PRESIDENCY UNIVERSITY

Presidency University Act, 2013 of the Karnataka Act No. 41 of 2013 | Established under Section 2(f) of UGC Act, 1956
Approved by AICTE, New Delhi

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

BACHELOR OF TECHNOLOGY

Lab Manual

Course Code	CSE3216
Course Name	Mastering Object Oriented Concepts in Python
Credit Structure	0-0-2-1
Year / Semester	II/IV
Specialization	B.Tech all programs

Prepared by	Ms. Yogeetha B R

VISION OF PRESIDENCY SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

To be a value based, practice-driven School of Computer Science and Engineering, committed to developing globally-competent Professionals, dedicated to applying Modern Information Science for Social Benefit

MISSION OF PRESIDENCY SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

- Cultivate a practice-driven environment with an Information-Technology-based pedagogy, integrating theory and practice.
- Attract and nurture world-class faculty to excel in Teaching and Research, in the Information Science Domain.
- Establish state-of-the-art facilities for effective Teaching and Learning experiences.
- Promote Interdisciplinary Studies to nurture talent for global impact.

- Instil Entrepreneurial and Leadership Skills to address Social, Environmental and Community-needs.

PROGRAM OUTCOMES :

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. [H]

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. [M]

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations. [L]

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice. [L]

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings. [L]

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES:

At the end of the B. Tech. Program in Computer Science and Engineering and CSE-Allied the students shall:

PSO1: [Problem Analysis]: Identify, formulate, research literature, and analyze complex engineering problems related to Software Engineering principles and practices, Programming and Computing technologies reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PSO2: [Design/development of Solutions]: Design solutions for complex engineering problems related to Software Engineering principles and practices, Programming and Computing technologies and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PSO3: [Modern Tool usage]: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities related to Software Engineering principles and practices, Programming and Computing technologies with an understanding of the limitations.

COURSE OBJECTIVES:

The objective of the course is to familiarize the learners with the concepts of Mastering Object Oriented Concepts in Python and attain Skill Development through Experiential Learning.

COURSE OUTCOMES:

On successful completion of the course the students shall be able to:

TABLE 1: COURSE OUTCOMES		
CO Number	**CO**	**BLOOMS LEVEL**
CO1	Explain features of Oops along with creation of Python classes and objects to represent real world Objects.	**Understand**
CO2	Demonstrate inheritance, polymorphism, and abstraction in Python to build maintainable and extendable software systems.	**Apply**
CO3	Demonstrate exception handling in Python to build robust error-handling mechanisms and debugging tool and Assess various file handling techniques in Python.	**Apply**

MAPPING OF C.O. WITH P.O :

[H-HIGH , M- MODERATE, L-LOW]

TABLE 2a: CO PO Mapping ARTICULATION MATRIX												
CO. No.	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	**H**	**M**	-	**-**	-	**L**	**-**	**-**	**L**	**-**	**-**	**M**
CO2	**H**	**M**	**-**	**-**	**L**	**L**	-	-	**L**	**-**	**-**	**M**
CO3	**H**	**M**	**-**	**-**	**L**	**L**	-	-	**L**	**-**	**-**	**M**

MAPPING OF C.O. WITH PSO :

TABLE 2b: CO PSO Mapping ARTICULATION MATRIX			
CO.No.	**PSO-1**	**PSO-2**	**PSO-3**
CO1	**H**	-	-
CO2	**H**	**L**	**L**
CO3	**L**	**H**	-

Assessment Component

ASSESSMENT SCHEDULE							
Sl.No	**Assessment type**	**Contents**	**Course outcome Number**	**Duration In Hours**	**Marks**	**Weightage**	**Tentative Date**
1	CA1/Lab Test1	Module 1	CO1	30	30	30%	20-02-2025
2	CA2/Lab Test2	Module 2	CO1 and CO2	30	30	30%	25-03-2025
3	CA3/Lab Test 3	Module 3	CO1, CO2 and CO3	30	40	40%	10-05-2025

