Experiment No.—1: Operators, Conditional Statement and Loop

Aim:

1. Create a program that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn 100 years old.
2. Enter the number from the user and depending on whether the number is even or odd, print out an appropriate message to the user.
3. Write a function to check the input value is Armstrong and also write the function for Palindrome
4. Write a recursive function to print the factorial for a given number.

Tools Used:

- **Python IDE or Anaconda Jupyter Lab**

Learning Objectives:

1. Learn the operation of operators.
2. Learn Working of Conditional Statement
3. Learn executions and use of loops

Theory

Operators: - Operators in Python are special symbols that carry out arithmetic or logical computation. The value that the operator operates on is called the operand. Python provides various operators categorized into several types:

1. Arithmetic Operators: Used to perform mathematical operations. (+,-,*,?,%)
2. Comparison Operators: Used to compare two values. (<,>,<=,>=,==,!=)
3. Logical Operations: Used to combine conditional statements. (&&,||,~)
4. Assignment Operators: Used to assign values to variables. (=)
5. Bitwise Operators: Used to perform bit-level operations. (<<,>>)

Conditional Statements:- Conditional statements enable the execution of code to change based on the evaluation of expressions. In Python, the main conditional statements are `**if**`, `**elif**`, and `**else**`.

1. **if statement:** Evaluates a condition and runs the associated block of code if the condition is true.

```
# Code block to execute if the condition is True
statement1
statement2
```

```
        # More statements
```

2. **elif statement:** Checks another condition if the previous if or elif condition was false.

```
if condition:
        statement 1
elif another_condition:
         statement 2
else
        statement 3
#end of elif statement
```

3. **else statement:** Executes a block of code if none of the previous conditions are true.

```
if condition:
        statement 1
else
        statement 2
#end of if else statement
```

Loops:- Loops in Python are used to iterate through a sequence (such as a list, tuple, dictionary, set, or string) or to repeatedly execute a block of code.

1. **for loop:** Iterates over each item in a sequence.

```
for variable in sequence:
        #code to be execute
        statements
#end of for
```

2. **while loop:** Repeatedly executes a block of code as long as a condition is true.

```
counter_variable
while condition:
        #code to be execute
        increment/ decrement counter_variable
```

3. **break and continue:** Control the flow of loops.

```
for counter_variable in sequence:
        if condition: break/continue
                statements
#end
```

4. **else clause in loops**: Executes after the loop completes normally, but not when terminated by a **break** statement.

```
for counter_varibale in sequence:
```

```
        statements
else:
        statements
#end
```

Practical 1

```
# Create a program that asks the user to enter their name and their birth year.
# Print out a message addressed to them that tells them the year they will turn 100 years old.

# Get user name and birth year
name = input("Enter your name: ")
birth_year = int(input("Enter the birth year: "))

# Calculate the year when the user will turn 100
year_when_100 = birth_year + 100

# Print result
print(f"{name}, you will turn 100 years old in the year {year_when_100}")
```

Output:-

```
Name: Aman Bhagat
Birth year: 2006
Aman Bhagat, you will turn 100 years old in the year 2106
```

Practical 2

```
# Program to check if a number is even or odd

# Get input from the user
number = int(input("Enter a number: "))

# Check if the number is even or odd
if number % 2 == 0:
    print(f"{number} is even.")
else:
    print(f"{number} is odd.")
```

Output:-

```
Enter a number: 7
7 is odd.
```

Practical 3

```python
# Function to check if a number is an Armstrong number
def is_armstrong(number):
    num_str = str(number)
    num_digits = len(num_str)
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
    return sum_of_powers == number

# Function to check if a number is a Palindrome
def is_palindrome(number):
    num_str = str(number)
    return num_str == num_str[::-1]

# Main program
def main():
    # Get input from the user
    number = int(input("Enter a number: "))

    # Menu to choose the type of check
    print("Choose an option:")
    print("1. Check if Armstrong")
    print("2. Check if Palindrome")
    choice = input("Enter your choice (1 or 2): ")

    # Check based on user's choice
    if choice == "1":
        if is_armstrong(number):
            print(f"{number} is an Armstrong number.")
        else:
            print(f"{number} is not an Armstrong number.")
    elif choice == "2":
        if is_palindrome(number):
            print(f"{number} is a palindrome.")
        else:
            print(f"{number} is not a palindrome.")
    else:
        print("Invalid choice. Please enter 1 or 2.")

# Run the main program
main()
```

Output:-

```
Enter a number: 153
Choose an option:
1. Check if Armstrong
2. Check if Palindrome
Enter your choice (1 or 2): 1
153 is an Armstrong number.
```

Learning Outcomes:

1. Accurately use arithmetic, relational, logical, and bitwise operators to perform various computational tasks and data manipulations.
2. Design and apply conditional statements (e.g., if, elif, else, switch) to control the flow of a program based on different conditions.
3. Efficiently use loops (e.g., for, while) to execute repetitive tasks and manage iterations in a program. Course Outcomes: To master the use of operators, conditional statements, and loops for effective problem-solving in programming.

Conclusion:

Viva Questions:

1. **What is an arithmetic operator in Python?**
2. **How do you write an if-else statement in Python?**
3. **What is the purpose of a for loop?**
4. **What does the `not` operator do in Python?**
5. **How would you exit a loop prematurely in Python?**

For Faculty use:

Correction Parameters	**Formative Assessment [40%]**	**Timely Completion of Practical [40%]**	**Attendance Learning Attitude [20%]**

Experiment No.—2: Functions and String

Aim:

1. Write a function that takes a character (i.e. a string of length 1) and returns True if it is a vowel, False otherwise.
2. Enter the number from the user and depending on whether the number is even or odd, print out an appropriate message to the user.
3. Write a function to check the input value is Armstrong and also write the function for Palindrome
4. A pangram is a sentence that contains all the letters of the English alphabet at least once, for example: The quick brown fox jumps over the lazy dog. Your task here is to write a function to check a sentence to see if it is a pangram or not.

Tools Used:

● Python IDE or Anaconda jupyter lab

Learning Objectives:

1. Learn the parameterized function.
2. Learn the argument function.
3. Learn the for in function.

Theory

In Python, functions and strings are essential elements that are key to programming. Functions enable us to organize code into reusable blocks, while strings offer a means to work with and modify text data. Together, they facilitate efficient text processing and automation across a range of applications, including web development, data analysis, and scripting.

Functions in Python

A function in Python is a block of structured, reusable code designed to perform a specific action or task. Functions are defined using the `def` keyword, followed by the function name and parentheses `()`, which can optionally include parameters.

Basic Syntax:

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

- **Function Name:** The name given to the function.

- **Parameters:** Optional inputs passed to the function.
- **Return Statement:** The output or result of the function.

Strings in Python

Strings are sequences of characters enclosed within single, double, or triple quotes. They are one of the most commonly used data types for handling textual data.

