Experiment No.:1 **Date:**

Experiment Title: Create a menu driven dictionary of elements having words and their corresponding meaning. Develop a menu driven program that offers user with the following options

1. **Add a new word to the dictionary.**
2. **Display the contents of dictionary.**
3. **Delete a word from the dictionary.**
4. **Search for a word in dictionary.**
5. **Modify the meaning of the word.**
6. **Quit.**

Program should continue to work until User wishes to quit.

Objectives:

- Understand the concept of dictionary and different methods in dictionary in Python.
- Understand the concept of loop control statements in Python.
- Learn to create a menu driven program in python.
- Learn to use membership operator appropriately

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment: Concepts used in the program

Dictionary data structure:

A **dictionary** in Python is an unordered collection of key-value pairs, where each key is unique, and it is associated with a corresponding value. Dictionaries are used to store and retrieve data efficiently.

Key Features of a Dictionary:

- Uses **curly braces {}** or the dict() constructor.
- Elements are stored as **key-value pairs**.
- Keys must be **immutable** (strings, numbers, tuples).
- Values can be of **any data type**.

Basic Dictionary Operations:

Creating a Dictionary:

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

Accessing Values:

```
print(my_dict["name"])  # Output: Alice
print(my_dict.get("age"))  # Output: 25
```

Adding/Modifying a Key-Value Pair:

```
my_dict["age"] = 26  # Modifies existing value
my_dict["profession"] = "Engineer"  # Adds a new key-value pair
```

Deleting an Element:

```
del my_dict["city"]  # Removes the key 'city'
```

Checking for a Key in Dictionary:

```
if "name" in my_dict:
        print("Key exists")
```

Looping Through a Dictionary:

```
for key, value in my_dict.items():
        print(key, ":", value)
```

While Loop

A **while loop** is used when a block of code needs to be executed repeatedly as long as a certain condition holds true.

Syntax:

```
while condition:
   # Code to execute
```

Example:

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

Explanation: The loop prints numbers from 1 to 5 and stops when counter exceeds 5.

A while loop is useful when we don’t know the number of iterations in advance, such as in user-driven menu-based programs.

If-Else Statements

An **if-else statement** is used to make decisions based on conditions. It helps execute different blocks of code depending on the user's input or program logic.

Syntax:

```
if condition:
   # Code block if condition is True
elif another_condition:
   # Code block if the second condition is True
else:
   # Code block if none of the conditions are True
```

Example:

```
num = 10
if num > 0:
```

```
    print("Positive number")
elif num < 0:
    print("Negative number")
else:
    print("Zero")
```

Explanation: The program checks whether num is positive, negative, or zero and executes the appropriate block.

User Input Handling

Python provides the input() function to take user input and store it as a string.

Example:

```
user_input = input("Enter your name: ")
print("Hello, " + user_input)
```

If numerical input is needed, it should be converted using int() or float():

```
num = int(input("Enter a number: "))
```

Learning Conclusion:

Course Outcome Attained:

Marks Allotted:

Activity	Marks
1. Conduction & Execution (Max: 5)	

Date: **Faculty Signature**

Experiment No.: 2 **Date:**

Experiment Title: write a menu driven python program with the following options using a user defined function for each task:

a. **Check is a number is prime.**
b. **Find the factorial of the number**
c. **Check whether a given number is even or odd**
d. **Check whether a given number is perfect number.**
e. **Exit the program**

Objectives:

1. **Understand and Implement User-Defined Functions** – Learn to write separate functions for each task, improving code modularity and reusability.
2. **Develop a Menu-Driven Program** – Use loops and conditionals to allow users to choose different operations dynamically.
3. **Apply Mathematical Concepts in Python** – Implement logic for prime numbers, factorials, even/odd checks, and perfect numbers using efficient algorithms.

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment: Concepts used

Functions (User-Defined Functions)

Functions allow code reusability and modularity by breaking the program into smaller, manageable parts. Each task in the menu (checking prime, factorial, even/odd, and perfect number) can be written as a separate function.