List of Experiments

S.No	Name of the Experiment
1.	Demonstrate a Python program that defines a class Car with a default constructor that initializes attributes for brand and year, and a method display_info to print the car's details. Create an object of the class and call the method to display the information.
2.	Program to demonstrate Instance Variables and class Variables.
3.	Program to demonstrate Class Methods, Instance Methods and Static Methods
4.	Program to demonstrate Inner Classes and Passing Members from one Class to another class
5.	Demonstrate the functionality of the Bank class by creating an instance of the class(name,accno,phno), and showcase how to access its public, private, and protected attributes.
6.	Demonstrate a mini project Bank Account class(All attributes should be private) to simulate basic banking operations such as depositing money, withdrawing money, checking account balance, and displaying account details. Create an instance of the class with initial values and demonstrate each of these operations.
7.	Demonstrate a Python program using a class called Car that includes attributes for make, model, mileage, and price. The program should include methods to display the car's details, start and stop the car, and update its price. Provide an example of creating two car objects and calling their methods.
8.	Create a Python program with a class Employee that includes a class attribute company set to "Tech Corp". The class should have private attributes __emp_id and __name, as well as a public attribute salary. Implement getter and setter methods for emp_id to control access to this private attribute. Add a method display_employee to print employee details, including the company name.
9.	Create a Python application for a simple Student Management System. The application should include the following functionalities: Class Definition: Define a Student class that: Has instance variables name and age. Has a class variable college_name that is shared among all instances. Methods: Implement the following methods within the Student class: Instance Methods: get_details(): Returns the details of the student (name, age, and college name).

	Class Methods: change_college(new_college): Allows changing the college_name for all instances of Student. Static Methods: is_adult(age): Takes an age as input and returns True if the age is 18 or above, otherwise returns False. Inner Class: Create an inner class called Address that has attributes city and state. It should also have a method get_address() to return the address details. Data Transfer Between Classes: Define another class called Course. This class should: Accept a Student object during initialization and store its details. Have a method get_student_info() that returns the details of the student in that course.
10.	Demonstrate how to use super keyword and how method overriding in inheritance works by using the class names as parent. Parent class person as attribute name and age Create child class employee with attributes emp_id ,salary and show the method overriding.
11.	Write a Python program that demonstrates abstraction by creating an abstract class Vehicle with an abstract method number_of_wheels(). Then, implement two concrete classes Car and Bike that inherit from Vehicle and define the number of wheels for each. Instantiate the Car and Bike classes and display the number of wheels for each vehicle.
12.	Create an Employee class with attributes: `employee_id`, `name`, `department`, and `salary` and add a method calculate_salary that calculates the employee's total salary. The basic salary is a required input, but the employee may also receive a bonus and overtime pay. If only the basic salary is provided, the total salary is equal to the basic salary.If a bonus is provided, the total salary should include the bonus. If overtime hours are provided, the total salary should also include the overtime pay, calculated at a rate of INR 750 per hour.
13.	Create a base class Shape and derived classes Circle and Rectangle. The Shape class should have a method to calculate area . The Circle class should inherit from Shape and have an attribute for the radius. The Rectangle class should inherit from Shape and have attributes for length and width. Formula: circle: 3.14 X radius X radius rectangle: length X breadth
14.	Demonstrate a program using a class named Person where: a. An attribute name is initialized with the value "Raja".

