RAMAIAH College of Arts, Science & Commerce - Autonomous

DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS

PYTHON LAB MANUAL

Lab Instructions to Students

1) **Mobile phones are strictly prohibited** during lab sessions. Keep them securely inside your bag.
2) Bags should be stored in the designated area and not kept on desks or lab tables.
3) Maintain a **separate notebook** titled **"Observation Notebook"**.
4) Write down the program you intend to execute in the lab in the observation notebook before coming to the lab.
5) Clearly mention the program's objective, code, and expected outcome in the observation notebook.
6) Before leaving the lab, make sure to:
    i. **Turn off the system** properly.
    ii. Arrange the chair and keyboard neatly in their respective positions.
    iii. Ensure the workstation is clean and tidy.

1. Write a program to analyze student scores for a class, using arrays and string operations. The program should:

a. Accept a list of student names and their respective scores.
b. Perform basic operations like finding the highest score, lowest score, and average score.
c. Identify students who scored above the average.
d. Use functions to modularize the code.

Solution:

```
# Function to accept student names and scores
def accept_scores():
    n = int(input("Enter the number of students: "))
    students = []
    scores = []

    for _ in range(n):
        name = input("Enter the student's name: ")
        score = float(input(f"Enter {name}'s score: "))
        students.append(name)
        scores.append(score)
    return students, scores
```

```
# Function to find the highest score
def highest_score(scores):
    return max(scores)

# Function to find the lowest score
def lowest_score(scores):
    return min(scores)

# Function to calculate the average score
def average_score(scores):
    return sum(scores) / len(scores)

# Function to find students who scored above the average
def students_above_average(students, scores, avg):
    return [students[i] for i in range(len(scores)) if scores[i] > avg]

# Main function
def main():
    # Accept data
    students, scores = accept_scores()

    # Perform basic operations
    high = highest_score(scores)
    low = lowest_score(scores)
    avg = average_score(scores)
    above_avg_students = students_above_average(students, scores, avg)

    # Display results
    print("\n--- Results ---")
    print(f"Highest Score: {high}")
    print(f"Lowest Score: {low}")
```

```
    print(f"Average Score: {avg:.2f}")
    print("Students scoring above average:")
    for student in above_avg_students:
        print(f"- {student}")

# Run the program
if __name__ == "__main__":
    main()
```

Sample output:

```
Enter the number of students: 3
Enter the student's name: Alice
Enter Alice's score: 85
Enter the student's name: Bob
Enter Bob's score: 90
Enter the student's name: Charlie
Enter Charlie's score: 78
```

Result:

```
Highest Score: 90.0
Lowest Score: 78.0
Average Score: 84.33
Students scoring above average:
- Alice
- Bob
```

Explanation

- **`range(n)`**: This generates a sequence of numbers from `0` to `n-1`. For example, if `n = 5`, `range(n)` produces `[0, 1, 2, 3, 4]`.
- `_`: This is a placeholder variable. It is used when the variable itself is not needed or when we don't intend to use it inside the loop.

Essentially, `for _ in range(n):` means, "repeat the loop `n` times, but we don't care about the loop variable."

The construct:

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

is a special Python idiom used to execute a block of code only when a Python file is run directly, and not when it is imported as a module. Let's break it down in detail:

2. a. Program to Create Numpy Arrays and perform arithmetic operations

Note: Install numpy : pip install numpy

Solution:

```
import numpy as np

# Function to read and create numpy arrays from user input
def create_array(prompt):
    print(prompt)
    elements = input("Enter the numbers separated by spaces: ").split()
    return np.array([int(x) for x in elements])

# Reading data from the user
array1 = create_array("Enter elements for Array 1:")
array2 = create_array("Enter elements for Array 2:")

# Performing arithmetic operations
sum_array = array1 + array2
diff_array = array1 - array2
prod_array = array1 * array2
div_array = array1 / array2

# Displaying results
print("\nArray 1:", array1)
print("Array 2:", array2)
print("\nSum of Arrays:", sum_array)
print("Difference of Arrays:", diff_array)
print("Product of Arrays:", prod_array)
print("Division of Arrays:", div_array)

# Additional operations
print("\nMean of Array 1:", np.mean(array1))
print("Max of Array 2:", np.max(array2))
```

SAMPLE OUTPUT:

```
Enter elements for Array 1:
```

```
10 20 30 40
Enter elements for Array 2:
1 2 3 4

Array 1: [10 20 30 40]
Array 2: [1 2 3 4]

Sum of Arrays: [11 22 33 44]
Difference of Arrays: [ 9 18 27 36]
Product of Arrays: [ 10  40  90 160]
Division of Arrays: [10. 10. 10. 10.]

Mean of Array 1: 25.0
Max of Array 2: 4
```

