection, control, and loop statements** while identifying players in the Financial Services sector, we can create a Python program that retrieves company data, applies conditions (selection), and uses loops to process multiple tickers. We'll use control structures (if-else, while, and for) along with the yfinance library.

Below is an example code that demonstrates the use of **selection**, **control**, and **loop** statements:

```
import yfinance as yf
import pandas as pd


# List of sample tickers from different sectors (some financial, some non-
financial)
tickers = ['JPM', 'AAPL', 'BAC', 'GS', 'GOOG', 'WFC', 'MS', 'TSLA', 'V']
# Function to get sector information for the financial services sector using loops
and control statements
def get_financial_sector_players(tickers):
    data = []
    count = 0  # Initialize a counter to control the number of processed companies
    for ticker in tickers:  # for loop to iterate over all tickers
        count += 1
        print(f"\nProcessing {count}/{len(tickers)}: {ticker}")
        stock = yf.Ticker(ticker)
        info = stock.info
            # Selection control using if-else statements to check if the sector is
Financial Services
        if 'sector' in info:
            if info['sector'] == 'Financial Services':  # Filter by Financial Services
sector
                print(f"-> {info['longName']} is in the Financial Services sector.")
                data.append({
                    'Company': info.get('longName'),
                    'Ticker': ticker,
                    'Sector': info.get('sector'),
                    'Industry': info.get('industry'),
                    'Market Cap': info.get('marketCap')
                })
            else:
                print(f"-> {info['longName']} is NOT in the Financial Services sector.")
```

```
        else:
            print(f"-> No sector data found for {ticker}")

        # Control structure to limit processing to the first 5 companies (using break)
        if count >= 5:
            print("Processed 5 companies, stopping further processing.")
            break  # Exit the loop after processing 5 companies

    return pd.DataFrame(data)

# While loop demonstration: retry if no data
attempts = 0
max_attempts = 3
while attempts < max_attempts:
    try:
        # Get financial sector players
        print(f"Attempt {attempts + 1}/{max_attempts}: Fetching company data")
        df = get_financial_sector_players(tickers)
        if not df.empty:
            print("\nFinancial Services Sector Players:")
            print(df)
        else:
            print("\nNo financial companies found.")
        break  # Exit the loop if successful
    except Exception as e:
        print(f"Error occurred: {e}")
        attempts += 1
        if attempts == max_attempts:
            print("Max attempts reached. Exiting program.")
        else:
            print("Retrying...\n")
```

OUTPUT

```
Attempt 1/3: Fetching company data
Processing 1/9: JPM
-> JPMorgan Chase & Co. is in the Financial Services sector.
Processing 2/9: AAPL
-> Apple Inc. is NOT in the Financial Services sector.
Processing 3/9: BAC
-> Bank of America Corporation is in the Financial Services sector.
Processing 4/9: GS
-> Goldman Sachs Group, Inc. is in the Financial Services sector.
Processing 5/9: GOOG
-> Alphabet Inc. is NOT in the Financial Services sector.
Processed 5 companies, stopping further processing.
Financial Services Sector Players:
                Company Ticker          Sector          Industry     Market Cap
0    JPMorgan Chase & Co.    JPM   Financial Services  Banks—Diversified
476542734500
1  Bank of America Corporation  BAC  Financial Services  Banks—Diversified
330700765500
2    Goldman Sachs Group, Inc.    GS   Financial Services  Capital Markets
115450726000
```

Let Us Sum Up

This delves into the fundamental control structures in Python, focusing on expressions, loops, and selection statements. Students will explore definite iteration using the for loop, which allows repeated execution of a block of code for a specified number of times or over a sequence. The unit will also cover selection statements such as if and if-else, enabling the execution of code based on specific conditions. Additionally, conditional iteration with the while loop will be introduced, allowing code to repeat as long as a certain condition is true. These concepts are essential for creating dynamic and responsive programs, enabling more complex and versatile coding solutions.

Check Your Progress

1. What does a for loop in Python typically iterate over?

    A) Integers only

    B) Strings only

    C) Lists, tuples, dictionaries, and ranges

    D) Floating-point numbers

2. What is the correct syntax for a for loop in Python?

    A) for (i = 0; i < 10; i++)

    B) for i in range(10):

    C) foreach i in range(10)

    D) for i to 10:

3. Which statement is used to terminate a loop prematurely in Python?

    A) continue

    B) exit

    C) break

    D) stop

4. What will the following code print? for i in range(3): print(i)

    A) 1 2 3

    B) 0 1 2

    C) 0 1 2 3

    D) 1 2

5. Which of the following keywords is used for selection statements in Python?

    A) select

    B) switch

C) if

D) choice

6. What is the output of the following code? if 5 > 3: print("Hello")

   A) Hello

   B) Nothing

   C) Error

   D) 5 > 3

7. Which of the following is correct syntax for an if-else statement in Python?

   A) if x > y: print("x is greater") else: print("y is greater")

   B) if (x > y) { print("x is greater"); } else { print("y is greater"); }

   C) if x > y print "x is greater" else print "y is greater"

   D) if x > y: print("x is greater") else: print("y is greater")

8. Which loop would you use for conditional iteration in Python?

   A) for

   B) while

   C) do-while

   D) foreach

9. What does the following code do? while x < 10: print(x) x += 1

   A) Prints numbers from 1 to 10

   B) Prints numbers from 0 to 9

   C) Causes an infinite loop

   D) Syntax error

10. How do you skip the current iteration in a loop and proceed to the next one?

   A) break

B) continue

C) skip

D) pass

11. What will be the output of the following code? for i in range(2): for j in range(2): print(i, j)

A) (0, 1) (0, 2) (1, 1) (1, 2)

B) (0, 0) (0, 1) (1, 0) (1, 1)

C) (1, 0) (2, 0) (1, 1) (2, 1)

D) (1, 1) (2, 2) (1, 2) (2, 1)

12. What is the result of if not (4 == 4): print("No") else: print("Yes")?

A) No

B) Yes

C) Error

D) Nothing

13. Which of the following is a valid while loop?

A) while x: print(x)

B) while (x): print(x)

C) while x == 5: print(x)

D) while (x == 5) { print(x); }

14. What does the following code print? if 3 < 2: print("A") elif 3 == 3: print("B") else: print("C")

A) A

B) B

C) C

D) Nothing

15. What is the output of the following code? for i in range(5): if i == 3: break print(i)

    A) 0 1 2 3 4

    B) 0 1 2 3

    C) 0 1 2

    D) 1 2 3

16. How would you write a for loop to iterate over each character in a string s?

    A) for c in s:

    B) for each c in s:

    C) foreach c in s:

    D) for c to s:

17. Which of the following statements is true about the else clause in a loop?

    A) It executes only if the loop terminates normally without a break.

    B) It executes only if the loop contains a continue.

    C) It executes only if the loop is infinite.

    D) It executes before the loop starts.

18. What will the following code output? x = 0 while x < 3: x += 1 print(x)

    A) 1 2 3

    B) 0 1 2

    C) 1 2

    D) 0 1 2 3

19. Which of the following is a correct syntax for an infinite loop using while?

    A) while (true):

    B) while True:

    C) while (1):

D) while (True):

20. What does the pass statement do in a loop?

A) Terminates the loop.

B) Skips the current iteration and continues with the next.

C) Does nothing and continues to the next statement.

D) Causes a syntax error.

21. Which keyword is used to check multiple conditions in an if statement?

A) elif

B) elseif

C) else

D) elseif

22. What is the output of the following code? for i in range(3): if i == 1: continue print(i)

A) 0 1 2

B) 0 1

C) 0 2

D) 1 2

23. Which statement is used to execute a block of code only if a specified condition is false?

A) if

B) else

C) elif

D) unless

24. What will be the output of the following code? x = 5 if x > 2: print("Greater") else: print("Smaller")

    A) Greater

    B) Smaller

    C) 5

    D) Error

25. Which of the following can be used to combine multiple conditions in an if statement?

    A) and, or

    B) plus, minus

    C) & , |

    D) add, subtract

26. What will be the output of the following code? count = 0 while count < 3: print("looping") count += 1

    A) looping

    B) looping looping looping

    C) looping 0 looping 1 looping 2

    D) looping 1 looping 2 looping 3

27. Which of the following statements is used to exit a loop?

    A) exit

    B) stop

    C) break

    D) finish

28. What does the following code do? for i in range(5): if i == 2: break else: print(i)

    A) Prints 0, 1, 2

B) Prints 0, 1

C) Prints 0, 1, 3, 4

D) Causes an error

29. Which of the following is true about if-else statements in Python?

A) else must always follow an if or elif block.

B) if can exist without else.

C) elif can be used without if.

D) else can exist without if.

30. What will the output of this code be? for i in range(4): print(i) else: print("Done")

A) 0 1 2 3 Done

B) 0 1 2 3

C) Done

D) Error

Unit Summary

This unit covers fundamental concepts in computer science, focusing on basic programming constructs and data types. It begins with an introduction to strings, assignments, and comments, essential for documenting and structuring code. Numeric data types and character sets are discussed, highlighting how different types of data are represented and manipulated. The unit explores expressions and operators, providing the tools to perform calculations and logic operations. Loop constructs such as the for loop for definite iteration and the while loop for conditional iteration are introduced, allowing for repetitive tasks and complex flow control. Selection statements, including if and if-else, are covered to facilitate decision-making in programs. Overall, this unit lays a solid foundation for understanding and writing basic programs.

Glossary

- **String:** A sequence of characters used to represent text in programming. Enclosed in quotes (' or ").
- **Assignment:** The process of assigning a value to a variable using the = operator.
- **Comment:** A non-executable statement in code, used for documentation. In Python, comments start with #.
- **Numeric Data Types**: Data types that represent numbers. Common types include int (integers) and float (floating-point numbers).
- **Character Set:** A set of characters recognized by the computer, including letters, digits, and symbols. ASCII and Unicode are common character sets.
- **Expression:** A combination of variables, operators, and values that yields a result value.
- **Operator:** A symbol that performs operations on variables and values. Examples include +, -, *, /, and **.
- **For Loop**: A control flow statement for definite iteration, executing a block of code a specific number of times.
- **While Loop**: A control flow statement for conditional iteration, executing a block of code as long as a condition is true.
- **If Statement:** A selection statement that executes a block of code if a specified condition is true.
- **If-Else Statement**: A selection statement that executes one block of code if a condition is true and another block if the condition is false.
- **Elif:** Short for "else if," used in Python to check multiple conditions in sequence.
- **Break:** A statement used to exit a loop prematurely.
- **Continue:** A statement used to skip the rest of the current iteration of a loop and proceed to the next iteration.
- **Range():** A function that generates a sequence of numbers, commonly used in for loops.
- **Len():** A function that returns the length of a string, list, tuple, or other collections.
- **Ord():** A function that returns the ASCII value of a character.

- **Str():** A function that converts a value to a string representation.

Self Assessment Questions

1. Evaluate the importance of comments in programming and explain how they contribute to code readability and maintainability.
2. Analyze the role of strings in Python programming and compare the usage of single quotes and double quotes for string declaration.
3. Assess the significance of assignment statements in Python and illustrate with examples how variables are initialized and assigned values.
4. Compare and contrast the main numeric data types in Python, analyzing their characteristics and use cases.
5. Evaluate the purpose of expressions in Python and explain how they are used to perform computations and evaluate conditions.
6. Analyze the syntax and functionality of the for loop in Python, comparing it with other iterative constructs.
7. Assess the effectiveness of selection statements (if and if-else) in Python for making decisions based on conditional expressions.
8. Compare the use of the while loop and the for loop in Python, evaluating their strengths and limitations in different scenarios.
9. Evaluate the significance of definite iteration using the for loop and explain how it differs from conditional iteration with the while loop.
10. Assess the effectiveness of using comments to document Python code and compare different commenting styles for clarity and readability.
11. Analyze the role of numeric data types and character sets in Python programming, illustrating their importance in data manipulation and representation.
12. Evaluate the impact of expressions on program efficiency and performance, comparing simple arithmetic operations with complex expressions.

Activities / Exercises / Case Studies

Activities

1. Create a Python script that prompts the user for their first name and last name, then prints a greeting message that includes both names in uppercase.
2. Write a Python program to count the number of vowels in a given string.
3. Take a simple Python script and add comments explaining each line of code. Ensure that the comments describe the purpose and functionality of the code.
4. Write a Python program that converts a temperature from Celsius to Fahrenheit and vice versa. Use functions for the conversions and print the results.
5. Create a Python script that evaluates and prints the result of various mathematical expressions, including addition, subtraction, multiplication, division, and exponentiation.

Exercises

1. Write a Python program that prints the multiplication table for numbers 1 through 10 using a for loop. Modify the program to take an input number from the user and print its multiplication table.
2. Write a Python program that checks if a given year is a leap year. Use an if-else statement to determine and print whether the year is a leap year. Extend the program to check if the input is a valid year.
3. Create a Python script that asks the user to guess a number between 1 and 100. Use a while loop to give the user multiple attempts to guess the number, and provide feedback if the guess is too high or too low. Implement a counter to track the number of attempts and display it once the user guesses the correct number.
4. Write a Python program that prints the ASCII values of all characters from 'a' to 'z'. Modify the program to print the characters for a given range of ASCII values.

Case Studies

1. Case Study: Basic Calculator Program

- Develop a Python-based calculator that performs basic arithmetic operations: addition, subtraction, multiplication, and division. Use functions for each operation and include input validation.
- Add a feature to handle invalid inputs gracefully and display appropriate error messages.

2. Case Study: Student Grade Management System

- Design a Python program that allows a teacher to input students' names and their corresponding grades. Store the data in a dictionary and provide functionalities to:
- Display all students and their grades.
- Calculate and display the average grade.
- Find and print the highest and lowest grades along with the respective students' names.
- Enhance the program to allow updating and deleting student records.

3. Case Study: Simple Banking System

- Create a Python program to simulate a simple banking system. The program should allow users to:
- Create an account with an initial balance.
- Deposit money into the account.
- Withdraw money from the account, ensuring the balance does not go negative.
- Check the account balance.
- Implement input validation and handle edge cases such as attempting to withdraw more money than available in the account.

Answers for check your progress

Modules	S. No.	Answers
Module 1	1.	B) To provide explanations or annotations in the code for human readers.
	2.	A) 'Hello, World!'

	3.	C) float
	4.	C) #
	5.	C) To assign a value to a variable.
	6.	C) int
	7.	B) It assigns the value 5 to the variable x.
	8.	A) Unicode
	9.	C) a + b * c
	10.	A) int
	11.	B) int()
	12.	D) 15
	13.	A) Using single or double quotes.
	14.	C) _var1
	15.	A) Calculates the length of a string.
	16.	D) string
	17.	B) **
	18.	B) HelloWorld
	19.	A) upper()
	20.	A) \n
	21.	C) def
	22.	C) False
	23.	B) 8
	24.	A) # This is a comment
	25.	B) Using triple quotes """ or '''
	26.	A) ord()
	27.	C) print(type(x))
	28.	B) 105
	29.	A) strip()
	30.	B) +
Module 2	**1.**	C) Lists, tuples, dictionaries, and ranges
	2.	B) for i in range(10):
	3.	C) break
	4.	B) 0 1 2

	5.	C) if
	6.	A) Hello
	7.	A) if x > y: print("x is greater") else: print("y is greater")
	8.	B) while
	9.	B) Prints numbers from 0 to 9
	10.	B) continue
	11.	B) (0, 0) (0, 1) (1, 0) (1, 1)
	12.	D) Nothing
	13.	A) while x: print(x)
	14.	B) B
	15.	B) 0 1 2 3
	16.	A) for c in s:
	17.	A) It executes only if the loop terminates normally without a break.
	18.	C) 1 2
	19.	B) while True:
	20.	C) Does nothing and continues to the next statement.
	21.	A) elif
	22.	B) 0 1
	23.	B) else
	24.	A) Greater
	25.	A) and, or
	26.	B) looping looping looping
	27.	C) break
	28.	B) Prints 0, 1
	29.	B) if can exist without else.
	30.	A) 0 1 2 3 Done

Suggested Readings

1. Milliken, C. P. (2019). *Python projects for beginners: a ten-week bootcamp approach to Python programming*. Apress.

2. Sweigart, A. (2019). *Automate the boring stuff with Python: practical programming for total beginners*. no starch press.
3. Lutz, M. (2013). *Learning python: Powerful object-oriented programming*. " O'Reilly Media, Inc.".

Open-Source E-Content Links

1. https://docs.python.org/3/
2. https://www.w3schools.com/python/
3. **https://www.geeksforgeeks.org/python-programming-language-tutorial/**

References

1. MIT OpenCourseWare: Introduction to Computer Science and Programming Using Python
2. Coursera: Python for Everybody Specialization
3. Codecademy Python Course

UNIT II – STRINGS AND TEXT FILES

Unit – II: Strings and Text Files: Accessing Characters and substrings in strings - Data encryption-Strings and Number systems- String methods - Text - Lists and Dictionaries: Lists - Dictionaries - Design with Functions: A Quick review - Problem Solving with top-Down Design - Design with recursive Functions - Managing a Program's namespace - Higher-Order Functions

Strings and Text Files

Section	Topic	Page No.
	UNIT – II	
	Unit Objectives	
Section 2.1	**Strings and Text Files**	**64**
2.1.1	Accessing Characters and substrings in strings	65
2.1.2	Data Encryption	67
2.1.3	Strings and Number systems and String Methods	69
2.1.4	Text	75
	Let Us Sum Up	77
	Check Your Progress	77
Section 2.2	**Lists and Dictionaries**	**80**
2.2.1	Lists	80
2.2.2	Dictionaries	82
	Let Us Sum Up	86
	Check Your Progress	86
Section 2.3	**Design with Functions**	89
2.3.1	A Quick Review	89
2.3.2	Problem Solving with Top Down Design	90
2.3.3	Design with Recursive Functions	92
2.3.4	Managing a Program's Namespace	94
2.3.5	Higher Order Function	95
	Let Us Sum Up	97
	Check Your Progress	97
2.4	Unit- Summary	101
2.5	Glossary	102

2.6	Self- Assessment Questions	103
2.7	Activities / Exercises / Case Studies	104
2.8	Answers for Check your Progress	106
2.9	References and Suggested Readings	108

Unit Objective:

The course objectives encompass a comprehensive journey through Python programming, starting with foundational understanding and gradually progressing to advanced concepts. Students will master fundamental Python concepts, including data types and control structures, before delving into the intricacies of string manipulation, text file handling, and data structures like lists and dictionaries. With an emphasis on function design and implementation, students will learn to create modular and reusable code, essential for problem-solving in Python. Throughout the course, problem-solving techniques will be honed through top-down design strategies, empowering students to break down complex problems effectively. By exploring advanced topics such as namespace management, recursion, and higher-order functions, students will gain a deeper understanding of Python's capabilities, preparing them for real-world programming challenges with confidence and proficiency.

SECTION 2.1: STRINGS AND TEXT FILES

In the realm of Python programming, strings and text files are foundational elements crucial for various applications. Strings represent sequences of characters, enabling manipulation, analysis, and presentation of textual data. Python offers robust string manipulation capabilities, empowering developers to perform tasks like concatenation, slicing, and formatting with ease. Additionally, text files serve as a vital means of storing and accessing large volumes of textual data persistently. Python provides powerful tools for reading from and writing to text files, facilitating tasks such as data input/output operations, logging, and data storage. Mastery of string handling and text file processing is essential for building versatile and efficient Python applications across a wide range of domains, from data analysis and processing to web development and beyond.

2.1.1 ACCESSING CHARACTERS AND SUBSTRINGS IN STRINGS

Accessing characters and substrings in strings is a fundamental operation in Python. Here's how you can do it:

Accessing Characters:

Access individual characters in a string using indexing. Indexing in Python starts from 0.

```
s = "Hello, World!"
# Accessing individual characters
print (s [0])  # Output: 'H'
print (s [7])  # Output: 'W'
```

Accessing Substrings (Slicing):

Extract substrings from a string using slicing. Slicing syntax is string[start:end:step], where start is the starting index (inclusive), end is the ending index (exclusive), and step is the step size (default is 1).

```
s = "Hello, World!"
# Extracting substrings
print(s [7:12])   # Output: 'World'
print(s[:5])     # Output: 'Hello'
print(s[7:])     # Output: 'World!'
print(s[::2])    # Output: 'Hlo ol!'
```

Accessing Characters in Reverse:

Access characters in reverse order by using negative indices.

```
s = "Hello, Worl
# Accessing characters in reverse
print(s[-1])   # Output: '!'
print(s[-6])   # Output: 'W'
```

Iterating Through Characters:

Iterate through each character in a string using a loop.

```
s = "Hello"
```

```
# Iterating through characters
for char in s:
   print(char)
```

Length of a String:

Find the length of a string using the len() function.

```
s = "Hello, World!"
# Length of the string
print(len(s))   # Output: 13
```

Modifying Strings:

Strings in Python are immutable, cannot modify them in-place. However, it can create a new string with the desired modifications.

```
s = "Hello, World!"
# Modifying strings
modified_s = s[:5] + " Python!"
print(modified_s)  # Output: 'Hello Python!'
```

Working with Text Files:

To read text from a file and manipulate it as strings, use file handling in Python.

```
# Reading from a text file
with open ("example.txt", "r") as file:
data = file.read()
```

PROGRAM FOR ACCESS CHARACTERS AND SUBSTRINGS IN STRINGS USING INDEXING AND SLICING IN PYTHON:

```
# Printing the content of the file
print(data)
# Accessing characters in a string
my_string = "Hello, World!"
# Access individual characters
print("First character:", my_string[0])    # Output: 'H'
print("Eighth character:", my_string[7])  # Output: 'W'
print("Last character:", my_string[-1])    # Output: '!'
# Accessing substrings using slicing
```

```
print("Substring from index 0 to 4:", my_string[0:5])   # Output: 'Hello'
print("Substring from index 7 to 11:", my_string[7:12]) # Output: 'World'
# Slicing without start or end
print("Substring from start to index 4:", my_string[:5])   # Output: 'Hello'
print("Substring from index 7 to the end:", my_string[7:]) # Output: 'World!'
# Step slicing
print("Every second character:", my_string[::2])  # Output: 'Hlo ol!'
# Reverse the string using slicing
print("Reversed string:", my_string[::-1])  # Output: '!dlroW ,olleH'
```

2.1.2 DATA ENCRYPTION

In Python, data encryption involves converting plain text into a ciphertext using encryption algorithms and keys. Python provides several libraries and modules for data encryption, with the most commonly used one being the cryptography library. Here's a basic overview of how you can perform data encryption in Python using this library:

Using the cryptography Library:

1. **Installation**: If you haven't installed the cryptography library yet, you can do so using pip:

   ```
   pip install cryptography.
   ```

2. **Symmetric Encryption (AES)**: Symmetric encryption uses a single key for both encryption and decryption. AES (Advanced Encryption Standard) is one of the most widely used symmetric encryption algorithms.

   ```
   from cryptography. fernet import Fernet.
   # Generate a random key
       key = Fernet.generate_key()
   # Create a Fernet symmetric encryption object with the key
       cipher = Fernet(key)
   # Encrypting data
       plaintext = b"Hello, world!"
       ciphertext = cipher.encrypt(plaintext)
   # Decrypting data
       decrypted_text = cipher.decrypt(ciphertext)
       print("Plaintext:", plaintext.decode())
       print("Ciphertext:", ciphertext)
   ```

```
print("Decrypted text:", decrypted_text.decode())
```

3.Asymmetric Encryption (RSA): Asymmetric encryption uses a pair of public and private keys. RSA is one of the most commonly used asymmetric encryption algorithms.

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
# Generate RSA key pair
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
)
public_key = private_key.public_key()
# Encrypting data with the public key
ciphertext = public_key.encrypt(
    b"Hello, world!",
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Decrypting data with the private key
plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
print("Plaintext:", plaintext.decode())
print("Ciphertext:", ciphertext)
```

3. **Hashing**: While not technically encryption, hashing is commonly used to securely store passwords and verify data integrity.

```
import hashlib
# Hashing a string
```

```
plaintext = b"Hello, world!"
hashed = hashlib.sha256(plaintext).hexdigest()
print("Plaintext:", plaintext.decode())
print("Hashed:", hashed)
```

These are basic examples of data encryption in Python using the cryptography library. Encryption is crucial for protecting sensitive information and ensuring data confidentiality and integrity in various applications. Always ensure to use strong encryption algorithms and keep your encryption keys secure.

2.1.3 STRINGS AND NUMBER SYSTERMS , STRING METHODS

The below Python functions are used to change the case of the strings. Let’s look at some Python string methods with examples:

- **lower():** Converts all uppercase characters in a string into lowercase.
- **upper():** Converts all lowercase characters in a string into uppercase.
- **title():** Convert string to title case.
- **swapcase():** Swap the cases of all characters in a string.
- **capitalize():** Convert the first character of a string to uppercase.

Example:

```
# Python3 program to show the
# working of upper() function
text = 'geeKs For geEkS'
# upper() function to convert
# string to upper case
print("\nConverted String:")
print(text.upper())
# lower() function to convert
# string to lower case
print("\nConverted String:")
print(text.lower())
# converts the first character to
# upper case and rest to lower case
print("\nConverted String:")
print(text.title())
# swaps the case of all characters in the string
# upper case character to lowercase and vice versa
print("\nConverted String:")
```

```
        print(text. swapcase())
    # convert the first character of a string to uppercase
    print("\nConverted String:")
    print(text.capitalize())
    # original string never changes
    print("\nOriginal String")
    print(text)
```

Output :

```
Converted String:
GEEKS FOR GEEKS
Converted String:
geeks for geeks
Converted String:
Geeks For Geeks
Converted String:
GEEkS fOR GEeKs
Original String
```

geeKs For geEkSTime complexity: O(n) where n is the length of the string ‘text’
Auxiliary space: **O(1)**

List of String Methods :

Here is the list of in-built Python string methods, that you can use to perform actions on string:

1. String capitalize() - Converts first character to Capital Letter
2.String casefold() - Converts to case folded strings
3.String center() - Pads string with specified character
4.String count() - Returns occurrences of substring in string
5.String encode() - Returns encoded string of given string
6.String endswith() - Checks if String Ends with the Specified Suffix
7.String expandtabs() - Replaces Tab character With Spaces
8.String find()- Returns the index of first occurrence of substring
9.String format() - Formats string into nicer output
10.String format_map() - Formats the String Using Dictionary
11.String index() - Returns Index of Substring
12.String isalnum() - Checks Alphanumeric Character
13.String isalpha ()- Checks if All Characters are Alphabets
14.String is decimal()- Checks Decimal Characters
15.String is digit() - Checks Digit Characters
16.String is identifier()- Checks for Valid Identifier
17.String is lower()- Checks if all Alphabets in a String are Lowercase.

18.String is numeric()- Checks Numeric Characters
19.String is printable()- Checks Printable Character
20.String is space()- Checks Whitespace Characters
21.String is title ()- Checks for Title cased String.
22.String is upper() - Returns if all characters are uppercase characters.
23.String join () - Returns a Concatenated String
24.String l just () - Returns left-justified string of given width
25.String lower () - Returns lowercased string
26.String l strip ()- Removes Leading Characters
27.String make trans () - Returns a translation table.
28.String partition () - Returns a Tuple
29.String replace ()- Replaces Substring Inside
30.String r find () - Returns the Highest Index of Substring
31.String r index ()- Returns Highest Index of Substring
32.String r just () - Returns right-justified string of given width.
33.String r partition () - Returns a Tuple
34.String r split ()- Splits String from Right
35.String r strip ()- Removes Trailing Characters
36.String split () - Splits string into a list of substrings
37.String splitlines ()- Splits String at Line Boundaries
38.String starts with ()- Checks if String Starts with the Specified String
39.String strip () - Removes both leading and trailing characters.
40.String swapcase () - Swap uppercase characters to lowercase; vice versa.
41.String title ()- Returns a Title Cased String
42.String translate () - Returns mapped charactered string.
43.String upper () - Returns uppercased string

SAMPLE PROGRAM

```
# Define a sample string
s = "hello world"
# capitalize() - Converts first character to Capital Letter
print("capitalize():", s.capitalize())
# casefold() - Converts to case folded strings
print("casefold():", s.casefold())
# center() - Pads string with specified character
print("center():", s.center(20, '*'))
# count() - Returns occurrences of substring in string
print("count('l'):", s.count('l'))
# encode() - Returns encoded string of given string
print("encode():", s.encode())
```

```
# endswith() - Checks if String Ends with the Specified Suffix
print("endswith('world'):", s.endswith('world'))
# expandtabs() - Replaces Tab character With Spaces
print("expandtabs():", s.expandtabs(4))
# find() - Returns the index of first occurrence of substring
print("find('world'):", s.find('world'))
# format() - Formats string into nicer output
print("format():", "My name is {} and I am {} years old.".format("John", 30))
# format_map() - Formats the String Using Dictionary
point = {'x': 4, 'y': -5}
print("format_map():", '{x} {y}'.format_map(point))
# index() - Returns Index of Substring
print("index('world'):", s.index('world'))
# isalnum() - Checks Alphanumeric Character
print("isalnum():", s.isalnum())
# isalpha() - Checks if All Characters are Alphabets
print("isalpha():", s.isalpha())
# isdecimal() - Checks Decimal Characters
print("isdecimal():", s.isdecimal())
# isdigit() - Checks Digit Characters
print("isdigit():", s.isdigit())
# isidentifier() - Checks for Valid Identifier
print("isidentifier():", s.isidentifier())
# islower() - Checks if all Alphabets in a String are Lowercase.
print("islower():", s.islower())
# isnumeric() - Checks Numeric Characters
print("isnumeric():", s.isnumeric())
# isprintable() - Checks Printable Character
print("isprintable():", s.isprintable())
# isspace() - Checks Whitespace Characters
print("isspace():", s.isspace())
# istitle() - Checks for Title cased String.
print("istitle():", s.istitle())
# isupper() - Returns if all characters are uppercase characters.
print("isupper():", s.isupper())
# join() - Returns a Concatenated String
print("join():", '-'.join(s))
# ljust() - Returns left-justified string of given width
print("ljust():", s.ljust(20, '*'))
```

```
# lower() - Returns lowercased string
print("lower():", s.lower())
# lstrip() - Removes Leading Characters
print("lstrip():", s.lstrip('he'))
# maketrans() - Returns a translation table.
print("maketrans():", str.maketrans('abc', '123'))
# partition() - Returns a Tuple
print("partition():", s.partition(' '))
# replace() - Replaces Substring Inside
print("replace():", s.replace('world', 'python'))
# rfind() - Returns the Highest Index of Substring
print("rfind('world'):", s.rfind('world'))
# rindex() - Returns Highest Index of Substring
print("rindex('world'):", s.rindex('world'))
# rjust() - Returns right-justified string of given width.
print("rjust():", s.rjust(20, '*'))
# rpartition() - Returns a Tuple
print("rpartition():", s.rpartition(' '))
# rsplit() - Splits String from Right
print("rsplit():", s.rsplit(' '))
# rstrip() - Removes Trailing Characters
print("rstrip():", s.rstrip('d'))
# split() - Splits string into a list of substrings
print("split():", s.split())
# splitlines() - Splits String at Line Boundaries
print("splitlines():", "hello\nworld".splitlines())
# startswith() - Checks if String Starts with the Specified String
print("startswith('hello'):", s.startswith('hello'))
# strip() - Removes both leading and trailing characters.
print("strip():", s.strip('h'))
# swapcase() - Swap uppercase characters to lowercase; vice versa.
print("swapcase():", s.swapcase())
# title() - Returns a Title Cased String
print("title():", s.title())
# translate() - Returns mapped charactered string.
print("translate():", s.translate(str.maketrans('aeiou', '12345')))
# upper() - Returns uppercased string
print("upper():", s.upper())
```

OUTPUT