String Declaration:

```
string1 = 'hello'
string2 = "Aman"
string3 = '''Elon musk is the real
iron man'''
```

String Operations

Python provides a wide range of operations and methods to manipulate strings:

Concatenation: Joining strings together using the `+` operator

```
first_name = "Aman"
last_name = "Bhagat"
full_name = first_name + " " + last_name
print(full_name)  # Output: Aman Bhagat
```

Repetition: Repeating a string using the `*` operator.

```
word = "Hello "
repeated_word = word * 3
print(repeated_word)  # Output: Hello Hello Hello
```

Slicing: Extracting a portion of a string.

```
text = "Hello, World!"
# Extract "Hello"
sliced_text = text[0:5]
print(sliced_text)  # Output: Hello

# Extract "World" using negative indices
sliced_text = text[-6:-1]
print(sliced_text)  # Output: World
```

Practical 1:

```python
def is_vowel(char):
    vowels = "aeiouAEIOU"  # List of vowels including both lowercase and uppercase
    if len(char) == 1:  # Check if the input is a single character
        return char in vowels
    else:
        return False  # Return False if the input length is not 1

# Test the function
print(is_vowel("a"))  # Output: True
print(is_vowel("b"))  # Output: False
print(is_vowel("E"))  # Output: True
print(is_vowel("z"))  # Output: False
```

Output:-

```
is_vowel("a") returns True
is_vowel("b") returns False
is_vowel("E") returns True
is_vowel("z") returns False
```

Practical 2:

```python
def compute_length(data):
    return len(data)  # Using the built-in len() function

# Test the function with a string
print(compute_length("Hello, World!"))  # Output: 13

# Test the function with a list
print(compute_length([1, 2, 3, 4, 5]))  # Output: 5
```

Output:-

```
compute_length("Hello, World!") returns 13
compute_length([1, 2, 3, 4, 5]) returns 5
```

Practical 3:

```python
# Function to generate Fibonacci series
def fibonacci(n):
    fib_series = [0, 1]  # Starting values of the Fibonacci series
    while len(fib_series) < n:
        fib_series.append(fib_series[-1] + fib_series[-2])
    return fib_series

# Input for number of terms
num_terms = int(input("Enter the number of terms you want in the Fibonacci series: "))

# Generate and print the Fibonacci series
fib_sequence = fibonacci(num_terms)
print(f"The Fibonacci series with {num_terms} terms is: {fib_sequence}")
```

Output:-

```
Enter the number of terms you want in the Fibonacci series: 10
The Fibonacci series with 10 terms is: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Practical 4:

```python
import string

# Function to check if a sentence is a pangram
def is_pangram(sentence):
    # Convert the sentence to lowercase and remove non-alphabetic characters
    sentence = sentence.lower()

    # Create a set of all alphabetic characters
    alphabet_set = set(string.ascii_lowercase)

    # Create a set of all unique alphabetic characters in the sentence
    sentence_set = set(c for c in sentence if c.isalpha())

    # Check if sentence contains all letters of the alphabet
    return sentence_set == alphabet_set

# Input sentence
sentence = input("Enter a sentence: ")

# Check and display if the sentence is a pangram
if is_pangram(sentence):
    print("The sentence is a pangram.")
else:
    print("The sentence is not a pangram.")
```

Output:-

```
Enter a sentence: The quick brown fox jumps over the lazy dog
The sentence is a pangram.
```

Learning Outcomes:

1. Understand and define functions in Python using the def keyword.
2. Implement basic and advanced string operations such as concatenation, slicing, and formatting.
3. Apply common string methods like. upper (), lower (), strip (), split (), replace (), and. join ().

Course Outcomes:
To master defining functions and manipulating strings using Python's built-in methods for solving common text-based problems

Conclusion:

Viva Questions:

1. **What is the keyword used to define a function in Python?**
2. **Which of the following is the correct syntax for defining a function in Python that takes two parameters and returns their sum?**
3. **What will the following Python function return when called with the argument 5?**
4. **What is the scope of a variable declared inside a function?**
5. **What is the result of the following function call?**

For Faculty use:

Correction Parameters	**Formative Assessment [40%]**	**Timely Completion of Practical [40%]**	**Attendance Learning Attitude [20%]**

Experiment No.—3: List

Aim:

1. Define a procedure histogram () that takes a list of integers and prints a histogram to the screen. For example, histogram ([4, 9, 7]) should print the following: **** ********* ****
2. Take a list, say for example this one: a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89] and write a program that prints out all the elements of the list that are less than 5.
3. Write a program that takes two lists and returns True if they have at least one common member. Write a Python program to print a specified list after removing the 0th, 2nd, 4th and 5th elements.
4. Write a Python program to clone or copy a list.

Tools Used:

- Python IDE or Anaconda Jupyter Lab

Learning Objectives:

1. Learn the operations and manipulations of Python lists.
2. Understand the integration of lists with conditional statements.
3. Practice using loops for iterating over list elements effectively.

Theory

A list in Python is an ordered, mutable collection of items that can contain duplicate values. Lists are one of the most flexible and widely used data types in Python, allowing developers to store multiple elements in a single variable. The items in a list can be of any data type, such as integers, floats, strings, or even other lists.

List Characteristics

- **Ordered:** Lists maintain the order of the elements. When elements are added to the list, they are appended in a specific sequence, and that order remains the same unless explicitly changed.
- **Mutable:** This means the elements of a list can be changed after the list is created. You can add, remove, or modify the elements of a list freely.
- **Allows Duplicates:** Lists can store duplicate elements. This is unlike sets, which require all elements to be unique.

Creating a List Lists are created by placing elements inside square brackets (`[]`), separated by commas.

```
# example of list
list = [1,2,5,"Aman",9]
```

Accessing List Elements

Elements in a list are accessed using indexing. Python uses zero-based indexing, so the first element of a list is accessed with index `0`.

```
list[0] #output: 1
```

Negative indexing can be used to access elements from the end of the list

```
list[-1] #output: 9
```

List Operations

Several operations can be performed on lists:

1. **Adding Elements**
   - **`append () `**: Adds an element to the end of the list.
   - **` insert (index, element) `**: Adds an element at a specific position.
   - **`extend () `**: Adds all elements of another list to the end of the current list.

```
mylist.append = ("Elon musk")
mylist.insert = (1,"Mark zuckerberg")
mylist.extend = ([1,3])
```

2. **Removing Elements:**
   - `remove(element)`: Removes the first occurrence of the specified element.
   - **`pop(index)`**: Removes and returns the element at the specified index. If no index is specified, it removes the last element.
   - **`clear () `**: Removes all elements from the list.

```
mylist.remove = ("Larry elison")
mylist.pop = (2)
```

3. **Modifying Elements:**

   You can modify a list element by directly assigning a new value to a specific index.

```
mylist.remove[2] = "Musk"
```

List Slicing

Slicing allows you to access a sublist from the original list. It is done by specifying a range of indices:

```
mylist[1:2] #Output of elements from index 1 to 3
```

List Comprehensions

List comprehensions provide a concise way to create lists. They consist of brackets containing an expression followed by a for clause.

```
squared_list = [x**2 for x in range(5)] #output: [1,9,8,92,120]
```

Common List Methods

- **`len(list)`:** Returns the number of elements in the list.
- **`min(list) and max(list)`:** Return the smallest and largest elements in the list.
- **`list.index(element)`:** Returns the index of the first occurrence of an element.
- **`list.count(element)`:** Returns the number of occurrences of an element.

Practical 1:

```
# Define the histogram function
def histogram(values):
    for value in values:
        # Print a row of '*' characters corresponding to each value
        print('*' * value)

# Example usage
histogram([4, 9, 7])
```

Output:-

```
****
*********
*******
```

Practical 2:

```
# Given list
a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

# Loop through the list and print elements less than 5
for element in a:
    if element < 5:
        print(element)
```

Output:-
1

1
2
3

Practical 3:

```
# Function to check if two lists have at least one common member
def have_common_member(list1, list2):
    # Check if there is any common element between the two lists
    for item in list1:
        if item in list2:
            return True
    return False

# Example usage
list1 = [1, 2, 3, 4]
list2 = [3, 5, 6, 7]

# Check and print if the lists have a common member
result = have_common_member(list1, list2)
print(result)  # This will print True because both lists have 3 in common
```

Output:-
True

Practical 4:

```python
# Function to remove elements at specified indices
def remove_elements(lst):
    # Specify indices to remove (0th, 2nd, 4th, and 5th)
    indices_to_remove = [0, 2, 4, 5]

    # Create a new list with elements removed at the specified indices
    new_lst = [lst[i] for i in range(len(lst)) if i not in indices_to_remove]

    return new_lst

# Example list
lst = [10, 20, 30, 40, 50, 60, 70]

# Call the function and print the result
result = remove_elements(lst)
print("Updated list:", result)
```

Output:-

```
Updated list: [20, 40, 70]
```

Practical 5:

```python
# Original list
original_list = [1, 2, 3, 4, 5]

# Clone the list using the copy() method
cloned_list = original_list.copy()

# Print both lists
print("Original List:", original_list)
print("Cloned List:", cloned_list)
```

Output:-

```
Original List: [1, 2, 3, 4, 5]
Cloned List: [1, 2, 3, 4, 5]
```

Learning Outcomes:

1. Understand and create lists in Python using square brackets [].
2. Perform basic list operations like indexing, slicing, appending, and deleting elements.
3. Apply common list methods such as. append (), extend (), insert (), remove (), and. sort ().
4. Utilize list comprehensions for efficient and readable data processing.
5. Work with nested lists for handling multi-dimensional data effectively.

Course Outcomes:

To master creating, manipulating, and applying Python list operations and methods for efficient data processing and problem-solving.

Conclusion:

Viva Questions:

1. How do you create a list in Python, and what are its key characteristics?
2. What is list slicing, and how is it used?
3. Explain the difference between. append () and. extend () methods in lists.
4. How would you remove duplicates from a list?
5. What is a list comprehension, and why is it useful?

For Faculty use:

Correction Parameters	**Formative Assessment [40%]**	**Timely Completion of Practical [40%]**	**Attendance Learning Attitude [20%]**

Experiment No.—4: Dictionary

Aim:

1. Write a Python script to sort (ascending and descending) a dictionary by value (IT LAB)
2. Write a Python script to concatenate the following dictionaries to create a new one. Sample Dictionary: dic1= {1:10, 2:20} dic2= {3:30, 4:40} dic3= {5:50,6:60} Expected Result: {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60} (IT LAB)
3. Write a Python program to sum all the items in a dictionary. (HOMEWORK)

Tools Used:

- Python IDE or Anaconda Jupyter Lab

Learning Objectives:

1. Learn the operations and manipulation of Python dictionaries.
2. Understand how to use conditional statements with dictionaries.
3. Practice using loops to iterate through dictionary keys and values effectively.

Theory

A dictionary in Python is an unordered, mutable collection of key-value pairs. It allows efficient retrieval of values based on a unique key. Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable data type such as strings, numbers, or tuples.

Characteristics of Dictionaries

- Unordered: Prior to Python 3.7, dictionaries were unordered collections. Starting with Python 3.7+, dictionaries maintain the insertion order of keys, meaning elements will appear in the order they were added.
- Mutable: Dictionaries are mutable, meaning you can change, add, or remove key-value pairs after the dictionary has been created.
- Key-Value Pairs: Each entry in a dictionary consists of a key and a value. Keys must be unique and immutable (strings, numbers, or tuples), while values can be of any data type (including lists, other dictionaries, etc.).
- Fast Lookup: Python dictionaries allow fast access to data. The average time complexity for lookups, insertions, and deletions in a dictionary is O (1), making it an efficient data structure for many use cases.

Creating a Dictionary

Dictionaries are created by placing key-value pairs inside curly braces ` {} `, separated by commas. Each key is separated from its corresponding value by a colon `: `.

```
# Creating a simple dictionary
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
```

Accessing Dictionary

Elements You can access values in a dictionary by using the corresponding key in square brackets.

```
print(person["name"])  # Output: Alice
```

If you attempt to access a key that does not exist, a KeyError will be raised. To avoid this, you can use the get () method, which returns None (or a specified default value) if the key is not found.

```
print(person.get("name"))  # Output: Alice
print(person.get("email", "Not found"))  # Output: Not found
```

Modifying a Dictionary

Dictionaries are mutable, so you can add or change key-value pairs:

```
# Adding a new key-value pair
person["email"] = "alice@example.com"

# Modifying an existing value
person["age"] = 26
```

Removing Elements

There are several ways to remove elements from a dictionary:

- `del statement`: Removes a key-value pair by key.
- `pop () method`: Removes a key and returns its value.
- `popitem() method`: Removes and returns the last inserted key-value pair (useful for working with insertion-order dictionaries in Python 3.7+).
- `clear () method`: Removes all items from the dictionary.

```
# Remove a specific key-value pair
del person["city"]

# Remove and return a value by key
email = person.pop("email")

# Remove and return the last inserted key-value pair
last_item = person.popitem()

# Clear all items
person.clear()
```

Dictionary Methods

Some common dictionary methods include:

- `keys () `: Returns a view object of all keys in the dictionary.
- `values () `: Returns a view object of all values.
- `items () `: Returns a view object of key-value pairs.
- `update () `: Updates a dictionary with the key-value pairs from another dictionary or an iterabAle of key-value pairs.

```
# Example of using keys(), values(), and items()
keys = person.keys()
values = person.values()
items = person.items()

# Example of updating a dictionary
person.update({"age": 27, "city": "Los Angeles"})
```

Iterating Over a Dictionary

You can iterate over a dictionary's keys, values, or both (key-value pairs):

```
# Iterating over keys
for key in person:
    print(key)

# Iterating over values
for value in person.values():
    print(value)

# Iterating over key-value pairs
for key, value in person.items():
    print(f"{key}: {value}")
```

Practical 1:

```
"""
Write a Python script to sort (ascending and descending) a dictionary by value
"""


# Function to sort dictionary by value (ascending and descending)
def sort_dict_by_value(d):
    # Sort by value in ascending order
    ascending = dict(sorted(d.items(), key=lambda item: item[1]))

    # Sort by value in descending order
    descending = dict(sorted(d.items(), key=lambda item: item[1], reverse=True))

    return ascending, descending

# Example dictionary
my_dict = {'apple': 5, 'banana': 2, 'cherry': 7, 'date': 3}

# Sorting the dictionary
asc_sorted, desc_sorted = sort_dict_by_value(my_dict)

# Display results
print("Dictionary sorted by value (ascending):", asc_sorted)
print("Dictionary sorted by value (descending):", desc_sorted)
```

Output:-

```
Dictionary sorted by value (ascending): {'banana': 2, 'date': 3, 'apple': 5, 'cherry': 7}
Dictionary sorted by value (descending): {'cherry': 7, 'apple': 5, 'date': 3, 'banana': 2}
```

Practical 2:

```
"""
Write a Python script to concatenate following dictionaries to create a new one.
Sample Dictionary :
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
Expected Result : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60} (IT LAB)

"""

# Define the sample dictionaries
dic1 = {1: 10, 2: 20}
dic2 = {3: 30, 4: 40}
dic3 = {5: 50, 6: 60}

# Concatenate the dictionaries
# Using the update() method to merge dictionaries
concatenated_dict = {}
concatenated_dict.update(dic1)
concatenated_dict.update(dic2)
concatenated_dict.update(dic3)

# Print the concatenated dictionary
print("Concatenated Dictionary:", concatenated_dict)
```

Output:-

```
Concatenated Dictionary: {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
```

Practical 3:

```
"""
Write a Python program to sum all the items in a dictionary.
"""
# Function to sum all the values in a dictionary
def sum_dict_values(d):
    return sum(d.values())