b. Program to perform list operations such as slicing, appending, and nested lists.

```
# Function to create a list from user input
def create_list(prompt):
    print(prompt)
    elements = input("Enter the numbers separated by spaces: ").split()
    return [int(x) for x in elements]

# Reading list from the user
my_list = create_list("Enter elements for the list:")

# Slicing operations
print("\nOriginal List:", my_list)
start = int(input("Enter start index for slicing: "))
end = int(input("Enter end index for slicing: "))
step = int(input("Enter step value for slicing: "))
print(f"Sliced List [{start}:{end}:{step}]:", my_list[start:end:step])

# Appending an item
to_append = int(input("\nEnter a number to append to the list: "))
my_list.append(to_append)
print("List after appending:", my_list)
```

```
# Extending the list
to_extend = create_list("Enter additional elements to extend the list:")
my_list.extend(to_extend)
print("List after extending:", my_list)
# Nested lists
nested_list = [my_list, [100, 200, 300]]
print("\nNested List:", nested_list)
# Accessing elements in a nested list
nested_index1 = int (input ("\nEnter index to access from the first list: "))
nested_index2 = int (input ("Enter index to access from the second list: "))
print ("Element in Nested List [0][{}]: {}".format(nested_index1, nested_list[0][nested_index1]))
print ("Element in Nested List [1][{}]: {}".format(nested_index2, nested_list[1][nested_index2]))
```

SAMPLE OUTPUT:

```
Enter elements for the list:
10 20 30 40 50 60

Original List: [10, 20, 30, 40, 50, 60]
Enter start index for slicing: 1
Enter end index for slicing: 4
Enter step value for slicing: 1
Sliced List [1:4:1]: [20, 30, 40]

Enter a number to append to the list: 70
List after appending: [10, 20, 30, 40, 50, 60, 70]

Enter additional elements to extend the list:
80 90
List after extending: [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Nested List: [[10, 20, 30, 40, 50, 60, 70, 80, 90], [100, 200, 300]]

Enter index to access from the first list: 2

Enter index to access from the second list: 1

Element in Nested List [0][2]: 30

Element in Nested List [1][1]: 200

3.a. Program to Perform CRUD Operations on Dictionaries

```
def display_dict(dictionary):
    print("\nCurrent Dictionary:", dictionary)

# Create (C)
students = {}
n = int(input("Enter the number of students to add: "))
for _ in range(n):
    name = input("Enter student name: ")
    score = int(input(f"Enter {name}'s score: "))
    students[name] = score
display_dict(students)

# Read (R)
print("\nReading a specific student's score:")
name = input("Enter the student name to look up: ")
if name in students:
    print(f"{name}'s score is {students[name]}")
else:
    print(f"No record found for {name}.")

# Update (U)
print("\nUpdating a student's score:")
name = input("Enter the student name to update: ")
if name in students:
```

```
    new_score = int(input(f"Enter the new score for {name}: "))
    students[name] = new_score
    print(f"{name}'s score has been updated.")
else:
    print(f"No record found for {name}.")
display_dict(students)

# Delete (D)
print("\nDeleting a student's record:")
name = input("Enter the student name to delete: ")
if name in students:
    del students[name]
    print(f"Record for {name} has been deleted.")
else:
    print(f"No record found for {name}.")
display_dict(students)
```

SAMPLE OUTPUT:

```
Enter the number of students to add: 3
Enter student name: Alice
Enter Alice's score: 85
Enter student name: Bob
Enter Bob's score: 90
Enter student name: Charlie
Enter Charlie's score: 95

Current Dictionary: {'Alice': 85, 'Bob': 90, 'Charlie': 95}

Reading a specific student's score:
Enter the student name to look up: Bob
Bob's score is 90
```

Updating a student's score:

Enter the student name to update: Alice

Enter the new score for Alice: 88

Alice's score has been updated.

Current Dictionary: {'Alice': 88, 'Bob': 90, 'Charlie': 95}

Deleting a student's record:

Enter the student name to delete: Bob

Record for Bob has been deleted.

Current Dictionary: {'Alice': 88, 'Charlie': 95}

3.b. Program to Demonstrate Tuple Packing, Unpacking, and Nested Tuples

```
# Tuple Packing
name = "Alice"
age = 25
country = "USA"
packed_tuple = (name, age, country)
print("Packed Tuple:", packed_tuple)

# Tuple Unpacking
name, age, country = packed_tuple
print("\nUnpacked Values:")
print("Name:", name)
print("Age:", age)
print("Country:", country)

# Nested Tuples
nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True, False))
print("\nNested Tuple:", nested_tuple)

# Accessing Elements in Nested Tuples
```

```
print("\nAccessing Elements in Nested Tuple:")
print("First element of first tuple:", nested_tuple[0][0])
print("Second element of second tuple:", nested_tuple[1][1])
print("Third element of third tuple:", nested_tuple[2][2])  # Intentional index error to show safe handling

# Iterating Over Nested Tuples
print("\nIterating over nested tuple:")
for sub_tuple in nested_tuple:
    print("Sub-tuple:", sub_tuple)
```

sample output:

```
Packed Tuple: ('Alice', 25, 'USA')
Unpacked Values:
Name: Alice
Age: 25
Country: USA

Nested Tuple: ((1, 2, 3), ('a', 'b', 'c'), (True, False))

Accessing Elements in Nested Tuple:
First element of first tuple: 1
Second element of second tuple: b