Defining and Calling a Function:

```
def greet():
    print("Hello, welcome to the program!")

greet()  # Function call
```

Function with Parameters:

```
def square(num):
    return num * num

result = square(4)  # Output: 16
```

Conditional Statements (If-Else)

Conditional statements help in making decisions based on user input. They are essential for checking if a number is **prime, even/odd, or perfect**.

Loops (While and For Loops)

Loops help in repetitive tasks like checking divisibility for prime numbers, calculating factorial, and verifying perfect numbers.

Taking User Input

The `input()` function is used to accept user choices and numbers for operations. Since `input()` returns a string, we use `int()` for numerical input.

```
choice = int(input("Enter your choice: "))

num = int(input("Enter a number: "))
```

Exit Condition in a Menu-Driven Program

A menu-driven program runs in a loop until the user chooses to exit. The `break` statement helps terminate the loop when needed.

```
while True:

    choice = int(input("Enter choice: "))

    if choice == 5:

        print("Exiting program...")

        break
```

Learning Conclusion:

Course Outcome Attained:

Marks Allotted:

Activity	Marks
1. Conduction & Execution (Max: 5)	

Date: **Faculty Signature**

Experiment No.: 3 **Date:**

Experiment Title: Create a table to store the population and land area of the Karnataka state (Assume Data).

a) **Create a new database called census.db.**
b) **Make a database table called Density that will hold the name of the state or territory (TEXT), the population (INTEGER), and the land area (REAL).**
c) **Insert data into the table.**
d) **Display the contents of the table.**
e) **Display the populations.**
f) **Display the states that have populations of less than 1 million.**
g) **Display the states that have populations less than 1 million or greater than 5 million.**
h) **Display the states that do not have populations less than 1 million or greater than 5 million.**
i) **Display the populations of states that have a land area greater than 200,000 square kilometers.**
j) **Display the states along with their population densities (population divided by land area).**

Objectives:

1. **Create and manage a database** – Learn to set up a SQLite database (`census.db`) and define tables.
2. **Store and retrieve structured data** – Insert, update, and display census information efficiently.
3. **Apply filtering conditions** – Use SQL `WHERE` clauses to extract specific population and land area data.
4. **Perform calculations in SQL** – Compute population density dynamically within queries.
5. **Enhance SQL querying skills** – Practice writing and executing queries for real-world data analysis.

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment:

Concepts Used in the Program

To implement this program, the following database and SQL concepts are required:

1. SQLite Database

SQLite is a lightweight database management system used to store and manage structured data. In this program, we create a database (`census.db`) to store population and land area details.

Creating a Database in SQLite:

```
import sqlite3
conn = sqlite3.connect("census.db")  # Creates a database file
cursor = conn.cursor()
```

2. Creating a Table

A table (`Density`) is created to store data using the `CREATE TABLE` SQL command. It consists of three columns:

- `State` (TEXT) – Stores the name of the state.
- `Population` (INTEGER) – Stores the population count.
- `Land Area` (REAL) – Stores the land area in square kilometers.

3. Inserting Data into the Table

The `INSERT INTO` command is used to add records to the table.

4. Retrieving and Displaying Data

The `SELECT` statement is used to fetch records from the table.

5. Filtering Data Using `WHERE` Clause

The `WHERE` clause helps in filtering data based on specific conditions.

6. Performing Calculations in SQL

SQL supports calculations using expressions. For example, population density can be calculated dynamically.

7. Using Logical Operators (`AND`, `OR`, `NOT`)

Logical operators refine search conditions.

8. Closing the Database Connection

After performing all operations, it is important to close the connection to free resources.

```
conn.close()
```

This program applies essential database concepts, including **database creation, table management, data insertion, retrieval, filtering, and calculations** using SQL queries.