# Example dictionary
my_dict = {'a': 10, 'b': 20, 'c': 30, 'd': 40}

# Calculate the sum of all values
total_sum = sum_dict_values(my_dict)

# Display the result
print("The sum of all values in the dictionary is:", total_sum)
```

Output:-

```
The sum of all values in the dictionary is: 100
```

Learning Outcomes:

1. Understand the structure and purpose of dictionaries in Python.
2. Perform basic dictionary operations such as adding, updating, and deleting key-value pairs.
3. Utilize conditional statements to check for the presence of keys and manage dictionary data.
4. Apply loops to efficiently iterate over dictionary keys, values, and items.
5. Implement dictionary methods like. get (), keys (), values (), and. items () for data retrieval and manipulation.

Course Outcomes:

To develop the ability to create, manipulate, and apply Python dictionaries in conjunction with conditional statements and loops for efficient data management.

Conclusion:

Viva Questions:

1. What is a dictionary in Python, and how does it differ from a list?
2. How can you add a key-value pair to a dictionary?
3. How do you check if a specific key exists in a dictionary using a conditional statement?
4. Explain how to iterate through a dictionary using a for loop to access keys and values.
5. What is the difference between. get () and directly accessing a value using a key (e.g., dict[key])?

For Faculty use:

Correction Parameters	Formative Assessment [40%]	Timely Completion of Practical [40%]	Attendance Learning Attitude [20%]

Experiment No.—5: File Operation

Aim:

1. Write a Python program to read an entire text file. (IT LAB)
2. Write a Python program to append text to a file and display the text. (IT LAB)
3. Write a Python program to read the last *n* lines of a file. (HOMEWORK)

Tools Used:

- Python IDE or Anaconda Jupyter Lab

Learning Objectives:

1. Understand file handling basics, including opening, reading, writing, and closing files.
2. Differentiate file modes (read, write, append, binary) and their effects on file content.
3. Apply error handling to manage exceptions in file operations like missing files or permission errors.

Theory

File handling is a crucial aspect of Python programming, allowing developers to work with files stored on a disk. Whether it's reading from a file, writing data to a file, or manipulating file content, Python provides a simple and efficient way to manage these tasks through its built-in functions.

1. File Modes in Python

Before performing any operation, it's essential to open a file using Python's `open()` function. The `open()` function takes two parameters: the filename and the mode in which the file should be opened. The most common file modes include:

- **'r'**: Read mode (default mode) - Opens the file for reading. It raises an error if the file does not exist.
- **'w'**: Write mode - Opens the file for writing. If the file already exists, it truncates the file (deletes its contents). If it does not exist, a new file is created.
- **'a'**: Append mode - Opens the file for writing but does not truncate it. New data is added at the end of the file.
- **'r+'**: Read and write mode - Allows both reading and writing. The file pointer is placed at the beginning.

- **'w+'**: Write and read mode - Opens the file for reading and writing. It truncates the file if it exists; otherwise, it creates a new one.
- **'a+'**: Append and read mode - Opens the file for reading and writing. The file pointer is at the end if the file exists, allowing for data to be appended.

2. Basic File Operations

Python's file operations include reading from and writing to files. Here are some common operations:

- Opening a File:

```
file = open("example.txt", "r")
```

Reading from a File: Python provides several methods to read files:
`. read (): Reads the entire file content as a single string.

```
content = file.read()
print(content)
```

`.readline()`: Reads one line at a time.

```
line = file.readline()
print(line)
```

`.readlines()`**: Reads all lines as a list of strings

```
lines = file.readlines()
print(lines)
```

- Writing to a File :To write data to a file, you can use the `.write()` or `.writelines()` methods:

```
file = open("example.txt", "w")
file.write("Hello, Python!")
file.close()
```

`. write () `: Writes a single string to the file.
`. writelines()`: Writes a list of strings to the file.

```
lines = ["First line\n", "Second line\n"]
file.writelines(lines)
```

- Appending to a File**: Using append mode (`'a'`), data can be added to the end of an existing file without overwriting its content:

```
file = open("example.txt", "a")
file.write("This is an appended line.\n")
file.close()
```

3. Closing a File

After performing any file operation, it is good practice to close the file using the `. close () ` method to free up system resources:

```
file.close()
```

Alternatively, Python provides a safer and more efficient way to handle files using the `with` statement**. This approach automatically closes the file after the block of code is executed:

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

4. Working with File Paths

Python can work with both relative and absolute file paths. Using the **\`os\`** module, developers can manipulate file paths and directories:

```
import os
path = os.path.join("folder", "example.txt")
file = open(path, "r")
```

5. Error Handling in File Operations

It is common to encounter errors when working with files, such as a file not being found. Python allows handling such errors gracefully using `try-except` blocks:

```
try:
    file = open("nonexistent.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found. Please check the filename.")
finally:
    file.close()
```

6. Reading and Writing Binary Files

Python can also handle binary files, such as images or executable files. To work with binary data, open the file in binary mode by using `'rb'` for reading and `'wb'` for writing:

Practical 1:

```python
# Write a Python program to read an entire text file. (IT LAB)

def read_file ():
  filename = input("Enter Filename to read it: ")

  try:
    file = open(filename, "r+")
    print(f"File Readed successfully:=> \n{file.read()}")
  except FileNotFoundError :
    print(f"{filename} is not found")

read file()
```

Output:-

```
Enter Filename to read it: ./Text.txt
File Readed successfully:=>
Aman Bhagat
```

Practical 2:

```python
# Write a Python program to append text to a file and display the text. (IT LAB)

def read_file():
    file = open("./Text.txt", "r")
    print(file.read())

def append_file(filename, text):
    try:
        file = open(filename, "a")
        file.write(text + "\n")
    except Exception:
        print("Error Appending File")

append_file("./Text.txt", "Aman")
read_file()
```

Output:-

Aman Bhagat

Practical 3:

```
# Write a Python program to read the last n lines of a file. (HOMEWORK)

def read_last_line(filename = "./Text.txt"):
  try:
    file = open(filename, "r")
    line = file.readlines()
    last_line = line[-1].splitlines()
    return last_line
  except FileNotFoundError:
    print("File not found")
    return None

print(f"Read Last line of file :=> \n{read_last_line()}")
```

Output:-

Read last line of the file => [‘Elon musk is the real life iron man’]

Learning Outcomes:

1. Perform basic file operations like reading, writing, and appending data accurately.
2. Implement error handling to manage common issues in file processing.
3. Utilize context managers and file modes effectively for safe and efficient file handling.

Course Outcomes:

To proficiently perform file operations with effective error handling and utilize context managers for secure and efficient data management in programming.

Conclusion:

Viva Questions:

1. What are the differences between the file modes 'r', 'w', and 'a'?
2. How does a context manager (e.g., `with` statement in Python) help in file handling?
3. What is the purpose of error handling when working with files, and how would you implement it?
4. How can you read a file line by line without loading the entire file into memory?
5. What are absolute and relative file paths, and when would you use each?

For Faculty use:

Correction Parameters	Formative Assessment [40%]	Timely Completion of Practical [40%]	Attendance Learning Attitude [20%]

Experiment No.—6: Object-Oriented Programming

Aim:

1. Design a class that stores the information of students and displays the same (IT LAB).
2. Implement the concept of inheritance using Python (IT LAB).
3. Create a class called `Numbers`, which has a single class attribute `MULTIPLIER`, and a constructor that takes the parameters `x` and `y` (these should all be numbers).
    - a. Write a method called `add` which returns the sum of the attributes `x` and `y`.
    - b. Write a class method called `multiply`, which takes a single number parameter and returns the product of it and `MULTIPLIER`.
    - c. Write a static method called `subtract`, which takes two number parameters, `b` and `c`, and returns `b - c`.
    - d. Write a method called `value` which returns a tuple containing the values of `x` and `y`. Make this method into a property, and write a setter and a deleter for manipulating the values of `x` and `y`. **(HOMEWORK)**
4. Open a new file in IDLE (“New Window” in the “File” menu) and save it as `geometry.py` in the directory where you keep the files you create for this course. Then copy the functions you wrote for calculating volumes and areas in the “Control Flow and Functions” exercise into this file and save it.
   Now open a new file and save it in the same directory. You should now be able to import your own module like this: `import geometry`.
   Try and add `print(dir(geometry))` to the file and run it. Now write a function `pointyShapeVolume(x, y, squareBase)` that calculates the volume of a square pyramid if `squareBase` is True and of a right circular cone if `squareBase` is False. `x` is the length of an edge on a square if `squareBase` is True and the radius of a circle when `squareBase` is False. `y` is the height of the object. First, use `squareBase` to distinguish the cases. Use the `circleArea` and `squareArea` from the geometry module to calculate the base areas. **(HOMEWORK)**

Tools Used:

- Python IDE or Anaconda Jupyter Lab

Learning Objectives:

1. Learn the fundamental principles of OOP, including encapsulation, inheritance, polymorphism, and abstraction.

2. Learn how to design and implement classes and objects to model real-world entities in code.
3. Learn to apply OOP concepts to create reusable, maintainable, and efficient code structures.

Theory:

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data, or objects, rather than functions and logic. Objects are instances of classes, which define the properties (attributes) and behaviors (methods) of the object. OOP helps to create modular, reusable, and maintainable code. Python, being an object-oriented language, supports OOP principles.

The four main principles of OOP in Python are:

- Classes and Objects
- Encapsulation
- Inheritance
- Polymorphism

2. Classes and Objects
A class is a blueprint for creating objects (a specific data structure), defining the properties (attributes) and methods (functions) that the object will have.

Creating a Class

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display_info(self):
        print(f"Car: {self.brand} {self.model}")
```