```
capitalize(): Hello world
casefold(): hello world
center(): ****hello world*****
count('l'): 3
encode(): b'hello world'
endswith('world'): True
expandtabs(): hello   world
find('world'): 6
format(): My name is John and I am 30 years old.
format_map(): 4 -5
index('world'): 6
isalnum(): False
isalpha(): False
isdecimal(): False
isdigit(): False
isidentifier(): False
islower(): True
isnumeric(): False
isprintable(): True
isspace(): False
istitle(): False
isupper(): False
join(): h-e-l-l-o- -w-o-r-l-d
ljust(): hello world*********
lower(): hello world
lstrip(): llo world
maketrans(): {97: 49, 98: 50, 99: 51}
partition(): ('hello', ' ', 'world')
replace(): hello python
rfind('world'): 6
rindex('world'): 6
rjust(): *********hello world
rpartition(): ('hello', ' ', 'world')
rsplit(): ['hello', 'world']
rstrip(): hello worl
split(): ['hello', 'world']
splitlines(): ['hello', 'world']
startswith('hello'): True
```

```
strip(): ello worl
swapcase(): HELLO WORLD
title(): Hello World
translate(): h2ll4 w4rld
upper(): HELLO WORLD
```

2.1.4 TEXT

Python Program for Text Processing:

This program demonstrates:

- **Basic text operations**: Finding words, counting characters, and modifying text.
- **Control structures**: Loops and conditional statements to process the text

PROGRAM FOR TEXT PROCESSING

```
# Sample text for processing
text = """
Financial services companies provide a wide range of services to individuals and businesses, including banking, insurance, and investment management.
Some well-known companies include JPMorgan Chase, Goldman Sachs, and Bank of America.
"""
# 1. Word and character counting
def text_statistics(text):
   # Removing extra spaces and splitting text into words
   words = text.split()
   num_words = len(words)  # Count of words
   num_chars = len(text)  # Count of characters (including spaces)
   return num_words, num_chars
# 2. Check if certain words are present in the text
def word_search(text, word):
   # Selection statement to check if the word is present
   if word.lower() in text.lower():
      return f"'{word}' is found in the text."
   else:
      return f"'{word}' is NOT found in the text."
# 3. Modify the text by replacing a word
```

```
def replace_word(text, old_word, new_word):
    modified_text = text.replace(old_word, new_word)
    return modified_text
# 4. Loop through the text to find the occurrence of words longer than 7 characters
def find_long_words(text):
    words = text.split()
    long_words = []
    # Using a for loop to find words longer than 7 characters
    for word in words:
        if len(word) > 7:  # Selection within loop
            long_words.append(word)
    return long_words
# Main program demonstrating text operations
if __name__ == "__main__":
    # Getting text statistics
    word_count, char_count = text_statistics(text)
    print(f"Word count: {word_count}, Character count: {char_count}\n")
    # Searching for specific words in the text
    word_to_search = "banking"
    print(word_search(text, word_to_search))
    # Replacing a word in the text
    old_word = "JPMorgan Chase"
    new_word = "Morgan Stanley"
    modified_text = replace_word(text, old_word, new_word)
    print(f"\nModified Text:\n{modified_text}\n")
    # Finding and displaying long words
    long_words = find_long_words(text)
    print(f"Words longer than 7 characters: {long_words}")
```

OUTPUT

```
Word count: 37, Character count: 279
'banking' is found in the text.
Modified Text:
Financial services companies provide a wide range of services to individuals and
businesses, including banking, insurance, and investment management.
Some well-known companies include Morgan Stanley, Goldman Sachs, and Bank
of America.
Words longer than 7 characters: ['Financial', 'services', 'companies', 'businesses',
'including', 'insurance', 'investment', 'management', 'well-known', 'companies',
'Goldman', 'America']
```

Let Us Sum Up

In this unit, we delve into the core concepts of strings and text file manipulation in Python. Initially, we explore methods for accessing characters and substrings within strings, essential for various string manipulation tasks. Furthermore, we discuss data encryption techniques, highlighting the importance of secure data handling practices. Additionally, we cover the interplay between strings and number systems, facilitating conversions and manipulations between different numerical representations within string contexts. Throughout the unit, we delve into a plethora of string methods, empowering learners with the tools to manipulate and transform textual data efficiently. Overall, this unit equips students with fundamental skills in string manipulation and text file handling, laying a solid foundation for more advanced topics in Python programming.

Check Your Progress

1. How do you access the first character of a string in Python?
   A) s[0]
   B) s[1]
   C) s[-1]
   D) s[first]
2. Which method in Python is used for data encryption?
   A) encrypt()
   B) encode()
   C) decrypt()
   D) cipher()
3. In Python, which method is used to convert a string to lowercase?
   A) to_lower()
   B) lower()
   C) convert_lower()
   D) casefold()
4. What is the purpose of the ord() function in Python?
   A) Converts a string to uppercase
   B) Converts a string to lowercase
   C) Returns the Unicode code point for a character
   D) Returns the ASCII value for a character

5. Which method in Python is used to check if a string contains only numeric characters?
    A) isnumeric()
    B) isdigit()
    C) isnumber()
    D) isnumericstr()
6. What does the split() method in Python do?
    A) Joins two strings together
    B) Splits a string into a list of substrings
    C) Replaces characters in a string
    D) Removes leading and trailing characters
7. How do you access the last character of a string in Python?
    A) s[-1]
    B) s[0]
    C) s[length-1]
    D) s[last]
8. Which method in Python is used to remove leading and trailing whitespace characters from a string?
    A) trim()
    B) rstrip()
    C) lstrip()
    D) strip()
9. What is the output of the join() method in Python?
    A) Returns a concatenated string
    B) Splits a string into a list of substrings
    C) Removes leading and trailing characters
    D) Returns the index of the first occurrence of a substring
10. Which method is used to pad a string with a specified character to a certain length in Python?
    A) pad()
    B) fill()
    C) center()
    D) padding()
11. In Python, which method is used to check if a string starts with a specified substring?
    A) startswith()
    B) start()
    C) beginwith()
    D) begin()

12. What is the purpose of the format() method in Python?
    A) Formats a string into a specified width
    B) Converts a string to uppercase
    C) Formats a string into nicer output
    D) Removes leading and trailing characters
13. Which method in Python is used to replace occurrences of a substring inside a string?
    A) replace()
    B) substitute()
    C) swap()
    D) sub()
14. How do you convert a string to uppercase in Python?
    A) to_upper()
    B) uppercase()
    C) convert_upper()
    D) upper()
15. What is the purpose of the strip() method in Python?
    A) Converts a string to lowercase
    B) Removes both leading and trailing characters
    C) Splits a string into a list of substrings
    D) Returns the index of the first occurrence of a substring
16. In Python, which method is used to check if a string ends with a specified suffix?
    A) endwith()
    B) ends()
    C) end()
    D) endswith()
17. What does the count() method in Python do?
    A) Returns occurrences of a substring in a string
    B) Joins two strings together
    C) Splits a string into a list of substrings
    D) Removes leading and trailing characters
18. Which method in Python is used to convert a string to title case?
    A) title()
    B) to_title()
    C) convert_title()
    D) case_title()
19. How do you remove leading characters from a string in Python?
    A) strip()

B) lstrip()
C) rstrip()
D) trim()

20. What is the output of the find() method in Python?
A) Returns occurrences of a substring in a string
B) Replaces occurrences of a substring inside a string
C) Returns the index of the first occurrence of a substring
D) Checks if a string starts with a specified substring

SECTION 2.2: LISTS AND DICTIONARIES

2.2.1 LIST

In Python, a list is a versatile and mutable collection of items. It's one of the built-in data structures and is commonly used to store a collection of related items. Lists can contain elements of different data types, including integers, floats, strings, and even other lists. They are defined by enclosing a comma-separated sequence of items within square brackets [].

Here's a basic example of creating a list:

```
my_list = [1, 2, 3, 4, 5]
```

Lists are mutable, meaning can change, add, or remove elements after the list is created. Here are some common operations you can perform on lists:

- **Accessing Elements**: Access individual elements in a list using indexing. Indexing starts at 0 for the first element.
  ```
  first_element = my_list[0] # Access the first element
  ```
- **Slicing**: You can extract a sub list (a slice) from a list using slicing notation.
  ```
  sub list = my_list [1:4]  # Extract elements from index 1 to index 3 (inclusive)
  ```
- **Appending and Extending**: Add elements to the end of a list using the append () method or extend a list with the elements of another list using the extend() method.
  ```
  my_list.append(6)         # Adds 6 to the end of the list.
  my_list.extend([7, 8, 9]) # Extends the list with [7, 8, 9]
  ```

- **Inserting Elements**: Insert elements at a specific position in the list using the insert () method.

  my_list.insert(2, 'hello') # Inserts 'hello' at index 2

- **Removing Elements**: Remove elements by value using the remove () method, by index using the pop() method, or by clearing the entire list using the clear() method.

  my_list.remove(3) # Removes the first occurrence of 3 from the list

  popped_element = my_list.pop(1) # Removes and returns the element at index 1

  my_list.clear() # Removes all elements from the list

- **Other Operations**: Find the length of a list using the len() function, check if an item is in a list using the in keyword, and more.

Lists are fundamental in Python and are used extensively in various programming tasks due to their flexibility and ease of use.

Python Program Demonstrating List Operations:

```
# Initialize a list
my_list = [10, 20, 30, 40, 50]
# 1. Accessing elements
print("Element at index 2:", my_list[2])  # Output: 30
# 2. Slicing
print("Sliced list (index 1 to 3):", my_list[1:4])  # Output: [20, 30, 40]
# 3. Append: Add an element to the end
my_list.append(60)
print("After appending 60:", my_list)  # Output: [10, 20, 30, 40, 50, 60]
# 4. Insert: Insert at index 2
my_list.insert(2, 25)
print("After inserting 25 at index 2:", my_list)  # Output: [10, 20, 25, 30, 40, 50,
60]
# 5. Extend: Add multiple elements
my_list.extend([70, 80])
print("After extending with [70, 80]:", my_list)  # Output: [10, 20, 25, 30, 40, 50,
60, 70, 80]
# 6. Remove: Remove the first occurrence of 40
my_list.remove(40)
print("After removing 40:", my_list)  # Output: [10, 20, 25, 30, 50, 60, 70, 80]
# 7. Pop: Remove the element at index 3
```

```
removed_element = my_list.pop(3)
print("Popped element at index 3:", removed_element)  # Output: 30
print("List after popping:", my_list)  # Output: [10, 20, 25, 50, 60, 70, 80]
*# 8. Index: Find the index of element 60
index_60 = my_list.index(60)
print("Index of element 60:", index_60)  # Output: 4
# 9. Count: Count occurrences of 70
count_70 = my_list.count(70)
print("Count of 70 in the list:", count_70)  # Output: 1
# 10. Sort: Sort the list
my_list.sort()
print("Sorted list:", my_list)  # Output: [10, 20, 25, 50, 60, 70, 80]
# 11. Reverse: Reverse the list
my_list.reverse()
print("Reversed list:", my_list)  # Output: [80, 70, 60, 50, 25, 20, 10]
# 12. Clear: Remove all elements from the list
my_list.clear()
print("List after clearing:", my_list)  # Output: []
```

OUTPUT

```
Element at index 2: 30
Sliced list (index 1 to 3): [20, 30, 40]
After appending 60: [10, 20, 30, 40, 50, 60]
After inserting 25 at index 2: [10, 20, 25, 30, 40, 50, 60]
After extending with [70, 80]: [10, 20, 25, 30, 40, 50, 60, 70, 80]
After removing 40: [10, 20, 25, 30, 50, 60, 70, 80]
Popped element at index 3: 30
List after popping: [10, 20, 25, 50, 60, 70, 80]
Index of element 60: 4
Count of 70 in the list: 1
Sorted list: [10, 20, 25, 50, 60, 70, 80]
Reversed list: [80, 70, 60, 50, 25, 20, 10]
List after clearing: []
```

2.2.2 DICTIONARIES

Python dictionaries are unordered collections of key-value pairs. They are used to store and retrieve data efficiently, providing fast lookup times for keys. Dictionaries are mutable, meaning their elements can be modified after creation. Keys in

dictionaries must be unique, and they are typically immutable data types like strings or numbers, while values can be of any data type.

Basic Operations:

Creating a Dictionary:

```
# Empty dictionary
my_dict = {}
# Dictionary with initial values
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

Accessing Elements:

```
# Accessing value using key
print(my_dict['name'])  # Output: John
# Using get() method (returns None if key doesn't exist)
print(my_dict.get('age'))  # Output: 30
```

Adding or Modifying Elements:

```
# Adding a new key-value pair
my_dict['gender'] = 'Male'
# Modifying value for an existing key
my_dict['age'] = 35
```

Removing Elements:

```
# Removing a key-value pair
del my_dict['city']
# Removing all key-value pairs
my_dict.clear()
```

Iterating Over a Dictionary:

```
# Iterating over keys
for key in my_dict:
    print(key)
# Iterating over values
for value in my_dict.values():
    print(value)
# Iterating over key-value pairs
for key, value in my_dict.items():
    print(key, value)
```

Checking Membership:

```
# Checking if a key exists
if 'name' in my_dict:
    print("Key 'name' exists")
```

DICTIONARY METHODS:

keys(): Returns a view of all keys in the dictionary.
values(): Returns a view of all values in the dictionary.
items(): Returns a view of all key-value pairs in the dictionary.
update(): Updates the dictionary with the key-value pairs from another dictionary or iterable.
pop(): Removes and returns the value for a given key.
popitem(): Removes and returns the last inserted key-value pair.

Example :

```
# Creating a dictionary
student = {'name': 'Alice', 'age': 20, 'major': 'Computer Science'}
# Accessing elements
print(student['name'])  # Output: Alice
# Adding a new key-value pair
student['gpa'] = 3.8
# Modifying value for an existing key
student['age'] = 21
# Removing a key-value pair
del student['major']
# Iterating over key-value pairs
for key, value in student.items():
    print(key, value)
```

Dictionaries are versatile data structures in Python, widely used for various purposes such as storing configuration settings, caching data, and representing structured data. Understanding how to effectively work with dictionaries is essential for Python developers.

```
# Initialize a dictionary
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# 1. Accessing elements
print("Name:", my_dict['name'])  # Output: 'Alice'
# 2. Adding/Updating elements
my_dict['age'] = 26  # Update the value for 'age'
my_dict['email'] = 'alice@example.com'  # Add a new key-value pair
print("Updated dictionary:", my_dict)
# 3. Removing elements
del my_dict['city']  # Remove 'city' from the dictionary
print("After removing 'city':", my_dict)
```

```
# 4. Get all keys
keys = my_dict.keys()
print("Keys:", keys)
# 5. Get all values
values = my_dict.values()
print("Values:", values)
# 6. Get all key-value pairs
items = my_dict.items()
print("Key-value pairs:", items)
# 7. Check key existence
key_exists = 'email' in my_dict
print("Is 'email' key present?:", key_exists)
# 8. Length of the dictionary
length = len(my_dict)
print("Number of key-value pairs:", length)
# 9. Clear the dictionary
my_dict.clear()
print("Dictionary after clearing:", my_dict)
# Reinitialize for further operations
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# 10. Pop a key and return its value
age = my_dict.pop('age')
print("Popped 'age':", age)
print("After popping 'age':", my_dict)
# 11. Pop the last inserted item
last_item = my_dict.popitem()
print("Popped last item:", last_item)
print("After popping last item:", my_dict)
# 12. Update with another dictionary
my_dict.update({'name': 'Bob', 'age': 30})
print("After updating with new data:", my_dict)
```

OUTPUT

```
Name: Alice
Updated dictionary: {'name': 'Alice', 'age': 26, 'email': 'alice@example.com'}
After removing 'city': {'name': 'Alice', 'age': 26, 'email': 'alice@example.com'}
Keys: dict_keys(['name', 'age', 'email'])
Values: dict_values(['Alice', 26, 'alice@example.com'])
Key-value pairs: dict_items([('name', 'Alice'), ('age', 26), ('email', 'alice@example.com')])
```

Is 'email' key present?: True
Number of key-value pairs: 3
Dictionary after clearing: {}
Popped 'age': 25
After popping 'age': {'name': 'Alice', 'city': 'New York'}
Popped last item: ('city', 'New York')
After popping last item: {'name': 'Alice'}
After updating with new data: {'name': 'Bob', 'age': 30}

Let Us Sum Up

In Python, lists are ordered collections of items, while dictionaries are unordered collections of key-value pairs. Lists are mutable, allowing for dynamic changes, while dictionaries are mutable as well, but their keys must be immutable. Both lists and dictionaries offer versatile data storage and retrieval options. Lists maintain elements in the order they were added, accessible via indices, while dictionaries enable quick lookup of values using unique keys. Lists are denoted by square brackets [ ], whereas dictionaries use curly braces { }. Understanding their differences and applications enhances data manipulation and program efficiency.

Check Your Progress

1. What is the primary difference between a list and a dictionary in Python?
   A) Lists are ordered collections, while dictionaries are unordered.
   B) Lists can store only integers, while dictionaries can store any data type.
   C) Lists can be accessed using keys, while dictionaries can be accessed using indices.
   D) Lists can only contain single elements, while dictionaries can contain key-value pairs.
2. Which of the following operations is NOT supported by dictionaries in Python?
   A) Adding new key-value pairs
   B) Removing key-value pairs
   C) Accessing elements by index
   D) Updating existing key-value pairs
3. How do you access a specific element in a list by its index in Python?
   A) Using the get() method
   B) Using square brackets [] notation
   C) Using the find() method
   D) Using the access() method

4. What happens if you try to access a key in a dictionary that does not exist?
   A) It returns None
   B) It raises a KeyError exception
   C) It returns an empty dictionary
   D) It creates a new key with a None value
5. Which method in Python is used to add a new element to the end of a list?
   A) append()
   B) insert()
   C) add()
   D) extend()
6. How do you remove the last element from a list in Python?
   A) Using the remove() method
   B) Using the pop() method with no arguments
   C) Using the pop() method with the index of the last element
   D) Using the delete() method
7. What is the time complexity for accessing an element by index in a list in Python?
   A) O(1)
   B) O(log n)
   C) O(n)
   D) O(n^2)
8. Which of the following is a valid way to create an empty dictionary in Python?
   A) dict = {}
   B) dict = {[]}
   C) dict = ()
   D) dict = []
9. How do you check if a key exists in a dictionary in Python?
   A) Using the contains() method
   B) Using the exists() method
   C) Using the in keyword
   D) Using the has_key() method
10. What does the keys() method return for a dictionary in Python?
    A) All the values in the dictionary
    B) All the keys in the dictionary
    C) The length of the dictionary
    D) The maximum key in the dictionary
11. How do you remove a key-value pair from a dictionary in Python?
    A) Using the remove() metho
    B) Using the pop() method with the key as an argument

   C) Using the delete() method
   D) Using the clear() method
12. Which method in Python is used to merge two dictionaries?
   A) merge()
   B) combine()
   C) concat()
   D) update()
13. What is the output of the len() function when called on a dictionary in Python?
   A) The number of keys in the dictionary
   B) The number of values in the dictionary
   C) The total number of elements in the dictionary
   D) The maximum key in the dictionary
14. What does the values() method return for a dictionary in Python?
   A) All the keys in the dictionary
   B) All the values in the dictionary
   C) The length of the dictionary
   D) The minimum value in the dictionary
15. How do you check if a value exists in a dictionary in Python?
   A) Using the contains_value() method
   B) Using the in keyword with the dictionary
   C) Using the has_value() method
   D) Using the exists() method
16. Which method in Python is used to get a list of all key-value pairs in a dictionary?
   A) items()
   B) pairs()
   C) keys()
   D) values()
17. What is the time complexity for adding a new key-value pair to a dictionary in Python?
   A) O(1)
   B) O(log n)
   C) O(n)
   D) O(n^2
18. How do you update the value of a specific key in a dictionary in Python?
   A) Using the set() method
   B) Using the change() method
   C) Using square brackets [] notation
   D) Using the update() method

19. Which of the following is a valid way to create a list with initial values in Python?
    A) list = []
    B) list = {}
    C) list = [1, 2, 3]
    D) list = ()
20. What is the time complexity for removing a key-value pair from a dictionary in Python?
    A) O(1)
    B) O(log n)
    C) O(n)
    D) O(n^2)

SECTION 2.3: DESIGN WITH FUNCTIONS

2.3.1- DESIGN WITH FUNCTIONS

"Design with Functions" in Python refers to a programming paradigm or approach where you design your code structure around functions. This approach emphasizes breaking down a program into smaller, reusable functions, each responsible for performing a specific task. Here's a breakdown of what it involves:

1. **Modularity**: The code is organized into modular units, with each unit encapsulating a specific functionality. These units are often implemented as functions, allowing you to isolate and debug specific parts of your code more easily.
2. **Reusability**: Functions can be reused across different parts of your program or even in different programs altogether. By designing functions with a clear purpose and scope, you can avoid duplicating code and promote code reuse.
3. **Readability**: Breaking down the code into smaller functions can improve readability and maintainability. Each function should ideally perform a single, well-defined task, making it easier for other developers (including your future self) to understand the code.
4. **Testing and Debugging**: With a modular design, it becomes easier to test and debug your code. You can test each function independently, providing specific inputs and checking the outputs, which helps in identifying and fixing bugs more efficiently.

5. **Scalability**: As your program grows in complexity, a function-based design can help manage that complexity by organizing code into smaller, manageable pieces. This makes it easier to extend and maintain the codebase over time.

Here's a simple example to illustrate how you might design a program with functions in Python:

```
def calculate_area(radius):
    """Calculate the area of a circle."""
    return 3.14 * radius * radius.
 def calculate_volume(radius, height):
    """Calculate the volume of a cylinder."""
    base_area = calculate_area(radius)
    return base_area * height
def main ():
    """Main function to demonstrate the use of functions."""
    radius = float(input("Enter the radius of the cylinder: "))
    height = float(input("Enter the height of the cylinder: "))
    volume = calculate_volume(radius, height)
    print("The volume of the cylinder is:", volume)
if __name__ == "__main__":
    main ()
```

In this example, the code is organized around three functions: calculate_area, calculate_volume, and main. Each function has a specific purpose, making the code easier to understand, test, and maintain. The main function acts as the entry point to the program, orchestrating the execution by calling other functions as needed. Overall, designing with functions in Python promotes code organization, reusability, and maintainability, leading to more robust and scalable software solutions.

2.3.2 PROBLEM SOLVING WITH TOP DOWN DESIGN

Top-Down Design in Python follows the same principles described earlier, but it involves implementing the design using Python's syntax and features. Let's walk through an example of implementing a simple program to calculate the total cost of a shopping cart using a top-down design approach in Python:

1. **High-Level Decomposition**:
   - ✓ Define the main function calculate_total_cost.

- ✓ Identify sub-tasks: get_item_prices, calculate_subtotal, apply_discounts, calculate_tax, calculate_total.

2. **Submodule Decomposition**:
   - ✓ get_item_prices: Prompt the user to input prices for each item.
   - ✓ calculate_subtotal: Add up the prices to get the subtotal.
   - ✓ apply_discounts: Apply any discounts to the subtotal.
   - ✓ calculate_tax: Calculate the tax based on the subtotal.
   - ✓ calculate_total: Add tax to the discounted subtotal to get the total cost.

Example:

```
def get_item_prices():
    """Prompt the user to input prices for each item."""
    prices = []
    num_items = int(input("Enter the number of items: "))
    for i in range(num_items):
        price = float (input (f"Enter the price of item {i+1}: "))
        prices. append(price)
    return prices.
def calculate_subtotal(prices):
    """Calculate the subtotal."""
    return sum(prices)
def apply_discounts (subtotal, discount):
    """Apply any discounts to the subtotal."""
    return subtotal * (1 - discount)
def calculate_tax (subtotal, tax_rate):
    """Calculate the tax based on the subtotal."""
    return subtotal * tax_rate
def calculate_total_cost():
    """Calculate the total cost of the shopping cart."""
    prices = get_item_prices()
    subtotal = calculate_subtotal(prices)
    subtotal_after_discount = apply_discounts(subtotal, 0.1) # Assuming a 10%
discount
    tax = calculate_tax (subtotal_after_discount, 0.08) # Assuming an 8% tax rate
    total_cost = subtotal_after_discount + tax
    return total_cost
def main ():
    """Main function to execute the program."""
    total_cost = calculate_total_cost()
    print("Total cost of the shopping cart:", total_cost)
```

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

1. **Integration**:
   - ✓ Functions are called within the calculate_total_cost function to perform the calculations.
2. **Testing and Debugging**:
   - ✓ Test each function independently and then test the integrated solution with different inputs to ensure correctness.

This example demonstrates how top-down design can be implemented in Python by breaking down a problem into smaller, manageable tasks and implementing each task as a separate function. This approach leads to modular, readable, and maintainable code.

2.3.3 DESIGN WITH RECURSIVE FUNCTIONS

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

The following image shows the working of a recursive function called recurse.

```
def recurse():
    ...
    recurse()
    ...

recurse()
```

recursive call

Example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example:

```
def factorial(x):
    """This is a recursive function.
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
```

```
        return (x * factorial(x-1))
num = 3
print ("The factorial of", num, "is", factorial(num))
Out Put:
The factorial of 3 is 6
```

In the above example, factorial () is a recursive function as it calls itself.

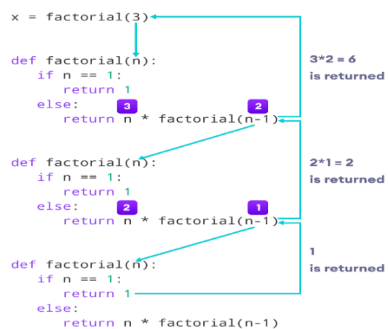
When we call this function with a positive integer, it will recursively call itself by decreasing the number. Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial (3)            # 1st call with 3
3 * factorial (2)        # 2nd call with 2
3 * 2 * factorial (1)    # 3rd call with 1
3 * 2 * 1                # return from 3rd call as number=1
3 * 2                    # return from 2nd call
6                        # return from 1st call
```

Let's look at an image that shows a step-by-step process of what is going on:

Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.



By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in Recursion Error. Let's look at one such condition.

Example:

```
def recursor():
    recursor()
     recursor()
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded.
```

2.3.4 MANAGING A PROGRAM'S NAMESPACE

In Python, a namespace is a mapping from names to objects. It's like a dictionary that associates variable names with objects (such as variables, functions, classes, modules, etc.). Managing a program's namespace involves understanding how names are bound to objects and how you can manipulate these bindings within your program.

Here are some key aspects of managing a program's namespace in Python:

1. **Local Namespace**: Each function call in Python creates its own local namespace, which contains the names defined within that function. These names are local to the function and are not accessible outside of it.
2. **Global Namespace**: The global namespace contains the names defined at the top level of a module or script. These names are accessible from any part of the module or script.
3. **Built-in Namespace**: Python also has a built-in namespace that contains the names of built-in functions, exceptions, and other objects provided by the Python interpreter. These names are always available without the need for importing.
4. **Name Resolution**: When you reference a name in Python, the interpreter looks for that name first in the local namespace, then in the global namespace, and

finally in the built-in namespace. If the name is not found in any of these namespaces, a NameError is raised.

5. **Scope**: Scope refers to the region of code where a name is valid and accessible. Variables defined within a function have local scope and are only accessible within that function. Variables defined at the top level of a module have global scope and are accessible throughout the module.
6. **global and nonlocal Keywords**: In Python, you can use the global keyword inside a function to indicate that a variable should be treated as global, even if it's assigned a value within the function. Similarly, the nonlocal keyword allows you to modify variables in the outer (enclosing) scope within a nested function.

Here's a simple example demonstrating the management of namespaces in Python:

```
# Global namespace
global_variable = 10
def my_function ():
# Local namespace
    local_variable = 20
    print ("Inside my_function:")
    print ("Local variable:", local_variable)
    print ("Global variable:", global_variable)
# Accessing global variable
print ("Outside my_function (before calling):")
print ("Global variable:", global_variable)
# Calling the function
my_function ()
# Accessing global variable again
print ("Outside my_function (after calling):")
print ("Global variable:", global_variable)
```

In this example, global_variable is defined in the global namespace and is accessible from both inside and outside the function my_function. However, local_variable is defined only within the function's local namespace and is not accessible outside the function.

2.3.5 HIGHER ORDER FUNCTIONS

In Python, a higher-order function is a function that takes one or more functions as arguments or returns a function as its result. Essentially, it treats functions as first-class citizens, allowing them to be manipulated and passed around like any other value.

Here are some key points about higher-order functions in Python:

1. **Functions as Arguments**: Higher-order functions can accept other functions as arguments. This enables powerful abstractions and allows you to parameterize behaviour by passing functions as arguments.
2. **Functions as Return Values**: Higher-order functions can also create and return functions dynamically. This is particularly useful for creating functions tailored to specific contexts or configurations.
3. **Abstraction**: Higher-order functions promote abstraction by separating concerns and allowing you to express complex behaviour in terms of simpler functions.
4. **Common Higher-Order Functions**: Python's standard library provides several higher-order functions, such as map(), filter(), reduce(), and sorted(), which take functions as arguments to apply operations on iterables.
5. **Functional Programming**: Higher-order functions are a fundamental concept in functional programming, a programming paradigm that emphasizes the use of functions as building blocks for creating software systems.

Here's a simple example to illustrate higher-order functions in Python:

Example:

```
def apply_operation(func, x, y):
    """Apply a binary operation function to two operands."""
    return func (x, y)
```

```
def add (x, y):
    """Add two numbers."""
    return x + y
def subtract (x, y):
    """Subtract one number from another."""
    return x - y
```

Using apply_operation with different operations