	b. Attempt is made to access another attribute age which is not defined. c. Handle the resulting AttributeError using a try-except block and provide an appropriate message.
15.	Demonstrate a program using a class Math where: a. Two numbers n1 and n2 are passed during object creation and stored as attributes. b. The class includes two methods: cal_add (intended for addition) and cal_sub (intended for subtraction). c. Due to a logical error, cal_add performs subtraction and cal_sub performs addition.
16.	Demonstrate a program where: a. The user is prompted to input a number. b. An attempt is made to divide a fixed value (10) by the entered number. c. Use a try-except block to handle multiple exceptions: ZeroDivisionError: Raised when the user enters 0. ValueError: Raised when the input is not a valid integer. It should be satisfied with that following inputs: • A valid positive integer (e.g., 5) • 0 • An invalid input (e.g., "abc")
17.	Demonstrate a program to understand File open and closing in Python
18.	Demonstrate Student Data Management System where the following operations need to be implemented: 1. **Store Student Information:** o Create a text file (`students.txt`) to store student information in the format: ▪ Student Name ▪ Age ▪ Course o Store multiple students' data in this text file. 2. **Update Student Information:** o Append additional information (like new students) to the existing `students.txt` file. o You must be able to update a student's course or age, based on the student name, by editing the respective record. 3. **Search for Student by Name:** o Read the `students.txt` file and search for a student's information by their name. o If the student is found, display their information. o If the student is not found, print an appropriate message. 4. **Student Information Backup (Serialization):** o Serialize the student data to a binary file (`students_backup.pickle`) using `pickle` to create a backup of the current student records. 5. **Restore Student Data from Backup:** o Unpickle the backup file (`students_backup.pickle`) and restore the data back to the system. This will allow you to read the students' information and display it. 6. **Seek and Tell Usage:** o Demonstrate how to use the `seek()` and `tell()` methods

	while reading the student data file. Use `seek()` to navigate to specific positions in the file and use `tell()` to get the current file pointer position

SPECIFIC GUIDELINES TO STUDENTS:

- Maintain Minimum 75% Attendance: Regular attendance is mandatory to ensure continuity in learning and understanding of the experiments.
- Missing lab sessions can lead to gaps in understanding and affect your ability to complete experiments and assignments.
- Carefully follow the instructions provided by the course instructor during both lectures and lab sessions.
- Ensure all assignments, reports, and projects are submitted before the deadline. Late submissions may result in penalties.
- Treat all lab equipment, including VR headsets, controllers, and computers, with care to avoid damage.

Experiment No : 1

Name of the Experiment : Demonstrate a Python program that defines a class Car with a default constructor that initializes attributes for brand and year, and a method display_info to print the car's details. Create an object of the class and call the method to display the information.

Source Code:

```
class Car:
    # Default constructor to initialize brand and year
    def __init__(self, brand="Toyota", year=2020):
        self.brand = brand
        self.year = year
    # Method to display car information
    def display_info(self):
        print(f"Car Brand: {self.brand}")
        print(f"Year: {self.year}")
# Create an object of the Car class
my_car = Car("Honda", 2022)
# Call the display_info method to print car details
my_car.display_info()
```

Sample Output:

Car Brand: Honda

Year: 2022

Experiment No : 2

Name of the Experiment : Program to demonstrate Instance Variables and class Variables.

Source Code:

```
class Car:
    # Class variable (shared among all instances)
    wheels = 4
    def __init__(self, brand, year):
        # Instance variables (specific to each instance)
        self.brand = brand
        self.year = year
    # Method to display car details
    def display_info(self):
        print(f"Car Brand: {self.brand}")
        print(f"Year: {self.year}")
        print(f"Wheels: {self.wheels}")
# Creating two instances of the Car class
car1 = Car("Toyota", 2020)
car2 = Car("Honda", 2022)
# Accessing instance variables and class variable
car1.display_info()
car2.display_info()
# Modifying the class variable using the class name
Car.wheels = 6
print("\nAfter changing class variable 'wheels' to 6:")
car1.display_info()
car2.display_info()
```

Sample Output:

```
Car Brand: Toyota
Year: 2020
Wheels: 4
Car Brand: Honda
Year: 2022
Wheels: 4
```

After changing class variable 'wheels' to 6:

Car Brand: Toyota

Year: 2020

Wheels: 6

Car Brand: Honda

Year: 2022

Wheels: 6

Experiment No : 3

Name of the Experiment : Program to demonstrate Class Methods, Instance Methods and Static Methods

Source Code:

```
class Car:
    # Class variable
    wheels = 4

    def __init__(self, brand, year):
        # Instance variables
        self.brand = brand
        self.year = year

    # Instance method (takes 'self' as the first argument)
    def display_info(self):
        print(f"Car Brand: {self.brand}")
        print(f"Year: {self.year}")
        print(f"Wheels: {self.wheels}")

    # Class method (takes 'cls' as the first argument)
    @classmethod
    def change_wheels(cls, new_wheels):
        cls.wheels = new_wheels
        print(f"Wheels updated to {cls.wheels} for all cars")

    # Static method (doesn't take 'self' or 'cls' as the first argument)
    @staticmethod
```

```
    def is_eco_friendly(brand):
        eco_friendly_brands = ["Tesla", "Nissan", "Chevrolet"]
        if brand in eco_friendly_brands:
            return True
        return False
```