Iterating over nested tuple:
Sub-tuple: (1, 2, 3)
Sub-tuple: ('a', 'b', 'c')
Sub-tuple: (True, False)
```

Explanation

- **3 a: CRUD Operations on Dictionaries**:

- **Create**: Add student names and scores.
- **Read**: Retrieve the score of a specific student.
- **Update**: Modify the score of a student.
- **Delete**: Remove a student's record.
- **Display**: View the current state of the dictionary.

- **3 b: Tuple Operations**:
  - **Packing**: Combining multiple values into a tuple.
  - **Unpacking**: Assigning tuple elements to separate variables.
  - **Nested Tuples**: A tuple containing other tuples.
  - **Access and Iteration**: Demonstrates accessing elements and looping through nested tuples.

4.a. Program to Find the Mean, Median, and Mode for a Given Set of Numbers

```
from statistics import mean, median, mode

# Function to calculate mean

def calculate_mean(numbers):

    return mean(numbers)


# Function to calculate median

def calculate_median(numbers):

    return median(numbers)


# Function to calculate mode

def calculate_mode(numbers):

    try:

        return mode(numbers)

    except:

        return "No unique mode (multiple or no repeating elements)"


# Input: Accept a list of numbers from the user

numbers = list(map(int, input("Enter numbers separated by spaces: ").split()))


# Calculations

mean_value = calculate_mean(numbers)

median_value = calculate_median(numbers)

mode_value = calculate_mode(numbers)
```

```
# print values
print(f"\nNumbers: {numbers}")
print(f"Mean: {mean_value}")
print(f"Median: {median_value}")
print(f"Mode: {mode_value}")
```

Sample Output:

```
Enter numbers separated by spaces: 10 20 30 20 10 40 50 20

Numbers: [10, 20, 30, 20, 10, 40, 50, 20]
Mean: 25
Median: 20.0
Mode: 20
```

4.b. Program to Define a Function to Find All Duplicate Values in a List

```
# Function to find duplicate values
def find_duplicates(numbers):
    duplicates = set()
    seen = set()
    for num in numbers:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)
    return list(duplicates)

# Input: Accept a list of numbers from the user
numbers = list(map(int, input("Enter numbers separated by spaces: ").split()))
```

```
# Find duplicates
duplicates = find_duplicates(numbers)

# print values
print(f"\nNumbers: {numbers}")
if duplicates:
    print(f"Duplicate Values: {duplicates}")
else:
    print("No duplicate values found.")
```

Sample output:

```
Enter numbers separated by spaces: 10 20 30 40 20 50 30 40

Numbers: [10, 20, 30, 40, 20, 50, 30, 40]
Duplicate Values: [20, 30, 40]
```

Explanation

4.a. Mean, Median, and Mode

1. **Mean**:
   - Calculated as the sum of the numbers divided by the count.
   - Uses `statistics.mean()`.
2. **Median**:
   - The middle value in an ordered list.
   - Uses `statistics.median()`.
3. **Mode**:
   - The value that occurs most frequently.
   - Uses `statistics.mode()`.
   - If there's no unique mode, it gracefully handles exceptions.

4.b. Finding Duplicate Values

- **Logic**:

- Use two sets: `seen` to track numbers already encountered, and `duplicates` to store numbers that repeat.
- Iterate through the list and check for duplicates.

- **Output**:
  - Returns a list of duplicate values or indicates no duplicates.

5. Using the OOPs concepts, write a program for basic working of an ATM Machine.

Solution:

```
class ATM:
    def __init__(self, pin, balance=0):
        self.__pin = pin  # Private attribute for PIN
        self.__balance = balance  # Private attribute for balance

    # Function to verify the entered PIN
    def verify_pin(self, pin):
        return self.__pin == pin

    # Function to check balance
    def check_balance(self):
        return self.__balance

    # Function to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"₹{amount} deposited successfully.")
        else:
            print("Invalid deposit amount.")

    # Function to withdraw money
    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient balance.")
        elif amount <= 0:
            print("Invalid withdrawal amount.")
        else:
            self.__balance -= amount
```

```
            print(f"₹{amount} withdrawn successfully.")

# Main Program
def main():
    # Read PIN and initial balance from the user
    user_pin = int(input("Set your ATM PIN: "))
    initial_balance = float(input("Set your initial balance (₹): "))

    # Initialize ATM object with user-provided PIN and balance
    atm = ATM(pin=user_pin, balance=initial_balance)

    print("\nWelcome to the ATM!")
    entered_pin = int(input("Enter your PIN: "))

    # Verify PIN
    if not atm.verify_pin(entered_pin):
        print("Incorrect PIN. Access denied.")
        return

    while True:
        print("\nChoose an option:")
        print("1. Check Balance")
        print("2. Deposit Money")
        print("3. Withdraw Money")
        print("4. Exit")

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

        if choice == 1:
            # Check balance
            print(f"Your current balance is: ₹{atm.check_balance()}")
        elif choice == 2:
            # Deposit money
            amount = float(input("Enter the amount to deposit: ₹"))
            atm.deposit(amount)
        elif choice == 3:
```

```
            # Withdraw money
            amount = float(input("Enter the amount to withdraw: ₹"))
            atm.withdraw(amount)
        elif choice == 4:
            # Exit
            print("Thank you for using the ATM. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

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

output:

```
Set your ATM PIN: 1234
Set your initial balance (₹): 5000

Welcome to the ATM!
Enter your PIN: 1234

Choose an option:
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Enter your choice: 1
Your current balance is: ₹5000

Choose an option:
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Enter your choice: 2
Enter the amount to deposit: ₹1500
₹1500 deposited successfully.

Choose an option:
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Enter your choice: 3
Enter the amount to withdraw: ₹2000
₹2000 withdrawn successfully.
```

```
Choose an option:
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Enter your choice: 1
Your current balance is: ₹4500

Choose an option:
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Enter your choice: 4
Thank you for using the ATM. Goodbye!
```

6. Program to Implement a).single inheritance b). multilevel inheritance and c). multiple inheritance with constructors and overridden methods.