```
result1 = apply_operation (add, 5, 3)       # Equivalent to add (5, 3)
result2 = apply_operation (subtract, 10, 4) # Equivalent to subtract (10,
4)
print ("Result of addition:", result1)      # Output: 8
print ("Result of subtraction:", result2)   # Output: 6
```

In this example, apply_operation is a higher-order function because it takes another function (add or subtract) as an argument. Depending on which function is passed to apply_operation, it applies that operation to the given operands x and y.

Higher-order functions are powerful tools for writing expressive and modular code in Python, enabling you to create flexible and reusable components. They are widely used in functional programming paradigms and can lead to more concise and elegant solutions to various programming problems.

Let Us Sum Up

In the realm of functional programming, understanding the core concepts is pivotal. Designing with functions involves breaking down complex tasks into smaller, manageable functions, aiding in modularity and reusability. Employing top-down design enables tackling larger problems by breaking them into smaller, more solvable sub-problems. Recursive functions offer elegant solutions by calling themselves within their definition, often used in scenarios like tree traversal or factorial computation. Managing a program's namespace involves handling variable scopes, ensuring clarity and avoiding naming conflicts. Higher-order functions elevate functionality by treating

functions as first-class citizens, allowing operations like passing functions as arguments or returning them from other functions. Mastering these concepts enriches programming practices, fostering efficient and scalable code development.

CHECK YOUR PROGRESS

1. What is the primary advantage of designing with functions in programming?
   - A) Improved program speed
   - B) Enhanced program readability and organization
   - C) Decreased program memory usage
   - D) Simplified debugging process
2. What is top-down design primarily used for in programming?
   - A) Breaking down a problem into smaller, solvable sub-problems
   - B) Optimizing program performance
   - C) Reorganizing existing code
   - D) Generating random test cases
3. Which of the following best describes a recursive function?
   - A) A function that calls itself within its definition
   - B) A function with a complex algorithm
   - C) A function that takes no arguments
   - D) A function that returns a boolean value
4. What is namespace management in programming primarily concerned with?
   - A) Optimizing code execution speed
   - B) Avoiding memory leaks
   - C) Handling variable scopes and avoiding naming conflicts
   - D) Managing network resources
5. What are higher-order functions capable of doing in programming?
   - A) Treating functions as first-class citizens
   - B) Ignoring function calls
   - C) Running functions in parallel
   - D) Restricting function access

6. Which function design approach breaks down a problem into smaller, more manageable sub-problems?
   A) Bottom-up design
   B) Top-down design
   C) Recursive design
   D) Procedural design
7. In top-down design, what is the first step typically involved in?
   A) Defining the main program logic
   B) Implementing the smallest sub-problems
   C) Analyzing the problem statement
   D) Writing documentation
8. What characteristic distinguishes a recursive function from a non-recursive one?
   A) It must contain a loop
   B) It must call itself within its definition
   C) It must have a fixed number of arguments
   D) It must return a boolean value
9. What is the primary goal of managing a program's namespace?
   A) Reducing code redundancy
   B) Ensuring program portability
   C) Minimizing memory usage
   D) Preventing variable name conflicts
10. Which of the following functions is an example of a higher-order function?
   A) add_numbers(a, b)
   B) sort_list(list)
   C) map(function, iterable)
   D) calculate_average(list)
11. In top-down design, what is the process of dividing a large problem into smaller sub-problems called?
   A) Decomposition

B) Composition

C) Abstraction

D) Recursion

12. What is the primary purpose of a recursive function in programming?

A) To improve program performance

B) To simplify code structure

C) To handle repetitive tasks

D) To solve problems that can be broken down into smaller instances of the same problem

13. Which of the following statements best describes namespace in programming?

A) It defines the scope within which names can be referenced

B) It represents a specific memory location

C) It is a reserved keyword in Python

D) It stores the value of a variable

14. How does a higher-order function differ from a regular function?

A) It can only accept one argument

B) It can only return one value

C) It can accept other functions as arguments or return them

D) It cannot be called from another function

15. What does the acronym DRY stand for in programming?

A) Don't Repeat Yourself

B) Do Repeat Yourself

C) Duplicate Repeated Yield

D) Do Reuse Yourself

16. Which of the following is NOT a benefit of using functions in programming?

A) Improved code readability

B) Enhanced code modularity

C) Increased program speed

D) Simplified debugging process

17. In recursive functions, what is the base case?

A) The case where the function returns a value
B) The case where the function calls itself
C) The case where the function terminates without calling itself
D) The case where the function has no arguments

18. What is the primary purpose of a higher-order function?
A) To improve program efficiency
B) To reduce code redundancy
C) To treat functions as first-class citizens
D) To handle memory management

19. Which function design approach emphasizes solving the problem at the highest level first?
A) Bottom-up design
B) Recursive design
C) Procedural design
D) Top-down design

20. What does recursion involve in programming?
A) Repeating a sequence of instructions
B) Calling a function from within itself
C) Iterating through a list of elements
D) Breaking down a problem into smaller sub-problems

Unit Summary

In this unit, we delved into the fundamental concepts of Python programming, covering a diverse array of topics essential for building a solid understanding of the language. We began by exploring the intricacies of strings and text manipulation, learning how to effectively work with textual data through methods like concatenation, slicing, and formatting. Understanding these operations provided a crucial foundation for more complex data handling tasks. Moving forward, we delved into the versatile world of lists and dictionaries, two fundamental data structures in Python. With lists, we learned how to manage ordered collections of items, perform various operations

such as appending, inserting, and accessing elements by index. Dictionaries introduced us to the concept of key-value pairs, enabling efficient mapping and retrieval of data, essential for a wide range of applications. Functions emerged as a pivotal aspect of our journey, offering a modular approach to programming. Through functions, we encapsulated reusable blocks of code, enhancing code readability, modularity, and maintainability. We explored different aspects of function design, including parameter passing, return values, and the importance of proper function documentation. Moreover, we delved into problem-solving techniques, emphasizing strategies like top-down design and recursive thinking. These problem-solving methodologies equipped us with the tools to tackle complex programming challenges systematically, enabling efficient algorithmic thinking and program design.

Glossary

- **Strings:** Sequences of characters used to represent textual data in Python programs. Strings support various operations such as concatenation, slicing, and formatting.
- **Text Manipulation:** The process of modifying or extracting information from strings. Text manipulation techniques include converting case, splitting, stripping whitespace, and searching for substrings.
- **Lists:** Ordered collections of items in Python, allowing for the storage and manipulation of multiple elements. Lists support operations like appending, inserting, accessing elements by index, and slicing.
- **Dictionaries:** Data structures that store key-value pairs, enabling efficient mapping and retrieval of data. Dictionaries provide a flexible way to organize and access information based on unique keys.
- **Functions:** Reusable blocks of code that perform specific tasks. Functions enhance code modularity, readability, and reusability by encapsulating logic into named units.

- **Parameters:** Variables defined within the parentheses of a function definition, used to pass data into the function. Parameters enable functions to operate on different inputs dynamically.
- **Return Values:** Data or objects returned by a function after it completes its execution. Return values allow functions to communicate results or outputs back to the caller.
- **Modular Programming:** A programming paradigm that emphasizes breaking down complex systems into smaller, manageable modules or functions. Modular programming enhances code organization, maintainability, and scalability.
- **Top-Down Design:** A problem-solving approach where a complex problem is broken down into smaller, more manageable subproblems, which are further decomposed until they can be easily solved using code.
- **Recursive Functions:** Functions that call themselves within their definition, allowing for the solution of problems that can be broken down into smaller instances of the same problem.
- **Namespace:** A container that holds a set of identifiers, such as variable names and function names, and their corresponding objects. Namespaces prevent naming conflicts and facilitate code organization and management.
- **Higher-Order Functions:** Functions that take other functions as arguments or return functions as results. Higher-order functions enable powerful programming techniques like functional programming and callback mechanisms.

Self – Assessment Questions

1. Compare and contrast strings and lists in Python. How do their characteristics and operations differ?
2. Explain the importance of dictionaries in Python programming. Analyze scenarios where dictionaries are preferred over lists for data storage and retrieval.

3. Evaluate the role of functions in modular programming. How do functions enhance code organization and reusability?
4. Analyze the benefits of top-down design in problem-solving. How does breaking down complex problems into smaller subproblems facilitate program development?
5. Explain the concept of recursion in programming. Analyze scenarios where recursive functions are advantageous over iterative approaches.
6. Compare and contrast the del keyword and the pop() method for removing elements in Python dictionaries. When would you choose one over the other?
7. Evaluate the efficiency of different string manipulation methods in Python, such as split(), join(), and strip(). How do these methods impact code performance and readability?
8. Analyze the use of higher-order functions in Python. Provide examples of scenarios where higher-order functions offer advantages over traditional approaches.
9. Explain how namespaces are managed in Python programs. Analyze the implications of namespace conflicts and strategies to avoid them.
10. Compare the efficiency of list comprehension and traditional loops for creating lists in Python. Evaluate factors such as readability, performance, and memory usage.

Activities / Exercises / Case Studies

Activities

1. String Manipulation Challenge: Provide a set of string manipulation tasks where students need to perform operations like concatenation, splitting, stripping whitespace, and formatting to achieve specific outputs.
2. List Operations Lab: Create a lab activity where students practice various list operations such as appending, inserting, removing, and slicing elements. Provide real-world scenarios where these operations are applicable.

3. Dictionary Adventure: Design a scavenger hunt activity where students use dictionaries to navigate through a virtual world, retrieving information and solving puzzles along the way.
4. Function Design Workshop: Organize a workshop where students collaborate to design and implement functions for solving specific tasks. Encourage them to document their functions and discuss best practices for function design.
5. Problem-Solving Hackathon: Host a problem-solving hackathon where students work in teams to tackle coding challenges using top-down design and recursive thinking. Provide prizes for the most elegant and efficient solutions.

Exercises

1. String Practice Problems: Assign a set of practice problems involving string manipulation, including tasks like reversing strings, counting occurrences of substrings, and validating input formats.
2. List Challenges: Provide exercises where students must manipulate lists to perform tasks such as sorting elements, finding duplicates, and implementing algorithms like binary search or merge sort.
3. Dictionary Drills: Present exercises where students practice using dictionaries to solve real-world problems, such as building a contact management system or implementing a word frequency analyzer.
4. Function Implementation Tasks: Assign tasks that require students to implement specific functions to accomplish defined objectives. For example, create functions to calculate factorial, find prime numbers, or validate email addresses.
5. Algorithmic Exercises: Challenge students with algorithmic exercises that require recursive thinking, such as implementing Fibonacci sequence generation, tower of Hanoi problem, or depth-first search algorithm.

Case Studies

1. Inventory Management System: Present a case study where students design and implement an inventory management system using lists and dictionaries. They must create functions for adding, updating, and querying inventory items.
2. Text Processing Application: Provide a case study where students develop a text processing application capable of performing various string operations like word count, character frequency analysis, and text encryption/decryption.
3. Student Gradebook System: Describe a case study scenario where students build a student gradebook system using dictionaries to store student information and grades. They must implement functions for calculating averages, generating reports, and handling data updates.
4. Recursive Problem Solver: Challenge students with a case study involving the development of a recursive problem-solving tool. They must implement functions to solve classic recursive problems like factorial calculation, Fibonacci sequence generation, and binary search.

Answers for check your progress

Modules	S. No.	Answers
Module 1	1.	A) s[0]
	2.	A) encrypt()
	3.	B) lower()
	4.	D) Returns the ASCII value for a character
	5.	B) isdigit()
	6.	B) Splits a string into a list of substrings
	7.	A) s[-1]
	8.	D) strip()
	9.	A) Returns a concatenated string
	10.	C) center()

	11.	A) startswith()
	12.	C) Formats a string into nicer output
	13.	A) replace()
	14.	D) upper()
	15.	B) Removes both leading and trailing characters
	16.	D) endswith()
	17.	A) Returns occurrences of a substring in a string
	18.	A) title()
	19.	B) lstrip()
	20.	C) Returns the index of the first occurrence of a substring
Module 2	**1.**	A) **s[0]**
	2.	C) **decrypt()**
	3.	B) **lower()**
	4.	C) Returns the Unicode code point for a character
	5.	B) isdigit()
	6.	B) Splits a string into a list of substrings
	7.	A) s[-1]
	8.	D) strip()
	9.	A) Returns a concatenated string
	10.	C) center()
	11.	A) startswith()
	12.	C) Formats a string into nicer output
	13.	A) replace()
	14.	D) upper()
	15.	B) Removes both leading and trailing characters

	16.	D) endswith()
	17.	A) Returns occurrences of a substring in a string
	18.	A) title()
	19.	B) lstrip()
	20.	C) Returns the index of the first occurrence of a substring
Module 3.	**1.**	B) Enhanced program readability and organization
	2.	A) Breaking down a problem into smaller, solvable sub-problems
	3.	A) A function that calls itself within its definition
	4.	C) Handling variable scopes and avoiding naming conflicts
	5.	A) Treating functions as first-class citizens
	6.	B) Top-down design
	7.	C) Analyzing the problem statement
	8.	B) It must call itself within its definition
	9.	D) Preventing variable name conflicts
	10.	C) **map(function, iterable)**
	11.	A) Decomposition
	12.	D) To solve problems that can be broken down into smaller instances of the same problem
	13.	A) It defines the scope within which names can be referenced
	14.	C) It can accept other functions as arguments or return them
	15.	A) Don't Repeat Yourself
	16.	C) Increased program speed

	17.	C) The case where the function terminates without calling itself
	18.	C) To treat functions as first-class citizens
	19.	D) Top-down design
	20.	B) Calling a function from within itself

Suggested Readings

1. Ramalho, L. (2022). *Fluent python*. " O'Reilly Media, Inc.".
2. Sweigart, A. (2019). *Automate the boring stuff with Python: practical programming for total beginners*. no starch press.
3. Downey, A. B. (2003). *How to think like a computer scientist*.

Open-Source E-Content Links

1. https://docs.python.org/3/
2. https://www.w3schools.com/python/
3. **https://www.geeksforgeeks.org/python-programming-language-tutorial/**

References

1. "Python for Everybody" by Dr. Charles Severance
2. Coursera: Python for Everybody Specialization
3. Codecademy Python Course
4. Coursera: Python for Everybody Specialization

UNIT III – DESIGN WITH CLASSES

Unit III: Design with Classes: Getting inside Objects and Classes - Data-Modeling Examples - Building a New Data Structure - The Two - Dimensional Grid - Structuring Classes with Inheritance and Polymorphism-Graphical User Interfaces-The Behavior of terminal-Based programs and GUI-Based programs - Coding Simple GUI-Based programs - Windows and Window Components - Command Buttons and responding to events.

Design with Classes

Section	Topic	Page No.
	UNIT – III	
	Unit Objectives	
Section 3.1	**Design with Classes**	**77**
3.1.1	Getting Inside objects and Classes	77
3.1.2	Data Modeling Examples	79
3.1.3	Building a New Data Structure	81
3.1.4	The Two Dimensional Grid	82
3.1.5	Structuring Classes with Inheritance and Polymorphism	86
	Let Us Sum Up	89
	Check Your Progress	89
Section 3.2	**Graphical User Interfaces**	**93**
3.2.1	The Behavior of Terminal Based Programs and GUI Based Programs	95
3.2.2	Coding Simple GUI Based Programs	101
3.2.3	Windows and Window Components	104
3.2.4	Command Buttons and responding to events	119
	Let Us Sum Up	120
	Check Your Progress	120
3.3	Unit- Summary	125
3.4	Glossary	126
3.5	Self- Assessment Questions	127
3.6	Activities / Exercises / Case Studies	127
3.7	Answers for Check your Progress	129
3.8	References and Suggested Readings	131

Unit Objective

In the unit "Design with Classes," students will delve into the fundamental concepts of Object-Oriented Programming (OOP) by exploring classes, objects, inheritance, and polymorphism. They will learn how to model real-world entities using classes and design efficient data structures. Through hands-on examples and projects, students will gain proficiency in structuring classes, building data structures like grids, and implementing inheritance and polymorphism to create reusable and extensible code. Furthermore, they will explore the development of graphical user interfaces (GUIs) and learn to design and code simple GUI-based programs, including the creation of windows, window components, command buttons, and event handling. By the end of the unit, students will have a comprehensive understanding of OOP principles and practical skills in developing both terminal-based and GUI-based applications.

SECTION 3. 1: DESIGN WITH CLASSES

3.1.1 DESIGN WITH CLASSES : GETTING INSIDE OBJECTS AND CLASSES

Designing with classes involves creating modular and reusable code using object-oriented programming (OOP) principles. To get inside objects and classes, let's explore how to design classes effectively, how to define attributes and methods, and how to use inheritance and encapsulation to build flexible and maintainable code.

Here's a breakdown of designing with classes:

1. **Identify Objects**: Identify the real-world entities or concepts you want to model in your program. Each of these entities can be represented as an object.
2. **Define Classes**: For each identified object, create a class that defines its attributes (data) and methods (behaviour). Classes serve as blueprints for creating objects.
3. **Attributes**: Attributes represent the state of an object. They are defined as variables within a class and hold data specific to each object.
4. **Methods**: Methods are functions defined within a class that define the behavior of the objects. They operate on the attributes of the object and can perform various actions.

5. **Encapsulation**: Encapsulation is the bundling of data (attributes) and methods that operate on that data within a single unit (a class). It hides the internal state of the object and exposes only certain operations through methods.
6. **Inheritance**: Inheritance is a mechanism where a new class (subclass) can inherit attributes and methods from an existing class (superclass). This promotes code reuse and enables hierarchical relationships between classes.
7. **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the same method name to behave differently based on the object it is called on.

Let's illustrate these concepts with an example:

```
class Animal:
                              def __init__(self, name):
                              self.name = name
                              def speak(self):
                              pass
# Abstract method, to be overridden in subclasses
                              class Dog (Animal):
                               def speak(self):
                               return f"{self.name} says woof!"
                              class Cat (Animal):
                               def speak(self):
                                return f"{self.name} says meow!"
# Create instances of Dog and Cat
                              dog = Dog("Buddy")
                              cat = Cat("Whiskers")
# Call the speak method for each object
                              print(dog.speak())  # Output: Buddy says woof!
                              print(cat.speak())  # Output: Whiskers says meow!
```

In this example, we have a superclass Animal with a common attribute name and an abstract method speak (). The subclasses Dog and Cat inherit from Animal and provide their own implementation of the speak () method. This demonstrates inheritance and polymorphism in action.

By designing with classes in Python, you can create modular, reusable, and maintainable code that accurately models real-world entities and promotes code organization and abstraction.

3.1.2 DATA MODELING EXAMPLES

Data modeling is the process of creating a conceptual representation of data and its relationships to better understand how to organize and manipulate that data. In Python, data modeling often involves defining classes and their relationships to model real-world entities and their interactions.

Here are a few examples of data modeling in Python:

1. **Employee Management System**:
   - ✓ Classes: Employee, Department, Manager
   - ✓ Attributes: name, employee_id, salary, department, etc.
   - ✓ Methods: calculate_bonus (), update_salary (), etc.
   - ✓ Relationships: Employee belongs to a Department, Department is managed by a manager, etc.
2. **Library Management System**:
   - ✓ Classes: Book, Library, Member
   - ✓ Attributes: title, author, ISBN, due_date, borrower, etc.
   - ✓ Methods: check_out (), return_book(), calculate_fine(), etc.
   - ✓ Relationships: Book is owned by Library, Member borrows Book, etc.
3. **Banking System**:
   - ✓ Classes: Account, Customer, Transaction
   - ✓ Attributes: account_number, balance, customer, transaction_type, etc.
   - ✓ Methods: deposit (), withdraw (), transfer(), etc.
   - ✓ Relationships: Account belongs to Customer, Transaction is associated with Account, etc.
4. **E-commerce Platform**:
   - ✓ Classes: Product, User, Order
   - ✓ Attributes: name, price, quantity, user_id, order_date, etc.
   - ✓ Methods: add_to_cart (), checkout (), cancel_order(), etc.
   - ✓ Relationships: Product is in Order, Order is associated with User, etc.

Let's take a closer look at an example of modeling a simple library system:

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.checked_out = False
    def check_out(self):
        self.checked_out = True
    def return_book(self):
        self.checked_out = False
class Library:
    def __init__(self, name):
        self.name = name
```

```
        self.books = []
    def add_book (self, book):
        self.books.append(book)
    def find_available_books(self):
        return [book for book in self. books if not
book.checked_out]
class Member:
    def __init__(self, name, member_id):
        self.name = name
        self.member_id = member_id
        self.checked_out_books = []
    def check_out_book (self, book):
        if not book.checked_out:
            book.check_out()
            self.checked_out_books.append(book)
            return True
        else:
            return False
    def return_book(self, book):
        if book in self.checked_out_books:
            book.return_book()
            self.checked_out_books. remove(book)
            return True
        else:
            return False
# Usage
book1 = Book ("Python Crash Course", "Eric Matthes",
"978-1593279288")
book2 = Book ("Clean Code", "Robert C. Martin", "978-
0132350884")
library = Library ("Central Library")
library.add_book(book1)
library.add_book(book2)
member = Member ("John Doe", "123456")
available_books = library.find_available_books()
print ("Available Books:", [book. title for book in
available_books])
member.check_out_book(book1)
available_books = library. find_available_books ()
print ("Available Books after checkout:", [book.title for
book in available_books])
member.return_book(book1)
available_books = library. find_available_books ()
```

```
print ("Available Books after return:", [book.title for book
in available_books])
```

3.1.3 BUILDINGA A NEW DATA STRUCTURE

EXAMPLE OF BUILDING A NEW DATA STRUCTURE FOR MODELING A BOOKSTORE INVENTORY:

Bookstore Inventory Data Structure:

```
class Book:
    def __init__(self, title, author, isbn, price, quantity):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.price = price
        self.quantity = quantity
    def display_info(self):
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"ISBN: {self.isbn}")
        print(f"Price: ${self.price}")
        print(f"Quantity: {self.quantity}")
class BookstoreInventory:
    def __init__(self):
        self.books = {}
    def add_book(self, book):
        if book.isbn not in self.books:
            self.books[book.isbn] = book
            print(f"Added '{book.title}' to inventory.")
        else:
            print(f"Book with ISBN '{book.isbn}' already exists
in inventory.")
    def remove_book(self, isbn):
        if isbn in self.books:
            del self.books[isbn]
            print(f"Book with ISBN '{isbn}' removed from
inventory.")
        else:
            print(f"No book found with ISBN '{isbn}' in
inventory.")
    def update_quantity(self, isbn, quantity):
        if isbn in self.books:
self.books[isbn].quantity = quantity
print(f"Quantity updated for book with ISBN '{isbn}'.")
else:
```

```
            print(f"No book found with ISBN '{isbn}' in inventory.")
    def display_inventory(self):
        print("Bookstore Inventory:")
        for book in self.books.values():
            book.display_info()
            print("-------------------------")
# Example usage:
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", "9780743273565", 12.99, 50)
book2 = Book("To Kill a Mockingbird", "Harper Lee", "9780061120084", 10.99, 30)
inventory = BookstoreInventory()
inventory.add_book(book1)
inventory.add_book(book2)
inventory.display_inventory()
inventory.update_quantity("9780743273565", 40)
inventory.remove_book("9780061120084")
inventory.display_inventory()
```

In this example, we've created two classes: Book and Bookstore Inventory. The Book class represents individual books with attributes like title, author, ISBN, price, and quantity. The Bookstore Inventory class manages a collection of books using a dictionary where the ISBN serves as the key.

The Bookstore Inventory class provides methods to add, remove, update quantity, and display the inventory of books. Each method performs operations on the dictionary of books based on the provided ISBN. This data structure allows for efficient management of a bookstore inventory, enabling operations such as adding new books, updating quantities, and displaying the current inventory status.

3.1.3 THE TWO-DIMENSIONAL GRID

A two-dimensional grid is a common data structure used in various applications, such as games, simulations, image processing, and more. It is essentially a matrix or a table of elements arranged in rows and columns.

Here's a step-by-step guide on how to create and manipulate a two-dimensional grid in Python:

Step 1: Creating a 2D Grid.

You can create a 2D grid using a list of lists. Each inner list represents a row in the grid.

Example:

```
# Create a 3x3 grid initialized with zeros
grid = [ [0, 0, 0],
         [0, 0, 0],
         [0, 0, 0]]
# Alternatively, use a loop to create a grid of given dimensions
def create_grid(rows, cols, initial_value=0):
    return [[initial_value for _ in range(cols)] for _ in range(rows)]
# Create a 3x3 grid initialized with zeros
grid = create_grid(3, 3)
print(grid)
```

Step 2: Accessing and Modifying Elements

You can access and modify elements in the grid using row and column indices.

Example:

```
# Accessing an element at row 1, column 2
print(grid[1][2]) # Output: 0
# Modifying an element at row 1, column 2
grid [1][2] = 5
print (grid [1][2]) # Output: 5
```

Step 3: Displaying the Grid

You can display the grid in a readable format using nested loops.

Example:

```
def display_grid(grid):
    for row in grid:
        print (" ".join(map (str, row)))
display_grid(grid)
# Output:
# 0 0 0
# 0 0 5
# 0 0 0
```

Step 4: Example Operations

Here are a few example operations on a 2D grid:

Filling the Grid with Values

```
# Fill the grid with consecutive numbers
value = 1
for row in range(len(grid)):
    for col in range(len(grid[row])):
        grid[row][col] = value
        value += 1
display_grid(grid)
# Output:
# 1 2 3
# 4 5 6
```

```
# 7 8 9
```

Summing Elements in the Grid

```
def sum_grid(grid)
    total = 0
    for row in grid:
        total += sum(row)
    return total
print(sum_grid(grid)) # Output: 45
```

Finding the Maximum Element

```
def max_in_grid(grid):
    max_value = float('-inf')
    for row in grid:
        for value in row:
            if value > max_value:
                max_value = value
    return max_value
print(max_in_grid(grid)) # Output: 9
```

Step 5: More Complex Operations

You can also perform more complex operations such as searching for a specific value, counting occurrences of a value, or implementing algorithms like pathfinding on the grid.

Counting Occurrences of a Value

```
def count_value (grid, target):
    count = 0
for row in grid:
count += row.count(target)
 return count
# Example grid
grid = [   [1, 2, 3],[4, 5, 6], [7, 8, 9]]
print (count_value (grid, 5)) # Output: 1
```

A two-dimensional grid is a versatile data structure that can be used to model various problems and scenarios. By understanding how to create, manipulate, and perform operations on a 2D grid, you can solve many computational problems efficiently.

Here's the complete code:

```
# Create a grid
def create_grid(rows, cols, initial_value=0):
    return [[initial_value for _ in range(cols)] for _ in range(rows)]
# Display the grid
def display_grid(grid):
    for row in grid:
```

```
print (" “. join (map (str, row)))
# Fill the grid with consecutive numbers
def fill_grid(grid):
    value = 1
    for row in range(len(grid)):
        for col in range(len(grid[row])):
            grid[row][col] = value
            value += 1
# Sum elements in the grid
def sum_grid(grid):
    total = 0
    for row in grid:
        total += sum(row)
    return total
# Find the maximum element in the grid
def max_in_grid(grid):
    max_value = float('-inf')
    for row in grid:
        for value in row:
            if value > max_value:
                max_value = value
    return max_value
# Count occurrences of a value in the grid
def count_value (grid, target):
    count = 0
    for row in grid:
        count += row. count(target)
    return count.
# Main function to demonstrate the grid operations
def main ():
    grid = create_grid (3, 3)
    print ("Initial Grid:")
    display_grid(grid)
    grid [1][2] = 5
    print ("\nModified Grid:")
    display_grid(grid)
    fill_grid(grid)
    print ("\nFilled Grid:")
    display_grid(grid)
    print ("\nSum of elements in the grid:", sum_grid(grid))
    print ("Maximum element in the grid:", max_in_grid(grid))
    print ("Count of value 5 in the grid:", count_value (grid, 5))
if __name__ == "__main__":
```

```
main ()
```

This code demonstrates how to create, manipulate, and perform various operations on a two-dimensional grid in Python.

3.1.4 STRUCTURING CLASSES WITH INHERITANCE AND POLYMORPHISM

Structuring classes with inheritance and polymorphism is a fundamental concept in object-oriented programming (OOP). These principles help create flexible and reusable code by allowing classes to share functionality and enabling methods to behave differently based on the object that invokes them.

Inheritance

Inheritance allows a class (the child class) to inherit attributes and methods from another class (the parent class). This promotes code reuse and establishes a hierarchical relationship between classes.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the same method to perform different operations based on the object it is called on.

Let's walk through an example to illustrate these concepts:

Example Scenario: Animal Hierarchy

We'll create a class hierarchy for different types of animals. All animals have some common behaviors (methods), but each type of animal also has specific behaviors.

1. **Define the Base Class (Parent Class)**

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        raise Not Implemented Error ("Subclass must implement abstract
method")
    def move(self):
        print(f"{self.name} moves")
```

The Animal class defines common attributes and methods for all animals. The speak method is intended to be overridden by subclasses, making it an abstract method.

2.Define Derived Classes (Child Classes)

```
class Dog (Animal):
    def speak(self):
        return f"{self.name} says woof!"
    def fetch(self):
        return f"{self.name} is fetching the ball!"
```

```
class Cat(Animal):
   def speak(self):
      return f"{self.name} says meow!"
   def climb(self):
      return f"{self.name} is climbing the tree!"
class Bird (Animal):
   def speak(self):
      return f"{self.name} says tweet!"
   def fly(self):
      return f"{self.name} is flying!"
```

The Dog, Cat, and Bird classes inherit from the Animal class. Each subclass provides its own implementation of the speak method and may include additional methods specific to the subclass.

3.Using Polymorphism

You can treat objects of different subclasses as objects of the parent class and call overridden methods, which will behave according to the object's actual class.

Example:

```
def animal_sound(animal):
   print (animal. speak ())
# Create instances of each subclass
dog = Dog("Buddy")
cat = Cat("Whiskers")
bird = Bird("Tweety")
# Use polymorphism to call the speak method
animal_sound(dog) # Output: Buddy says woof!
animal_sound(cat) # Output: Whiskers says meow!
animal_sound(bird) # Output: Tweety says tweet!
```

In this example, the animal_sound function accepts an Animal object but behaves differently based on the actual subclass of the object passed to it.

4.Combining Inheritance and Polymorphism

You can create a list of animals and iterate over it, calling methods defined in the parent class. Polymorphism ensures that the correct method implementation is called for each object.

Example:

```
animals = [dog, cat, bird]
for animal in animals:
   print (animal. speak ())
   animal.move ()
# Output:
# Buddy says woof!
```

```
# Buddy moves
# Whiskers says meow!
# Whiskers moves
# Tweety says tweet!
# Tweety moves
```

Complete Example

Here is the complete code for the example:

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        raise Not Implemented Error ("Subclass must implement abstract
method")
    def move(self):
        print(f"{self.name} moves")
class Dog (Animal):
    def speak(self):
        return f"{self.name} says woof!"
    def fetch(self):
        return f"{self.name} is fetching the ball!"
class Cat(Animal):
    def speak(self):
        return f"{self.name} says meow!"
    def climb(self):
        return f"{self.name} is climbing the tree!"
class Bird(Animal):
    def speak(self):
        return f"{self.name} says tweet!"
    def fly(self):
        return f"{self.name} is flying!"
def animal_sound(animal):
    print(animal.speak())
# Create instances of each subclass
dog = Dog("Buddy")
cat = Cat("Whiskers")
bird = Bird("Tweety")
# Use polymorphism to call the speak method
animal_sound(dog) # Output: Buddy says woof!
animal_sound(cat) # Output: Whiskers says meow!
animal_sound(bird) # Output: Tweety says tweet!
# List of animals
animals = [dog, cat, bird]
for animal in animals:
```

```
    print(animal.speak())
    animal.move()
# Output:
# Buddy says woof!
# Buddy moves
# Whiskers says meow!
# Whiskers moves
# Tweety says tweet!
# Tweety moves
```

- **Inheritance**: Allows classes to inherit attributes and methods from a parent class, promoting code reuse.
- **Polymorphism**: Allows objects of different classes to be treated as objects of a common superclass, enabling methods to behave differently based on the object.