Sample Output:

```
Car 1 Info:
Car Brand: Toyota
Year: 2020
Wheels: 4
Car 2 Info:
Car Brand: Tesla
Year: 2022
Wheels: 4
Calling class method to update wheels...
Wheels updated to 6 for all cars
Is Tesla eco-friendly?  True
Is Toyota eco-friendly?  False
```

Experiment No : 4

Name of the Experiment : Program to demonstrate Inner Classes and Passing Members from one Class to another class

Source Code:

Class Person:

```
        def __init__(self):
                self.name = 'Charles'
                self.db = self.Dob()
        def display(self):
                print('Name= ', self.name)
        #this is inner class
        class Dob:
                def __init__(self):
                        self.dd =10
                        self.mm = 5
                        self.yy = 1987
```

```
            def display(self):
                print('Dob = {}/{}/{}'.format(self.dd, self.mm,self.yy))
#creating Person class object
p = Person()
p .display()
#create inner class object
x = p.db
x.display()
```

Sample Output:

```
Name: Charles
Dob: 10/05/1987
```

Experiment No : 5

Name of the Experiment : Demonstrate the functionality of the Bank class by creating an instance of the class (name,accno,phno), and showcase how to access its public, private, and protected attributes.

Source Code:

```
class Bank:
    # Public attribute
    def __init__(self, name, accno, phno):
        self.name = name        # Public attribute
        self.accno = accno      # Public attribute
        self.phno = phno        # Public attribute
        self._balance = 0       # Protected attribute (by convention)
        self.__pin = "1234"     # Private attribute (by convention)

    # Method to deposit money (updates the protected attribute)
    def deposit(self, amount):
        self._balance += amount
        print(f"Deposited: {amount}. New balance: {self._balance}")

    # Method to get the pin (accessing the private attribute inside the class)
    def get_pin(self):
        return self.__pin

    # Method to display information (demonstrating public and protected attributes)
```

```
    def display_info(self):
        print(f"Customer Name: {self.name}")
        print(f"Account Number: {self.accno}")
        print(f"Phone Number: {self.phno}")
        print(f"Balance: {self._balance}")

# Creating an instance of the Bank class
customer = Bank("John Doe", "123456789", "9876543210")

# Accessing public attributes
print(f"Public Attribute (Name): {customer.name}")
print(f"Public Attribute (Account Number): {customer.accno}")
print(f"Public Attribute (Phone Number): {customer.phno}")

# Accessing protected attribute (conventionally discouraged but possible)
print(f"Protected Attribute (Balance): {customer._balance}")

# Accessing private attribute (this will raise an error if accessed directly)
# print(f"Private Attribute (Pin): {customer.__pin}")  # This would raise an AttributeError

# Accessing private attribute using a method
print(f"Private Attribute (Pin) through method: {customer.get_pin()}")

# Demonstrating deposit method and updated balance
customer.deposit(500)

# Displaying customer information using the public method
customer.display_info()
```

Sample Output:

```
Public Attribute (Name): John Doe
Public Attribute (Account Number): 123456789
Public Attribute (Phone Number): 9876543210
Protected Attribute (Balance): 0
Private Attribute (Pin) through method: 1234
```

Deposited: 500. New balance: 500

Customer Name: John Doe

Account Number: 123456789

Phone Number: 9876543210

Balance: 500

Experiment No : 6

Name of the Experiment : Demonstrate a mini project Bank Account class(All attributes should be private) to simulate basic banking operations such as depositing money, withdrawing money, checking account balance, and displaying account details. Create an instance of the class with initial values and demonstrate each of these operations.