Solution:

a. Single Inheritance

```
# Base class
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Human created: {self.name}, Age: {self.age}")

    def speak(self):
        print(f"{self.name} can speak.")

# Derived class
class Man(Human):
    def __init__(self, name, age, profession):
        super().__init__(name, age)  # Call Human constructor
        self.profession = profession
        print(f"Man '{self.name}' works as a {self.profession}")

    def speak(self):  # Overriding the method
        print(f"{self.name}, the man, is speaking about {self.profession}.")
```

```
# Main function
def main():
    print("\n=== Single Inheritance ===")
    name = input("Enter name: ")
    age = int(input("Enter age: "))
    profession = input("Enter profession: ")
    man = Man(name, age, profession)
    man.speak()

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

output:

```
=== Single Inheritance ===
Enter name: sachin
Enter age: 45
Enter profession: cricketer
Human created: sachin, Age: 45
Man 'sachin' works as a cricketer
sachin, the man, is speaking about cricketer.
```

b. Multilevel Inheritance

```
# Base class
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Human created: {self.name}, Age: {self.age}")

    def speak(self):
        print(f"{self.name} can speak.")

# Derived class
```

```
class Man(Human):
    def __init__(self, name, age, profession):
        super().__init__(name, age)  # Call Human constructor
        self.profession = profession
        print(f"Man '{self.name}' works as a {self.profession}")

    def speak(self):  # Overriding the method
        print(f"{self.name}, the man, is speaking about {self.profession}.")

# Further derived class
class Sportsman(Man):
    def __init__(self, name, age, profession, sport):
        super().__init__(name, age, profession)
        self.sport = sport
        print(f"Sportsman '{self.name}' plays {self.sport}")

    def show_skills(self):
        print(f"{self.name} demonstrates his skills in {self.sport}.")

# Main function
def main():
    print("\n=== Multilevel Inheritance ===")
    name = input("Enter name: ")
    age = int(input("Enter age: "))
    profession = input("Enter profession: ")
    sport = input("Enter sport: ")
    sportsman = Sportsman(name, age, profession, sport)
    sportsman.speak()
    sportsman.show_skills()

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

output:

```
=== Multilevel Inheritance ===
```

```
Enter name: sachin
Enter age: 65
Enter profession: cricketer
Enter sport: cricket
Human created: sachin, Age: 65
Man 'sachin' works as a cricketer
Sportsman 'sachin' plays cricket
sachin, the man, is speaking about cricketer.
sachin demonstrates his skills in cricket.
```

c) Multiple Inheritance

```
# Class 1
class Sportsman:
    def __init__(self, name, sport):
        self.name = name
        self.sport = sport
        print(f"Sportsman '{self.name}' plays {self.sport}")

    def show_skills(self):
        print(f"{self.name} demonstrates his skills in {self.sport}.")

# Class 2
class Artist:
    def __init__(self, name, art_form):
        self.name = name
        self.art_form = art_form
        print(f"Artist '{self.name}' specializes in {self.art_form}")

    def perform(self):
        print(f"{self.name} performs a masterpiece in {self.art_form}.")

# Multiple inheritance
class TalentedMan(Sportsman, Artist):
    def __init__(self, name, sport, art_form):
```

```
        Sportsman.__init__(self, name, sport)
        Artist.__init__(self, name, art_form)
        print(f"Talented man '{self.name}' excels in both {self.sport} and {self.art_form}.")

    def introduce(self):
        print(f"My name is {self.name}. I'm talented in both {self.sport} and {self.art_form}.")

# Main function
def main():
    print("\n=== Multiple Inheritance ===")
    name = input("Enter name: ")
    sport = input("Enter sport: ")
    art_form = input("Enter art form: ")
    talented_man = TalentedMan(name, sport, art_form)
    talented_man.introduce()
    talented_man.show_skills()
    talented_man.perform()

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

output

```
=== Multiple Inheritance ===
Enter name: Sachin
Enter sport: Cricket
Enter art form: painting
Sportsman 'Sachin' plays Cricket
Artist 'Sachin' specializes in painting
Talented man 'Sachin' excels in both Cricket and painting.
My name is Sachin. I'm talented in both Cricket and painting.
Sachin demonstrates his skills in Cricket.
Sachin performs a masterpiece in painting.
```

7. Program to Demonstrate Polymorphism (method overloading, method overriding, and operator overloading.)

Solution:

```
# Method Overloading
class Calculator:
    def add(self, a, b, c=0):
        """Performs addition of two or three numbers."""
        return a + b + c

# Method Overriding
class Animal:
    def speak(self):
        print("Animal makes a sound.")

class Dog(Animal):
    def speak(self):
        print("Dog barks.")

class Cat(Animal):
    def speak(self):
        print("Cat meows.")