By structuring classes with inheritance and polymorphism, you can create flexible and maintainable code that models real-world relationships and behaviors effectively.

Let Us Sum Up

This unit delves into the core principles of Object-Oriented Programming (OOP), focusing on designing and implementing classes and objects. Students explore data-modeling examples to understand how real-world entities can be represented using classes. They also learn to build new data structures, such as a two-dimensional grid, enhancing their ability to manage complex data. The unit emphasizes structuring classes using inheritance and polymorphism, allowing for code reuse and the creation of flexible, extensible software designs. Through practical examples, students gain hands-on experience in creating sophisticated programs that leverage the power of OOP.

Check Your Progress

1. What is the primary purpose of using classes in programming?
   A) To increase program speed
   B) To organize code and encapsulate data and behavior
   C) To decrease memory usage
   D) To simplify syntax
2. What does the term "object" refer to in object-oriented programming?
   A) A variable that holds multiple values
   B) An instance of a class
   C) A function within a class
   D) A type of data structure
3. What is inheritance in object-oriented programming?

A) The process of creating new classes from existing ones
B) The ability of a function to call itself
C) The encapsulation of data and methods
D) The act of defining a function within a class

4. Which keyword is used to create a class in Python?
A) class
B) def
C) function
D) new

5. What is polymorphism in the context of OOP?
A) The ability to define multiple methods with the same name
B) The ability to use a single function name for multiple types
C) The ability to change an object's type at runtime
D) The ability to create classes from other classes

6. What is the purpose of the __init__ method in Python classes?
A) To initialize a class attribute
B) To define a class method
C) To initialize a new object instance
D) To delete an object instance

7. In a class definition, what does self represent?
A) The class itself
B) A global variable
C) The current instance of the class
D) A local variable

8. How do you call a method myMethod of an object obj in Python?
A) myMethod(obj)
B) obj.myMethod()
C) obj->myMethod()
D) obj

9. What is encapsulation in OOP?
A) The ability to inherit methods from a parent class
B) The bundling of data and methods into a single unit
C) The ability to define multiple methods with the same name
D) The creation of new objects

10. What is a class attribute?
A) A function defined inside a class
B) A variable that is shared among all instances of a class
C) A variable that is specific to an instance of a class
D) A method that initializes class instances

11. Which method is used to represent an object as a string in Python?
A) str
B) repr
C) init

D) format

12. How can you prevent a method from being overridden in a subclass?
    A) By defining it as private
    B) By using the final keyword
    C) By using the static keyword
    D) By defining it as protected
13. What does super() do in a method of a subclass?
    A) It accesses the methods of the current class
    B) It calls a method of the superclass
    C) It deletes an instance of a class
    D) It creates a new superclass
14. In Python, how do you create a new instance of a class MyClass?
    A) MyClass.new()
    B) MyClass()
    C) new MyClass()
    D) MyClass.create()
15. What is the primary purpose of a constructor in a class?
    A) To allocate memory for the object
    B) To provide default values for object attributes
    C) To initialize the object's attributes
    D) To define methods for the class
16. What is a destructor in a class?
    A) A method that initializes a class
    B) A method that deletes an object
    C) A method that updates object attributes
    D) A method that is automatically called when an object is destroyed
17. Which of the following best describes method overloading?
    A) Defining multiple methods with the same name but different parameters
    B) Defining multiple classes with the same name
    C) Creating a method that calls another method
    D) Using the same method name in both parent and child classes
18. How do you access a class attribute?
    A) ClassName.attribute
    B) self.attribute
    C) ClassName.method()
    D) objectName.attribute
19. What is the purpose of the @staticmethod decorator in Python?
    A) To define a method that can be called without an instance
    B) To define a method that cannot be overridden
    C) To define a method that modifies class attributes
    D) To define a method that initializes class instances
20. Which of the following statements is true about classes and objects?

A) Classes are instances of objects
B) Objects are instances of classes
C) Classes and objects are the same thing
D) Classes can only have methods, not attributes

21. What is a subclass in OOP?
A) A class that is used to instantiate objects
B) A class that inherits from another class
C) A class that cannot be extended
D) A class with no methods

22. Which of the following is a benefit of using inheritance in OOP?
A) Reduced program execution time
B) Improved code readability and reuse
C) Increased memory usage
D) Simplified syntax

23. How do you define a private attribute in a Python class?
A) Using the private keyword
B) Starting its name with a single underscore _
C) Starting its name with two underscores __
D) Using the protected keyword

24. What is the purpose of the self parameter in class methods?
A) To refer to the class itself
B) To refer to the instance calling the method
C) To refer to a global variable
D) To refer to a local variable

25. Which method in a class is called when an object is created?
A) str
B) init
C) del
D) new

26. What does method overriding allow you to do in OOP?
A) Define multiple methods with the same name in a class
B) Change the implementation of an inherited method
C) Use a method without an instance
D) Prevent a method from being inherited

27. How can you call a parent class method from a child class in Python?
A) ParentClass.method()
B) super().method()
C) self.method()
D) base.method()

28. What is multiple inheritance?
A) A class inheriting from multiple parent classes
B) A class inheriting multiple methods
C) A class with multiple attributes

D) A class with multiple instances

29. Which of the following is a correct way to define a class method in Python?
    A) def method(self):
    B) def method():
    C) def method(cls):
    D) def method(static):
30. What does the term "encapsulation" refer to in OOP?
    A) The process of defining multiple methods with the same name
    B) The bundling of data and methods within a single unit or class
    C) The ability to use a single function name for multiple types
    D) The act of creating new objects from a class

SECTION 3.2: GRAPHICAL USER INTERFACES

Most of the programs we have done till now are text-based programming. But many applications need GUI (Graphical User Interface). Python provides several different options for writing GUI based programs. These are listed below:

- ✓ **Tkinter**: It is easiest to start with. Tkinter is Python's standard GUI (graphical user interface) package. It is the most commonly used toolkit for GUI programming in Python.
- ✓ **JPython**: It is the Python platform for Java that is providing Python scripts seamless access o Java class Libraries for the local machine.
- ✓ **wxPython**: It is an open-source, cross-platform GUI toolkit written in C++. It is one of the alternatives to Tkinter, which is bundled with Python.

There are many other interfaces available for GUI. But these are the most commonly used ones. In this, we will learn about the basic GUI programming using Tkinter.

Using Tkinter

It is the standard GUI toolkit for Python. Fredrik Lundh wrote it. For modern Tk binding, Tkinter is implemented as a Python wrapper for the Tcl Interpreter embedded within the interpreter of Python. Tk provides the following widgets:

- ✓ button
- ✓ canvas
- ✓ combo-box
- ✓ frame
- ✓ level
- ✓ check-button.
- ✓ entry

- ✓ level-frame.
- ✓ menu
- ✓ list - box.
- ✓ menu button
- ✓ message
- ✓ tk_optoinMenu
- ✓ progress-bar
- ✓ radio button.
- ✓ scroll bar
- ✓ separator
- ✓ tree-view, and many more.

Creating a GUI program using this Tkinter is simple. For this, programmers need to follow the steps mentioned below:

- ✓ Import the module Tkinter
- ✓ Build a GUI application (as a window)
- ✓ Add those widgets that are discussed above.
- ✓ Enter the primary, i.e., the main event's loop for taking action when the user triggered the event.

Sample program Tkinter:

In this program, it is shown how Tkinter is used via Python to build windows along with some buttons and the events that are programmed using these buttons.

```
import tkinter as tk
from tkinter import *
from tkinter import ttk
class karl( Frame ):
    def __init__( self ):
        tk.Frame.__init__(self)
        self.pack()
        self.master.title("Karlos")
        self.button1 = Button( self, text = "CLICK HERE", width =
25,
                                    command = self.new_window)
        self. button1.grid( row = 0, column = 1, column span = 2,
sticky = W+E+N+S )
    def new_window(self):
        self.newWindow = karl2()
class karl2(Frame):
```

```
    def __init__(self):
        new =tk.Frame.__init__(self)
        new = Top level(self)
        new.title("karlos More Window")
        new.button = tk.Button( text = "PRESS TO CLOSE", width
= 25,  command = self.close_window)
        new.button.pack()
    def close_window(self):
        self.destroy()
def main():
    karl().mainloop()
if __name__ == '__main__':
    main ()
```

Standard attributed for GUI:

- ✓ Dimensions
- ✓ Fonts
- ✓ Colors
- ✓ Cursors
- ✓ Anchors
- ✓ Bitmaps

Methods for geometry management

- ✓ The pack (): This method manages the geometry of widgets in blocks.
- ✓ The grid (): This method organizes widgets in a tabular structure.
- ✓ The place (): This method organizes the widgets to place them in a specific position.

3.2.1 THE BEHAVIOR OF TERMINAL BASED PROGRAM

Terminal-based programs in Python are programs that interact with the user through a text-based interface in the terminal or command line. These programs can be as simple as printing text to the screen or as complex as full-fledged text-based user interfaces. Below, we will explore the behavior of terminal-based programs in Python, including input and output operations, argument parsing, and building text-based user interfaces.

Basic Input and Output

Printing to the Terminal

The print () function is used to display text in the terminal.

Example

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

Reading User Input

The input () function reads a line of text from the terminal.

Example

```
name = input ("Enter your name: ")
print (f"Hello, {name}!")
```

Argument Parsing

Many terminal-based programs accept arguments from the command line. The argparse module provides a way to handle these arguments.

Using argparse

Example

```
import argparse
# Create the parser
parser = argparse.ArgumentParser(description="A simple example
program")
# Add arguments
parser.add_argument('name', type=str, help="Your name")
parser.add_argument('--greet', action='store_true', help="Greet the
user")
# Parse the arguments
args = parser.parse_args ()
# Use the arguments
if args.greet:
   print (f"Hello, {args.name}!")
else:
   print (f"Name: {args.name}")
```

Save this script as example.py and run it from the terminal:

```
Sh
python example.py John --greet
# Output: Hello, John!
```

TEXT-BASED USER INTERFACES

For more complex terminal-based applications, you can create text-based user interfaces (TUI) using libraries like curses or textual.

Using curses

The curses module provides functions to create text-based user interfaces.

```
import curses
def main(stdscr):
   stdscr.clear()
   stdscr.addstr(0, 0, "Hello, Curses!")
   stdscr.refresh()
   stdscr.getkey()
```

```
curses.wrapper(main)
```

Run this script, and it will create a simple text-based interface that displays "Hello, Curses!" and waits for a key press.

Using textual

Textual is a high-level TUI framework that makes creating complex interfaces easier.

```
from textual.app import App
from textual.widgets import Header, Footer, Button
class MyApp(App):
    async def on_mount(self) -> None:
        await self. view. dock (Header (), edge="top")
        await self. view. dock (Footer (), edge="bottom")
        await self. view. dock(Button ("Click me"), edge="left")
MyApp.run ()
```

Handling Signals

Terminal-based programs may need to handle signals, such as interrupts (Ctrl+C). You can handle these using the signal module.

```
import signal
import sys
def signal_handler(sig, frame):
    print ('You pressed Ctrl+C!')
    sys. exit (0)
signal. signal (signal.SIGINT, signal_handler)
print('Press Ctrl+C')
signal.pause()
```

Advanced Input Handling

For more advanced input handling, you can use the readline module, which provides line editing and history capabilities.

```
import readline
def completer (text, state):
    options = [i for i in ['apple', 'banana', 'grape', 'orange'] if
i.startswith(text)]
    try:
        return options[state]
    except Index Error:
        return None
readline.set_completer(completer)
readline.parse_and_bind('tab: complete')
while True:
    user_input = input ('Enter a fruit: ')
    print(f'You entered: {user_input}')
```

Terminal-based programs in Python can range from simple scripts that print text and read user input to complex applications with text-based user interfaces. Key elements include:

- ✓ **Basic I/O**: Using print () and input().
- ✓ **Argument Parsing**: Using argparse to handle command-line arguments.
- ✓ **Text-Based UIs**: Using libraries like curses and textual.
- ✓ **Signal Handling**: Using the signal module to handle interrupts.
- ✓ **Advanced Input Handling**: Using readline for line editing and history.

By leveraging these tools and techniques, you can create robust and interactive terminal-based applications in Python.

GRAPHICAL USER INTERFACE

Graphical User Interface (GUI) programs in Python allow users to interact with applications in a more intuitive way compared to text-based interfaces. These programs use windows, icons, buttons, and other graphical elements to facilitate user interaction. Python offers several libraries for creating GUIs, with tkinter being the most commonly used due to its inclusion with Python. Other popular libraries include PyQt, wxPython, and Kivy. Below, I'll provide an overview of creating GUI-based programs with tkinter, followed by brief introductions to PyQt and Kivy.

Using tkinter

Step 1: Importing tkinter

First, import the tkinter module.

```
import tkinter as tk
from tkinter import message box
```

Step 2: Creating the Main Window

Create the main application window.

```
root = tk.Tk()
root.title("Simple GUI")
root.geometry("300x200")
```

Step 3: Adding Widgets

Label

```
label = tk.Label(root, text="Hello, Tkinter!")
label.pack(pady=10)
```

Button

```
def on_button_click():
    messagebox.showinfo("Information", "Button clicked!")
button = tk.Button(root, text="Click Me", command=on_button_click)
button.pack(pady=10)
```

Entry (Text Field)

```
entry = tk.Entry(root)
entry.pack(pady=10)
```

Step 4: Running the Main Event Loop

The event loop keeps the application running and waits for user interactions.

```
root.mainloop()
```

Complete tkinter Example

Here’s a complete example that includes a label, a button, and a text field:

```
import tkinter as tk
from tkinter import message box
def on_button_click ():
    user_input = entry.get ()
    messagebox.showinfo("Information", f"You entered: {user_input}")
# Create the main window
root = tk.Tk()
root.title("Simple GUI")
root.geometry("300x200")
# Add a label
label = tk.Label(root, text="Hello, Tkinter!")
label.pack(pady=10)
# Add an entry (text field)
entry = tk.Entry(root)
entry.pack(pady=10)
# Add a button
button = tk.Button(root, text="Click Me", command=on_button_click)
button.pack(pady=10)
# Run the main event loop
root.mainloop()
```

Using PyQt

PyQt is a set of Python bindings for the Qt application framework, allowing you to create sophisticated and visually appealing GUIs.

Installation

```
pip install PyQt5
```

Basic Example:

```
import sys
from PyQt5. QtWidgets import QApplication, QWidget, QLabel,
QPushButton, QVBoxLayout, QLineEdit, QMessageBox
def on_button_click():
    user_input = entry.text()
    QMessageBox. Information (window, "Information", f"You entered:
{user_input}")
app = QApplication(sys.argv)
window = QWidget ()
window.setWindowTitle('Simple PyQt5 GUI')
```

```
window.setGeometry(100, 100, 300, 200)
layout = QVBoxLayout()
label = QLabel('Hello, PyQt5!')
layout.addWidget(label)
entry = QLineEdit()
layout.addWidget(entry)
button = QPushButton('Click Me')
button.clicked.connect(on_button_click)
layout.addWidget(button)
window.setLayout(layout)
window. show ()
sys.exit(app.exec_())
```

Using Kivy

Kivy is an open-source Python library for developing multitouch applications. It is great for creating applications with advanced user interfaces.

Installation

```
pip install kivy
Basic Example
from kivy.app import App
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.popup import Popup
class MyApp(App):
    def build(self):
        layout = BoxLayout(orientation='vertical', padding=10,
spacing=10)
        self.label = Label(text="Hello, Kivy!")
        layout.add_widget(self.label)
        self.text_input = TextInput(hint_text="Enter something")
        layout.add_widget(self.text_input)
        button = Button (text="Click Me")
        button.bind(on_press=self.on_button_click)
        layout.add_widget(button)
        return layout
    def on_button_click(self, instance):
        user_input = self.text_input.text
        popup = Popup(title='Information',
                      content=Label(text=f"You entered: {user_input}"),
                      size_hint=(None, None), size=(200, 200))
```

```
        popup.open()
if __name__ == '__main__':
    MyApp().run()
```

- **tkinter**: Best for simple to moderately complex GUIs. It is included with Python and is easy to use.
- **PyQt**: Excellent for complex and feature-rich applications. It offers a wide range of tools and a modern look.
- **Kivy**: Ideal for applications with innovative user interfaces and multi-touch support. Great for mobile applications.

By selecting the appropriate library for your needs, you can create powerful and user-friendly GUI applications in Python. Each of these libraries has extensive documentation and communities to help you get started and troubleshoot any issues you might encounter.

3.2.2– CODING SIMPLE GUI BASED PROGRAMS

Creating simple GUI-based programs in Python is straightforward with libraries like tkinter, PyQt, and Kivy. Below, I'll guide you through the process of building a simple GUI with each of these libraries. Each example will include a window with a label, a text entry field, and a button that, when clicked, displays the text entered by the user in a message box or popup.

Simple GUI with tkinter

First, let's create a simple GUI using tkinter.

Step-by-Step Guide

1. Import tkinter**.**
2. Create the main window.
3. Add widgets (Label, Entry, Button)
4. Define the button click event handler.
5. Run the main event loop.

```
import tkinter as tk
```

```
from tkinter import messagebox
def on_button_click():
    user_input = entry.get()
    messagebox.showinfo("Information", f"You entered: {user_input}")
# Create the main window
root = tk.Tk()
root.title("Simple GUI")
root.geometry("300x200")
# Add a label
label = tk.Label(root, text="Enter something:")
label.pack(pady=10)
# Add an entry (text field)
entry = tk.Entry(root)
entry.pack(pady=10)
# Add a button
button = tk.Button(root, text="Click Me", command=on_button_click)
button.pack(pady=10)
# Run the main event loop
root.mainloop()
```

Simple GUI with PyQt

Next, we'll create a similar GUI using PyQt.

Step-by-Step Guide

1. Install PyQt

```
pip install PyQt5
```

2.Import PyQt5 modules

3.Create the main application and window

4.Add widgets (Label, Entry, Button)

5.Define the button click event handler

6.Run the application

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QLabel,
QPushButton, QVBoxLayout, QLineEdit, QMessageBox
def on_button_click():
    user_input = entry.text ()
```

```
        QMessageBox.information(window, "Information", f"You entered:
    {user_input}")
# Create the main application
    app = QApplication(sys.argv)
# Create the main window
    window = QWidget ()
    window.setWindowTitle('Simple PyQt5 GUI')
    window.setGeometry(100, 100, 300, 200)
# Create a layout
    layout = QVBoxLayout()
# Add a label
    label = QLabel ('Enter something:')
    layout.addWidget(label)
# Add an entry (text field)
    entry = QLineEdit ()
    layout.addWidget(entry)
# Add a button
    button = QPush Button('Click Me')
    button.clicked.connect(on_button_click)
    layout.addWidget(button)
# Set the layout and show the window
    window.setLayout(layout)
    window.show()
# Run the application
    sys.exit(app.exec_())
```

Simple GUI with Kivy

Lastly, we'll create a simple GUI using Kivy.

Step-by-Step Guide

1. Install Kivy

```
pip install kivy
```

1. Import Kivy modules
2. Create the main application class
3. Add widgets (Label, TextInput, Button)
4. Define the button click event handler
5. Run the application

```
from kivy.app import App
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.popup import Popup
class MyApp(App):
    def build(self):
        layout = BoxLayout (orientation='vertical', padding=10,
spacing=10)
        self.label = Label (text="Enter something:")
        layout.add_widget(self.label)
        self.text_input = TextInput (hint_text="Enter something")
        layout.add_widget(self.text_input)
        button = Button (text="Click Me")
        button.bind(on_press=self.on_button_click)
        layout.add_widget(button)
        return layout
    def on_button_click (self, instance):
        user_input = self.text_input.text
        popup = Popup (title='Information',
                    content=Label (text=f"You entered: {user_input}"),
                    size_hint= (None, None), size= (200, 200))
        popup.open()
if __name__ == '__main__':
    MyApp(). run()
```

By using these libraries, you can quickly create simple GUI applications in Python:

- **tkinter**: Great for simple applications and is included with Python.
- **PyQt**: Ideal for more advanced applications with a polished look.
- **Kivy**: Perfect for touch applications and cross-platform development.

Each example demonstrates how to create a main window, add basic widgets (Label, Entry, Button), handle button clicks, and display user input. This should give you a solid foundation to build upon for more complex GUI applications.

3.2.3– WINDOWS AND WINDOW COMPONENTS

Tkinter Widget

There are a number of widgets which you can put in your tkinter application. Some of the major widgets are explained below:

1. Label

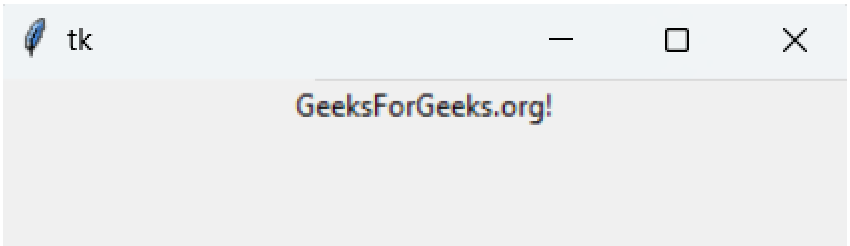
It refers to the display box where you can put any text or image which can be updated any time as per the code. The general syntax is:

```
w=Label(master, option=value)
```

master is the parameter used to represent the parent window.

```
from tkinter import *
root = Tk()
w = Label(root, text='GeeksForGeeks.org!')
w.pack()
root.mainloop()
```

Output



2. Button

To add a button in your application, this widget is used. The general syntax is:

```
w=Button(master, option=value)
```

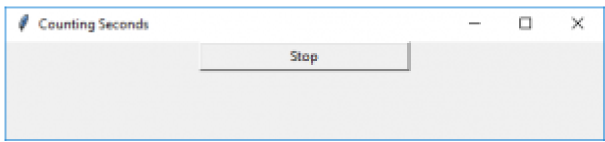
master is the parameter used to represent the parent window. There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas.

```
import tkinter as tk

r = tk.Tk()
r.title('Counting Seconds')
```

```
button = tk.Button(r, text='Stop', width=25, command=r.destroy)
button.pack()
r.mainloop()
```

Output



3. Entry

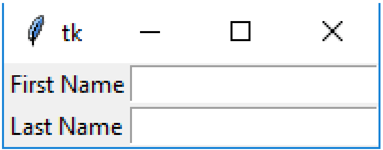
It is used to input the single line text entry from the user.. For multi-line text input, Text widget is used. The general syntax is:

```
w=Entry(master, option=value)
```

master is the parameter used to represent the parent window. There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
master = Tk()
Label(master, text='First Name').grid(row=0)
Label(master, text='Last Name').grid(row=1)
e1 = Entry(master)
e2 = Entry(master)
e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
mainloop()
```

Output



4. Check Button

To select any number of options by displaying a number of options to a user as toggle buttons. The general syntax is:

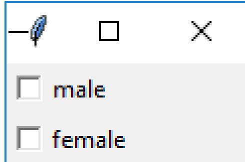
```
w = CheckButton (master, option=value)
```

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *

master = Tk ()
var1 = IntVar ()
check button (master, text='male', variable=var1).grid(row=0, sticky=W)
var2 = IntVar ()
Check button (master, text='female', variable=var2).grid(row=1, sticky=W)
mainloop ()
```

Output



5. Radio Button

It is used to offer multi-choice option to the user. It offers several options to the user and the user has to choose one option. The general syntax is:

```
w = Radio Button(master, option=value)
```

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *

root = Tk()
v = IntVar ()
Radio button (root, text='GfG', variable=v, value=1). pack(anchor=W)
Radio button (root, text='MIT', variable=v, value=2). pack(anchor=W)
mainloop ()
```

Output

6. List box

It offers a list to the user from which the user can accept any number of options. The general syntax is:

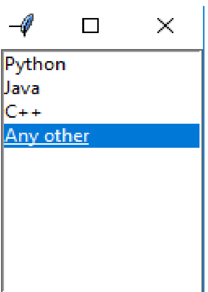
```
w = List box(master, option=value)
master is the parameter used to represent the parent window.
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *

top = Tk()
Lb = List box(top)
Lb.insert(1, 'Python')
Lb.insert(2, 'Java')
Lb.insert(3, 'C++')
Lb.insert(4, 'Any other')
Lb.pack()
top.mainloop()
```

Output



7. Scrollbar

It refers to the slide controller which will be used to implement listed widgets. The general syntax is:

```
w = Scrollbar (master, option=value)
master is the parameter used to represent the parent window.
```

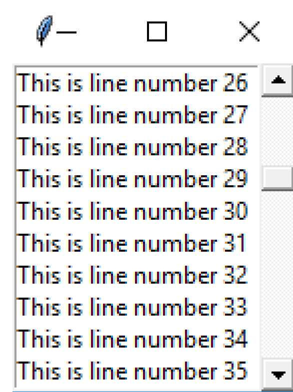
There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
```

```
root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack(side=RIGHT, fill=Y)
mylist = List box (root, yscrollcommand=scrollbar.set)

for line in range(100):
   mylist.insert(END, 'This is line number' + str(line))
   mylist.pack(side=LEFT, fill=BOTH)
scrollbar.config(command=mylist.yview)
mainloop ()
```

Output



8. Menu

It is used to create all kinds of menus used by the application. The general syntax is:

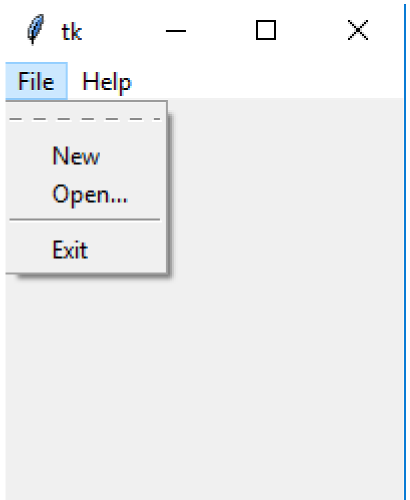
w = Menu (master, option=value)
master is the parameter used to represent the parent window.

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
root = Tk()
menu = Menu(root)
root.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New')
filemenu.add_command(label='Open...')
filemenu.add_separator()
filemenu.add_command(label='Exit', command=root.quit)
help menu = Menu(menu)
menu.add_cascade (label='Help', menu=help menu)
helpmenu.add_command(label='About')
```

```
mainloop ()
```

Output



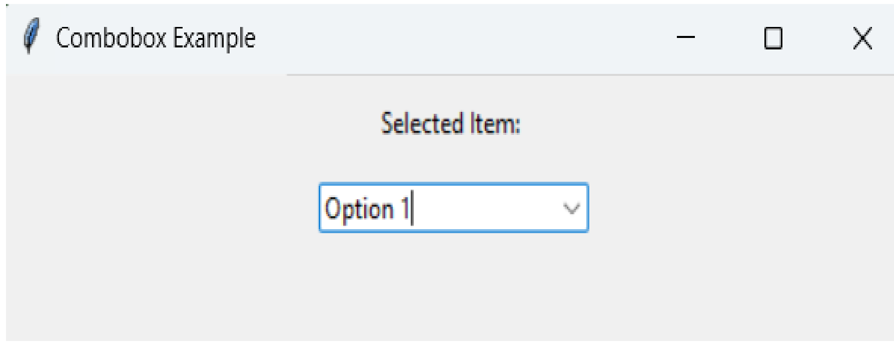
9. Combobox

Combo box widget is created using the ttk. Combo box class from the tkinter.ttk module. The values for the Combo box are specified using the values parameter. The default value is set using the set method. An event handler function on_select is bound to the Combo box using the bind method, which updates a label with the selected item whenever an item is selected.

```
import tkinter as tk
from tkinter import ttk
def on_select(event):
selected_item = combo_box.get ()
  label.config (text="Selected Item: " + selected_item)
root = tk.Tk()
root.title("Combobox Example")
# Create a label
label = tk.Label(root, text="Selected Item: ")
label.pack(pady=10)
# Create a Combobox widget
combo_box = ttk.Combobox(root, values=["Option 1", "Option 2",
"Option 3"])
combo_box.pack(pady=5)
# Set default value
combo_box.set ("Option 1")
# Bind event to selection
```

```
combo_box.bind("<<ComboboxSelected>>", on_select)
root.mainloop()
```

Output



10. Scale

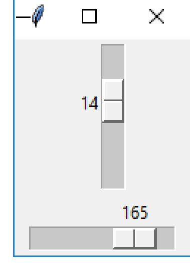
It is used to provide a graphical slider that allows to select any value from that scale. The general syntax is:

w = Scale (master, option=value) master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
master = Tk ()
w = Scale (master, from_=0, to=42)
w.pack()
w = Scale (master, from_=0, to=200, orient=HORIZONTAL)
w.pack()
mainloop ()
```

Output



11. Top Level

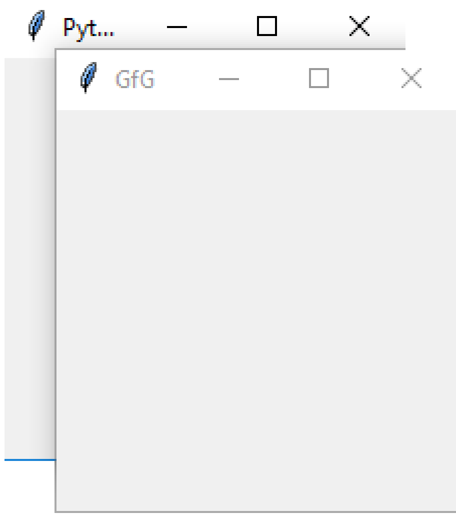
This widget is directly controlled by the window manager. It don’t need any parent window to work on. The general syntax is:

w = Top Level (master, option=value)

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
root = Tk ()
root. title('GfG')
top = Top level ()
top. Title('Python')
top. mainloop ()
```

Output



12. Message

It refers to the multi-line and non-editable text. It works same as that of Label. The general syntax is:

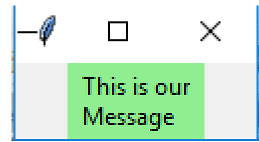
w = Message (master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
main = Tk()
ourMessage = 'This is our Message'
messageVar = Message(main, text=ourMessage)
messageVar.config(bg='lightgreen')
messageVar.pack()
main. mainloop ()
```

Output



13. Menu Button

It is a part of top-down menu which stays on the window all the time. Every menu button has its own functionality. The general syntax is:

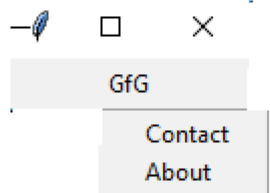
w = Menu Button (master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
top = Tk()
mb = Menubutton ( top, text = "GfG")
mb.grid()
mb. menu = Menu (mb, tearoff = 0 )
mb["menu"] = mb.menu
cVar = IntVar()
aVar = IntVar()
mb.menu.add_checkbutton (label ='Contact', variable = cVar )
mb.menu.add_checkbutton (label = 'About', variable = aVar )
mb.pack()
top. mainloop()
```