Source Code:

```
class BankAccount:
    def __init__(self, name, accno, initial_balance):
        # Private attributes (by convention)
        self.__name = name
        self.__accno = accno
        self.__balance = initial_balance

    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: {amount}. New balance: {self.__balance}")
        else:
            print("Deposit amount must be positive!")

    # Method to withdraw money
    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive!")
        elif amount > self.__balance:
            print("Insufficient funds!")
        else:
            self.__balance -= amount
            print(f"Withdrawn: {amount}. New balance: {self.__balance}")
```

```
    # Method to check balance
    def check_balance(self):
        print(f"Current balance: {self.__balance}")

    # Method to display account details
    def display_account_details(self):
        print(f"Account Holder: {self.__name}")
        print(f"Account Number: {self.__accno}")
        print(f"Account Balance: {self.__balance}")


# Creating an instance of BankAccount
account = BankAccount("Alice Johnson", "987654321", 1000)

# Displaying account details
account.display_account_details()

# Depositing money
account.deposit(500)

# Withdrawing money
account.withdraw(200)

# Checking balance
account.check_balance()

# Trying to withdraw more than available balance
account.withdraw(1500)

# Checking balance again
account.check_balance()
```

Sample Output:

```
Account Holder: Alice Johnson
```

Account Number: 987654321

Account Balance: 1000

Deposited: 500. New balance: 1500

Withdrawn: 200. New balance: 1300

Current balance: 1300

Insufficient funds!

Current balance: 1300

Experiment No : 7

Name of the Experiment : Demonstrate a Python program using a class called Car that includes attributes for make, model, mileage, and price. The program should include methods to display the car's details, start and stop the car, and update its price. Provide an example of creating two car objects and calling their methods.

Source Code:

```
class Car:
    def __init__(self, make, model, mileage, price):
        # Initialize the car's attributes
        self.make = make
        self.model = model
        self.mileage = mileage
        self.price = price
        self.is_running = False  # Car is initially stopped

    def display_details(self):
        # Display the car's details
        print(f"Car Details:\nMake: {self.make}\nModel: {self.model}\nMileage: {self.mileage} miles\nPrice: ${self.price}")
        print(f"Status: {'Running' if self.is_running else 'Stopped'}\n")

    def start(self):
        # Start the car
        if not self.is_running:
            self.is_running = True
            print(f"{self.make} {self.model} has started.\n")
        else:
```

```
            print(f"{self.make} {self.model} is already running.\n")

    def stop(self):
        # Stop the car
        if self.is_running:
            self.is_running = False
            print(f"{self.make} {self.model} has stopped.\n")
        else:
            print(f"{self.make} {self.model} is already stopped.\n")

    def update_price(self, new_price):
        # Update the car's price
        self.price = new_price
        print(f"The price of {self.make} {self.model} has been updated to ${self.price}.\n")


# Example of creating two car objects
car1 = Car("Toyota", "Camry", 25000, 22000)
car2 = Car("Honda", "Civic", 30000, 18000)

# Calling methods on car1
car1.display_details()
car1.start()
car1.update_price(21000)
car1.display_details()
car1.stop()

# Calling methods on car2
car2.display_details()
car2.start()
car2.stop()
car2.update_price(17000)
car2.display_details()
```

Sample Output:

Car Details:

Make: Toyota

Model: Camry

Mileage: 25000 miles

Price: $22000

Status: Stopped

Toyota Camry has started.

The price of Toyota Camry has been updated to $21000.

Car Details:

Make: Toyota

Model: Camry

Mileage

Experiment No : 8

Name of the Experiment : Create a Python program with a class Employee that includes a class attribute company set to "Tech Corp". The class should have private attributes __emp_id and __name, as well as a public attribute salary. Implement getter and setter methods for emp_id to control access to this private attribute. Add a method display_employee to print employee details, including the company name.