# Operator Overloading
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        """Overloads the '+' operator to add two complex numbers."""
        return ComplexNumber(self.real + other.real, self.imag + other.imag)

    def __str__(self):
        """Overloads the string representation for printing complex numbers."""
        return f"{self.real} + {self.imag}i"
```

```
# Main function
if __name__ == "__main__":
    # Method Overloading Example
    calc = Calculator()
    print("Addition of two numbers:", calc.add(10, 20))
    print("Addition of three numbers:", calc.add(10, 20, 30))

    # Method Overriding Example
    animal = Animal()
    dog = Dog()
    cat = Cat()

    animal.speak()  # Output: Animal makes a sound.
    dog.speak()     # Output: Dog barks.
    cat.speak()     # Output: Cat meows.

    # Operator Overloading Example
    c1 = ComplexNumber(2, 3)
    c2 = ComplexNumber(4, 5)
    c3 = c1 + c2  # '+' operator overloaded
    print("Sum of complex numbers:", c3)
```

output:

```
Addition of two numbers: 30
Addition of three numbers: 60
Animal makes a sound.
Dog barks.
Cat meows.
Sum of complex numbers: 6 + 8i
```

8. Program to create an abstract class with abstract methods and demonstrate inheritance with concrete subclasses.

Solution:

- **Abstract Class** → A class that cannot be instantiated and contains at least one abstract method.
- **Abstract Method** → A method that must be implemented by subclasses.
- **Concrete Subclass** → A subclass that provides implementations for all abstract methods.

```
from abc import ABC, abstractmethod

# Abstract class
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        """Abstract Method: Must be implemented by subclasses"""
        pass

    def sleep(self):
        """Concrete Method: Can be used directly by subclasses"""
        print("Sleeping...")

# Concrete subclass 1
class Dog(Animal):
    def make_sound(self):
        print("Dog says: Woof!")

# Concrete subclass 2
class Cat(Animal):
    def make_sound(self):
        print("Cat says: Meow!")

# Concrete subclass 3
class Cow(Animal):
    def make_sound(self):
        print("Cow says: Moo!")

# Creating objects of concrete subclasses and calling methods
dog = Dog()
dog.make_sound()
dog.sleep()

cat = Cat()
cat.make_sound()
cat.sleep()

cow = Cow()
cow.make_sound()
cow.sleep()
```

OutPut:

```
Dog says: Woof!
Sleeping...
Cat says: Meow!
Sleeping...
Cow says: Moo!
Sleeping...
```

9. Program to create Data Frames from dictionaries, lists, or CSV files and perform basic operations like filtering and grouping Using Pandas

Solution:

```
import pandas as pd

#  Creating a DataFrame from a Dictionary
data = {
    "Name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "Age": [25, 30, 35, 40, 29],
    "Department": ["HR", "IT", "Finance", "IT", "HR"],
    "Salary": [50000, 60000, 70000, 65000, 48000]
}

df = pd.DataFrame(data)
print("\n=== DataFrame from Dictionary ===")
print(df)

# Saving DataFrame to a CSV file, it create employees.csv automatically, if it not exits
df.to_csv("employees.csv", index=False)  # Save without index

#  Reading DataFrame from a CSV file
df_csv = pd.read_csv("employees.csv")
print("\n=== DataFrame from CSV ===")
print(df_csv)

#  Filtering Data: Employees with Salary > 50,000
filtered_df = df[df["Salary"] > 50000]
print("\n=== Employees with Salary > 50,000 ===")
print(filtered_df)

#  Grouping Data by Department
grouped_df = df.groupby("Department")["Salary"].mean()
print("\n=== Average Salary per Department ===")
print(grouped_df)
```

Output:

```
=== DataFrame from Dictionary ===

      Name  Age Department  Salary
0   Alice   25         HR   50000
1     Bob   30         IT   60000
2 Charlie   35    Finance   70000
3   David   40         IT   65000
4     Eve   29         HR   48000

=== DataFrame from CSV ===


      Name  Age Department  Salary
0   Alice   25         HR   50000
1     Bob   30         IT   60000
2 Charlie   35    Finance   70000
3   David   40         IT   65000
4     Eve   29         HR   48000

=== Employees with Salary > 50,000 ===

      Name  Age Department  Salary

1     Bob   30         IT   60000
2 Charlie   35    Finance   70000
3   David   40         IT   65000

=== Average Salary per Department ===

Department
Finance    70000.0
HR         49000.0
IT         62500.0

Name: Salary, dtype: float64
```

Explanation of the Code:

1. **Creating a DataFrame from a Dictionary:**
   - The dictionary contains columns: "Name", "Age", "Department", and "Salary".
   - pd.DataFrame(data) converts it into a Pandas DataFrame.
2. **Saving the DataFrame to a CSV file:**
   - df.to_csv("employees.csv", index=False) saves the DataFrame to a CSV file without row indices.
3. **Reading Data from a CSV file:**
   - pd.read_csv("employees.csv") reads the CSV file into a Pandas DataFrame.

4. **Filtering Data:**
    - df[df["Salary"] > 50000] filters employees who have a salary greater than 50,000.
5. **Grouping Data:**
    - df.groupby("Department")["Salary"].mean() groups data by "Department" and calculates the average salary for each department.

10. Program to perform file operations such as reading, writing, and appending text and binary files.

Solution

- `write_text_file()`: Writes data to a text file.
- `read_text_file()`: Reads data from a text file.
- `append_text_file()`: Appends new data to the existing file.
- `write_binary_file()`: Writes binary data to a binary file.
- `read_binary_file()`: Reads binary data from a file.