Output:

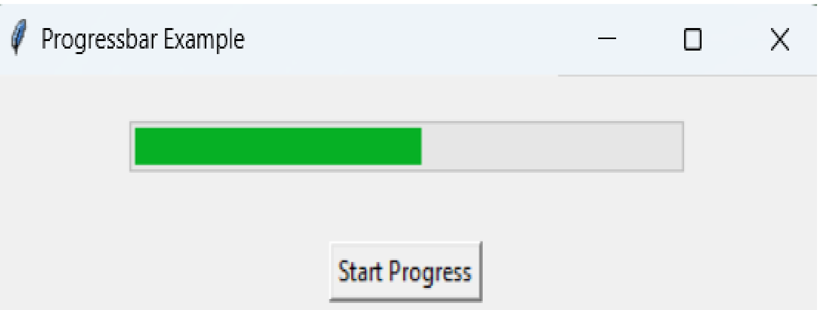


14. Progress bar

Tkinter application with a Progress bar widget and a button to start the progress. When the button is clicked, the progress bar fills up to 100% over a short period, simulating a task that takes time to complete.

```
                    import tkinter as tk
                    from tkinter import ttk
                    import time
                    def start_progress():
                    progress.start()
# Simulate a task that takes time to complete
                    for i in range(101):
# Simulate some work
        time.sleep(0.05)
        progress['value'] = i
# Update the GUI
                    root.update_idletasks()
                    progress. stop()
                    root = tk. Tk()
                    root.title("Progressbar Example")
# Create a progress bar widget
                    progress = ttk. Progressbar (root, orient="horizontal", length=300,
mode="determinate")
                    progress. Pack(pady=20)
# Button to start progress
                    start_button = tk. Button (root, text="Start Progress",
command=start_progress)
                    start_button. Pack(pady=10)
                    root. mainloop ()
```

Output:



15. Spin Box

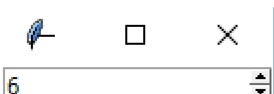
It is an entry of 'Entry' widget. Here, value can be input by selecting a fixed value of numbers. The general syntax is:

```
w = Spin Box(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
master = Tk ()
w = Spin box (master, from_=0, to=10)
w. pack ()
mainloop ()
```

Output:



16. Text

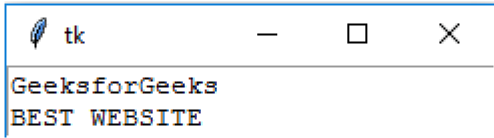
To edit a multi-line text and format the way it has to be displayed. The general syntax is:

```
w =Text(master, option=value)
```

There are number of options which are used to change the format of the text. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
root = Tk ()
T = Text (root, height=2, width=30)
T. pack ()
T. insert (END, 'GeeksforGeeks\nBEST WEBSITE\n')
mainloop ()
```

Output



17. Canvas

It is used to draw pictures and other complex layout like graphics, text and widgets. The general syntax is:

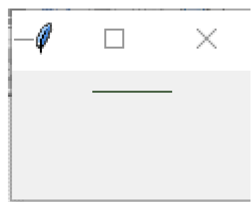
w = Canvas (master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
master = Tk ()
w = Canvas (master, width=40, height=60)
w. pack ()
canvas_height=20
canvas_width=200
y = int (canvas_height / 2)
w. create_line (0, y, canvas_width, y )
main loop ()
```

Output



18. Panned Window

It is a container widget which is used to handle number of panes arranged in it. The general syntax is:

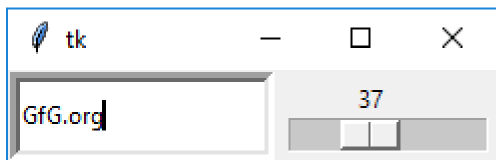
w = Panned Window (master, option=value)

Master is the parameter used to represent the parent window. There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

```
from tkinter import *
m1 = Paned Window ()
m1. pack (fill=BOTH, expand=1)
left = Entry (m1, bd=5)
```

```
m1.add(left)
m2 = Paned Window (m1, orient=VERTICAL)
m1.add(m2)
top = Scale (m2, orient=HORIZONTAL)
m2.add(top)
mainloop ()
```

Output



Color Option in Tkinter

This example demonstrates the usage of various color options in Tkinter widgets, including active background and foreground colours, background and foreground colours, disabled state colors, and selection colors. Each widget in the example showcases a different color option, providing a visual representation of how these options affect the appearance of the widgets.

```
import tkinter as tk
root = tk. Tk ()
root. title ("Color Options in Tkinter")
# Create a button with active background and foreground colors
button = tk. Button (root, text="Click Me", active background="blue",
active foreground="white")
button. pack ()
# Create a label with background and foreground colors
label = tk. Label (root, text="Hello, Tkinter!", bg="light Gray", fg="black")
label. pack ()
# Create an Entry widget with selection colors
entry = tk. Entry (root, select background="light blue", select
foreground="black")
entry. pack ()
root. mainloop ()
```

Output

Learn more to Improve Font: Tkinter Font

Geometry Management

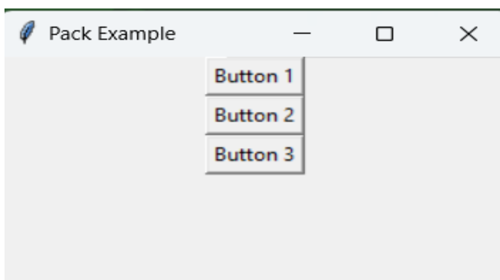
Tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

pack () method.

It organizes the widgets in blocks before placing in the parent widget.

```
        import tkinter as tk
        root = tk. Tk()
        root. title ("Pack Example")
# Create three buttons
        button1 = tk. Button (root, text="Button 1")
        button2 = tk. Button (root, text="Button 2")
        button3 = tk. Button (root, text="Button 3")
# Pack the buttons vertically
        button1.pack()
        button2.pack()
        button3.pack()
        root.mainloop()
```

Output



grid () method

It organizes the widgets in grid (table-like structure) before placing in the parent widget.

```
        import tkinter as tk
        root = tk.Tk()
        root.title("Grid Example")
# Create three labels
        label1 = tk.Label(root, text="Label 1")
        label2 = tk.Label(root, text="Label 2")
        label3 = tk.Label(root, text="Label 3")
# Grid the labels in a 2x2 grid
        label1.grid(row=0, column=0)
        label2.grid(row=0, column=1)
        label3.grid(row=1, column=0, column span=2)
```

```
root. mainloop()
```

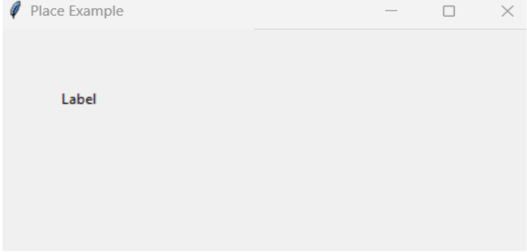
Output

place () method

It organizes the widgets by placing them on specific positions directed by the programmer.

```
import tkinter as tk
root = tk. Tk ()
root. title ("Place Example")
# Create a label
label = tk. Label (root, text="Label")
# Place the label at specific coordinates
label. place (x=50, y=50)
root. mainloop ()
```

Output



3.2.4– COMMAND BUTTONS AND RESPONDING TO EVENTS

Creating command buttons and responding to events in a GUI application is a fundamental aspect of graphical user interface (GUI) programming. Here's a simplified example using Python's Tkinter library:

```
import tkinter as tk
# Function to handle button click event
def on_button_click():
    label.config(text="Button Clicked!")
# Create main window
root = tk.Tk()
root.title("Command Buttons Example")
# Create label
```

```
label = tk.Label(root, text="Click the button")
label.pack(pady=10)
# Create command button
button = tk.Button(root, text="Click Me", command=on_button_click)
button.pack()
# Run the application
root.mainloop()
```

Explanation:

First, we import the tkinter module as tk.

We define a function on_button_click() to handle the button click event. In this example, it simply changes the text of a label widget.

We create the main window using tk.Tk() and set its title.

Next, we create a label widget (tk.Label()) to display text and pack it into the main window with some padding.

Then, we create a command button (tk.Button()) with the text "Click Me" and specify the command parameter to call the on_button_click() function when the button is clicked.

Finally, we start the GUI application by calling the mainloop() method on the main window.

When the button is clicked, the on_button_click() function is invoked, updating the label text accordingly. This demonstrates how to create command buttons and respond to events in a Tkinter-based GUI application.

Let Us Sum Up

In this unit, we explored fundamental Python programming concepts, including object-oriented design, data modeling, and graphical user interface (GUI) development. We learned to create custom data structures using classes, implement inheritance and polymorphism for class hierarchies, and design GUI applications using

Tkinter. Through hands-on exercises, we practiced building GUI-based programs with windows, widgets, and event-driven functionality, including command buttons and

event handling. This unit equipped us with essential skills to structure classes effectively, model data structures, and develop interactive GUI applications, laying a solid foundation for further exploration in Python programming and software development.

Check Your Progress

1. What is the primary difference between terminal-based programs and GUI-based programs?
   A) GUI-based programs are faster
   B) Terminal-based programs use graphical elements
   C) GUI-based programs use graphical elements
   D) Terminal-based programs are easier to use
2. Which of the following is a common library used for creating GUI-based programs in Python?
   A) NumPy
   B) Matplotlib
   C) Tkinter
   D) Pandas
3. What is the main component of a window in a GUI-based program?
   A) Terminal
   B) Canvas
   C) Widget
   D) Event
4. In Tkinter, which method is used to start the main event loop?
   A) mainloop()
   B) start()
   C) run()
   D) execute()
5. What is the purpose of an event in a GUI-based program?
   A) To create a window
   B) To handle user interactions
   C) To design the interface
   D) To execute background tasks

6. Which of the following is a common type of widget in GUI-based programming?
    A) Label
    B) Array
    C) Dictionary
    D) Tuple
7. How do you create a button in Tkinter?
    A) Button(window)
    B) window.Button()
    C) TkButton(window)
    D) create_button(window)
8. What is the function of a callback in GUI programming?
    A) To handle errors
    B) To execute code in response to an event
    C) To update the GUI
    D) To initialize the GUI
9. Which of the following is an example of an event in GUI programming?
    A) Opening a file
    B) Clicking a button
    C) Printing to the console
    D) Sorting a list
10. What is the purpose of the pack() method in Tkinter?
    A) To handle events
    B) To arrange widgets in a window
    C) To create a new widget
    D) To start the main loop
11. How do you create a label in Tkinter?
    A) Label(window, text="Hello")
    B) window.Label(text="Hello")
    C) TkLabel(window, text="Hello")
    D) create_label(window, "Hello")
12. Which method is used to place widgets at specific coordinates in Tkinter?
    A) pack()

B) grid()
C) place()
D) position()

13. What does GUI stand for?
A) General User Interface
B) Graphical User Interface
C) General Utility Interface
D) Graphical Utility Interface

14. In a GUI-based program, what is an event loop?
A) A loop that waits for and dispatches events or messages in a program
B) A function that initializes the GUI
C) A method to create widgets
D) A process to handle errors

15. What is the primary purpose of a command button in a GUI?
A) To display text
B) To receive user input
C) To execute a command or function
D) To create a new window

16. How do you set the title of a Tkinter window?
A) window.setTitle("Title")
B) window.title("Title")
C) setTitle(window, "Title")
D) set_window_title("Title")

17. What is the purpose of the bind() method in Tkinter?
A) To bind data to a widget
B) To bind an event to a function
C) To create a new widget
D) To start the main loop

18. Which of the following widgets is commonly used to allow the user to input text in Tkinter?
A) Label
B) Entry
C) Button

D) Frame

19. How do you create a window in Tkinter?
    A) window = Tkinter()
    B) window = Window()
    C) window = Tk()
    D) window = CreateWindow()
20. What is the primary difference between pack() and grid() methods in Tkinter?
    A) pack() arranges widgets in columns and rows; grid() stacks them vertically
    B) pack() stacks widgets vertically; grid() arranges them in columns and rows
    C) pack() and grid() are used for different types of widgets
    D) pack() is used to create widgets; grid() is used to handle events
21. Which widget is used to create a menu bar in a Tkinter window?
    A) Menu
    B) Label
    C) Button
    D) Entry
22. What is the purpose of the after() method in Tkinter?
    A) To execute a function after a specified time delay
    B) To create a new widget
    C) To start the main event loop
    D) To handle errors
23. How do you close a Tkinter window?
    A) window.close()
    B) window.destroy()
    C) close(window)
    D) window.quit()
24. Which widget in Tkinter is used to create a scrollable list?
    A) Listbox
    B) Entry
    C) Scrollbar

D) Text

25. What is the purpose of the withdraw() method in Tkinter?

A) To close the window

B) To minimize the window

C) To hide the window without destroying it

D) To maximize the window

26. How do you update the text of a label in Tkinter?

A) label.setText("New Text")

B) label.updateText("New Text")

C) label.config(text="New Text")

D) label.changeText("New Text")

27. Which of the following methods is used to change the background color of a widget in Tkinter?

A) widget.bgcolor("color")

B) widget.config(bg="color")

C) widget.setBackground("color")

D) widget.setColor("color")

28. How do you create a pop-up message box in Tkinter?

A) messagebox.show("Message")

B) popup.showinfo("Title", "Message")

C) messagebox.showinfo("Title", "Message")

D) popup.message("Title", "Message")

29. Which of the following methods is used to make a widget visible in Tkinter?

A) widget.show()

B) widget.display()

C) widget.pack()

D) widget.present()

30. What is the function of the focus_set() method in Tkinter?

A) To highlight a widget

B) To set the focus on a widget

C) To create a new widget

D) To start the main loop

Unit Summary

In this unit, we delved into essential concepts of Python programming, encompassing object-oriented principles, GUI development, and event-driven programming. From understanding classes, inheritance, and polymorphism to building graphical user interfaces using Tkinter, learners explored the intricacies of structuring classes, modeling data, and responding to user interactions. Through practical exercises, they gained proficiency in designing interactive GUI-based programs, mastering the creation of windows, window components, command buttons, and event Handling, setting a solid foundation for advanced python application development.

Glossary

- **Class:** A blueprint for creating objects in Python, defining attributes and methods.
- **Object:** An instance of a class, encapsulating data and behavior.
- **Inheritance:** The mechanism by which a class can inherit attributes and methods from another class.
- **Polymorphism:** The ability of different classes to be treated as instances of a common superclass, enabling flexibility and code reuse.
- **Constructor:** A special method in Python (__init__) used to initialize object state when an instance is created.
- **Encapsulation:** The bundling of data and methods that operate on the data, hiding implementation details from the outside world.
- **Method:** A function defined within a class, associated with instances of the class.
- **Superclass:** A class from which other classes inherit properties and behaviors.
- **Subclass:** A class that inherits properties and behaviors from a superclass.
- **Instance Variable:** A variable associated with a specific instance of a class.
- **GUI (Graphical User Interface):** A type of interface that allows users to interact with electronic devices through graphical icons and visual indicators.
- **Event Handling:** The process of responding to user actions or system events, such as button clicks or mouse movements, in a GUI application.

- **Tkinter:** A standard GUI toolkit for Python, providing a set of modules for creating graphical user interfaces.
- **Button:** A GUI widget used to trigger an action when clicked by the user.
- **Event:** An action or occurrence detected by a program, typically requiring some response or processing.
- **Event Handler:** A function or method that is executed in response to a specific event, such as a button click or keypress.
- **Widget:** A graphical component used to interact with the user in a GUI application, such as buttons, labels, or entry fields.

Self – Assessment Questions

1. Evaluate the importance of inheritance in object-oriented programming. How does it promote code reusability and maintainability?
2. Analyze the concept of polymorphism in Python. How does it enable flexibility in programming and enhance code readability?
3. Compare and contrast encapsulation and inheritance. How do they contribute to building robust and modular software systems?
4. Explain the significance of constructors in Python classes. How are they used to initialize object state and ensure proper object creation?
5. Elucidate the role of event handling in GUI-based programming. How does it enable interactive user interfaces and responsiveness in applications?
6. Evaluate the advantages and disadvantages of using Tkinter for GUI development in Python compared to other libraries such as PyQt or wxPython.
7. Analyze the process of binding event handlers to GUI widgets in Tkinter. How does it facilitate the handling of user interactions in graphical applications?
8. Compare and contrast GUI-based programming with terminal-based programming. What are the key differences in their user interaction paradigms and development approaches?

Activities / Exercises / Case Studies

Activities

1. Task students with designing a class hierarchy for a real-world scenario, such as a library management system or a banking application. Encourage them to incorporate concepts like inheritance, encapsulation, and polymorphism.

2. Divide students into groups and assign each group a real-world scenario. For example, one group could focus on vehicles (Car, Truck, Bicycle), while another group could focus on animals (Dog, Cat, Bird). Students should identify common attributes and behaviours and design a class hierarchy showcasing inheritance relationships.

3. Organize a code review session where students evaluate each other's code for adherence to object-oriented principles and best practices. Encourage constructive feedback and discussions on how to improve code quality and design. This activity promotes collaboration and peer learning.

4. Provide a superclass with several methods and ask students to create subclasses that override these methods with their own implementations. Encourage them to demonstrate how method overriding enables customization and flexibility in class behavior

Exercise:

1. Provide a set of classes representing different shapes (e.g., Circle, Square, Triangle) with a common method calculate_area(). Ask students to create instances of these classes and demonstrate polymorphic behavior by calling the calculate_area() method on each object.

2. Provide a set of GUI widgets (e.g., Button, Entry, Label) and ask students to create a simple Tkinter application. They should implement event handlers for various user interactions, such as button clicks or text entry events. This exercise helps reinforce the understanding of event-driven programming.

CASE STUDY:

1. Present a case study where students are required to develop a simple GUI-based application using Tkinter. Provide a scenario such as a to-do list manager or a

calculator. Students should design the GUI layout, implement event handlers for user interactions, and ensure the application's responsiveness.

2. Present a case study involving sensitive data handling, such as a user authentication system or a banking transaction system. Students should design classes that encapsulate sensitive data and methods for secure access. Emphasize the importance of encapsulation in maintaining data integrity and security.

3. Provide a partially implemented Tkinter application and ask students to enhance its functionality. For example, they could add new features, improve user interface design, or optimize event handling. This case study encourages students to apply their knowledge to real-world application scenarios.

Answers for check your progress

Modules	S. No.	Answers
Module 1	1.	B) To organize code and encapsulate data and behavior
	2.	B) An instance of a class
	3.	A) The process of creating new classes from existing ones
	4.	A) class
	5.	B) The ability to use a single function name for multiple types
	6.	C) To initialize a new object instance
	7.	C) The current instance of the class
	8.	B) obj.myMethod()
	9.	B) The bundling of data and methods into a single unit
	10.	B) A variable that is shared among all instances of a class
	11.	A) str
	12.	B) By using the final keyword
	13.	B) It calls a method of the superclass
	14.	B) MyClass()
	15.	C) To initialize the object's attributes
	16.	D) A method that is automatically called when an object is destroyed

	17.	A) Defining multiple methods with the same name but different parameters
	18.	A) ClassName.attribute
	19.	A) To define a method that can be called without an instance
	20.	B) Objects are instances of classes
	21.	B) A class that inherits from another class
	22.	B) Improved code readability and reuse
	23.	C) Starting its name with two underscores __
	24.	B) To refer to the instance calling the method
	25.	B) init
	26.	B) Change the implementation of an inherited method
	27.	B) super().method()
	28.	A) A class inheriting from multiple parent classes
	29.	C) def method(cls):
	30.	B) The bundling of data and methods within a single unit or class
Module 2.	**1.**	C) GUI-based programs use graphical elements
	2.	C) Tkinter
	3.	C) Widget
	4.	A) mainloop()
	5.	B) To handle user interactions
	6.	A) Label
	7.	A) Button(window)
	8.	B) To execute code in response to an event
	9.	B) Clicking a button
	10.	B) To arrange widgets in a window
	11.	A) Label(window, text="Hello")
	12.	C) place()
	13.	B) Graphical User Interface
	14.	A) A loop that waits for and dispatches events or messages in a program
	15.	C) To execute a command or function
	16.	B) window.title("Title")

	17.	B) To bind an event to a function
	18.	B) Entry
	19.	C) window = Tk()
	20.	B) pack() stacks widgets vertically; grid() arranges them in columns and rows
	21.	A) Menu
	22.	A) To execute a function after a specified time delay
	23.	B) window.destroy()
	24.	A) Listbox
	25.	C) To hide the window without destroying it
	26.	C) label.config(text="New Text")
	27.	B) widget.config(bg="color")
	28.	C) messagebox.showinfo("Title", "Message")
	29.	C) widget.pack()
	30.	B) To set the focus on a widget

Suggested Readings

1. Ramalho, L. (2022). *Fluent python*. " O'Reilly Media, Inc.".
2. Sweigart, A. (2019). *Automate the boring stuff with Python: practical programming for total beginners*. no starch press.
3. Lutz, M. (2013). *Learning python: Powerful object-oriented programming*. " O'Reilly Media, Inc.".

Open-Source E-Content Links

1. https://docs.python.org/3/
2. https://www.w3schools.com/python/
3. https://www.geeksforgeeks.org/python-programming-language-tutorial/
4. https://realpython.com/python3-object-oriented-programming/
5. https://docs.python.org/3/library/tkinter.html

References

1. EdX: Introduction to Computer Science and Programming Using Python
2. Coursera: Python for Everybody Specialization
3. Codecademy Python Course
4. Coursera: Python for Everybody Specialization

UNIT IV – WORKING WITH PYTHON PACKAGES

Unit IV: Working with Python Packages: NumPy Library-Ndarray- Basic Operations - Indexing, Slicing and Iteration - Array manipulation - Pandas - The Series - The DataFrame - The Index Objects - Data Vizualization with Matplotlib- The Matplotlib Architecture -Pyplot- The Plotting Window - Adding Elements to the Chart - Line Charts - Bar Charts - Pie charts

Working With Python Packages

Section	Topic	Page No.
	UNIT – IV	
	Unit Objectives	
Section 4.1	**Working With Python Packages**	**133**
4.1.1	Working with Python Packages	133
4.1.2	Numpy Library	134
4.1.3	Ndarray	136
4.1.4	Basic Operations : Indexing , Slicing and Iteration	138
4.1.5	Array Manipulation	143
	Let Us Sum Up	146
	Check Your Progress	146
Section 4.2	**Pandas**	**147**
4.2.1	The Series, The Dataframe	152
4.2.3	The Index Objects	155
	Let Us Sum Up	159
	Check Your Progress	159
Section 4.3	**Data Visualization with Matplotlib**	164
4.3.1	The Matplotlib Architecture- Pyplot	169
4.3.2	The Plotting Window - Adding Elements to the Chart	173
4.3.3	Line Charts – Bar Charts – Pie Charts	176
	Let Us Sum Up	178
	Check Your Progress	178
4.4	Unit- Summary	183
4.5	Glossary	183

4.6	Self- Assessment Questions	185
4.7	Activities / Exercises / Case Studies	186
4.8	Answers for Check your Progress	186
4.9	References and Suggested Readings	190

UNIT OBJECTIVES

This unit aims to provide a comprehensive understanding of essential Python packages used in data analysis and visualization. Students will learn to effectively use the NumPy library for handling large multi-dimensional arrays and performing basic operations such as indexing, slicing, and iteration. They will gain skills in manipulating arrays to suit various data processing needs. The course will introduce Pandas for data manipulation and analysis, focusing on key components like Series, DataFrame, and Index objects. Additionally, students will explore data visualization techniques using Matplotlib, learning about its architecture and how to use Pyplot to create and customize various types of charts including line charts, bar charts, and pie charts. By the end of this unit, students will be proficient in leveraging these powerful tools to manage, analyze, and visualize data effectively.

SECTION 4.1: WORKING WITH PYTHON PACKAGES

4.1.1 WORKING WITH PYTHON PACKAGES

Python packages are essential tools that significantly enhance the capabilities of the Python programming language, especially in the fields of data analysis and visualization. This unit focuses on three pivotal packages: NumPy, Pandas, and Matplotlib.

NumPy is the cornerstone for numerical computing in Python, providing support for large multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

Pandas is a powerful data manipulation and analysis library that offers data structures like Series and DataFrame, enabling efficient handling and analysis of structured data.

Matplotlib is a versatile plotting library used to create static, interactive, and animated visualizations in Python. It provides a framework for creating a wide variety of plots and charts, essential for data visualization.

Together, these packages form a robust foundation for performing complex data analysis and creating insightful visualizations, making them indispensable tools for data scientists and analysts.

4.1.2 NUMPY LIBRARY

Working with the NumPy library in Python involves understanding its core functionalities for numerical computations, such as array operations, mathematical functions, and linear algebra operations. Below is an overview of how to install, import, and utilize NumPy effectively. Installing NumPy. First, you need to install NumPy if it is not already installed. You can do this using pip install numpy

Importing NumPy: To use NumPy in your Python script, you need to import it. The conventional alias for NumPy is np.

```
import numpy as np
```

Basic Operations with NumPy: Creating Arrays NumPy provides various ways to create arrays. From a Python list:

```
coder = np. array ([1, 2, 3, 4, 5])
print(arr)
```

Using built-in function s:np. zeros: Creates an array filled with zeros.np. ones: Creates an array filled with ones.np. arange: Creates an array with a range of values.np.

Lin space: Creates an array with linearly spaced values.

```
zeros_array = np. zeros((3, 3))
ones_array = np.ones((2, 4))
range_array = np.arange(10)
linspace_array = np.linspace(0, 1, 5)
print(zeros_array)
```

```
print(ones_array)
print(range_array)
print(linspace_array)
```

Array Operations NumPy allows for element-wise operations on arrays.

```
a = np. array ([1, 2, 3])
b = np. array([4, 5, 6])
# Addition       c = a + b
print(c) # Output: [5 7 9]
# Multiplication
d = a * b
print(d)  # Output: [ 4 10 18]
```

Mathematical Functions NumPy provides a wide range of mathematical functions.

```
arr = np. array([1, 2, 3, 4, 5])
# Square root
sqrt_arr = np.sqrt(arr)
print(sqrt_arr)
# Exponential
exp_arr = np.exp(arr)
print(exp_arr)
# Trigonometric functions
sin_arr = np.sin(arr)
print(sin_arr)
```

Array Indexing and Slicing Access elements, rows, columns, or subarrays using indexing and slicing.

```
arr = np. array ([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Accessing an element
element = arr [1, 2]
print(element) # Output: 6
# Slicing
subarray = arr [0:2, 1:3]
print(subarray)
```

Reshaping and Resizing Change the shape of an array using reshape or resize.pythonCopy codearr = np.arange(12)

```
# Reshape
reshaped_arr = arr.reshape((3, 4))
print(reshaped_arr)
# Resize
arr.resize((2, 6))
print(arr)
```

Linear Algebra NumPy includes various functions for linear algebra.

```
from numpy.linalg
import inv, eig
matrix = np. array ([[1, 2], [3, 4]])
# Inverse
inverse_matrix = inv(matrix)
print(inverse_matrix)
# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = eig(matrix)
print(eigenvalues)
print(eigenvectors)
```

Broadcasting allows for operations on arrays of different shapes.

```
arr1 = np. array ([1, 2, 3])
arr2 = np. array ([[1], [2], [3]])
# Broadcasting addition
result = arr1 + arr2
print(result)
```

NumPy is a powerful library for numerical computations in Python. It provides functionalities such as: Array creation and manipulation. Mathematical operations. Indexing and slicing. Linear algebra operations. Broadcasting for operations on arrays of different shapes. By understanding and utilizing these features, you can perform efficient numerical computations in your Python programs.

4.1.3 NDARRAY

The ndarray (n-dimensional array) is a central feature of the NumPy library in Python, enabling efficient storage and manipulation of large datasets. Here’s a detailed guide

on working with ND array in NumPy. Creating Nd arrays From Lists You can create an ndarray from a Python list using the np.

```
array () function.
 import numpy as np
# 1D array
arr1d = np. array([1, 2, 3, 4, 5])
print(arr1d)
# 2D array
arr2d = np. array([[1, 2, 3], [4, 5, 6]])
print(arr2d)
```

Using Built-in Functions NumPy provides several functions to create arrays with specific patterns or values.

```
# Array of zeros
zeros_array = np. zeros((2, 3))
print(zeros_array)
# Array of ones
ones_array = np. ones ((2, 3))
print(ones_array)
# Array with a range of values
range_array = np. arange(0, 10, 2)
print(range_array)
# Array with linearly spaced values
linspace_array = np. linspace (0, 1, 5)
print(linspace_array)
```

Array Properties Understanding the properties of an ndarray is crucial for effective use.

```
codearr = np. array ([[1, 2, 3], [4, 5, 6]])
# Shape of the array
print ("Shape:", arr.shape)
# Number of dimensions
print ("Number of dimensions:", arr.ndim)
# Size of the array (total number of elements)
print ("Size:", arr. size)
# Data type of elements
print ("Data type:", arr.dtype)
```

4.1.4 BASIC OPERATIONS : INDEXING , SLICING AND ITERATION

INDEXING AND SLICING

Basic Indexing Access elements using square brackets.

```
arr = np.array([1, 2, 3, 4, 5])
# Accessing a single element
print (arr [0]) # Output: 1
# Accessing multiple elements
print (arr [1:4]) # Output: [2 3 4]
```

Multi-dimensional Indexing for multi-dimensional arrays, provide a tuple of indices.

```
arr = np. array ([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Accessing an element
print (arr [1, 2]) # Output: 6
# Slicing a subarray
print (arr [0:2, 1:3]) # Output: [[2 3]
[5 6]]
```

Array Operations Arithmetic Operations NumPy supports element-wise arithmetic operations.

```
arr1 = np. array ([1, 2, 3])
arr2 = np. array ([4, 5, 6])
# Addition
print (arr1 + arr2) # Output: [5 7 9]
# Multiplication
print (arr1 * arr2)  # Output: [ 4 10 18]
# Scalar operations
print (arr1 * 2) # Output: [2 4 6]
```

Mathematical Functions Apply mathematical functions elementwise.

```
arr = np. array ([1, 2, 3, 4, 5])
# Square root
print (np. sqrt(arr))
# Exponential
print(np.exp(arr))
```

```
# Trigonometric functions
print(np.sin(arr))
```

Reshaping Arrays Change the shape of an array without changing its data.

```
arr = np. arange (12)
# Reshaping
reshaped_arr = arr. reshapes ((3, 4))
print(reshaped_arr)
```

Broadcasting allows for operations on arrays of different shapes.

```
codearr1 = np. array([1, 2, 3])
arr2 = np. array ([[1], [2], [3]])
# Broadcasting addition
result = arr1 + arr2
print(result)
```

Advanced Indexing and Boolean Array Advanced Indexing Access elements using arrays of indices.

```
arr = np. array ([10, 20, 30, 40, 50])
# Index array
indices = np. array ([0, 2, 4])
print(arr[indices]) # Output: [10 30 50]
```

Boolean Indexing Use Boolean arrays for indexing.

```
arr = np.array([1, 2, 3, 4, 5])
# Boolean array
bool_arr = arr &gt; 3
print(arr[bool_arr]) # Output: [4 5]
```

Linear Algebra Operations NumPy provides functions for linear algebra.

```
from numpy.linalg import inv, eig
matrix = np. array ([[1, 2], [3, 4]])
# Inverse
inverse_matrix = inv(matrix)
print(inverse_matrix)
# Eigenvalues and eigenvectors
```

```
eigenvalues, eigenvectors = eig(matrix)
print(eigenvalues)
print(eigenvectors)
```

The ndarray is a powerful feature of the NumPy library, offering a wide range of functionalities for numerical computations:

- Creating arrays: From lists or using built-in functions.
- Array properties: Shape, dimensions, size, and data type.
- Indexing and slicing: Accessing and modifying array elements.
- Array operations: Arithmetic, mathematical functions, reshaping, and broadcasting.
- Advanced indexing: Using arrays of indices and Boolean arrays.
- Linear algebra: Matrix operations, inverses, eigenvalues, and eigenvectors. By mastering these features, you can leverage NumPy for efficient and effective numerical computations in Python.