Learning Conclusion:

Course Outcome Attained:

Marks Allotted:

Activity	Marks
1. Conduction & Execution (Max: 5)	

Date: **Faculty Signature**

Experiment No.: 4 **Date:**

Experiment Title: Create a Python program to simulate a bank account system with the following functionalities:

1. **Create Account**
2. **Deposit Money**
3. **Withdraw Money**
4. **Check Balance**
5. **Display Account Details**
6. **Exit**

Implement a menu where users can select options to perform these tasks.
The program should continue running until the user chooses to exit.

Objectives:

1. Learn the foundational concepts of OOP, including classes, objects, methods, and constructors.
2. Understand how to define a class and create objects to represent a bank account in the program.
3. Learn how to implement methods inside the class to perform tasks like deposit, withdrawal, balance checking, etc.
4. Use a constructor (`__init__`method) to initialize object attributes like account number, balance, etc., when creating a new account.
5. Practice implementing basic banking functionalities within an object-oriented structure.

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment

Concepts Required for the Experiment:

1. **Object-Oriented Programming (OOP):**
   - OOP is a programming paradigm that uses objects and classes to structure software programs. In this experiment, you will define a class (`BankAccount`) to simulate a bank account, and objects of this class will represent individual bank accounts.
2. **Class:**
   - A **class** is a blueprint for creating objects. It defines attributes (properties) and methods (functions) that operate on the data.
   - In this experiment, the `BankAccount` class will represent a bank account and contain methods for operations like deposit, withdrawal, and balance checking.
3. **Objects:**
   - **Objects** are instances of a class. Each object has its own set of data (attributes) and can perform the methods defined in the class.
   - For example, a specific bank account object might represent a customer's account, with attributes like `account_number` and `balance`.
4. **Methods:**
   - A **method** is a function defined inside a class that operates on the data of the class.

- For this experiment, methods will include `deposit()`, `withdraw()`, `check_balance()`, etc., which will manipulate and retrieve information related to a bank account.

5. **Constructor (`__init__`):**
   - The **constructor** (`__init__`) is a special method used to initialize an object's attributes when it is created.
   - When creating a new bank account, the constructor will initialize attributes like `account_number` and `balance` for each account.
6. **Input Handling and User Interaction:**
   - You will use basic input handling (`input()` function) to interact with the user. The program will display a menu of operations and prompt the user to select an option to perform on their account.
7. **Loops and Conditionals:**
   - To continuously offer menu options until the user chooses to exit, you will need to implement **loops** (e.g., `while` loop) to keep the program running.
   - **Conditionals** (`if-else`) will be used to determine which operation to execute based on user input.
8. **Exception Handling (Optional):**
   - To make the program robust, you might use exception handling (e.g., `try-except`) to deal with potential errors, like attempting to withdraw more money than available in the account.

Key Tasks in the Experiment:

- Define the BankAccount class with a constructor and methods for account operations.
- Create objects for individual bank accounts.
- Implement a menu-driven system to perform operations like creating an account, depositing, withdrawing, and checking balances.
- Continuously interact with the user until they choose to exit the program.

Learning Conclusion:

Course Outcome Attained:

Marks Allotted:

Activity	Marks
1. Conduction & Execution (Max: 5)	

Date: Faculty Signature

Experiment No.: 5 **Date:**

Write a Python program to demonstrate operator overloading by overloading the + operator to add two objects of a class Distance.

Objectives:

- Learn how to apply key OOP concepts such as classes, objects, and methods in solving real-world problems.
- Gain an understanding of how to overload operators to customize the behavior of common operators for user-defined objects.
- Improve the ability to break down problems and implement solutions using Python's object-oriented features, focusing on time management in this case.

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment:

Concept of Operator Overloading

Operator overloading in Python allows you to define how operators such as +, -, *, /, etc., behave for user-defined objects. By default, operators in Python perform operations on built-in data types like integers, floats, and strings. However, when you define your own classes, you can override the behavior of these operators to work with objects of your class.

This is done by defining special methods (also known as dunder methods) in your class. These methods allow objects of your class to interact with operators in a customized manner.

Common Operator Overloading Functions (Dunder Methods)

__add__(self, other) – Used to overload the + operator for adding two objects of a class. This method should return the result of the addition.