```
def write_text_file(filename, content):
    with open(filename, "w") as file:
        file.write(content)
    print("Text file written successfully.")

def read_text_file(filename):
    with open(filename, "r") as file:
        content = file.read()
    print("Text file content:")
    print(content)

def append_text_file(filename, content):
    with open(filename, "a") as file:
        file.write(content)
    print("Content appended successfully.")

def write_binary_file(filename, data):
    with open(filename, "wb") as file:
        file.write(data)
    print("Binary file written successfully.")

def read_binary_file(filename):
    with open(filename, "rb") as file:
        data = file.read()
    print("Binary file content:")
    print(data)
```

```
# Example Usage
text_filename = "sample.txt"
binary_filename = "sample.bin"

write_text_file(text_filename, "Hello, this is a sample text file.\n")
read_text_file(text_filename)
append_text_file(text_filename, "Appending new text.\n")
read_text_file(text_filename)

write_binary_file(binary_filename, b"\x41\x42\x43\x44")  # Writing binary data
read_binary_file(binary_filename)
```

OutPut:

```
Text file written successfully.
Text file content:
Hello, this is a sample text file.

Content appended successfully.
Text file content:
Hello, this is a sample text file.
Appending new text.

Binary file written successfully.
Binary file content:
b'ABCD'
```

11. Program to Connect Python to a MySQL database, create tables, and perform insert, update, delete, and fetch operations.

NOTE: Instead of MySQL database using SQLite Database since it's a self-contained, serverless, and zero-configuration database management system **(DBMS).**

Solution

```
import sqlite3

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

# Create a table
cursor.execute("""
    CREATE TABLE IF NOT EXISTS employees (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
        name TEXT NOT NULL,
        age INTEGER,
        department TEXT
    )
""")
conn.commit()

# Insert data
cursor.execute("INSERT INTO employees (name, age, department) VALUES (?, ?, ?)",
               ("Alice", 30, "HR"))
cursor.execute("INSERT INTO employees (name, age, department) VALUES (?, ?, ?)",
               ("Bob", 25, "IT"))
conn.commit()

# Fetch and display data
cursor.execute("SELECT * FROM employees")
rows = cursor.fetchall()
print("Employee Records:")
for row in rows:
    print(row)

# Update a record
cursor.execute("UPDATE employees SET age = 35 WHERE name = 'Alice'")
conn.commit()

# Delete a record
cursor.execute("DELETE FROM employees WHERE name = 'Bob'")
conn.commit()

# Fetch updated data
cursor.execute("SELECT * FROM employees")
print("\nUpdated Employee Records:")
for row in cursor.fetchall():
    print(row)

# Close connection
conn.close()
```

OUTPUT:

```
Employee Records:
(1, 'Alice', 30, 'HR')
(2, 'Bob', 25, 'IT')