Slicing and Iteration

Slicing and Iteration in Python Slicing and iteration are fundamental techniques in Python for accessing and manipulating sequences such as strings, lists, tuples, and more. Below is a detailed guide on how to use slicing and iteration effectively. Slicing Slicing allows you to extract a part of a sequence by specifying a start, stop, and step. Basic Slicing Syntax The basic syntax for slicing is sequence [start: stop: step].start: The starting index of the slice (inclusive).stop:

The ending index of the slice (exclusive).step:

The step or stride between each index.

```
# Example list
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# Slicing from index 2 to 5
print(lst[2:6]) # Output: [2, 3, 4, 5]
```

```
# Slicing with a step of 2
print(lst[1:8:2])  # Output: [1, 3, 5, 7]
# Slicing from the beginning to index 4
print(lst[:5])  # Output: [0, 1, 2, 3, 4]
# Slicing from index 5 to the end
print(lst[5:]) # Output: [5, 6, 7, 8, 9]
# Slicing with negative indices
print(lst[-5:])  # Output: [5, 6, 7, 8, 9]
# Reversing the list
print(lst[::-1])  # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Slicing String Slicing can also be applied to strings.

```
codes = "Hello, World!"
# Slicing from index 7 to 12
print(s[7:12])  # Output: World
# Slicing with a step of 2
print(s[::2])  # Output: Hlo ol!
# Reversing the string
print(s[::-1])  # Output: !dlroW ,olleH
```

Iteration involves going through each element of a sequence one by one. Python provides several constructs for iteration, such as for loops and comprehensions. Iterating Over Lists

```
# Example list
lst = [0, 1, 2, 3, 4, 5]
# Iterating using a for loop
for item in lst:
    print(item)
```

Iterating with Indexes the enumerate () function to get both the index and the value.

```
# Example list
lst = ['a', 'b', 'c']
# Iterating with index
for index, value in enumerate(lst):
 print (f"Index: {index}, Value: {value}")
Iterating Over Strings
 codes = "Hello"
# Iterating through each character
```

```
for char in s:
    print(char)
```

Iterating Over Dictionaries Use items () to iterate over key-value pairs in a dictionary.

```
d = {'a': 1, 'b': 2, 'c': 3}
# Iterating through dictionary items
for key, value in d. items():
 print (f"Key: {key}, Value: {value}")
```

List Comprehensions List comprehensions provide a concise way to create lists.

```
# Creating a list of squares.
squares = [x**2 for x in range (10)]
print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
# Filtering even numbers
evens = [x for x in range (10) if x % 2 == 0]
print(evens) # Output: [0, 2, 4, 6, 8]
```

Dictionary Comprehension Similar to list comprehensions, but for dictionaries.

```
# Creating a dictionary of squares
squares_dict = {x: x**2 for x in range (10)}
print(squares_dict)
```

Iterating with while Loops Use while loops for more complex iterations.

```
i = 0
while i &lt; 5:
    print(i)
    i += 1
```

Extracts parts of sequences using sequence [start: stop step]. Iteration: Traverses elements of a sequence, often using for or while loops.

List Comprehensions:

Create lists concisely. Dictionary Comprehensions: Create dictionaries concisely. Enumerate: Iterate with both index and value.

4.1.5 Array Manipulation

Array manipulation is a fundamental aspect of data processing and analysis in Python, especially when using libraries like NumPy, which provides efficient and flexible array operations. Below is a comprehensive guide to array manipulation using NumPy. Introduction to NumPy Arrays First, ensure that NumPy is installed and imported.

```
import numpy as np
```

Creating Arrays can be created from lists, tuples, or using NumPy's built-in functions.

```
# From a list
arr1 = np. array ([1, 2, 3, 4, 5])
print(arr1)
# From a tuple
arr2 = np. array((1, 2, 3, 4, 5))
print(arr2)
# Using built-in functions
arr3 = np. zeros ((2, 3))
print(arr3)
arr4 = np. ones ((2, 3))
print(arr4)
arr5 = np. arange (10)
print(arr5)
arr6 = np. linspace(0, 1, 5)
print(arr6)
```

Reshaping Arrays You can change the shape of an array using reshape.

```
arr = np. arange (12)
print(arr)
reshaped_arr = arr. reshape ((3, 4))
print(reshaped_arr)
```

Flattening Arrays Convert a multi-dimensional array into a one-dimensional array using flatten or ravel.

```
arr = np. array ([[1, 2, 3], [4, 5, 6]])
```

```
flattened_arr = arr. flatten ()
print(flattened_arr)
raveled_arr = arr. ravel ()
print(raveled_arr)
```

Concatenating Arrays Combine arrays using concatenate, vstack, or hstack.

```
arr1 = np. array ([[1, 2], [3, 4]])
arr2 = np. array ([[5, 6], [7, 8]])
# Concatenate along the first axis
concatenated_arr = np.concatenate((arr1, arr2), axis=0)
print(concatenated_arr)
# Stack arrays vertically
vstacked_arr = np.vstack((arr1, arr2))
print(vstacked_arr)
# Stack arrays horizontally
hstacked_arr = np.hstack((arr1, arr2))
print(hstacked_arr)
```

Splitting ArraysSplit arrays into multiple sub-arrays using split, hsplit, or vsplit.

```
arr = np.arange(16).reshape((4, 4))
# Split array into 2 sub-arrays along the first axis
split_arr = np.split(arr, 2, axis=0)
print(split_arr)
# Split array into 2 sub-arrays along the second axis
hsplit_arr = np.hsplit(arr, 2)
print(hsplit_arr)
# Split array into 2 sub-arrays along the first axis
vsplit_arr = np.vsplit(arr, 2)
print(vsplit_arr)
```

Adding and Removing Elements. Use append, insert, and delete to modify arrays.

```
arr = np.array([1, 2, 3, 4, 5])
# Append element to array
appended_arr = np.append(arr, 6)
print(appended_arr)
# Insert element at specific position
inserted_arr = np.insert(arr, 2, 10)
print(inserted_arr)
# Delete element at specific position
```

```
deleted_arr = np.delete(arr, 2)
print(deleted_arr)
```

Array TranspositionTranspose arrays using transpose or T.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Transpose the array
transposed_arr = arr.transpose()
print(transposed_arr)
# Alternative way
print(arr.T)
```

Element-wise Operations : Perform element-wise operations such as addition, subtraction, multiplication, and division.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
# Element-wise addition
print(arr1 + arr2)  # Output: [5 7 9]
# Element-wise subtraction
print(arr1 - arr2)  # Output: [-3 -3 -3]
# Element-wise multiplication
print(arr1 * arr2)  # Output: [ 4 10 18]
# Element-wise division
print(arr1 / arr2)  # Output: [0.25 0.4  0.5 ]
```

BroadcastingBroadcasting allows operations on arrays of different shapes.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([[1], [2], [3]])
# Broadcasting addition
result = arr1 + arr2
print(result)
```

Universal Functions (ufuncs)NumPy provides many universal functions for element-wise operations.

```
arr = np.array([1, 2, 3, 4])
# Square root
print(np.sqrt(arr))
# Exponential
print(np.exp(arr))
# Sine
```

```
print(np.sin(arr))
```

NumPy provides powerful tools for array manipulation, including: Creating arrays: From lists, tuples, and using built-in functions. Reshaping and flattening arrays. Concatenating and splitting arrays. Adding, inserting, and deleting elements. Transposing arrays. Performing element-wise operations. Broadcasting: Enabling operations on arrays of different shapes. Using universal functions: For mathematical operations. These capabilities make NumPy an essential library for numerical computations and data manipulation in Python.

Let Us Sum Up

In Python, working with packages like NumPy facilitates efficient numerical computations. **NumPy** provides the **ndarray** object, a powerful multidimensional array structure that supports a variety of mathematical operations. Basic operations with ndarray include **indexing**, which allows access to individual elements, **slicing** to extract subarrays, and **iteration** over array elements. NumPy also offers comprehensive **array manipulation** functions, such as reshaping, concatenating, and broadcasting, making it ideal for complex data processing and mathematical tasks. This combination of functionalities makes NumPy an essential tool for scientific computing and data analysis in Python.

Check Your Progress

1. What does **np.ndarray** represent in NumPy?
A) List object
B) DataFrame
C) n-dimensional array
D) Dictionary

2. How do you create a 1D NumPy array?
A) np.array([1, 2, 3, 4])
B) np.zeros((4,))
C) np.empty(4)
D) np.linspace(4)

3.What function returns a NumPy array filled with zeros?
A) np.ones(shape)
B) np.empty(shape)
C) np.zeros(shape)
D) np.full(shape, 0)

4.How do you access the third element of a NumPy array **arr**?
A) arr[3]

B) arr[2]
C) arr(3)
D) arr[1]

5.What method is used to reshape a NumPy array?
A) reform()
B) resize()
C) reshape()
D) transform()

6.How do you slice the first three elements of a NumPy array **arr**?
A) arr[3:]
B) arr[:3]
C) arr[1:4]
D) arr[0:3]

7.Which function creates an array with a specified range of numbers?
A) np.linspace(start, stop, num)
B) np.arange(start, stop, step)
C) np.random.random(size)
D) np.full(size, fill_value)

8. What is the output of **np.eye(3)**?
A) A 3x3 matrix of zeros
B) A 3x3 identity matrix
C) A 3x3 matrix of ones
D) A 3x3 random matrix

9.How can you concatenate two NumPy arrays **a** and **b** horizontally?
A) np.vstack((a, b))
B) np.concatenate((a, b))
C) np.hstack((a, b))
D) np.split((a, b))

10.What function computes the mean of a NumPy array?
A) np.median(arr)
B) np.mean(arr)
C) np.average(arr)
D) np.sum(arr)

SECTION 4.2: PANDAS

Pandas is a powerful and flexible open-source data manipulation and analysis library for Python. It provides data structures like Data Frame and Series, which are designed to handle structured data intuitively and efficiently. Here's a comprehensive guide to using Pandas for data manipulation and analysis. Introduction to Pandas First, ensure you have Pandas installed and imported.

```
!pip install pandas
```

```
import pandas as pd
```

Data Structures in Pandas Series A Series is a one-dimensional array-like object that can hold any data type.

```
# Creating a Series from a list
data = [1, 2, 3, 4, 5]
series = pd.Series(data)
print(series)
# Creating a Series with a custom index
series = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])
print(series)
```

DataFrame: A DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure.

```
# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
print(df)
# Creating a DataFrame from a list of lists
data = [
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'Los Angeles'],
    ['Charlie', 35, 'Chicago']
]
df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
print(df)
```

Reading and Writing DataReading DataPandas can read data from various file formats.

```
# Reading a CSV file
df = pd.read_csv('file.csv')
# Reading an Excel file
df = pd.read_excel('file.xlsx')
# Reading a JSON file
df = pd.read_json('file.json')
# Reading from a SQL database
```

```
import sqlite3
conn = sqlite3.connect('database.db')
df = pd.read_sql('SELECT * FROM table', conn)
```

Writing DataPandas can also write data to various file formats.

```
# Writing to a CSV file
df.to_csv('file.csv', index=False)
# Writing to an Excel file
df.to_excel('file.xlsx', index=False)
# Writing to a JSON file
df.to_json('file.json', orient='records')
# Writing to a SQL database
df.to_sql('table', conn, if_exists='replace', index=False)
```

Basic Data Operations Viewing Data

```
# Viewing the first few rows
print(df.head())
# Viewing the last few rows
print(df.tail())
# Getting information about the DataFrame
print(df.info())
# Getting basic statistics
print(df.describe())
Selecting Data
# Selecting a single column
print(df['Name'])
# Selecting multiple columns
print(df[['Name', 'Age']])
# Selecting rows by index
print(df.iloc[0])  # First row
print(df.iloc[0:2])  # First two rows
# Selecting rows by label
print(df.loc[0])  # First row
print(df.loc[0:2])  # First three rows
# Conditional selection
print(df[df['Age'] &gt; 30])
```

Modifying DataAdding and Modifying Columns

```
# Adding a new column
df['Country'] = ['USA', 'USA', 'USA']
print(df)
# Modifying an existing column
df['Age'] = df['Age'] + 1
print(df)
```

Dropping Columns and Rows

```
# Dropping a column
df = df.drop('Country', axis=1)
print(df)
# Dropping a row
df = df.drop(0, axis=0)
print(df)
```

Handling Missing Data

```
# Checking for missing values
print(df.isnull())
# Dropping rows with missing values
df = df.dropna()
print(df)
# Filling missing values
df = df.fillna(0)
print(df)
```

Grouping and Aggregating Data

```
# Grouping data by a column
grouped = df.groupby('City')
print(grouped.mean())
# Aggregating data with multiple functions
agg = grouped.agg({'Age': ['mean', 'max'], 'Name': 'count'})
print(agg)
```

Merging and Joining DataPandas provides several functions for combining DataFrames.

```
# Merging DataFrames
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['A', 'B', 'D'], 'value2': [4, 5, 6]})
merged = pd.merge(df1, df2, on='key', how='inner')
print(merged)
# Concatenating DataFrames
```

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'A': ['A3', 'A4', 'A5'], 'B': ['B3', 'B4', 'B5']})
concatenated = pd.concat([df1, df2])
print(concatenated)
```

Working with Time Series DataPandas has robust support for working with time series data.

```
# Creating a time series
dates = pd.date_range('20230101', periods=6)
df = pd.DataFrame(np.random.randn(6, 4), index=dates,
columns=list('ABCD'))
print(df)
# Resampling time series data
resampled = df.resample('M').mean()
print(resampled)
```

Advanced Indexing MultiIndex Pandas supports hierarchical indexing for working with high-dimensional data.

```
# Creating a MultiIndex DataFrame
arrays = [
   ['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
index = pd.MultiIndex.from_arrays(arrays, names=('first', 'second'))
df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A',
'B'])
print(df)
# Accessing data in a MultiIndex DataFrame
print(df.loc['bar'])
print(df.loc[('bar', 'one')]
```

Pandas is an essential tool for data manipulation and analysis in Python, providing powerful data structures and a wealth of functionalities for: Creating and manipulating Series and Data Frames. Reading from and writing to various file formats. Performing basic and advanced data operations. Handling missing data. Grouping, aggregating, merging, and joining data. Working with time series data. Advanced indexing techniques. By mastering Pandas, you can efficiently manage and analyze large datasets, making it an indispensable library for data science and analytics.

4.2.1– THE SERIES – THE DATAFRAME

Pandas Series and Data Frame Pandas provides two primary data structures for data manipulation: Series and Data Frame. Understanding these structures is fundamental to working effectively with Pandas. The Series A Series is a one-dimensional array-like object capable of holding any data type (integers, strings, floating-point numbers, Python objects, etc.). It is similar to a column in a Data Frame or a one-dimensional array in NumPy. Creating a Series You can create a Series from a list, dictionary, or scalar value.

```
import pandas as pd
# From a list
data = [1, 2, 3, 4, 5]
series = pd.Series(data)
print(series)
# From a list with a custom index
series = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])
print(series)
# From a dictionary
data = {'a': 1, 'b': 2, 'c': 3}
series = pd. Series(data)
print(series)
# From a scalar value
series = pd.Series(5, index=['a', 'b', 'c'])
print(series)
```

Accessing Data in a Series You can access data in a Series using the index.

```
series = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
# Accessing by label
print(series['a'])  # Output: 1
# Accessing by position
print(series[0])  # Output: 1
# Accessing multiple elements
print(series[['a', 'c', 'e']])  # Output: [1, 3, 5]
# Slicing
print(series['b':'d'])  # Output: b    2, c    3, d    4
```

Vectorized Operations Series support vectorized operations, making element-wise operations easy.

```
series = pd.Series([1, 2, 3, 4, 5])
# Adding a scalar
print (series + 1)  # Output: [2, 3, 4, 5, 6]
# Element-wise addition
print(series + series)  # Output: [2, 4, 6, 8, 10]
# Applying functions
print (series.apply(lambda x: x**2))  # Output: [1, 4, 9, 16, 25]
```

The Data Frame A DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or SQL table, or a dictionary of Series objects. Creating a DataFrameYou can create a DataFrame from dictionaries, lists, or other DataFrames.

```
# From a dictionary
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
print(df)
# From a list of dictionaries
data = [
   {'Name': 'Alice', 'Age': 25, 'City': 'New York'},
   {'Name': 'Bob', 'Age': 30, 'City': 'Los Angeles'},
   {'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}
]
df = pd.DataFrame(data)
print(df)
# From a list of lists
data = [
   ['Alice', 25, 'New York'],
   ['Bob', 30, 'Los Angeles'],
   ['Charlie', 35, 'Chicago']
]
df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
print(df)
```

Accessing Data in a DataFrameYou can access data in a DataFrame using labels and positions.

```
# Accessing columns
print(df['Name'])  # Output: Series of names
# Accessing multiple columns
print(df[['Name', 'Age']])  # Output: DataFrame with Name and Age c
columns
# Accessing rows by index
print(df.iloc[0])  # Output: First row
# Accessing rows by label
df.index = ['row1', 'row2', 'row3']
print(df.loc['row1'])  # Output: First row
# Accessing a specific value
print(df.at['row1', 'Name'])  # Output: Alice
print(df.iat[0, 0])  # Output: Alice
```

Modifying Data You can modify the DataFrame by adding or removing columns and rows.

```
# Adding a new column
df['Country'] = ['USA', 'USA', 'USA']
print(df)
# Modifying an existing column
df['Age'] = df['Age'] + 1
print(df)
# Dropping a column
df = df.drop('Country', axis=1)
print(df)
# Dropping a row
df = df.drop('row1', axis=0)
print(df)
```

Operations on DataFramesDataFrames support a wide range of operations, including aggregation, filtering, and more.Aggregation

```
# Aggregation functions
print(df.sum())
print(df.mean())
print(df.describe())
# Grouping and aggregation
grouped = df.groupby('City')
print(grouped.mean())
```

Filtering

```
# Filtering rows
filtered_df = df[df['Age'] &gt; 30]
print(filtered_df)
```

Merging and Joining

```
# Merging DataFrames
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['A', 'B', 'D'], 'value2': [4, 5, 6]})
merged_df = pd.merge(df1, df2, on='key', how='inner')
print(merged_df)
# Concatenating DataFrames
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'A': ['A3', 'A4', 'A5'], 'B': ['B3', 'B4', 'B5']})
concatenated_df = pd.concat([df1, df2])
print(concatenated)
```

Time Series Data Pandas has robust support for handling time series data.

```
# Creating a time series DataFrame
dates = pd.date_range('2023-01-01', periods=6)
df = pd.DataFrame(np.random.randn(6, 4), index=dates,
columns=list('ABCD'))
print(df)
# Resampling time series data
resampled_df = df.resample('M').mean()
print(resampled_df)
```

Pandas Series and Data Frames are powerful tools for data manipulation and analysis.

Series: One-dimensional array-like structure with labelled indices.

DataFrame: Two-dimensional, mutable, and heterogeneous tabular structure.

Data Operations: Creating, accessing, modifying, aggregating, filtering, merging, and joining data.

Time Series: Handling and analyzing time-based data. Mastering these data structures and their operations will enable you to efficiently manipulate and analyze complex datasets in Python.

4.2.3– THE INDEX OBJECTS

The Index Objects in Pandas

In Pandas, the `Index` object is an essential part of both `Series` and `DataFrame`. It acts as the label for the rows and columns and provides the axis labels for the data structures.

Types of Index Objects

1. Range Index: A default index, like Python's built-in `range`.
2. Index: A generic, immutable sequence used for indexing and alignment.
3. MultiIndex: A hierarchical index for multi-dimensional data.
4. Datetime Index: An index of timestamp values.

Creating and Using Index Objects

```
import pandas as pd
#Creating a Series with a custom Index
data = [1, 2, 3, 4, 5]
index = pd.Index(['a', 'b', 'c', 'd', 'e'])
series = pd.Series(data, index=index)
print(series)
#Range Index
Creating a DataFrame with default Range Index
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
print(df.index)  # Output: Range Index(start=0, stop=3, step=1)
```

MultiIndex
Creating a MultiIndex

```
arrays = [
   ['bar', 'bar', 'baz', 'baz', 'foo', 'foo'],
   ['one', 'two', 'one', 'two', 'one', 'two']
]
multi_index = pd.MultiIndex.from_arrays(arrays, names=('first',
'second'))
```

Creating a Data Frame with MultiIndex

```
df = pd.DataFrame({'A': [1, 2, 3, 4, 5, 6]}, index=multi_index)
print(df)
```

DatetimeIndex

Creating a DatetimeIndex

```
dates = pd.date_range('2023-01-01', periods=6)
df = pd.DataFrame({'A': [1, 2, 3, 4, 5, 6]}, index=dates)
print(df.index) # Output: DatetimeIndex
print(df)
```

Index Object Methods and Properties

Basic Properties

```
#Creating an Index object
index = pd.Index(['a', 'b', 'c', 'd', 'e'])
#Accessing properties
print(index.size)  # Output: 5
print(index.shape)  # Output: (5,)
print(index.ndim)  # Output: 1
print(index.dtype)  # Output: object
```

Indexing and Slicing

Indexing and slicing Index objects

```
print(index[1])  # Output: b
print(index[1:3])  # Output: Index(['b', 'c'], dtype='object')
```

Operations on Index

Creating another Index object

```
index2 = pd.Index(['c', 'd', 'e', 'f', 'g'])
```

Intersection

```
print(index. intersection(index2))  # Output: Index(['c', 'd', 'e'],
dtype='object')
```

Union

```
print(index.Union(index2))  # Output: Index(['a', 'b', 'c', 'd', 'e', 'f', 'g'],
dtype='object')
```

Difference

```
print(index.difference(index2))  # Output: Index(['a', 'b'], dtype='object')
```

Working with MultiIndex
Creating and Accessing MultiIndex

```
# Creating a MultiIndex from tuples
tuples = [('bar', 'one'), ('bar', 'two'), ('baz', 'one'), ('baz', 'two')]
multi_index = pd.MultiIndex.from_tuples(tuples, names=['first',
'second'])
# Creating a DataFrame with MultiIndex
df = pd.DataFrame({'A': [1, 2, 3, 4]}, index=multi_index)
print(df)
# Accessing data in a MultiIndex DataFrame
print(df.loc['bar'])  # Accessing all 'bar' entries
print(df.loc[('bar', 'one')])  # Accessing a specific entry
```

Multi Index Methods

```
# Creating a MultiIndex
arrays = [
   ['bar', 'bar', 'baz', 'baz', 'foo', 'foo'],
   ['one', 'two', 'one', 'two', 'one', 'two']
]
multi_index = pd.MultiIndex.from_arrays(arrays, names=('first',
'second'))
# Getting levels and labels
print(multi_index.levels)  # Output: Levels of the MultiIndex
print(multi_index.labels)  # Deprecated, use multi_index.codes instead
print(multi_index.codes)   # Output: Codes of the MultiIndex
```

Handling Index and Columns in Data Frame

Setting and Resetting Index

```
#Setting a new index
df = pd.DataFrame({'A': ['foo', 'bar', 'baz'], 'B': [1, 2, 3]})
```

```
df = df.set_index('A')
print(df)
Resetting the index
df = df.reset_index()
print(df)
```

Renaming index and columns

```
df = pd.DataFrame({'A': [1, 2, 3]}, index=['a', 'b', 'c'])
df = df.rename(index={'a': 'x', 'b': 'y'}, columns={'A': 'Value'})
print(df)
```

Index objects in Pandas are a powerful feature for handling and manipulating data. Key points include:

1. Types of Indexes: `Range Index`, `Index`, `MultiIndex`, `DatetimeIndex`.
2. Creation and Access: You can create and access different types of Indexes using various methods.
3. Operations: Index objects support set operations like intersection, union, and difference.
4. MultiIndex: Allows for multi-dimensional data and provides methods for hierarchical indexing.
5. Index Manipulation: Setting, resetting, and renaming indexes and columns enhance data management.

Let Us Sum Up

In this unit, we explored the essential Python packages for data manipulation and analysis, primarily focusing on NumPy and Pandas. We delved into NumPy's ndarray, learning about its creation, basic operations, and advanced indexing techniques. We also covered array manipulation methods to reshape and merge arrays efficiently. Moving on to Pandas, we examined its core data structures: Series and DataFrame, along with their indexing and slicing operations. We learned how to perform various data manipulation tasks using these structures, including handling

missing data, grouping, and sorting. Finally, we discussed the Index object in Pandas and its role in enhancing data alignment and indexing.

Check Your Progress

1. What is the primary purpose of NumPy in Python?
   - A) Web development
   - B) Numerical and scientific computing
   - C) Machine learning algorithms
   - D) Text processing
2. Which object in NumPy is used to represent multi-dimensional arrays?
   - A) list
   - B) dict
   - C) ndarray
   - D) set
3. How do you create a NumPy array from a list [1, 2, 3]?
   - A) np.array([1, 2, 3])
   - B) np.create([1, 2, 3])
   - C) np.ndarray([1, 2, 3])
   - D) np.make([1, 2, 3])
4. Which function is used to perform element-wise addition of two NumPy arrays?
   - A) np.add()
   - B) np.sum()
   - C) np.plus()
   - D) np.concatenate()
5. What does the shape attribute of a NumPy array return?
   - A) The size of each element in bytes
   - B) The total number of elements
   - C) The dimensions of the array
   - D) The data type of the elements

6. How do you access the element at the second row and third column in a 2D NumPy array arr?
    A) arr[1, 2]
    B) arr[2, 3]
    C) arr[3, 2]
    D) arr[2, 1]
7. Which operation is used to combine multiple arrays along a specified axis in NumPy?
    A) np.concatenate()
    B) np.append()
    C) np.insert()
    D) np.expand()
8. What does the ndim attribute of a NumPy array indicate?
    A) Number of elements
    B) Data type of elements
    C) Number of dimensions
    D) Memory size of the array
9. How do you reshape a NumPy array arr of size 9 into a 3x3 matrix?
    A) arr.reshape(3, 3)
    B) arr.resize(3, 3)
    C) np.reshape(arr, 3, 3)
    D) arr.shape(3, 3)
10. Which function is used to find the maximum value in a NumPy array?
    A) np.max()
    B) np.maximum()
    C) np.argmax()
    D) np.maxvalue()
11. What is the primary data structure used in the Pandas library for 1-dimensional labeled data?
    A) DataFrame
    B) Series
    C) ndarray

D) Index

12. How do you create a Pandas DataFrame from a dictionary?

    A) pd.DataFrame.from_dict()

    B) pd.DataFrame(dict)

    C) pd.createDataFrame(dict)

    D) pd.DataFrame(dict)

13. Which method would you use to read a CSV file into a Pandas DataFrame?

    A) pd.read_csv()

    B) pd.load_csv()

    C) pd.import_csv()

    D) pd.load_csvfile()

14. How do you select the column 'age' from a Pandas DataFrame df?

    A) df.age

    B) df['age']

    C) df[['age']]

    D) All of the above

15. Which attribute of a Pandas DataFrame returns the column labels?

    A) df.columns

    B) df.index

    C) df.labels

    D) df.names

16. How do you add a new column 'total' to a DataFrame df by summing two existing columns 'A' and 'B'?

    A) df['total'] = df['A'] + df['B']

    B) df.add_column('total', df['A'] + df['B'])

    C) df.new_column('total', df['A'] + df['B'])

    D) df['total'] = df.sum(['A', 'B'])

17. What does the iloc function in Pandas do?

    A) Selects data by label

    B) Selects data by integer-location based indexing

    C) Inserts a column

    D) Loads data from a file

18. Which method would you use to handle missing data by filling with the mean value in a Pandas DataFrame?
    A) df.fillna(df.mean())
    B) df.replaceNaN(df.mean())
    C) df.interpolate(df.mean())
    D) df.fill_mean(df.mean())
19. How do you group data by a column 'group' in a Pandas DataFrame df?
    A) df.groupby('group')
    B) df.group_by('group')
    C) df.group('group')
    D) df.groupby_column('group')
20. Which method would you use to sort a DataFrame df by the values in column 'A'?
    A) df.sort_values('A')
    B) df.sort_by('A')
    C) df.sort_column('A')
    D) df.sort_by_column('A')
21. What is the primary purpose of the Index object in Pandas?
    A) To store data
    B) To label and align data
    C) To perform calculations
    D) To visualize data
22. How do you convert a Pandas DataFrame df to a NumPy array?
    A) df.to_numpy()
    B) df.to_ndarray()
    C) df.as_matrix()
    D) df.to_array()
23. What is the use of the head() method in Pandas?
    A) To get the first few rows of a DataFrame
    B) To get the last few rows of a DataFrame
    C) To get the column names
    D) To get the index names

24. Which Pandas function is used to concatenate DataFrames vertically?
    A) pd.concat()
    B) pd.append()
    C) pd.merge()
    D) pd.join()
25. What does the describe() method do in Pandas?
    A) Provides a summary of statistics for numerical columns
    B) Describes the data types of columns
    C) Provides a summary of text data
    D) Displays the column names
26. How do you rename the columns of a DataFrame df?
    A) df.rename(columns=new_names)
    B) df.rename_columns(new_names)
    C) df.columns.rename(new_names)
    D) df.set_columns(new_names)
27. What does the transpose() method do in a Pandas DataFrame?
    A) Swaps rows and columns
    B) Rotates the DataFrame 90 degrees
    C) Inverts the DataFrame
    D) Reverses the DataFrame
28. Which method in Pandas is used to merge two DataFrames based on a key?
    A) pd.merge()
    B) pd.concat()
    C) pd.join()
    D) pd.bind()
29. What is the use of the pivot_table() method in Pandas?
    A) To create a spreadsheet-style pivot table
    B) To plot data
    C) To transform data
    D) To filter data
30. How do you reset the index of a DataFrame df?
    A) df.reset_index()

B) df.set_index()

C) df.reset()

D) df.clear_index()

SECTION 4.3 DATA VISALIZATION WITH MATPLOTLIB

Matplotlib is a powerful and versatile plotting library for Python that enables you to create a wide variety of static, animated, and interactive visualizations. Here's a comprehensive guide to getting started with Matplotlib and using it to create various types of plots. Installation and Import First, ensure you have Matplotlib installed:shCopy codepip install matplotlib.

Then, import the necessary modules:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Basic Plotting Line Plot A simple line plot can be created using plt.plot.