An object is an instance of a class. Objects have attributes and methods defined by the class.

Creating an Object

```
my_car = Car("Tesla", "Model 3")
my_car.display_info()  # Output: Car: Tesla Model 3
```

3. Encapsulation

Encapsulation is the concept of restricting direct access to certain attributes or methods in a class, ensuring the data is protected. In Python, this can be achieved by using private variables (with double underscores).

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount("Alice", 1500)
account.deposit(500)
print(account.get_balance())  # Output: 2000
```

Here, the __balance attribute is private and cannot be accessed directly outside the class.

4. Inheritance

Inheritance allows one class (child class) to inherit the attributes and methods from another class (parent class). This promotes code reusability.

Example of Inheritance

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

    def speak(self):
        print("This animal makes a sound.")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} says woof!")

dog = Dog("Rex")
dog.speak()  # Output: Rex says woof!
```

Here, the Dog class inherits from the Animal class and overrides the speak () method.

5. Polymorphism

Polymorphism allows different classes to be treated as instances of the same class through inheritance. It enables the use of the same method name to behave differently based on the object it is acting on.

Example of Polymorphism

```python
class Bird:
    def fly(self):
        print("Birds can fly.")

class Ostrich(Bird):
    def fly(self):
        print("Ostriches cannot fly.")

bird = Bird()
ostrich = Ostrich()

for animal in (bird, ostrich):
    animal.fly()

# Output:
# Birds can fly.
# Ostriches cannot fly.
```

The fly method is polymorphic, behaving differently for Bird and Ostrich.

6. Abstraction

Abstraction hides the complex implementation details from the user and exposes only the essential features. This is done by creating abstract classes and methods that define the interface but not the implementation.

Example of Abstraction

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

rectangle = Rectangle(5, 10)
print(rectangle.area())  # Output: 50
```

In this example, the Shape class is abstract, meaning it cannot be instantiated, and the area () method is abstract, meaning it must be implemented by any subclass.

Practical 1:

```python
# Design a class that store the information of student and display the same (IT LAB)

# Define the Student class
class Student:
    # Constructor to initialize student details
    def __init__(self, name, roll_number, course, marks):
        self.name = name
        self.roll_number = roll_number
        self.course = course
        self.marks = marks

    # Method to display student details
    def display_info(self):
        print("Student Information")
        print("-------------------")
        print(f"Name         : {self.name}")
        print(f"Roll Number  : {self.roll_number}")
        print(f"Course       : {self.course}")
        print(f"Marks        : {self.marks}")

# Main function
if __name__ == "__main__":
    # Taking input from the user
    name = input("Enter the student's name: ")
    roll_number = input("Enter the roll number: ")
    course = input("Enter the course: ")
    marks = float(input("Enter the marks: "))

    # Creating an object of the Student class
    student = Student(name, roll_number, course, marks)

    # Displaying the student information
    student.display_info()
```

Output

```
Enter the student's name: Aman Bhagat
Enter the roll number: 20
Enter the course: Bsc IT
Enter the marks: 90

Student Information
-------------------
Name         : Aman Bhagat
Roll Number  : 20
Course       : Bsc IT
```

Practical 2:

```python
# Implement the concept of inheritance using python (IT LAB)
# Base class (Parent)
class Person:
    # Constructor to initialize person details
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Method to display person's information
    def display_info(self):
        print("Person Information")
        print("------------------")
        print(f"Name: {self.name}")
        print(f"Age : {self.age}")

# Derived class (Child)
class Student(Person):
    # Constructor to initialize student details, using inheritance
    def __init__(self, name, age, roll_number, course):
        # Call the constructor of the base class (Person)
        super().__init__(name, age)
        self.roll_number = roll_number
        self.course = course

    # Method to display student's information, extending the base class method
    def display_student_info(self):
        # Display person information from the base class
        self.display_info()
        print(f"Roll Number: {self.roll_number}")
        print(f"Course     : {self.course}")

# Main function
if __name__ == "__main__":
    # Creating an object of the Student class
    student_name = input("Enter the student's name: ")
    student_age = int(input("Enter the student's age: "))
    student_roll_number = input("Enter the roll number: ")
    student_course = input("Enter the course: ")

    # Create a Student object
    student = Student(student_name, student_age, student_roll_number, student_course)

    # Display the student's information
    student.display_student_info()
```

Output:-

```
Enter the student's name: Aman Bhagat
Enter the roll number: 20
Enter the course: Bsc IT
Enter the marks: 90
Enter the age: 18

Student Information
-------------------
Name            : Aman Bhagat
Roll Number     : 20
Course          : Bsc IT
Marks           : 90
Age             : 18
```

Practical 3:

```python
class Numbers:
    MULTIPLIER = 2  # Example class attribute, can be set to any constant value

    def __init__(self, x, y):
        self._x = x  # Instance attribute x
        self._y = y  # Instance attribute y

    def add(self):
        """Returns the sum of x and y."""
        return self._x + self._y

    @classmethod
    def multiply(cls, a):
        """Returns the product of a and MULTIPLIER."""
        return a * cls.MULTIPLIER

    @staticmethod
    def subtract(b, c):
        """Returns the result of b - c."""
        return b - c

    @property
    def value(self):
        """Returns a tuple containing the values of x and y."""
        return (self._x, self._y)

    @value.setter
    def value(self, values):
        """Sets the values of x and y."""
        self._x, self._y = values

    @value.deleter
    def value(self):
        """Deletes the values of x and y."""
        del self._x
        del self._y

# Example usage
numbers = Numbers(5, 10)
print("Addition:", numbers.add())  # Output: 15
print("Multiplication:", Numbers.multiply(3))  # Output: 6
print("Subtraction:", Numbers.subtract(10, 5))  # Output: 5
print("Values:", numbers.value)  # Output: (5, 10)

numbers.value = (7, 14)
print("Updated Values:", numbers.value)  # Output: (7, 14)

del numbers.value
```

Output:-

```
Addition: 15
Multiplication: 6
Subtraction: 5
Initial Values: (5, 10)
Updated Values: (7, 14)
Deleted Values: "Values have been deleted."
```

Practical 4:

```
# Import the custom geometry module
import geometry

# Display all the attributes and methods of the geometry module
print(dir(geometry))

# Function to calculate the volume of a square pyramid or a right circular
cone
def pointyShapeVolume(x, y, squareBase):
    if squareBase:
        # If squareBase is True, calculate the volume of a square pyramid
        base_area = geometry.squareArea(x)
        volume = (1/3) * base_area * y
        shape = "Square Pyramid"
    else:
        # If squareBase is False, calculate the volume of a right circular
        cone
        base_area = geometry.circleArea(x)
        volume = (1/3) * base_area * y
        shape = "Right Circular Cone"

    print(f"The volume of the {shape} is: {volume:.2f}")
    return volume

# Testing the function
if __name__ == "__main__":
    # Example 1: Square Pyramid
    pointyShapeVolume(5, 10, True)

    # Example 2: Right Circular Cone
    pointyShapeVolume(7, 12, False)
```

Output:-

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'circleA
ea', 'cubeVolume', 'math', 'sphereVolume', 'squareArea']
The volume of the Square Pyramid is: 83.33
The volume of the Right Circular Cone is: 615.75
```

Learning Outcomes:

1. **Understand OOP Principles:** Grasp the core concepts of encapsulation, abstraction, inheritance, and polymorphism.
2. **Design Object-Oriented Solutions:** Create modular, maintainable software using OOP concepts and design patterns.
3. **Optimize OOP Code:** Analyze and refactor code for improved efficiency, readability, and best practices adherence.