Updated Employee Records:
(1, 'Alice', 35, 'HR')
```

What is SQLite?

SQLite is a lightweight, self-contained, serverless, and zero-configuration **database management system** (DBMS). It is built into Python and many other programming languages, making it easy to use without installing additional software.

Unlike MySQL or PostgreSQL, **SQLite does not require a separate database server**. Instead, it stores the entire database as a **single file on disk** (`.db` file).

Why Use SQLite?

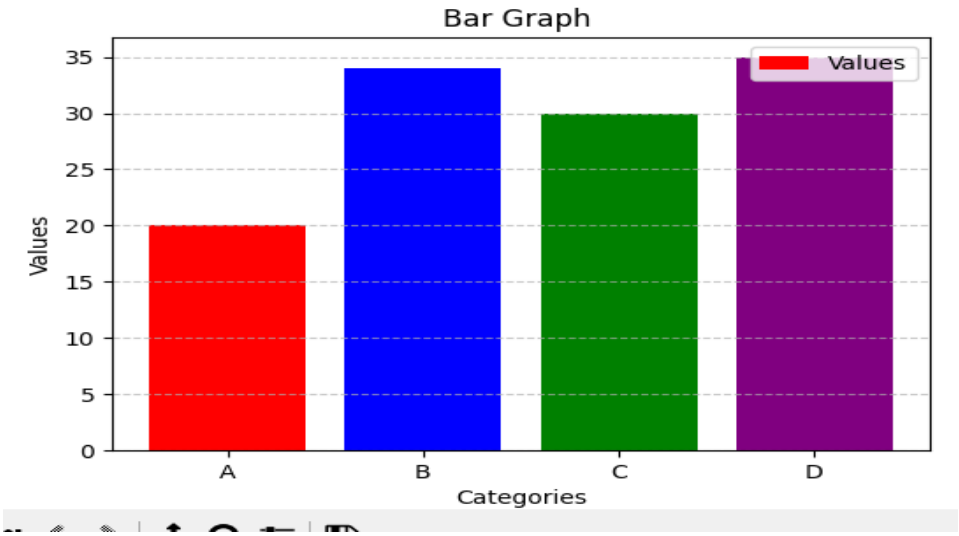
- **No need to install a separate database server** (comes built-in with Python).
- **Lightweight and easy to use** for small to medium-scale applications.
- **Uses a single file** (`database.db`) to store all data.

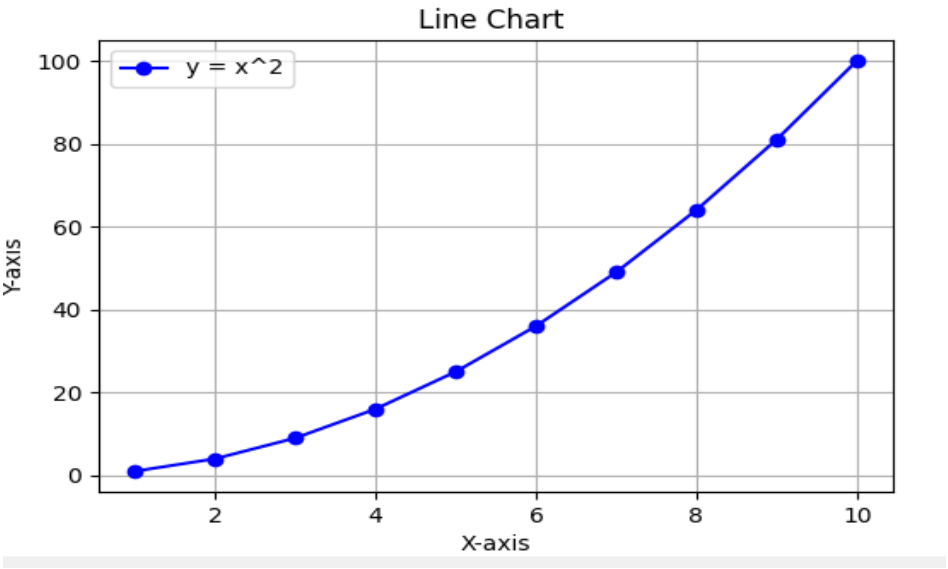
12. Program to create data visualizations using Matplotlib. Plot bar graphs, line charts, histograms, and pie charts with customized labels, titles, and legends.

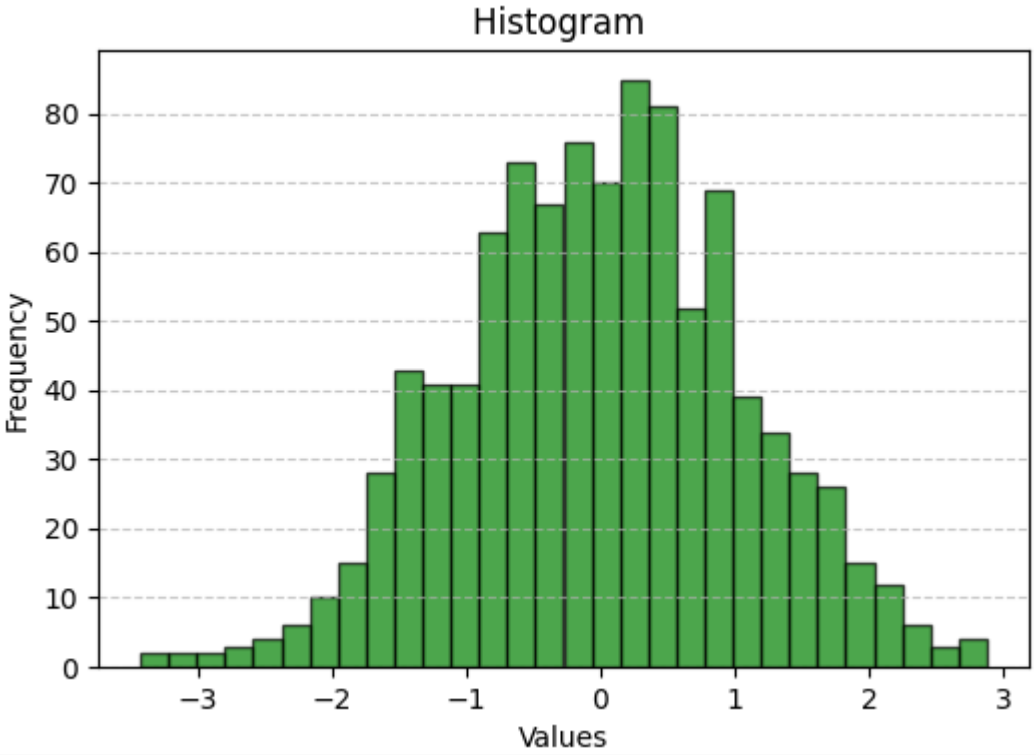
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
categories = ['A', 'B', 'C', 'D']
values = [20, 34, 30, 35]

# 1. **Bar Graph**
plt.figure(figsize=(6, 4))
plt.bar(categories, values, color=['red', 'blue', 'green', 'purple'])
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Bar Graph ")
plt.legend(["Values"], loc="upper right")
plt.grid(axis='y', linestyle="--", alpha=0.7)
plt.show()

# 2. **Line Chart**
x = np.arange(1, 11)
y = x ** 2  # Quadratic function

plt.figure(figsize=(6, 4))
plt.plot(x, y, marker='o', linestyle='-', color='b', label="y = x^2")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Line Chart ")
plt.legend()
plt.grid(True)
plt.show()

# 3. **Histogram**
```