Example: Overloading + to add two Time objects (HH:MM:SS).

```
def _add_(self, other):
   total_seconds = self.to_seconds() + other.to_seconds()
   return Time.from_seconds(total_seconds)
```

__sub__(self, other) – Used to overload the - operator to subtract one object from another. It is commonly used for finding the difference between two objects.

Example: Subtracting two Time objects to get the time difference.

```
def __sub__(self, other):
    total_seconds = self.to_seconds() - other.to_seconds()
    return Time.from_seconds(total_seconds)
```

__mul__(self, other) – Overloads the * operator to allow multiplying an object by a scalar or another object. This can be useful for scaling values.

Example: Multiplying a Time object by an integer to scale the time.

```
def __mul__(self, other):
    if isinstance(other, int):
        total_seconds = self.to_seconds() * other
        return Time.from_seconds(total_seconds)
```

__eq__(self, other) – Overloads the == operator to compare two objects for equality. This method should return True or False.

Example: Comparing two Time objects for equality.

```
def __eq__(self, other):
    return self.to_seconds() == other.to_seconds()
```

__str__(self) – Overloads the str() function to return a string representation of an object, which is useful for printing.

Example: Representing a Time object as a string in HH:MM:SS format.

```
def __str__(self):
    return f"{self.hours:02}:{self.minutes:02}:{self.seconds:02}"
```

Advantages of Operator Overloading

1. Makes code more readable and intuitive by using natural operators for specific types of objects.
2. Provides flexibility in defining custom behaviors for common operators.
3. Simplifies the code by eliminating the need for custom method calls for common operations.
4. Ensures consistency with built-in types, making custom objects easier to work with.
5. Reduces code duplication by allowing operators to handle multiple tasks in a single expression.

Learning Conclusion:

Course Outcome Attained:

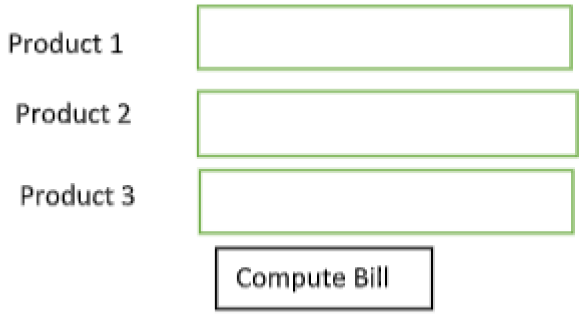
Marks Allotted:

Activity	Marks
1. Conduction & Execution (Max: 5)	

Date: **Faculty Signature**

Experiment No.: 6 **Date:**

Experiment Title: Create a GUI application using tkinter to design the following form and perform the actions mentioned below.



- Price of Product1=Rs.500/unit
- Price of Product2=Rs.50/unit upto 50 units, otherwise it is Rs.45
- Price of Product3=Rs.100/unit and minimum quantity to buy is 10 units

When Compute Bill button is clicked, final billing amount should be displayed.

Objectives:
- Learn to use tkinter module
- Understand how various widgets of tkinter work
- Learn to provide action to buttons

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment (Theory max. 500 words):
To solve the given problem, knowledge of following concepts is essential.
Tkinter is a standard Python library for creating graphical user interfaces (GUIs). It provides a set of widgets, such as buttons, labels, text boxes, and menus, that can be used to create interactive and responsive GUI applications.

Tkinter is based on the Tcl/Tk GUI toolkit, which is a cross-platform GUI toolkit that was developed for the Tcl scripting language. Tkinter provides a Python interface to the Tcl/Tk toolkit, which allows Python programmers to create GUI applications using the same toolkit.

Some of the features of Tkinter include:

1. Cross-platform support: Tkinter supports multiple platforms, including Windows, macOS, and Linux.

2. Easy to use: Tkinter provides a simple and easy-to-use API for creating GUI applications.

3. Customizable widgets: Tkinter provides a wide range of customizable widgets that can be used to create complex and interactive GUIs.