Source Code:

```
class Employee:
    # Class attribute
    company = "Tech Corp"

    def __init__(self, emp_id, name, salary):
        # Private attributes
        self.__emp_id = emp_id
        self.__name = name
        # Public attribute
        self.salary = salary

    # Getter method for emp_id
    def get_emp_id(self):
```

```
        return self.__emp_id

    # Setter method for emp_id
    def set_emp_id(self, emp_id):
        if isinstance(emp_id, int) and emp_id > 0:
            self.__emp_id = emp_id
        else:
            print("Invalid Employee ID. It should be a positive integer.")

    # Getter method for name
    def get_name(self):
        return self.__name

    # Setter method for name
    def set_name(self, name):
        if isinstance(name, str) and len(name) > 0:
            self.__name = name
        else:
            print("Invalid name. It should be a non-empty string.")

    # Method to display employee details
    def display_employee(self):
        print(f"Employee Details:\nCompany: {Employee.company}\nEmployee ID:
{self.__emp_id}\nName: {self.__name}\nSalary: ${self.salary}")


# Example of creating and using the Employee class
emp1 = Employee(101, "Alice Johnson", 60000)
emp2 = Employee(102, "Bob Smith", 55000)

# Display details of both employees
emp1.display_employee()
print("\n")
emp2.display_employee()
```

```
# Update employee details using setters
emp1.set_emp_id(103)
emp1.set_name("Alicia Johnson")
emp1.salary = 65000

# Display updated details of emp1
print("\nUpdated Employee Details:")
emp1.display_employee()
```

Sample Output:

```
Employee Details:
Company: Tech Corp
Employee ID: 101
Name: Alice Johnson
Salary: $60000
Employee Details:
Company: Tech Corp
Employee ID: 102
Name: Bob Smith
Salary: $55000
Updated Employee Details:
Employee Details:
Company: Tech Corp
Employee ID: 103
Name: Alicia Johnson
Salary: $65000
```


Experiment No : 9

Name of the Experiment : Create a Python application for a simple Student Management System. The application should include the following functionalities:

Class Definition: Define a Student class that:

Has instance variables name and age.

Has a class variable college_name that is shared among all instances.

Methods: Implement the following methods within the Student class:

Instance Methods:

get_details(): Returns the details of the student (name, age, and college name).

Class Methods:

change_college(**new_college**): Allows changing the college_name for all instances of Student.

Static Methods:

is_adult(age): Takes an age as input and returns True if the age is 18 or above, otherwise returns False.

Inner Class: Create an inner class called Address that has attributes city and state. It should also have a method get_address() to return the address details.

Data Transfer Between Classes: Define another class called Course. This class should:

Accept a Student object during initialization and store its details.

Have a method get_student_info() that returns the details of the student in that course.

Source Code:

```
class Student:
    # Class variable
    college_name = "ABC University"

    def __init__(self, name, age):
        # Instance variables
        self.name = name
        self.age = age

    # Instance method to get details of the student
    def get_details(self):
        return f"Name: {self.name}, Age: {self.age}, College: {Student.college_name}"

    # Class method to change the college name for all instances
    @classmethod
    def change_college(cls, new_college):
        cls.college_name = new_college

    # Static method to check if the student is an adult
```

```
    @staticmethod
    def is_adult(age):
        return age >= 18

    # Inner class for Address
    class Address:
        def __init__(self, city, state):
            self.city = city
            self.state = state

        # Method to return the address details
        def get_address(self):
            return f"City: {self.city}, State: {self.state}"


# Another class to associate a student with a course
class Course:
    def __init__(self, student):
        # Accepting a Student object during initialization
        self.student = student

    # Method to get student info in the course
    def get_student_info(self):
        return self.student.get_details()

# Creating student objects
student1 = Student("Alice", 20)
student2 = Student("Bob", 17)

# Display student details
print(student1.get_details())
print(student2.get_details())

# Checking if the students are adults
```

```
print(f"Alice is an adult: {Student.is_adult(student1.age)}")
print(f"Bob is an adult: {Student.is_adult(student2.age)}")

# Changing the college for all students
Student.change_college("XYZ Institute")

# Display updated student details after college change
print("\nAfter changing college:")
print(student1.get_details())
print(student2.get_details())

# Creating address for student1
address1 = student1.Address("New York", "NY")
print(address1.get_address())

# Creating Course object with student1
course = Course(student1)
print(f"Student info in the course: {course.get_student_info()}")
```

Sample Output:

```
Name: Alice, Age: 20, College: ABC University
Name: Bob, Age: 17, College: ABC University
Alice is an adult: True
Bob is an adult: False

After changing college:
Name: Alice, Age: 20, College: XYZ Institute
Name: Bob, Age: 17, College: XYZ Institute
City: New York, State: NY
Student info in the course: Name: Alice, Age: 20, College: XYZ Institute
```

Experiment No : 10

Name of the Experiment : Demonstrate how to use super keyword and how method overriding in inheritance works by using the class names as parent. Parent class person as attribute name and age Create child class employee with attributes emp_id ,salary and show the method overriding.