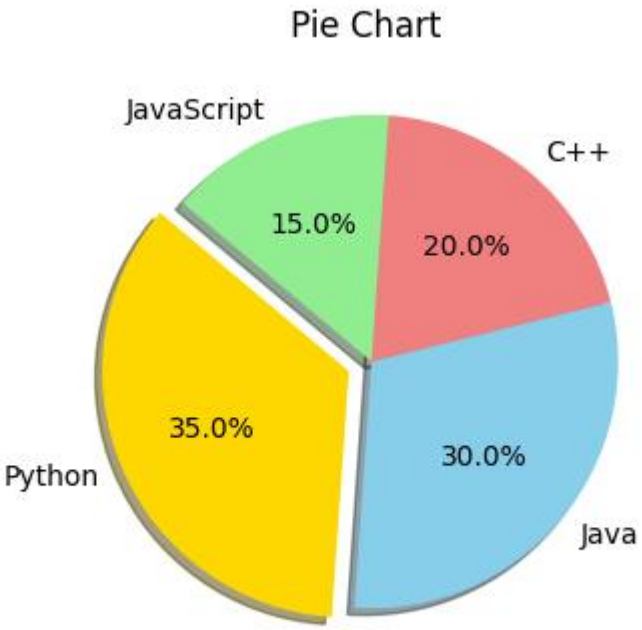
```
data = np.random.randn(1000)  # Generating random data
plt.figure(figsize=(6, 4))
plt.hist(data, bins=30, color='green', edgecolor='black', alpha=0.7)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Histogram ")
plt.grid(axis='y', linestyle="--", alpha=0.7)
plt.show()

# 4. **Pie Chart**
labels = ["Python", "Java", "C++", "JavaScript"]
sizes = [35, 30, 20, 15]
colors = ["gold", "skyblue", "lightcoral", "lightgreen"]
explode = (0.1, 0, 0, 0)  # Highlight the first slice

plt.figure(figsize=(6, 4))
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=colors, startangle=140,
explode=explode, shadow=True)
plt.title("Pie Chart ")
plt.show()
```

output:


Bar Graph
Values
35
30
25
20
15
10
5
0
Values
A
B
C
D
Categories


Line Chart
y = x^2
100
80
60
40
20
0
Y-axis
2
4
6
8
10
X-axis





Explanation of Each Chart

1. **Bar Graph:**
   - Displays values for four categories (`A, B, C, D`).
   - Custom colors, labels, and grid lines are used.
   - `plt.bar()` is used to create the bar graph.
2. **Line Chart:**

- Plots a quadratic function (`y = x^2`).
- Uses `plt.plot()` with markers, color, and labels.
- Grid and legend are enabled.

3. **Histogram:**
   - Uses `np.random.randn(1000)` to generate **normally distributed random data**.
   - `plt.hist()` is used to plot a frequency distribution with 30 bins.
4. **Pie Chart:**
   - Represents programming language popularity.
   - Uses **explode** to highlight the largest portion.
   - Adds percentage labels with `autopct='%1.1f%%'`

13. Program related to comprehension- map, filter and reduce.

```
from functools import reduce

# List Comprehension: Generate a list of squares from 1 to 10
squares = [x**2 for x in range(1, 11)]
print("List of squares:", squares)

# Map: Convert Celsius to Fahrenheit
temps_celsius = [0, 10, 20, 30, 40]
temps_fahrenheit = list(map(lambda c: (c * 9/5) + 32, temps_celsius))
print("Temperatures in Fahrenheit:", temps_fahrenheit)

# Filter: Get even numbers from a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print("Even numbers:", even_numbers)

# Reduce: Find the product of all numbers in a list
num_list = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, num_list)
print("Product of numbers:", product)
```

OUTPUT:

```
List of squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Temperatures in Fahrenheit: [32.0, 50.0, 68.0, 86.0, 104.0]
Even numbers: [2, 4, 6, 8, 10]
Product of numbers: 120
```

Explanation:

Comprehension, Map, Filter, and Reduce in Python

These are techniques in Python that help in writing **cleaner, more efficient, and readable** code for operations on lists and iterables.

- **List Comprehension**

List comprehension is a concise way to create lists in Python. It replaces for loops when you need to apply a simple transformation to an iterable.

```
# Generate a list of squares from 1 to 10 using list comprehension
squares = [x**2 for x in range(1, 11)]
print("List of squares:", squares)
```

- **map() Function**

The map() function applies a function to each item in an iterable (list, tuple, etc.).

```
# Convert a list of temperatures from Celsius to Fahrenheit using map()
temps_celsius = [0, 10, 20, 30, 40]
temps_fahrenheit = list(map(lambda c: (c * 9/5) + 32, temps_celsius))
print("Temperatures in Fahrenheit:", temps_fahrenheit)
```

- **filter() Function**

The filter() function filters elements based on a condition.

14. Implement Mini project based on the concept learnt in Theory.

Solution:

1. **Write about your Mini Project:**
   - **Title of the Project**
   - **Objective of the Project**
   - **Concepts Used in Implementation**
2. **Describe the Implementation:**
   - **Explanation of how the project is developed.**
   - **Key functionalities demonstrated.**
3. **Provide Output Details:**
   - **Expected results and observations.**
4. **Attach Screenshots:**
   - **Paste relevant screenshots of the code, execution, and final output.**