4. Event-driven programming: Tkinter uses an event-driven programming model, which means that the program responds to user actions such as button clicks, mouse movements, and keystrokes.

5. Object-oriented design: Tkinter is designed using object-oriented principles, which makes it easy to create and manage GUI components.

Here is an example of how to use Tkinter to create a simple GUI application that displays a label and a button:

```
import tkinter as tk

root = tk.Tk()

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

button = tk.Button(root, text="Click me!")
button.pack()

root.mainloop()
```

In this example, we first import the `tkinter` module and create a new `Tk` object, which is the main window of the GUI application. We then create a `Label` widget and a `Button` widget using the `Label()` and `Button()` functions, respectively. We use the `pack()` method to add the widgets to the main window.

Finally, we call the `mainloop()` method to start the GUI event loop, which waits for user input and responds to user actions. When the user clicks the button, the program can respond to the button click event using a callback function.

Here's an example program that demonstrates some of the common widgets available in Tkinter:

```
import tkinter as tk

# Create the main window
root = tk.Tk()

# Create a label widget
label = tk.Label(root, text="Welcome to my Tkinter demo!")
label.pack()

# Create a button widget with a callback function
def button_callback():
    print("Button was clicked!")

button = tk.Button(root, text="Click me!", command=button_callback)
button.pack()
```

```
# Create an entry widget for user input
entry = tk.Entry(root)
entry.pack()

# Create a checkbutton widget
checkbutton_var = tk.BooleanVar()
checkbutton_var.set(True)
checkbutton = tk.Checkbutton(root, text="Check me!", variable=checkbutton_var)
checkbutton.pack()

# Create a radiobutton widget
radiobutton_var = tk.StringVar()
radiobutton_var.set("option1")
radiobutton1 = tk.Radiobutton(root, text="Option 1", variable=radiobutton_var,
value="option1")
radiobutton2 = tk.Radiobutton(root, text="Option 2", variable=radiobutton_var,
value="option2")
radiobutton1.pack()
radiobutton2.pack()

# Create a listbox widget
listbox = tk.Listbox(root)
listbox.insert(1, "Item 1")
listbox.insert(2, "Item 2")
listbox.insert(3, "Item 3")
listbox.pack()

# Create a dropdown menu widget
def dropdown_callback(value):
    print(f"Selected value: {value}")

options = ["Option 1", "Option 2", "Option 3"]
dropdown_var = tk.StringVar()
dropdown_var.set(options[0])
dropdown = tk.OptionMenu(root, dropdown_var, *options,
command=dropdown_callback)
dropdown.pack()

# Start the main event loop
root.mainloop()
```

This program creates a main window, a label widget, a button widget with a callback function, an entry widget for user input, a checkbutton widget, a radiobutton widget with two options, a listbox widget, and a dropdown menu widget.

When the user clicks the button, the program prints a message to the console. When the user selects an option in the dropdown menu, the program prints the selected value to the console.

Result:

Learning Conclusion:

Course Outcome Attained:

Marks Allotted:

Activity	Marks
2. Conduction & Execution (Max: 5)	

Date: **Faculty Signature**

Experiment No.: 7 **Date:**

Experiment Title: Create a Python program for the **following exercises by using Numpy and pandas.**

- Create an identity matrix.
- Find the square root of each element in an array.
- Sort an array.
- Square each element in an array.
- Take logs of each element in an array.
- Create an array of zeros. Create an array of ones.
- Find the mean of array.
- Create two data frames and merge them.

Objectives:

- Understand the basics of NumPy and Pandas for data manipulation.
- Perform mathematical operations on arrays using NumPy functions.
- Create and modify arrays with specific properties (identity, zeros, ones).
- Implement sorting and statistical functions like mean.
- Merge and manipulate DataFrames using Pandas.

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment (Theory max. 500 words):

What is NumPy?

NumPy (Numerical Python) is a powerful Python library used for numerical computing. It provides efficient multi-dimensional array support and mathematical functions optimized for performance.

Why is NumPy useful?