Course Outcomes:

To understand and apply Object-Oriented Programming principles to design and build efficient software applications.

Conclusion:

Viva Questions:

1. What are the four main principles of Object-Oriented Programming?
2. How does inheritance work in Object-Oriented Programming?
3. What is the difference between encapsulation and abstraction?
4. Can you explain polymorphism with an example?
5. What is the purpose of a constructor in a class?

For Faculty Use:

Correction Parameters	Formative Assessment [40%]	Timely Completion of Practical [40%]	Attendance Learning Attitude [20%]

Experiment No. 7: Exception Handling

Aim:

1. Write a program to implement exception handling. (**IT LAB**)
2. Write a program for raising and handling unrelated exceptions.

Tools Used:

- Python IDE or Anaconda Jupyter Lab

Learning Objectives:

1. Learn the purpose and function of exception handling.
2. Learn how to implement try-catch blocks to manage errors.
3. Learn to create custom exceptions for specific error scenarios.

Theory:

Exception handling is a mechanism in Python that allows developers to manage and respond to runtime errors effectively. Instead of letting a program crash when an error occurs, exception handling enables the program to catch the error and handle it gracefully, providing meaningful feedback and maintaining the program's stability.

Key Concepts:

1. **Exception**
   An exception is an error that occurs during the execution of a program. When an exception is raised, the normal flow of the program is interrupted.
   Common Examples of Exceptions:
   - `ZeroDivisionError`: Raised when dividing by zero.
   - `ValueError`: Raised when an invalid value is given (e.g., non-integer input).
   - `TypeError`: Raised when an operation is performed on incompatible types.
   - `FileNotFoundError`: Raised when a file cannot be found.
2. **Try and Except Blocks**
   The `try` block contains code that may raise an exception. The `except` block catches and handles the exception.

```
try:
    number = int(input("Enter a number:
"))
    result = 10 / number
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Division by zero is
not allowed.")
```

Output:

```
Enter a number: 0
Error: Division by zero is not allowed.
```

3. Else Block

The `else` block executes if no exceptions are raised in the `try` block.

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print(f"Result: {result}")
```

4. Finally Block

The `finally` block executes regardless of whether an exception occurred or not. It is useful for cleanup actions like closing files.

```
try:
    file = open("example.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    print("Closing the file.")
    file.close()
```

5. Raising Exceptions

You can raise exceptions manually using the `raise` keyword. This is useful for handling specific conditions not covered by built-in exceptions.

```
def check_positive(number):
    if number < 0:
        raise ValueError("The number must be
positive.")

try:
    check_positive(-5)
except ValueError as e:
    print(e)
```

Output:

```
The number must be positive.
```

6. Custom Exceptions

Custom exceptions can be defined by creating a new class that inherits from the built-in `Exception` class.

```
class NegativeNumberError(Exception):
    pass

try:
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise NegativeNumberError("Negative numbers
are not allowed.")
except NegativeNumberError as e:
    print(e)
```

Practical 1:

```
# Write a program to implement exception handling. (IT LAB)
# Program to handle division by zero error

try:
    # Taking user input
    num1 = int(input("Enter the numerator: "))
    num2 = int(input("Enter the denominator: "))

    # Attempting division
    result = num1 / num2
    print(f"Result: {result}")

except ZeroDivisionError:
    # Handling division by zero error
    print("Error: Cannot divide by zero.")

except ValueError:
    # Handling invalid input error
    print("Error: Please enter valid integers.")

print("Program finished successfully.")
```

Output:-

```
Enter the numerator: 2
Enter the denominator: 3
Result: 0.6666666666666666
Program finished successfully.
```

Practical 2:

```
# Write a program for raising and handling unrelated exceptions.

# Custom Exception Class
class NegativeNumberError(Exception):
    pass

try:
    # Taking user input
    num = int(input("Enter a positive integer: "))

    # Raise custom exception if the input is negative
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")

    # Raise a ZeroDivisionError if the number is zero
    result = 10 / num
    print(f"Result: {result}")

except NegativeNumberError as e:
    # Handling custom exception
    print(f"Error: {e}")

except ZeroDivisionError:
    # Handling division by zero
    print("Error: Cannot divide by zero.")

except ValueError:
    # Handling invalid input
    print("Error: Please enter a valid integer.")

print("Program finished.")
```

Output:-

```
Enter a positive integer: 2
Result: 5.0
Program finished.
```

Learning Outcomes:

1. **Understand what errors are and why they happen in a program.**
2. **Learn how to use try-catch to fix errors without crashing the program.**
3. **Be able to create your own error messages to handle specific problems.**

Course Outcomes:

To master the use of try-catch blocks and custom exceptions for effective error handling and robust program execution.

Conclusion:

Viva Questions:

1. What is exception handling, and why is it important in programming?
2. Can you explain the difference between checked and unchecked exceptions?
3. How does a try-catch block work, and what is the purpose of the 'finally' block?
4. What is a custom exception, and when would you use it in a program?
5. What happens if an exception is not caught in a program? How does it affect program execution?

For Faculty Use:

Correction Parameters	**Formative Assessment [40%]**	**Timely Completion of Practical [40%]**	**Attendance Learning Attitude [20%]**

Experiment No.—8: Widget

Aim:

1. Configure the widget with various options like: `bg="red"`, `family="times"`, `size=18` (IT LAB).
2. Write a program to change the widget type and configuration options to experiment with other widget types like Message, Button, Entry, Checkbutton, Radiobutton, Scale, etc. (HOMEWORK).

Tools Used:

- Python IDE or Anaconda Jupyter Lab

Learning Objectives:

1. Learn the function and types of widgets used in user interfaces.
2. Learn how to create and implement widgets for interactive applications.
3. Learn how to customize and manage widget properties for better user experience.

Theory

In Python, Tkinter is the standard GUI (Graphical User Interface) library. It provides a wide range of widgets, which are elements of the GUI that allow users to interact with the application. Widgets in Tkinter are the building blocks for creating interactive applications. They can be buttons, labels, entry fields, checkboxes, radio buttons, and more.

Types of Widgets in Tkinter

Tkinter provides several types of widgets that serve different purposes in GUI applications. Below are some commonly used widgets:

---

1. **Label Widget**
   The `Label` widget is used to display text or images. It is a non-interactive widget and is commonly used to show information to the user.
   Example:

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

2. Button Widget

The `Button` widget is used to create buttons that users can click. Buttons can be configured with various properties such as text, color, and size, and can trigger functions when clicked.

Example:

```
button = tk.Button(root, text="Click Me", command=my_function)
button.pack()
```

3.Entry Widget

The `Entry` widget is used to accept single-line text input from the user. It is commonly used for forms or where the user needs to type data.

Example:

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

4. Checkbutton Widget

The `Checkbutton` widget allows users to select or deselect an option using a checkbox. It is commonly used for boolean choices like "Yes" or "No".

Example:

```
check_var = tk.IntVar()
checkbutton = tk.Checkbutton(root, text="Agree",
variable=check_var)
checkbutton.pack()
```

5. Radio Button Widget

The `Radio Button` widget allows users to choose one option from a set of choices. It is commonly used when only one option should be selected from a group.

Example:

```
var = tk.StringVar()
radio1 = tk.Radiobutton(root, text="Option 1",
variable=var, value="1")
radio2 = tk.Radiobutton(root, text="Option 2",
variable=var, value="2")
radio1.pack()
radio2.pack()
```

6. Scale Widget

The `Scale` widget allows users to select a value from a range by sliding a bar. This widget is useful for numeric input, such as selecting a volume level or a percentage.

Example:

```
scale = tk.Scale(root, from_=0, to=100,
orient="horizontal")
scale.pack()
```

7. Listbox Widget

The `Listbox` widget is used to display a list of options from which the user can select one or more.

Example:

```
listbox = tk.Listbox(root)
listbox.insert(1, "Option 1")
listbox.insert(2, "Option 2")
listbox.pack()
```

8. Text Widget

The `Text` widget is used for multi-line text input and display. It can handle large amounts of text and allows users to type and edit the text.

Example:

```
text = tk.Text(root, height=5, width=30)
text.pack()
```

9. Message Widget

The `Message` widget is similar to the `Label` widget but is used to display longer pieces of text. It is useful for showing multiple lines of text or paragraphs.

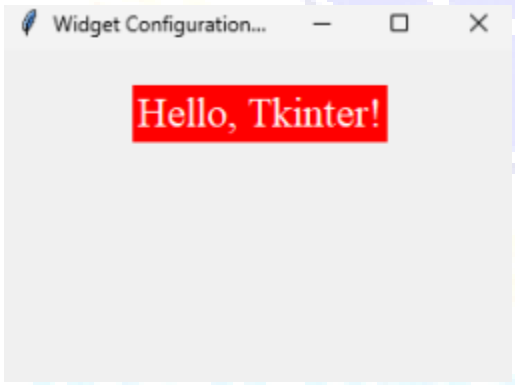
Example:

```
message = tk.Message(root, text="This is a message
widget", width=200)
message.pack()
```

Practical 1:

```
# Configure the widget with various options like: bg="red", family="times", size=18 (IT LAB)
import tkinter as tk

# Create the main application window
root = tk.Tk()
root.title("Widget Configuration Example")
root.geometry("300x200")

# Configure a Label widget with specified options
label = tk.Label(
    root,
    text="Hello, Tkinter!",
    bg="red",          # Background color
    fg="white",        # Text color
    font=("Times", 18) # Font family and size
)

# Place the label on the window
label.pack(pady=20)

# Run the Tkinter event loop
root.mainloop()
```

Output:-



Practical 2:

```
"""
Write a program to change the widget type and configuration options to experiment with other widget
types like Message, Button, Entry, Checkbutton, Radiobutton, Scale etc. (HOMEWORK)
"""
import tkinter as tk

# Create the main application window
root = tk.Tk()
root.title("Small Widget Experiment")
root.geometry("300x300")

# Button widget
button = tk.Button(root, text="Click Me", bg="blue", fg="white", font=("Arial", 12))
button.pack(pady=10)

# Entry widget
entry = tk.Entry(root, font=("Arial", 14))
entry.pack(pady=10)

# Label widget
label = tk.Label(root, text="This is a label", font=("Arial", 12), bg="yellow")
label.pack(pady=10)

# Checkbutton widget
check_var = tk.IntVar()
checkbutton = tk.Checkbutton(root, text="Agree", variable=check_var)
checkbutton.pack(pady=10)

# Run the Tkinter event loop
root.mainloop()
```

Output:-

Learning Outcomes:

1. Understand the role and types of widgets in building user interfaces.
2. Be able to create and implement interactive widgets in applications.
3. Customize widget properties to improve user experience and application functionality.

Course Outcomes: To master the use of widgets for creating interactive user interfaces and enhancing user experience in applications.

Conclusion:

Viva Questions:

1. What is a widget, and how is it used in user interface design?
2. Can you explain the difference between a button widget and a text field widget?
3. How do you customize the properties of a widget in an application?
4. What are some common types of widgets used in graphical user interfaces (GUIs)?
5. How do widgets improve the interactivity and usability of an application?

For Faculty Use:

Correction Parameters	Formative Assessment [40%]	Timely Completion of Practical [40%]	Attendance Learning Attitude [20%]

Experiment No.—9: Database Connection

Aim:

1. Design a simple database application that stores the records and retrieves the same. (IT LAB)
2. Design a database application to search the specified record from the database. (IT LAB)
3. Design a database application that allows the user to add, delete and modify the records. (HOMEWORK)

Tools Used:

- Python IDE or Anaconda Jupyter Lab

Learning Objectives:

1. Learn the process of establishing a connection between an application and a database.
2. Learn how to execute queries to retrieve, insert, update, and delete data in a database.
3. Learn how to handle errors and manage database connections for efficient data operations.

Theory

Database connection in Python allows an application to interact with databases and perform operations such as querying, inserting, updating, and deleting data. Python provides several libraries to establish a connection with various types of databases, such as SQLite, MySQL, PostgreSQL, etc. These libraries provide Python classes that represent databases, execute SQL commands, and return the results.

Types of Databases in Python

Python can connect to multiple types of databases, and the two most commonly used libraries for connecting to databases are:

1. **SQLite**: A lightweight database engine that is embedded within Python and requires no separate server.

2. **MySQL**: A popular relational database management system that requires an external server.

Connecting to SQLite Database

SQLite is a built-in database that comes with Python. To connect to an SQLite database, we can use the `sqlite3` module. The `sqlite3` module allows us to interact with SQLite databases directly without requiring a server.

Steps to connect to an SQLite Database:

1. **Import the sqlite3 module.**
2. **Create a connection** using `sqlite3.connect()`.
3. **Create a cursor object** using `connection.cursor()`, which is used to execute SQL queries.
4. **Execute SQL queries** using the `cursor.execute()` method.
5. **Commit changes** to the database if you perform any insert, update, or delete operations.
6. **Close the connection** to free up resources after database operations are complete.

Example: Connecting to SQLite Database

```python
import sqlite3

# Establish a connection to the SQLite database (
create one if it doesn't exist)
connection = sqlite3.connect('example.db')

# Create a cursor object
cursor = connection.cursor()

# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS
students (id INTEGER PRIMARY KEY, name TEXT, age
INTEGER)''')

# Insert data
cursor.execute("INSERT INTO students (name, age)
VALUES ('Alice', 22)")

# Commit changes
connection.commit()

# Query data
cursor.execute("SELECT * FROM students")
print(cursor.fetchall())

# Close the connection
connection.close()
```

Explanation:

- **sqlite3.connect('example.db'):** Establishes a connection to a database file example.db. If the file doesn't exist, it is created automatically.
- **cursor.execute():** Executes SQL commands, such as creating tables or inserting data.
- **connection.commit():** Commits any changes to the database (e.g., insertions).
- **cursor.fetchall()**: Retrieves all rows from the result of a SELECT query.

Connecting to MySQL Database

To connect to a MySQL database, you need the mysql-connector-python library (or PyMySQL). This library allows you to establish a connection to a MySQL server and execute SQL commands.

Steps to connect to a MySQL Database:

1. Install the mysql-connector-python library:

```
pip install mysql-connector-python
```

1. Import the connector module.
2. Create a connection using mysql.connector.connect().
3. Create a cursor object using connection.cursor().
4. Execute SQL queries using the cursor.

5. Commit changes to the database.
6. Close the connection

```
import mysql.connector

# Establish a connection to the MySQL database
connection = mysql.connector.connect(
    host="localhost",
    user="root",
    password="password",
    database="test_db"
)

# Create a cursor object
cursor = connection.cursor()

# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS
students (id INT AUTO_INCREMENT PRIMARY KEY, name
VARCHAR(100), age INT)''')