```
# Creating data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Plotting the data
plt.plot(x, y)
# Adding titles and labels
plt.title("Sine Wave")
plt.xlabel("x")
plt.ylabel("sin(x)")
# Displaying the plot
plt.show()
```

Customizing Plots Adding Legends and Grid

```
plt.plot(x, y, label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.legend()
plt.grid(True)
plt.title("Sine and Cosine Waves")
```

```
plt.xlabel("x")
plt.ylabel("Value")
plt.show()
```

Customizing Lines and Markers

```
plt.plot(x, y, color='blue', linestyle='--', linewidth=2, marker='o',
markersize=5, label='sin(x)')
plt.legend()
plt.title("Customized Sine Wave")
plt.show()
```

Subplots allow you to create multiple plots in a single figure.

```
# Creating a 2x1 subplot
fig, axs = plt.subplots(2, 1, figsize=(10, 8))
# Plotting on the first subplot
axs[0].plot(x, y, 'r')
axs[0].set_title('Sine Wave')
# Plotting on the second subplot
axs[1].plot(x, np.cos(x), 'b')
axs[1].set_title('Cosine Wave')
# Adding overall titles and labels
plt.suptitle("Sine and Cosine Waves")
plt.xlabel("x")
plt.ylabel("Value")
plt.show()
```

Different Types of Plots Scatter Plot

```
# Creating random data
x = np.random.rand(50)
y = np.random.rand(50)
colors = np.random.rand(50)
sizes = 1000 * np.random.rand(50)
# Creating a scatter plot
plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='viridis')
plt.colorbar()
plt.title("Scatter Plot")
plt.show()
```

Bar Plot

```
# Creating data
categories = ['A', 'B', 'C', 'D']
```

```
values = [5, 7, 8, 6]
# Creating a bar plot
plt.bar(categories, values, color='purple')
plt.title("Bar Plot")
plt.xlabel("Categories")
plt.ylabel("Values")
plt.show()
```

Histogram

```
# Creating random data
data = np.random.randn(1000)
# Creating a histogram
plt.hist(data, bins=30, color='green', alpha=0.7)
plt.title("Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

Box Plot

```
# Creating random data
data = [np.random.rand(50), np.random.rand(50), np.random.rand(50)]
# Creating a box plot
plt.boxplot(data)
plt.title("Box Plot")
plt.xlabel("Category")
plt.ylabel("Value")
plt.show()
```

Pie Chart

```
# Creating data
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
explode = (0.1, 0, 0, 0)  # Explode the first slice
# Creating a pie chart
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
autopct='%1.1f%%', shadow=True, startangle=140)
plt.title("Pie Chart")
plt.show()
```

Advanced Plot Customizations& Adding Annotations

```
# Creating data
x = np.linspace(0, 10, 100)
y = np.sin(x)
```

```
# Plotting the data
plt.plot(x, y)
# Adding an annotation
plt.annotate('Max Point', xy=(np.pi/2, 1), xytext=(np.pi/2 + 1, 1.5),
          arrowprops=dict(facecolor='black', shrink=0.05))
plt.title("Sine Wave with Annotation")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.show()
```

Using Logarithmic Scale

```
# Creating data
x = np.linspace(0.1, 10, 100)
y = np.exp(x)
# Plotting with logarithmic scale
plt.plot(x, y)
plt.yscale('log')
plt.title("Exponential Growth (Log Scale)")
plt.xlabel("x")
plt.ylabel("exp(x)")
plt.show()
```

Saving PlotsYou can save plots in various formats such as PNG, PDF, SVG, and more.

```
# Creating data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Plotting the data
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("x")
plt.ylabel("sin(x)")
# Saving the plot
plt.savefig("sine_wave.png")
plt.savefig("sine_wave.pdf")
plt.show()
```

Interactive Plots with WidgetsUsing the ipympl backend and Jupyter widgets, you can create interactive plots.

```
# Install ipympl
```

```
!pip install ipympl
# Enable the backend
%matplotlib widget
# Creating an interactive plot
plt.figure()
plt.plot(x, y)
plt.title("Interactive Sine Wave")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.show()
```

Matplotlib provides a comprehensive toolkit for creating a wide range of static, animated, and interactive plots. By mastering the basics and exploring its advanced features, you can create visually appealing and informative data visualizations. Here’s a quick recap of what we covered: Basic Plotting: Line plots, legends, grid. Customization: Line styles, markers, subplots. Various Plot Types: Scatter, bar, histogram, box, pie charts. Advanced Customizations: Annotations, logarithmic scale. Saving Plots: Exporting plots to different file formats. Interactive Plots: Using Jupyter widgets for interactivity.

4.3.1 THE MATPLOTLIB ARCHITECTURE : PYPLOT

Matplotlib, a comprehensive library for creating static, animated, and interactive visualizations in Python, is built on a layered architecture. Understanding this architecture helps in efficiently using Matplotlib for diverse plotting needs. The key component most users interact with is pyplot, which provides a high-level interface to the underlying architecture. Overview of Matplotlib Architecture Pyplot Interface (matplotlib.pyplot): A collection of functions that make Matplotlib work like MATLAB, designed to make creating common plots as simple as possible.Figure and Axes: Core objects in the Matplotlib object hierarchy, representing the entire figure and individual plots within a figure, respectively. Backend Layer: Handles all the drawing/rendering tasks. Includes several backends for different outputs (e.g., PNG, PDF, interactive interfaces).Artists: All visual elements in a plot (e.g., lines, text, markers).The Pyplot Interface pyplot is a state-based interface to Matplotlib. It keeps track of the current

figure and axes, making it easy to create and manage plots. Basic Structure Here's a simple example using pyplot:

```
import matplotlib.pyplot as plt
import numpy as np
# Creating data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Creating a plot
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("x")
plt.ylabel("sin(x)")
# Displaying the plot
plt.show()
```

Core ComponentsFigure: The entire figure, a container holding all plot elements. It's the top-level container.Axes: Represents a single plot. A figure can contain multiple axes (plots).Axis: Represents the x and y-axis.

```
# Creating a figure and axes
fig, ax = plt.subplots()
# Plotting on the axes
ax.plot(x, y)
ax.set_title("Sine Wave")
ax.set_xlabel("x")
ax.set_ylabel("sin(x)")
# Displaying the plot
plt.show()
```

Creating and Customizing Plots with PyplotCreating Multiple SubplotsYou can create multiple subplots within a single figure using plt.subplots.

```
# Creating multiple subplots
fig, axs = plt.subplots(2, 2)
# Plotting on each subplot
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Sine')
axs[0, 1].plot(x, np.cos(x), 'r')
axs[0, 1].set_title('Cosine')
axs[1, 0].plot(x, np.tan(x), 'g')
```

```
axs[1, 0].set_title('Tangent')
axs[1, 1].plot(x, -y, 'k')
axs[1, 1].set_title('Negative Sine')
# Adjusting layout
plt.tight_layout()
plt.show()
```

Customizing Plots You can customize plots using various pyplot functions.

```
# Creating a plot with customized lines and markers
plt.plot(x, y, label='Sine', color='blue', linestyle='--', marker='o',
markersize=5)
# Adding labels and title
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.title("Customized Sine Wave")
plt.legend()
# Adding grid
plt.grid(True)
# Displaying the plot
plt.show()
```

Advanced Plot Customizations Using Artists for Fine ControlArtists are the objects that represent everything you see on a plot (lines, text, etc.).

```
# Creating a figure and axes
fig, ax = plt.subplots()
# Creating a line artist
line, = ax.plot(x, y, label='Sine', color='blue')
# Customizing the line artist
line.set_linestyle('--')
line.set_marker('o')
# Adding text
ax.text(5, 0, "Center", fontsize=12, ha='center')
# Displaying the plot
plt.legend()
plt.show()
```

Handling Multiple Figures and Axes You can create and manage multiple figures and axes.

```
# Creating multiple figures
fig1, ax1 = plt.subplots()
fig2, ax2 = plt.subplots()
# Plotting on the first figure
ax1.plot(x, y)
ax1.set_title('Figure 1: Sine')
# Plotting on the second figure
ax2.plot(x, np.cos(x), 'r')
ax2.set_title('Figure 2: Cosine')
# Displaying both figures
plt.show()
```

Understanding the Backend LayerMatplotlib can render plots using different backends, which can be categorized into:Interactive Backends: For use with GUI interfaces (e.g., Qt5Agg, TkAgg).Non-Interactive Backends: For generating files (e.g., Agg for PNG, PDF for PDF files).You can switch backends using matplotlib.use.

```
import matplotlib
# Switching to the 'Agg' backend for file output
matplotlib.use('Agg')
import matplotlib.pyplot as plt
# Creating a plot
plt.plot(x, y)
plt.savefig("sine_wave.png")
```

Interactive PlotsMatplotlib integrates with interactive environments like Jupyter notebooks.

```
# Enabling interactive mode in Jupyter
%matplotlib notebook
# Creating an interactive plot
plt.plot(x, y)
plt.title("Interactive Sine Wave")
plt.show()
```

The Matplotlib architecture, particularly the pyplot interface, provides a powerful and flexible framework for creating a wide variety of plots in Python. Key components include:

1. Figure and Axes: Fundamental objects for creating plots.

2. Pyplot Interface: Simplifies creating common plots with state-based functions.
3. Customization: Extensive options for customizing plots, including lines, markers, and text.
4. Artists: Fine control over plot elements.
5. Backend Layer: Manages rendering and output, supporting various formats and interactive environments.

By understanding and utilizing these components, you can create sophisticated and customized visualizations to effectively communicate your data insights.

4.3.2– THE PLOTTING WINDOW: ADDING ELEMENT TO THE CHART WITH MATPLOTLIB

Adding various elements to a chart in Matplotlib can significantly enhance the clarity and informativeness of your visualizations. Here's a detailed guide on how to add and customize these elements, such as titles, labels, legends, grids, annotations, and more. Basic PlotLet's start with a simple plot to demonstrate the addition of various elements.

```
import matplotlib.pyplot as plt
import numpy as np
# Creating data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Creating a basic plot
plt.plot(x, y)
# Displaying the plot
plt.show()
```

Adding Titles and LabelsTitle: Add a title to the entire plot to provide context.

```
plt.plot(x, y)
plt.title("Sine Wave")
plt.show()
```

Axis Labels Label the x and y axes to indicate what each axis represents.

```
plt.plot(x, y)
```

```
plt.title("Sine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.show()
```

Adding a Legend A legend helps identify different data series in the plot. Use the label parameter in the plot function and then call plt.legend().

```
plt.plot(x, y, label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.title("Sine and Cosine Waves")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.legend()
plt.show()
```

Adding a Grid A grid improves the readability of the plot by making it easier to align data points with the axes.

```
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.grid(True)
plt.show()
```

Adding Annotations Annotations are useful for highlighting specific points or areas on the plot.

```
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
# Adding an annotation
plt.annotate('Max Point', xy=(np.pi/2, 1), xytext=(np.pi/2+1, 1.5),
arrowprops=dict(facecolor='black', shrink=0.05))
plt.grid(True)
plt.show()
```

Adding Text You can add custom text at any location on the plot using plt.text().

```
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
```

```
# Adding text
plt.text(5, 0, "Center", fontsize=12, ha='center')
plt.grid(True)
plt.show()
```

Multiple Subplots Create a grid of subplots using plt.subplots(), allowing multiple plots within a single figure.

```
fig, axs = plt.subplots(2, 2, figsize=(10, 8))
# Plotting on each subplot
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Sine')
axs[0, 1].plot(x, np.cos(x), 'r')
axs[0, 1].set_title('Cosine')
axs[1, 0].plot(x, np.tan(x), 'g')
axs[1, 0].set_title('Tangent')
axs[1, 1].plot(x, -y, 'k')
axs[1, 1].set_title('Negative Sine')
plt.tight_layout()
plt.show()
```

Customizing Plot Elements Line Styles and Colors Customize the appearance of the lines in your plot.

```
plt.plot(x, y, color='blue', linestyle='--', linewidth=2, marker='o',
markersize=5)
plt.title("Customized Sine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.grid(True)
plt.show()
```

Axis LimitsControl the range of the axes to focus on specific data ranges.

```
plt.plot(x, y)
plt.title("Sine Wave with Axis Limits")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
# Setting axis limits
plt.xlim(0, 5)
plt.ylim(-1, 1)
plt.grid(True)
```

```
plt.show()
```

Adding Multiple Figures You can create and manage multiple figures using plt.figure().

```
# First figure
plt.figure()
plt.plot(x, y)
plt.title("Figure 1: Sine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
# Second figure
plt.figure()
plt.plot(x, np.cos(x), 'r')
plt.title("Figure 2: Cosine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.show()
```

Saving Plots Save your plots to files using plt.savefig().

```
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.grid(True)
# Saving the plot
plt.savefig("sine_wave.png")
plt.savefig("sine_wave.pdf")
plt.show()
```

By adding various elements to your Matplotlib plots, you can significantly enhance their readability and informativeness. Here's a recap of what we covered:

1. Titles and Labels: Add context and describe your axes.
2. Legend: Identify different data series.
3. Grid: Improve readability.Annotations and Text: Highlight specific points and add custom notes.
4. Multiple Subplots: Create complex figures with multiple plots.
5. Customization: Adjust line styles, colors, and axis limits.
6. Multiple Figures: Manage multiple plots in separate windows.

7. Saving Plots: Export plots to different file formats.

Using these techniques, you can create detailed and highly informative visualizations tailored to your specific data analysis needs.

4.3.3– LINE CHARTS – BAR CHARTS – PIE CHARTS

Line Charts Line charts are suitable for showing trends over time or continuous data.

```
import matplotlib.pyplot as plt
import numpy as np
# Generating some data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Creating a line chart
plt.plot(x, y)
plt.title("Line Chart")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.grid(True)
plt.show()
```

Bar ChartsBar charts are effective for comparing categorical data or showing changes over time.

```
# Generating some data
categories = ['A', 'B', 'C', 'D']
values = [5, 7, 3, 9]
# Creating a bar chart
plt.bar(categories, values, color='skyblue')
plt.title("Bar Chart")
plt.xlabel("Categories")
plt.ylabel("Values")
plt.grid(axis='y')
plt.show()
```

Pie ChartsPie charts are useful for displaying proportions of a whole.

```
# Generating some data
```

```
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
explode = (0.1, 0, 0, 0)  # Explode the first slice
# Creating a pie chart
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
autopct='%1.1f%%', shadow=True, startangle=140)
plt.title("Pie Chart")
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a
circle.

plt.show()
```

Line Charts: Suitable for showing trends over time or continuous data.

Bar Charts: Effective for comparing categorical data or showing changes over time.

Pie Charts: Useful for displaying proportions of a whole.

LET US SUM UP

In this unit, we explored data visualization using Matplotlib, focusing on its architecture and the Pyplot module. We learned about the plotting window and how to add various elements like titles, labels, and legends to charts. We covered the creation of line charts to display trends, bar charts for comparing categorical data, and pie charts to illustrate proportions. Through practical examples, we saw how these visualizations help in interpreting and presenting data effectively. This foundational knowledge equips us with the skills to create informative and aesthetically pleasing visual representations of data.

Check your progress

1. What is the primary purpose of Matplotlib in Python?
   A) Data manipulation
   B) Data visualization
   C) Data storage
   D) Data encryption

2. Which module in Matplotlib is commonly used for plotting?
    A) pyplot
    B) plotter
    C) graph
    D) plotlib
3. What is the command to import the pyplot module from Matplotlib?
    A) import Matplotlib as plt
    B) import pyplot as plt
    C) from matplotlib import pyplot as plt
    D) import matplotlib.pyplot as plot
4. In a line chart, what does each point on the line represent?
    A) A different dataset
    B) A data value at a specific position
    C) A category label
    D) A legend item
5. Which method is used to create a new figure in Matplotlib?
    A) plt.show()
    B) plt.figure()
    C) plt.create()
    D) plt.new()
6. How can you add a title to a plot in Matplotlib?
    A) plt.add_title("Title")
    B) plt.title("Title")
    C) plt.set_title("Title")
    D) plt.name("Title")
7. What is the command to display a plot in Matplotlib?
    A) plt.show()
    B) plt.display()
    C) plt.render()
    D) plt.plot()
8. Which method is used to label the x-axis in a plot?
    A) plt.x_label("Label")

B) plt.xlabel("Label")
C) plt.axis_label("Label")
D) plt.xname("Label")

9. How do you create a bar chart in Matplotlib?
A) plt.plot()
B) plt.bar()
C) plt.line()
D) plt.barchart()

10. What function is used to create a pie chart in Matplotlib?
A) plt.pie()
B) plt.piechart()
C) plt.pie_plot()
D) plt.chart()

11. Which attribute in a bar chart specifies the heights of the bars?
A) heights
B) values
C) width
D) height

12. How can you add a legend to a plot?
A) plt.add_legend()
B) plt.legend()
C) plt.create_legend()
D) plt.show_legend()

13. What is the purpose of the 'label' parameter in plot functions?
A) To label data points
B) To create axis labels
C) To add a title to the plot
D) To specify the legend text

14. Which function would you use to set the limits of the x-axis?
A) plt.set_xlim()
B) plt.xlim()

C) plt.limit_x()

D) plt.set_axis_limit()

15. What does the 'color' parameter in a plotting function do?

A) Sets the background color

B) Sets the color of the plot lines or markers

C) Sets the axis color

D) Sets the title color

16. How do you save a plot as an image file in Matplotlib?

A) plt.save()

B) plt.savefig()

C) plt.saveimage()

D) plt.export()

17. Which function is used to create subplots in a figure?

A) plt.subplot()

B) plt.subplots()

C) plt.add_subplot()

D) plt.create_subplot()

18. In a pie chart, what does the 'autopct' parameter control?

A) The colors of the slices

B) The size of the pie

C) The format of the percentage labels

D) The spacing between slices

19. What is the 'figsize' parameter used for?

A) Setting the resolution of the plot

B) Setting the size of the figure

C) Setting the size of the labels

D) Setting the size of the legend

20. How do you set a grid on a plot?

A) plt.grid()

B) plt.add_grid()

C) plt.show_grid()

D) plt.set_grid()

21. What does the 'linewidth' parameter control in a plot?
    A) The width of the plot frame
    B) The width of the plot title
    C) The width of the lines in the plot
    D) The width of the legend border
22. In a scatter plot, which parameter controls the size of the points?
    A) size
    B) s
    C) point_size
    D) marker_size
23. How can you create a horizontal bar chart?
    A) plt.barh()
    B) plt.hbar()
    C) plt.bar(horizontal=True)
    D) plt.barsideways()
24. What does the 'alpha' parameter control in Matplotlib plots?
    A) Line thickness
    B) Transparency level
    C) Color intensity
    D) Plot resolution
25. How do you create a line chart with multiple lines?
    A) plt.lines()
    B) plt.multiline()
    C) plt.plot() multiple times
    D) plt.linechart()
26. In Matplotlib, what is the 'dpi' parameter used for?
    A) Dot per inch resolution
    B) Data point interval
    C) Depth pixel intensity
    D) Dynamic plot interval
27. Which method is used to clear the current figure?
    A) plt.clear()

B) plt.clf()

C) plt.reset()

D) plt.delete()

28. How do you set the font size of labels in a plot?

A) fontsize parameter

B) labelsize parameter

C) plt.set_labelsize()

D) plt.font()

29. What is the purpose of 'plt.tight_layout()'?

A) To fit the plot within the figure area

B) To adjust the padding between and around subplots

C) To reduce the margin size

D) To increase the plot size

30. How do you change the marker style in a plot?

A) marker parameter

B) style parameter

C) mark parameter

D) plt.markerstyle()

Unit Summary

In this unit, we delved into the fundamentals of working with Python packages such as NumPy and Pandas for data manipulation and visualization. We began by exploring NumPy, focusing on its primary data structure, the Ndarray, and performing basic operations like indexing, slicing, and iteration. We then transitioned to Pandas, where we learned about the Series and DataFrame, along with Index Objects for efficient data labeling and alignment. Next, we explored data visualization using Matplotlib, understanding its architecture, and utilizing Pyplot for creating various types of charts like line charts, bar charts, and pie charts. Through this unit, we gained essential skills in handling numerical data effectively and creating insightful visualizations for data analysis and interpretation.

Glossary

- **NumPy:** A Python library used for numerical computing, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- **Ndarray:** Short for "N-dimensional array," it is the primary data structure in NumPy for representing arrays of numerical data.
- **Indexing:** The process of accessing individual elements of an array by specifying their position within the array.
- **Slicing:** A technique in NumPy for extracting a portion of an array by specifying a range of indices.
- **Array Manipulation**: Operations performed on arrays to modify their shape, size, or content, such as reshaping, concatenating, or splitting arrays.
- **Pandas:** A Python library built on top of NumPy, providing data structures like Series and DataFrame for data manipulation and analysis.
- **Series:** A one-dimensional labeled array in Pandas, capable of holding data of any type.
- **DataFrame:** A two-dimensional labeled data structure in Pandas, resembling a spreadsheet or SQL table, capable of holding heterogeneous data types.
- **Index Objects**: Objects used in Pandas to label and align data within Series and DataFrame structures, facilitating efficient data manipulation and retrieval.
- **Matplotlib:** A comprehensive library for creating static, animated, and interactive visualizations in Python, commonly used for data visualization tasks.
- **Pyplot:** The Matplotlib module providing a MATLAB-like interface for creating plots and visualizations.
- **Plotting Window:** The area where plots and visualizations are rendered within Matplotlib, providing a canvas for displaying graphical representations of data.
- **Line Charts:** A type of chart used to display data as a series of data points connected by straight line segments, commonly used to visualize trends over time.

- **Bar Charts:** A type of chart used to represent categorical data with rectangular bars, where the length of each bar corresponds to the value being represented.
- **Pie Charts:** A circular statistical graphic divided into slices to illustrate numerical proportions, often used to show the composition of a categorical whole.

Self – Assessment Questions

2. Evaluate the Use of Ndarrays in NumPy: Discuss the advantages and disadvantages of using Ndarrays compared to traditional Python lists for numerical computations.
3. Summarize the Functionality of Pandas Series and DataFrames: Provide a brief overview of the features and capabilities of Pandas Series and DataFrames for data manipulation and analysis.
4. Compare Index Objects in Pandas and NumPy: Highlight the differences between Index Objects in Pandas and NumPy, and explain their respective roles in data manipulation.
5. Elucidate the Role of Matplotlib in Data Visualization: Explain how Matplotlib facilitates data visualization in Python, including its architecture and key components.
6. Explain the Process of Creating Line Charts in Matplotlib: Describe step-by-step how to create line charts using Matplotlib, including data preparation, plot creation, and customization options.
7. Compare Bar Charts and Pie Charts for Data Representation: Compare and contrast bar charts and pie charts in terms of their suitability for visualizing different types of data and conveying information effectively.
8. Evaluate the Performance of NumPy and Pandas for Data Manipulation: Assess the performance of NumPy and Pandas libraries in terms of speed, memory usage, and ease of use for common data manipulation tasks.

9. Summarize the Benefits of Using NumPy and Pandas Together: Summarize the advantages of using NumPy and Pandas together for data analysis and manipulation tasks, highlighting how they complement each other's functionality.

Activities / Exercises / Case Studies

1. NumPy Practice Exercises:

- Create a NumPy array with random integer values and perform basic arithmetic operations on it.
- Slice and index the array to extract specific elements or subsets of data.
- Reshape the array and perform array manipulation operations like concatenation and splitting.
- Use NumPy functions to compute statistical measures such as mean, median, and standard deviation of the array.

2.Pandas Case Study:

- Analyze a dataset using Pandas to gain insights into the data.
- Load a CSV file into a Pandas DataFrame and explore its structure and contents.
- Perform data cleaning tasks such as handling missing values and removing duplicates.
- Use Pandas functions to calculate descriptive statistics and visualize data using histograms and scatter plots.

3. Matplotlib Data Visualization Exercise:

- Generate synthetic data or use a dataset to create visualizations using Matplotlib.
- Create line charts to visualize trends over time or across categories.
- Use bar charts to compare different categories or groups within the data.
- Create pie charts to represent the composition of categorical data.

- Customize the appearance of the plots by adding titles, labels, legends, and adjusting colors and styles.

Answers for check your progress

Modules	S.No.	Answers
Module 1	1	C) n-dimensional array
	2	A) np.array([1, 2, 3, 4])
	3	C) np.zeros(shape)
	4	B) arr[2]
	5	C) reshape()
	6	B) arr[:3]
	7	B) np.arange(start, stop, step)
	8	B) A 3x3 identity matrix
	9	C) np.hstack((a, b))
	10	B) np.mean(arr)
Module 2	1.	B) Numerical and scientific computing
	2.	C) ndarray
	3.	A) np.array([1, 2, 3])
	4.	A) np.add()
	5.	C) The dimensions of the array
	6.	A) arr[1, 2]
	7.	A) np.concatenate()
	8.	C) Number of dimensions
	9.	A) arr.reshape(3, 3)
	10.	A) np.max()
	11.	B) Series
	12.	D) pd.DataFrame(dict)

	13.	A) pd.read_csv()
	14.	B) df['age']
	15.	A) df.columns
	16.	A) df['total'] = df['A'] + df['B']
	17.	B) Selects data by integer-location based indexing
	18.	A) df.fillna(df.mean())
	19.	A) df.groupby('group')
	20.	A) df.sort_values('A')
	21.	B) To label and align data
	22.	A) df.to_numpy()
	23.	A) To get the first few rows of a DataFrame
	24.	B) pd.append()
	25.	A) Provides a summary of statistics for numerical columns
	26.	A) df.rename(columns=new_names)
	27.	A) Swaps rows and columns
	28.	A) pd.merge()
	29.	A) To create a spreadsheet-style pivot table
	30.	A) df.reset_index()
Module 3	**1.**	B) Data visualization
	2.	A) pyplot
	3.	C) from matplotlib import pyplot as plt
	4.	B) A data value at a specific position
	5.	B) plt.figure()
	6.	B) plt.title("Title")
	7.	A) plt.show()

	8.	B) plt.xlabel("Label")
	9.	B) plt.bar()
	10.	A) plt.pie()
	11.	D) height
	12.	B) plt.legend()
	13.	D) To specify the legend text
	14.	B) plt.xlim()
	15.	B) Sets the color of the plot lines or markers
	16.	B) plt.savefig()
	17.	B) plt.subplots()
	18.	C) The format of the percentage labels
	19.	B) Setting the size of the figure
	20.	A) plt.grid()
	21.	C) The width of the lines in the plot
	22.	B) s
	23.	A) plt.barh()
	24.	B) Transparency level
	25.	C) plt.plot() multiple times
	26.	A) Dot per inch resolution
	27.	B) plt.clf()
	28.	B) labelsize parameter
	29.	B) To adjust the padding between and around subplots

	30.	A) marker parameter

Suggested Readings

1. Eidelman, A. (2020). Python data science handbook by jake VANDERPLAS (2016). *Statistique et Société*, *8*(2), 45-47.
2. McKinney, W. (2022). *Python for data analysis*. " O'Reilly Media, Inc.".
3. Idris, I. (2015). *NumPy: Beginner's Guide*. Packt Publishing Ltd.

Open-Source E-Content Links

1. https://numpy.org/doc/
2. https://pandas.pydata.org/docs/
3. https://matplotlib.org/stable/index.html

References

1. "Python Data Science Handbook" Online Version
2. NumPy User Guide
3. Pandas User Guide
4. Matplotlib User Guide
5. Kaggle: Python Data Science Tutorial

UNIT V – DJANGO

Unit V: Django: Installing Django- Building an Application - Project Creation - Designing the Data Schema - Creating an administration site for models - Working with QuerySets and Managers - Retrieving Objects - Building List and Detail Views

Django

Section	Topic	Page No.
UNIT – V		
	Unit Objectives	
Section 5.1	**Django**	**188**
5.1.1	Installing Django and Building an Application	188
5.1.2	Project Creation	193
5.1.3	Designing with Data Schema	195
5.1.4	Creating an administration site for models	200
5.1.5	Working with Querysets and Managers	203
5.1.6	Retrieving Objects	207
5.1.7	Building List and Detail Views	212
	Let Us Sum Up	216
	Check Your Progress	216
5.2	Unit- Summary	220
5.3	Glossary	220
5.4	Self- Assessment Questions	222
5.5	Activities / Exercises / Case Studies	223
5.6	Answers for Check your Progress	224
5.7	References and Suggested Readings	225

UNIT OBJECTIVES

In this unit, students will learn the fundamentals of Django, including installation and project creation. They'll delve into designing data schemas and creating an administration site for models. Through QuerySets and Managers, they'll grasp object retrieval concepts, culminating in constructing list and detail views for effective application development.

SECTION 5. 1: DJANGO

5.1.1– INSTALLING DJANGO AND BUILDING YOUR FIRST APPLICATION

Step 1: Install Django

Set up a virtual environment (recommended):

Open a terminal or command prompt.

Navigate to your desired project directory.

Create a virtual environment:

sh

```
python -m venv myenv
```

Activate the virtual environment:

On Windows:

sh

```
myenv\Scripts\activate
```

On macOS/Linux:

sh

```
source myenv/bin/activate
```

Install Django:

Once the virtual environment is activated, install Django using pip:

sh

```
pip install django
```

Verify installation:

Check the Django version to ensure it is installed correctly:

sh

```
python -m django --version
```

Step 2: Create a Django Project

Start a new project:

Use the django-admin command to create a new project:

sh

Periyar University – CDOE| Self-Learning Materialdjango-admin startproject myproject

Navigate into the project directory:

sh

```
cd myproject
```

Understand the project structure:

Your project directory should contain the following files and directories:

```
myproject/
    manage.py
    myproject/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Run the development server:

Use the manage.py script to run the server:

Sh :python manage.py runserver

Open your web browser and go to http://127.0.0.1:8000/ to see the Django welcome page.

Step 3: Create a Django Application

Use the start app command to create a new app within your project:

Sh :python manage.py startapp myapp

This will create a new directory called myapp with the following structure:

```
markdown
myapp/
    migrations/
        __init__.py
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
```

Register the app:

Open myproject/settings.py and add myapp to the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    ...
    'myapp',]
```

Create a view:

Open myapp/views.py and define a simple view:

```
from django. Http import HTTP Response
def index(request):
return HTTP Response ("Hello, world. You're at the myapp index.")
```

Map the view to a URL:

Create a new file myapp/urls.py and add the following code:

```
from Django. URLs import path.
from . import views
urlpatterns = [
   path ('', views.index, name='index'),]
```

Include the app's URLs in the project's main URL configuration. Open myproject/urls.py and modify it to include your app's URLs:

```
from django. contrib import admin
from django.urls import include, path
urlpatterns = [
   path('myapp/', include('myapp.urls')),
   path('admin/', admin.site.urls),]
```

View the application:

Run the server again if it's not running:

Sh :python manage.py runserver Navigate to http://127.0.0.1:8000/myapp/ to see your application in action.

Step 4: Create a Template

Create a template directory:

Inside myapp, create a directory called templates.