- Faster and more memory-efficient than Python lists.
- Supports mathematical operations on entire arrays (vectorized operations).
- Provides built-in functions for linear algebra, statistics, and data manipulation.

What is Pandas?

Pandas is a Python library built for data manipulation and analysis. It provides two main data structures:

- **Series** – A one-dimensional labeled array.
- **DataFrame** – A two-dimensional table similar to an Excel sheet or SQL table.

Why is Pandas useful?

- Makes data handling easy with indexing and filtering.
- Supports data cleaning, merging, grouping, and transformation.

- Provides seamless integration with NumPy for numerical operations.

Why Are They Important for This Experiment?

- **NumPy** helps perform array-based calculations efficiently, like square roots, sorting, and mean calculation.
- **Pandas** helps create and manipulate structured data, such as merging datasets.

Key Concepts Used in the Experiment

1. NumPy Arrays

NumPy provides an array object (ndarray) that is more efficient and faster than Python lists for handling large datasets. It supports various mathematical operations that can be performed on the entire array at once.

- **Creating Arrays**:
    - Identity matrix using np.eye()
    - Arrays of zeros using np.zeros()
    - Arrays of ones using np.ones()
- **Mathematical Operations**:
    - Square root of elements using np.sqrt()
    - Squaring elements using np.square()
    - Logarithm of elements using np.log()
- **Sorting and Statistical Operations**:
    - Sorting arrays using np.sort()
    - Finding the mean using np.mean()

2. Pandas DataFrames

Pandas provides the DataFrame structure, which is a table-like data structure similar to an Excel sheet or SQL table. It supports efficient data handling and manipulation.

- **Creating DataFrames** using pd.DataFrame()
- **Merging DataFrames** using pd.merge(), which allows combining two datasets based on a common column.

Learning Conclusion:

Course Outcome Attained:

Marks Allotted:

Activity	**Marks**
3. Conduction & Execution (Max: 5)	

Date: **Faculty Signature**

Experiment No.: 8

Date: **Experiment Title: Create the following plots by using Matplotlib.**

- Line plot
- Histogram
- Bar Chart
- Scatter plot
- Pie charts

Objectives:

- Understand the basics of data visualization using Matplotlib.
- Create and customize different types of plots.
- Interpret data effectively through visual representation.
- Learn how to modify axes, labels, and legends for clarity.
- Develop skills to visualize trends, distributions, and comparisons in data.

Software used: Anaconda Jupyter Notebook, python 3.7

About Experiment:

This experiment focuses on using **Matplotlib**, a powerful Python library for creating static, animated, and interactive visualizations. The key concepts involved are:

1. Matplotlib Basics

- **What is Matplotlib?**
  Matplotlib is a Python library used for creating various types of plots and charts. It provides control over figure properties, axes, colors, labels, and legends.
- **How to Use Matplotlib?**
  - matplotlib.pyplot is the primary module for creating plots.
  - Plots can be displayed using plt.show().

2. Types of Plots in the Experiment

- **Line Plot (plt.plot())** – Used to show trends over time or continuous data.
- **Histogram (plt.hist())** – Used to display frequency distributions of numerical data.
- **Bar Chart (plt.bar())** – Represents categorical data with rectangular bars.
- **Scatter Plot (plt.scatter())** – Shows relationships between two numerical variables.
- **Pie Chart (plt.pie())** – Displays proportions of categories in a dataset.

3. Customizing Plots

- **Adding Titles and Labels** (plt.title(), plt.xlabel(), plt.ylabel()).
- **Changing Colors, Line Styles, and Markers** for better visualization.
- **Adding Legends** (plt.legend()) to identify different data categories.
- **Adjusting Axes and Grid** (plt.xlim(), plt.ylim(), plt.grid()).

4. Importance of Data Visualization

- Helps in understanding patterns and trends in data.
- Makes complex data easier to interpret.
- Useful for decision-making and data analysis.

Learning Conclusion:

Course Outcome Attained:

Marks Allotted:

Activity	Marks
1. Conduction & Execution (Max: 5)	

Date: **Faculty Signature**