# Insert data
cursor.execute("INSERT INTO students (name, age)
VALUES ('Bob', 24)")

# Commit changes
connection.commit()

# Query data
cursor.execute("SELECT * FROM students")
print(cursor.fetchall())

# Close the connection
connection.close()
```

Explanation:

1. mysql.connector.connect(): Establishes a connection to the MySQL server using parameters like host, user, password, and database.
2. cursor.execute(): Executes SQL commands such as creating tables, inserting data, etc.
3. connection.commit(): Commits the transaction to the database.
4. cursor.fetchall(): Retrieves the results of a SELECT query.

Practical 1:

```
# Design a simple database application that stores the records and retrieves the same. (IT LAB)

import sqlite3

# Connect to SQLite database (creates file if it doesn't exist)
conn = sqlite3.connect('students.db')
cursor = conn.cursor()

# Create table if it doesn't exist
cursor.execute('''CREATE TABLE IF NOT EXISTS students
                  (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, age INTEGER)''')

# Function to add a new student
def add_student(name, age):
    cursor.execute('INSERT INTO students (name, age) VALUES (?, ?)', (name, age))
    conn.commit()
    print("Student added successfully!")

# Function to display all students
def view_students():
    cursor.execute('SELECT * FROM students')
    records = cursor.fetchall()
    print("\nID  | Name      | Age")
    print("----------------------")
    for record in records:
        print(f"{record[0]:<3} | {record[1]:<10} | {record[2]}")

# Adding a sample student
add_student('Alice', 20)

# Displaying all student records
view_students()

# Close the database connection
conn.close()
```

Output:-

```
ID  | Name      | Age
----------------------
1   | Alice     | 20
```

Practical 2:

```python
# Design a database application to search the specified record from the database. (IT LAB)

import sqlite3

# Connect to SQLite database (creates the file if it doesn't exist)
connection = sqlite3.connect('school.db')
cursor = connection.cursor()

# Create the 'students' table if it doesn't exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS students (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        age INTEGER
    )
''')

# Function to add a new student
def add_student(name, age):
    cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)", (name, age))
    connection.commit()
    print("Student added successfully!")

# Function to search for a student by name
def search_student(name):
    cursor.execute("SELECT * FROM students WHERE name = ?", (name,))
    records = cursor.fetchall()
    if records:
        print("\nFound Student Records:")
        print("ID | Name     | Age")
        print("--------------------")
        for record in records:
            print(f"{record[0]:<2} | {record[1]:<8} | {record[2]}")
    else:
        print("No student found with that name.")

# Function to view all students
def view_students():
    cursor.execute("SELECT * FROM students")
    records = cursor.fetchall()
    print("\nID | Name     | Age")
    print("--------------------")
    for record in records:
        print(f"{record[0]:<2} | {record[1]:<8} | {record[2]}")

# Main function for user interaction
def main():
    while True:
        print("\n1. Add Student")
        print("2. Search Student")
        print("3. View All Students")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            name = input("Enter student name: ")
            age = int(input("Enter student age: "))
            add_student(name, age)

        elif choice == '2':
            name = input("Enter student name to search: ")
            search_student(name)

        elif choice == '3':
            view_students()

        elif choice == '4':
            print("Exiting the program.")
            break

        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()

# Close the database connection
connection.close()
```

Output:-

```
1. Add Student
2. Search Student
3. View All Students
4. Exit
Enter your choice: 1
Enter student name: Aman Bhagat
Enter student age: 18
Student added successfully!

1. Add Student
2. Search Student
3. View All Students
4. Exit
Enter your choice: 2
Enter student name to search: Aman Bhagat

Found Student Records:
ID | Name        | Age
----------------------
1  | Aman Bhagat | 18
2  | Aman Bhagat | 18

1. Add Student
2. Search Student
3. View All Students
4. Exit
Enter your choice: 3

ID | Name        | Age
----------------------
1  | Aman Bhagat | 18
2  | Aman Bhagat | 18

1. Add Student
2. Search Student
3. View All Students
4. Exit
Enter your choice: 4
Exiting the program.
```

Practical 3:

```
# Design a database application to that allows the user to add, delete and modify the records. (HOMEWORK)

import sqlite3

# Connect to SQLite database (creates the file if it doesn't exist)
connection = sqlite3.connect('school.db')
cursor = connection.cursor()

# Create table if it doesn't exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS students (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        age INTEGER
    )
''')

# Function to add a new student
def add_student(name, age):
    cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)", (name, age))
    connection.commit()
    print("Student added successfully!")

# Function to search for a student by name
def search_student(name):
    cursor.execute("SELECT * FROM students WHERE name = ?", (name,))
    records = cursor.fetchall()
    if records:
        print("\nFound Student Records:")
        print("ID | Name     | Age")
        print("--------------------")
        for record in records:
            print(f"{record[0]:<2} | {record[1]:<8} | {record[2]}")
    else:
        print("No student found with that name.")

# Function to delete a student by ID
def delete_student(student_id):
    cursor.execute("DELETE FROM students WHERE id = ?", (student_id,))
    connection.commit()
    if cursor.rowcount > 0:
        print("Student deleted successfully!")
    else:
        print("No student found with that ID.")

# Function to update a student's information
def update_student(student_id, new_name, new_age):
    cursor.execute("UPDATE students SET name = ?, age = ? WHERE id = ?", (new_name, new_age, student_id))
    connection.commit()
    if cursor.rowcount > 0:
        print("Student record updated successfully!")
    else:
        print("No student found with that ID.")
```

```python
# Function to view all students
def view_students():
    cursor.execute("SELECT * FROM students")
    records = cursor.fetchall()
    print("\nID | Name     | Age")
    print("--------------------")
    for record in records:
        print(f"{record[0]:<2} | {record[1]:<8} | {record[2]}")

# Main function for user interaction
def main():
    while True:
        print("\n1. Add Student")
        print("2. Search Student")
        print("3. Update Student")
        print("4. Delete Student")
        print("5. View All Students")
        print("6. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            name = input("Enter student name: ")
            age = int(input("Enter student age: "))
            add_student(name, age)

        elif choice == '2':
            name = input("Enter student name to search: ")
            search_student(name)

        elif choice == '3':
            student_id = int(input("Enter student ID to update: "))
            new_name = input("Enter new name: ")
            new_age = int(input("Enter new age: "))
            update_student(student_id, new_name, new_age)

        elif choice == '4':
            student_id = int(input("Enter student ID to delete: "))
            delete_student(student_id)

        elif choice == '5':
            view_students()

        elif choice == '6':
            print("Exiting the program.")
            break

        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()

# Close the database connection
connection.close()
```

Output:-

```
1. Add Student
2. Search Student
3. Update Student
4. Delete Student
5. View All Students
6. Exit
Enter your choice: 1
Enter student name: Aman Bhagat
Enter student age: 18
Student added successfully!

1. Add Student
2. Search Student
3. Update Student
4. Delete Student
5. View All Students
6. Exit
Enter your choice: 2
Enter student name to search: Aman Bhagat
```

Learning Outcomes:

1. Understand how to establish and manage a database connection from an application.
2. Be able to execute SQL queries for data manipulation and retrieval.
3. Develop the ability to handle connection errors and optimize database interactions for performance.

Course Outcomes:

To master the process of establishing database connections, executing SQL queries, and managing data effectively for seamless application integration.

Conclusion:

Viva Questions:

1. What is a database connection, and why is it important in application development?
2. How do you establish a connection between a program and a database?
3. Can you explain the role of SQL queries in database operations?
4. What is the difference between a connection pool and a single database connection?
5. How do you handle database connection errors and ensure efficient resource management?

For Faculty use:

Correction Parameters	Formative Assessment [40%]	Timely Completion of Practical [40%]	Attendance Learning Attitude [20%]