Inside templates, create another directory named myapp.

Create a template file:

Create a file index.html inside myapp/templates/myapp and add some HTML content:

```
html
<!DOCTYPE html>
<html>
<head>
   <title>MyApp</title>
</head>
<body>
   <h1>Hello, world. You're at the myapp index.</h1>
</body>
</html>
```

Update the view to use the template:

Open myapp/views.py and modify the index view to render the template:

```
from django.shortcuts import render
def index(request):
 return render(request, 'myapp/index.html')
```

View the updated application:

Refresh the page http://127.0.0.1:8000/myapp/ to see the changes.

Congratulations! You've successfully installed Django, created a project, built an app, and rendered a template. From here, you can continue to expand your application by adding models, forms, and more complex views and templates.

5.1.2– PROJECT CREATION

Django project from scratch, including setting up the project, designing the data schema, and building a simple application.

Step 1: Set Up Your Django Environment

Set up a virtual environment:

Open a terminal or command prompt.

Navigate to your desired project directory.

Create a virtual environment:

```
sh
python -m venv myenv
```

Activate the virtual environment:

On Windows:

```
sh
myenv\Scripts\activate
```

On macOS/Linux:

```
sh
source myenv/bin/activate
```

Install Django:

With the virtual environment activated, install Django using pip:

sh

pip install django

Verify installation:

Check the Django version to ensure it is installed correctly:

sh

python -m django --version

Step 2: Create a Django Project

Start a new project:

Use the django-admin command to create a new project:

sh

django-admin startproject myproject

Navigate into the project directory:

sh
cd myproject

Run the development server:

Use the manage.py script to run the server:

sh

python manage.py runserver

Open your web browser and go to http://127.0.0.1:8000/ to see the Django welcome page.

Step 3: Create a Django Application

Create an app:

Use the startapp command to create a new app within your project:

sh

```
python manage.py startapp myapp
```

This will create a new directory called myapp with the following structure:

markdown

```
myapp/
    migrations/
        __init__.py
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
```

Register the app:

Open myproject/settings.py and add myapp to the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    ...
    'myapp',
]
```

5.1.3 DESIGNING THE DATA SCHEMA

Define your models:

In your app's models.py file, define the models. For example, let's create a simple blog with Author, Post, and Comment models:

```
# myapp/models.py
from django.db import models
```

```
class Author(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    def __str__(self):
        return self.name
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    def __str__(self):
        return self.title
class Comment(models.Model):
    post = models.ForeignKey(Post, related_name='comments',
on_delete=models.CASCADE)
    name = models.CharField(max_length=80)
    email = models.EmailField()
    body = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    def __str__(self):
        return f'Comment by {self.name} on {self.post}'
```

Create and apply migrations:

Generate migration files based on your models:

sh

```
python manage.py makemigrations
```

Apply the migrations to create the database tables:

sh

```
python manage.py migrate
```

Step 5: Admin Interface

Register models in the admin site:

To manage your models via Django’s admin interface, register them in the admin.py file:

```
# myapp/admin.py
from django.contrib import admin
from .models import Author, Post, Comment
admin.site.register(Author)
admin.site.register(Post)
admin.site.register(Comment)
```

Access the admin interface:
Create a superuser to access the admin interface:
sh

```
python manage.py createsuperuser
```

Follow the prompts to set up your superuser account.

Start the development server and log in to the admin site at http://127.0.0.1:8000/admin/ using the superuser credentials.

Step 6: Creating Views and Templates

Create a view:

Open myapp/views.py and define a simple view:

```
# myapp/views.py
from django.shortcuts import render
from .models import Post
def index(request):
    posts = Post.objects.all()
    return render(request, 'myapp/index.html', {'posts': posts})
```

Map the view to a URL:

Create a new file myapp/urls.py and add the following code:

```
# myapp/urls.py
from django.urls import path
from . import views
urlpatterns = [
    path('', views.index, name='index'),
]
```

- Include the app's URLs in the project's main URL configuration. Open `myproject/urls.py` and modify it to include your app's URLs:

```
# myproject/urls.py
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    path ('myapp/', include('myapp.urls')),
    path('admin/', admin.site.urls),]
```

Create a template:

Inside myapp, create a directory called templates, and within it create another directory named myapp.

Create a file index.html inside myapp/templates/myapp and add some HTML content:

```
html
<!-- myapp/templates/myapp/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>MyApp</title>
</head>
<body>
    <h1>Posts</h1>
    <ul>
        {% for post in posts %}
            <li>{{ post.title }} by {{ post.author.name }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

View the application:

Run the server again if it's not running:

sh

```
python manage.py runserver
```

Navigate to http://127.0.0.1:8000/myapp/ to see your application in action.

Step 7: Testing the Data Schema

Create test data:

Use the Django shell to create instances:

Sh

```
python manage.py shell
from myapp.models import Author, Post, Comment
author = Author.objects.create(name='John Doe', email='john@example.com')
post = Post.objects.create(title='First Post', content='This is the content of the first post.', author=author)
comment = Comment.objects.create(post=post, name='Jane Doe', email='jane@example.com', body='Great post!')
```

Query the database:

Use Django's ORM to fetch and display data.

```
posts = Post.objects.all ()
for post in posts:
    print (post. title, post.author.name)
```

By following these steps, you will have a fully functional Django project with a basic data schema, views, templates, and an admin interface. This foundation will allow you to expand your application further, adding more features and complexity as needed.

5.1.4– CREATING AN ADMINSTRATIVE SITE FOR MODELS

Creating an administration site for your models in Django involves using Django's built-in admin interface. This interface allows you to manage your application's data conveniently through a web-based interface. Here's a step-by-step guide to setting up the admin site for your models:

Step 1: Register Models with the Admin Site

To manage your models through the Django admin interface, you need to register them with the admin site. This is done in the admin.py file of your application.

Open admin.py:

Navigate to your app's admin.py file. For example, if your app is named myapp, open myapp/admin.py.

Register your models:

Import your models and register them with the admin site. Here's an example for a blog application with Author, Post, and Comment models:

```
# myapp/admin.py
from django.contrib import admin
from .models import Author, Post, Comment
# Register your models here.
admin.site.register(Author)
admin.site.register(Post)
admin.site.register(Comment)
```

Step 2: Customize the Admin Interface

To make the admin interface more user-friendly and informative, you can customize it by creating admin classes. These classes allow you to control how the models are displayed and managed in the admin interface.

Create admin classes:

Define admin classes to customize the model representations in the admin interface. For example, you might want to display specific fields, add search functionality, and list filters.

```
# myapp/admin.py
from django.contrib import admin
from .models import Author, Post, Comment
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('name', 'email')
    search_fields = ('name', 'email')
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'created_at')
    list_filter = ('author', 'created_at')
    search_fields = ('title', 'content')
class CommentAdmin(admin.ModelAdmin):
    list_display = ('post', 'name', 'email', 'created_at')
    list_filter = ('post', 'created_at')
    search_fields = ('name', 'email', 'body')
admin.site.register(Author, AuthorAdmin)
admin.site.register(Post, PostAdmin)
admin.site.register(Comment, CommentAdmin)
```

Step 3: Access the Admin Site

Create a superuser:

To access the admin interface, you need a superuser account. If you haven't created one yet, you can do so using the following command:

sh

```
python manage.py createsuperuser
```

Follow the prompts to create a superuser account with a username, email, and password.

Run the development server:

Start the Django development server if it’s not already running:

sh

```
python manage.py runserver
```

Log in to the admin site:

Open your web browser and navigate to http://127.0.0.1:8000/admin/.

Log in using the superuser credentials you created.

Step 4: Using the Admin Interface

Add, Edit, and Delete Records:

Once logged in, you will see your registered models in the admin interface.

You can add, edit, and delete records for your models using the provided forms.

Explore Admin Customization Options:

Django’s admin interface is highly customizable. Explore the Django admin documentation to learn about more advanced customization options like inlines, custom actions, and overriding templates.

Example: Complete admin.py File

Here's a complete example of a customized admin.py file for a blog application:

```
# myapp/admin.py
from django.contrib import admin
```

```
from .models import Author, Post, Comment
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('name', 'email')
    search_fields = ('name', 'email')
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'created_at')
    list_filter = ('author', 'created_at')
    search_fields = ('title', 'content')
class CommentAdmin(admin.ModelAdmin):
    list_display = ('post', 'name', 'email', 'created_at')
    list_filter = ('post', 'created_at')
    search_fields = ('name', 'email', 'body')
admin.site.register(Author, AuthorAdmin)
admin.site.register(Post, PostAdmin)
admin.site.register(Comment, CommentAdmin)
```

By following these steps, you will have a fully functional administration site for your Django models, allowing you to manage your application's data efficiently.

5.1.5– WORKING WITH QUERY SETS AND MANAGERS

Working with query sets and managers is a core part of interacting with your database in Django. Django’s ORM (Object-Relational Mapping) provides a powerful and intuitive way to retrieve and manipulate your database records. Here’s how to effectively work with query sets and managers to retrieve objects.

Understanding Query Sets

A Query Set represents a collection of objects from your database. It can have zero, one, or many filters to restrict the number of results returned. Query Sets are lazy, meaning that they don’t hit the database until they are actually evaluated.

Basic Query Set Methods

Retrieving all objects:

Use the all() method to retrieve all objects from a model.

```
from myapp.models import Post
all_posts = Post.objects.all()
```

Filtering objects:

Use the filter() method to retrieve objects that match certain criteria.

```
published_posts = Post.objects.filter(status='published')
```

Excluding objects:

Use the exclude() method to exclude objects that match certain criteria.

```
unpublished_posts = Post.objects.exclude(status='published')
```

Retrieving a single object:

Use the get() method to retrieve a single object that matches certain criteria. This method will raise Does Not Exist if no object is found and Multiple Objects Returned if more than one object is found.

```
post = Post.objects.get(id=1)
```

Ordering results:

Use the order_by() method to order the results.

```
ordered_posts = Post.objects.all().order_by('title')
```

Limiting results:

Use slicing to limit the number of results returned.

```
limited_posts = Post.objects.all()[:10]
```

Common QuerySet Methods

Count:

Use the count() method to get the number of objects in the Query Set.

```
post_count = Post.objects.count()
```

First and Last:

Use the first() and last() methods to get the first and last object in the Query Set.

```
first_post = Post.objects.first()
last_post = Post.objects.last()
```

Exists:

Use the exists() method to check if the Query Set contains any results.

```
has_posts = Post.objects.exists()
```

Distinct:

Use the distinct() method to remove duplicate results.

```
unique_authors = Post.objects.values('author').distinct()
```

MANAGERS IN DJANGO

Managers are the interface through which database query operations are provided to Django models. The default manager for every model is objects, but you can define your own managers to add custom query methods.

Custom Manager:

Define a custom manager by inheriting from models.Manager.

```
from django.db import models
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(status='published')
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    status = models.CharField(max_length=10, choices=(('draft', 'Draft'),
('published', 'Published')))
    published = PublishedManager()
```

Using the Custom Manager:

```
published_posts = Post.published.all()
```

Advanced QuerySet Operations

Chaining:

Query Sets can be chained to combine multiple filters and other methods.

```
recent_published_posts = Post.objects.filter(status='published').order_by('-created_at')[:5]
```

Annotations:

Use the annotate() method to add additional data to each object in the Query Set.

```
from django.db.models import Count
authors_with_post_count = Author.objects.annotate(post_count=Count('post'))
```

Select Related and Prefetch Related:

Use select_related() and prefetch_related() to optimize database access by reducing the number of queries.

```
# Eager load the related author for each post
posts_with_authors = Post.objects.select_related('author').all()
```

Example Usage

Let's consider an example with Author, Post, and Comment models.

Retrieve all posts:

```
all_posts = Post.objects.all()
```

Retrieve posts by a specific author:

```
author = Author.objects.get(name='John Doe')
johns_posts = Post.objects.filter(author=author)
```

Retrieve posts with comments:

```
posts_with_comments = Post.objects.prefetch_related('comments').all()
```

Retrieve the number of comments for each post:

```
posts_with_comment_count = Post.objects.annotate(comment_count=Count('comments'))
```

By mastering Query Sets and managers, you can efficiently interact with your Django models, performing complex queries and optimizations to ensure your application runs smoothly.

5.1.6– RETRIEVING OBJECTS

Retrieving objects from the database is a fundamental task when working with Django's ORM. Below is a detailed guide on how to retrieve objects using various Query Set methods and managers.

Basic Query Set Methods:

1. Retrieving All Objects

To retrieve all objects from a model, use the all() method:

```
from myapp.models import Post
all_posts = Post.objects.all()
```

2. Filtering Objects

To retrieve a subset of objects that match certain criteria, use the filter() method:

```
published_posts = Post.objects.filter(status='published')
```

3. Excluding Objects

To exclude objects that match certain criteria, use the exclude() method:

```
unpublished_posts = Post.objects.exclude(status='published')
```

4. Retrieving a Single Object

To retrieve a single object that matches certain criteria, use the get() method. This method will raise DoesNotExist if no object is found and MultipleObjectsReturned if more than one object is found:

```
post = Post.objects.get(id=1)
```

Ordering and Limiting QuerySets

5. Ordering Results

To order the results, use the order_by() method:

```
ordered_posts = Post.objects.all().order_by('title')
```

6. Limiting Results

To limit the number of results returned, use slicing:

```
limited_posts = Post.objects.all()[:10]
```

Aggregation and Annotations

7. Counting Objects

To count the number of objects in a QuerySet, use the count() method:

```
post_count = Post.objects.count()
```

8. Getting the First and Last Objects

To get the first and last object in a QuerySet, use the first() and last() methods:

```
first_post = Post.objects.first()
last_post = Post.objects.last()
```

9. Checking for Existence

To check if the QuerySet contains any results, use the exists() method:

```
has_posts = Post.objects.exists()
```

10. Removing Duplicates

To remove duplicate results, use the distinct() method:

```
unique_authors = Post.objects.values('author').distinct()
```

Using Managers in Django

Managers are the interface through which database query operations are provided to Django models. By default, every model in Django has at least one manager called objects. You can define your own managers to add custom query methods.

11. Custom Manager

To define a custom manager, inherit from models.Manager and override the get_queryset method:

```
from django.db import models
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(status='published')
class Post(models.Model):
    title = models.CharField(max_length=200)

    content = models.TextField()

    status = models.CharField(max_length=10, choices=(('draft', 'Draft'),
('published', 'Published')))
    published = PublishedManager()
```

12. Using the Custom Manager

To use the custom manager:

```
published_posts = Post.published.all()
```

Advanced QuerySet Operations

13. Chaining QuerySets

QuerySets can be chained to combine multiple filters and other methods:

```
recent_published_posts =
Post.objects.filter(status='published').order_by('-created_at')[:5]
```

14. Annotating QuerySets

Use the annotate() method to add additional data to each object in the QuerySet:

```
from django.db.models import Count
authors_with_post_count =
Author.objects.annotate(post_count=Count('post'))
```

15. Optimizing Queries with select_related and prefetch_related

Use select_related() and prefetch_related() to optimize database access by reducing the number of queries:

Eager load the related author for each post

```
posts_with_authors = Post.objects.select_related('author').all()
```

Example: Working with Author, Post, and Comment Models

Here's an example of how to use these techniques with Author, Post, and Comment models:

Retrieving All Posts

```
all_posts = Post.objects.all()
Retrieving Posts by a Specific Author
author = Author.objects.get(name='John Doe')
```

johns_posts = Post.objects.filter(author=author)

Retrieving Posts with Comments

posts_with_comments = Post.objects.prefetch_related('comments').all()

Retrieving the Number of Comments for Each Post

posts_with_comment_count =
Post.objects.annotate(comment_count=Count('comments'))

Using the Django Shell for Testing

To test these queries, use the Django shell:

sh

```
python manage.py shell
```

In the shell, you can run your queries:

```
from myapp.models import Author, Post, Comment
# Retrieve all posts

all_posts = Post.objects.all()

# Filter posts by status

published_posts = Post.objects.filter(status='published')

# Get a single post by ID

post = Post.objects.get(id=1)
# Count the number of posts
post_count = Post.objects.count()
# Get the first post
first_post = Post.objects.first()
# Check if any posts exist
has_posts = Post.objects.exists()
```

By mastering these Query Set methods and manager techniques, you can efficiently retrieve and manipulate data in your Django application, ensuring your application runs smoothly and effectively.

5.1.8 – BUILDING LIST AND DETAIL VIEWS

Building list and detail views in Django is an essential part of developing web applications that display collections of objects and their detailed information. Here's a step-by-step guide to creating list and detail views using Django's class-based views (CBVs) and function-based views (FBVs).

Using Class-Based Views (CBVs)

Django provides generic views that can significantly reduce the amount of code you need to write.

Step 1: Setting Up the Models

Assume you have a Post model as follows:

```
# myapp/models.py
from django.db import models
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey('Author', on_delete=models.CASCADE)
    def __str__(self):
        return self.title
```

Step 2: Creating List and Detail Views

List View: Use ListView to display a list of posts.

Detail View: Use DetailView to display the details of a single post.

```
# myapp/views.py
from django.views.generic import ListView, DetailView
from .models import Post
class PostListView(ListView):
    model = Post
    template_name = 'myapp/post_list.html'
    context_object_name = 'posts'
class PostDetailView(DetailView):
    model = Post
    template_name = 'myapp/post_detail.html'
    context_object_name = 'post'
```

Step 3: Setting Up URLs

Map the views to URLs in your app’s urls.py.

```
# myapp/urls.py
from django.urls import path
from .views import Post ListView, Post Detail View
urlpatterns = [
    path('', PostListView.as_view(), name='post_list'),
    path('post/<int:pk>/', PostDetailView.as_view(), name='post_detail'),
]
```

Step 4: Creating Templates

Create templates to render the views.

post_list.html:

```
html
<!-- myapp/templates/myapp/post_list.html -->
```

```
<!DOCTYPE html>
<html>
<head>
   <title>Post List</title>
</head>
<body>
   <h1>Post List</h1>
   <ul>
      {% for post in posts %}
         <li>
            <a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a>
         </li>
      {% endfor %}
   </ul>
</body>
</html>
```

post_detail.html:

```
html
<!-- myapp/templates/myapp/post_detail.html -->
<!DOCTYPE html>
<html>
<head>
   <title>{{ post.title }}</title>
</head>
<body>
   <h1>{{ post.title }}</h1>
   <p>{{ post.content }}</p>
   <p>Author: {{ post.author }}</p>
   <p>Created at: {{ post.created_at }}</p>
   <a href="{% url 'post_list' %}">Back to Post List</a>
```

```
</body>
</html>
```

Using Function-Based Views (FBVs)

If you prefer to use function-based views, here’s how you can achieve the same functionality.

Step 1: Creating List and Detail Views

List View: Use a function to display a list of posts.

Detail View: Use a function to display the details of a single post.

```
# myapp/views.py
from django.shortcuts import render, get_object_or_404
from .models import Post
def post_list(request):
    posts = Post.objects.all()
    return render(request, 'myapp/post_list.html', {'posts': posts})
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)
    return render(request, 'myapp/post_detail.html', {'post': post})
```

Step 2: Setting Up URLs

Map the views to URLs in your app’s urls.py.

```
# myapp/urls.py
from django.urls import path
from .views import post_list, post_detail
urlpatterns = [
    path('', post_list, name='post_list'),
    path('post/<int:pk>/', post_detail, name='post_detail'),
```

]

Step 3: Creating Templates

Use the same templates as shown above for class-based views.

Class-Based Views (CBVs): Use List View and Detail View to create concise and reusable views.

Function-Based Views (FBVs): Use traditional functions to handle requests and render templates.

Both methods have their place, and the choice depends on your project requirements and personal preference. By following these steps, you can effectively create list and detail views in Django to display collections of objects and their detailed information.

Let Us Sum Up

In this unit, we explored the process of working with Django, a high-level Python web framework. We began with the installation of Django and the creation of a new project. The process continued with building applications, which involves organizing code into models, views, templates, and URLs. We designed the data schema using Django's ORM (Object-Relational Mapping) by defining models that represent database tables. We also set up an administration site to manage our models easily. Working with Query Sets and Managers allowed us to retrieve and manipulate data efficiently. Finally, we created list and detail views to display collections and individual records of data, respectively. Through these steps, we learned how to leverage Django's powerful features to develop robust and scalable web applications.

Check Your Progress

1. What command is used to install Django?
   A) pip install Django
   B) django install
   C) pip get Django
   D) install django

2. How do you start a new Django project?
   A) django new projectname
   B) django-admin startproject projectname
   C) startproject projectname
   D) createproject projectname
3. Which file contains the settings for a Django project?
   A) urls.py
   B) views.py
   C) settings.py
   D) models.py
4. Which command is used to create a new Django app?
   A) django-admin startapp appname
   B) createapp appname
   C) startapp appname
   D) django createapp appname
5. What is the primary purpose of models in Django?
   A) To define URL patterns
   B) To handle HTTP requests
   C) To define the structure of the database
   D) To render HTML templates
6. How do you define a CharField in a Django model?
   A) models.TextField()
   B) models.CharField(max_length=100)
   C) models.CharField()
   D) models.StringField()
7. Which command is used to apply migrations in Django?
   A) python manage.py migrate
   B) python manage.py applymigrations
   C) python manage.py runmigrations
   D) python manage.py makemigrations
8. What is the purpose of the Django admin site?
   A) To handle static files
   B) To manage database tables and records
   C) To render HTML templates
   D) To manage URL patterns
9. How do you register a model with the admin site?
   A) admin.site.add_model(MyModel)
   B) admin.site.register(MyModel)
   C) admin.register(MyModel)
   D) admin.add_model(MyModel)
10. What does QuerySet represent in Django?
   A) A list of URLs
   B) A list of database objects
   C) A list of HTML templates

D) A list of views

11. Which method is used to filter QuerySets in Django?
    A) get()
    B) filter()
    C) find()
    D) select()
12. How do you retrieve a single object from a QuerySet in Django?
    A) get()
    B) filter()
    C) select()
    D) find()
13. What is the purpose of Django's URLs configuration?
    A) To define database models
    B) To define how URLs map to views
    C) To handle static files
    D) To manage the admin site
14. Which file typically contains URL patterns in a Django project?
    A) settings.py
    B) urls.py
    C) views.py
    D) models.py
15. How do you define a URL pattern in Django?
    A) urlpatterns = [url('pattern', view)]
    B) urlpatterns = [path('pattern/', view)]
    C) urls = [path('pattern', view)]
    D) urlpattern = [url('pattern/', view)]
16. What is the purpose of Django views?
    A) To define the database schema
    B) To handle HTTP requests and return responses
    C) To manage the admin site
    D) To configure settings
17. How do you render an HTML template in a Django view?
    A) render(request, 'template.html')
    B) return 'template.html'
    C) request('template.html')
    D) template('template.html')
18. What function is used to create a new superuser in Django?
    A) python manage.py createsuperuser
    B) python manage.py createadmin
    C) python manage.py newsuperuser
    D) python manage.py makesuperuser
19. Which template engine is used by default in Django?
    A) Jinja2
    B) Mako

C) Django Template Language (DTL)
D) Mustache

20. How do you pass context data to a template in a Django view?
    A) context = {'key': 'value'}; render(request, 'template.html', context)
    B) render(request, 'template.html', {'key': 'value'})
    C) context = {'key': 'value'}; template('template.html', context)
    D) render(request, 'template.html', key='value')
21. Which file in a Django project contains database configuration settings?
    A) urls.py
    B) views.py
    C) models.py
    D) settings.py
22. How do you start the Django development server?
    A) python manage.py runserver
    B) python manage.py startserver
    C) python manage.py devserver
    D) python manage.py server
23. What is the default port for the Django development server?
    A) 8000
    B) 8080
    C) 5000
    D) 3000
24. What does the makemigrations command do in Django?
    A) Applies database migrations
    B) Creates new migration files based on changes in models
    C) Deletes existing migrations
    D) Resets the database
25. How do you retrieve all objects from a model in Django?
    A) Model.objects.all()
    B) Model.objects.get()
    C) Model.all()
    D) Model.objects.retrieve()
26. Which of the following is a built-in Django field for storing date and time?
    A) DateField
    B) DateTimeField
    C) TimeField
    D) TimestampField
27. What is the purpose of the Django shell?
    A) To manage static files
    B) To run administrative commands
    C) To interact with the database from the command line
    D) To configure project settings
28. How do you define a foreign key relationship in a Django model?
    A) models.ForeignKey('ModelName')

B) models.ForeignKey(ModelName)
C) models.ForeignKey('ModelName', on_delete=models.CASCADE)
D) models.ForeignKey(ModelName, on_delete=models.CASCADE)

29. What is the purpose of the collectstatic command in Django?
A) To delete static files
B) To collect all static files into a single location
C) To serve static files during development
D) To compress static files

30. How do you start a new Django project using a specific template?
A) django-admin startproject projectname --template=template_name
B) django-admin startproject --template=template_name projectname
C) django-admin startproject projectname template_name
D) django-admin startproject template_name projectname

Unit Summary

In this unit, we delved into Django, a high-level Python web framework. We began by installing Django and creating a new project. Key configurations are handled in the settings.py file, while applications within the project are initiated using the startapp command. We explored how to design a data schema using Django's ORM, defining models to structure our database. An administration site was set up to manage these models easily. We learned to work with QuerySets and Managers for database interactions, retrieving objects with methods like filter() and get(). Finally, we built list and detail views, linking URLs to views and rendering templates to create dynamic web pages.

Glossary

- **Django:** A high-level Python web framework that encourages rapid development and clean, pragmatic design.
- **pip:** A package installer for Python that is used to install and manage software packages written in Python.
- **Project:** In Django, a project is a collection of settings for an instance of Django, including database configuration, Django-specific options, and application-specific settings.

- **Application (App):** A web application that does something, such as a blog system, a database of public records, or a simple poll app. A project can contain multiple apps.
- **Model:** A single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing.
- **Migration:** A way to propagate changes made to models into the database schema. Django provides tools to automatically create and apply migrations.
- **Admin Site:** A built-in feature of Django that provides a web interface for managing site content. It's highly customizable and supports authentication and authorization.
- **QuerySet:** A collection of database queries to retrieve objects from your database. QuerySets allow you to read the data from the database, filter it, and order it.
- **Manager**: A class that manages QuerySets and provides methods to interact with the database, like creating and retrieving objects.
- **URLconf:** A mapping between URL patterns and the views that should handle them. It defines how Django handles different URLs.
- **View:** A function or class in Django that receives a web request and returns a web response. Views access the data needed to fulfill the request via models and render a template to generate the response.
- **Template:** A text file that defines the structure or layout of a file (such as an HTML page) and uses placeholders to dynamically insert content.
- **Static Files:** Files like CSS, JavaScript, and images that are used to style and provide interactivity to web pages but do not change dynamically with user interaction.
- **Settings.py:** A configuration file for a Django project that includes all the settings for the project, such as database configurations, static files settings, and middleware configurations.
- **Migrate:** A command used in Django to apply migrations and sync database schema with the current state of the models.

- **Superuser:** An administrative user with all permissions, created using the createsuperuser command, who can log into the Django admin site and manage all data.
- **Render:** A function used in views to generate an HTTP response with a template and a context.

Self – Assessment Questions

1. Evaluate the advantages and disadvantages of using Django as a web framework for your project. What features of Django make it suitable for rapid development? Are there any potential limitations or challenges you might face when using Django?
2. Summarize the steps required to create a new Django project and application. What commands are used to start a new project and application? How do you configure the settings for the new project?
3. Compare Django's Model-View-Template (MVT) architecture with the traditional model-view-controller (MVC) architecture. What are the key differences and similarities between MVT and MVC? How does Django's approach benefit web development?
4. Elucidate the process of creating and applying migrations in Django. How do you create a migration for a new model? What are the steps to apply the migration and ensure the database schema is updated? Explain the purpose and functionality of the Django admin site. How do you enable the admin site for your Django project?
5. What are some common tasks you can perform using the Django admin interface? Analyze the role of QuerySets and Managers in Django's ORM.
6. How do QuerySets facilitate database operations?What is the function of Managers in managing QuerySets?
7. Evaluate the methods for handling URLs and views in Django. How do you define URL patterns and associate them with views?What is the importance of URLconf in a Django project?

8. Compare class-based views and function-based views in Django. What are the pros and cons of using class-based views over function-based views? Provide examples of scenarios where one might be preferred over the other.
9. Elucidate the process of rendering templates in Django. How do you use the render function to generate an HTTP response? What are the best practices for organizing and using templates in a Django project?
10. Explain the concept and importance of CSRF tokens in Django forms. How does Django implement CSRF protection? What are the potential risks if CSRF tokens are not used?

Activities / Exercises / Case Studies

Activities

1. Creating a Simple Django Project. To set up a new Django project and create a basic application.
2. Designing and Implementing Models. To design a data schema using Django models and apply migrations.

Exercises

1. Building List and Detail Views. To create list and detail views for displaying data from models.
2. Setting Up the Admin Site. To enable and customize the Django admin site for managing models.

Case Studies

1. Case Study: Developing a To-Do List Application. To apply Django concepts in building a complete web application.

Answers For Check Your Progress

Modules	S.No.	Answers
Module 1	1.	A) pip install django
	2.	B) django-admin startproject projectname
	3.	C) settings.py
	4.	A) django-admin startapp appname
	5.	C) To define the structure of the database
	6.	B) models.CharField(max_length=100)
	7.	A) python manage.py migrate
	8.	B) To manage database tables and records
	9.	B) admin.site.register(MyModel)
	10.	B) A list of database objects
	11.	B) filter()
	12.	A) get()
	13.	B) To define how URLs map to views
	14.	B) urls.py
	15.	B) urlpatterns = [path('pattern/', view)]
	16.	B) To handle HTTP requests and return responses
	17.	A) render(request, 'template.html')
	18.	A) python manage.py createsuperuser
	19.	C) Django Template Language (DTL)
	20.	B) render(request, 'template.html', {'key': 'value'})
	21.	D) settings.py
	22.	A) python manage.py runserver
	23.	A) 8000
	24.	B) Creates new migration files based on changes in models
	25.	A) Model.objects.all()
	26.	B) DateTimeField

	27.	C) To interact with the database from the command line
	28.	D) models.ForeignKey(ModelName, on_delete=models.CASCADE)
	29.	B) To collect all static files into a single location
	30.	B) django-admin startproject --template=template_name projectn

Suggested Readings

1. Vincent, W. S. (2022). *Django for Beginners: Build websites with Python and Django*. WelcomeToCode.
2. Pinkham, A. (2015). *Django unleashed*. Sams Publishing.
3. Vainikka, J. (2018). Full-stack web development using Django REST framework and React.
4. Antonio Mele, “Django 3 By Example”, Third Edition, 2020.

Open-Source E-Content Links

1. https://docs.djangoproject.com/en/5.0/

2. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django

References

1. Official Django Documentation
   - Django Documentation
2. Django Project on GitHub
   - Django GitHub
3. Full Stack Python: Django
   - A comprehensive guide to using Django within the broader context of full-stack development.
4. Django Packages
   - Django Packages

5. PyCon Django Videos
6. Django REST Framework
    - Django REST Framework