Source Code:

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

    # Method to display details of the person
    def show_details(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")

class Employee(Person):
    def __init__(self, name, age, emp_id, salary):
        # Call the parent class constructor using super() to initialize name and age
        super().__init__(name, age)
        self.emp_id = emp_id
        self.salary = salary

    # Override the show_details method from the parent class
    def show_details(self):
        # Call the parent class's show_details using super() to include name and age
        super().show_details()
        print(f"Employee ID: {self.emp_id}")
        print(f"Salary: {self.salary}")

# Create instances of Person and Employee
person1 = Person("John", 30)
employee1 = Employee("Alice", 25, 101, 50000)

# Display details of both person and employee
print("Person Details:")
person1.show_details()

print("\nEmployee Details:")
employee1.show_details()
```

Sample Output:

Person Details:

Name: John

Age: 30

Employee Details:

Name: Alice

Age: 25

Employee ID: 101

Salary: 50000

Experiment No : 11

Name of the Experiment : Write a Python program that demonstrates abstraction by creating an abstract class Vehicle with an abstract method number_of_wheels(). Then, implement two concrete classes Car and Bike that inherit from Vehicle and define the number of wheels for each. Instantiate the Car and Bike classes and display the number of wheels for each vehicle.

Source Code:

```
from abc import ABC, abstractmethod


# Abstract class Vehicle
class Vehicle(ABC):


    @abstractmethod
    def number_of_wheels(self):
        pass


# Concrete class Car
class Car(Vehicle):
    def number_of_wheels(self):
        return 4  # Cars typically have 4 wheels


# Concrete class Bike
class Bike(Vehicle):
    def number_of_wheels(self):
        return 2  # Bikes typically have 2 wheels
```

```
# Instantiate Car and Bike
car = Car()
bike = Bike()

# Display number of wheels for each vehicle
print(f"A car has {car.number_of_wheels()} wheels.")
print(f"A bike has {bike.number_of_wheels()} wheels.")
```

Sample Output:

```
A car has 4 wheels.
A bike has 2 wheels.
```

Experiment No : 12

Name of the Experiment : Create an Employee class with attributes: `employee_id`, `name`, `department`, and `salary` and add a method calculate_salary that calculates the employee's total salary. The basic salary is a required input, but the employee may also receive a bonus and overtime pay. If only the basic salary is provided, the total salary is equal to the basic salary.If a bonus is provided, the total salary should include the bonus. If overtime hours are provided, the total salary should also include the overtime pay, calculated at a rate of INR 750 per hour.

Source Code:

```
class Employee:
    def __init__(self, employee_id, name, department, basic_salary, bonus=0, overtime_hours=0):
        # Attributes
        self.employee_id = employee_id
        self.name = name
        self.department = department
        self.basic_salary = basic_salary
        self.bonus = bonus
        self.overtime_hours = overtime_hours

    # Method to calculate the total salary
    def calculate_salary(self):
        # Overtime pay is calculated at a rate of INR 750 per hour
        overtime_pay = self.overtime_hours * 750
        # Total salary includes basic salary, bonus, and overtime pay
        total_salary = self.basic_salary + self.bonus + overtime_pay
```

```
        return total_salary

    # Method to display employee details along with their total salary
    def display_employee_details(self):
        print(f"Employee ID: {self.employee_id}")
        print(f"Name: {self.name}")
        print(f"Department: {self.department}")
        print(f"Basic Salary: INR {self.basic_salary}")
        print(f"Bonus: INR {self.bonus}")
        print(f"Overtime Hours: {self.overtime_hours}")
        print(f"Total Salary: INR {self.calculate_salary()}")


# Creating employee objects and calculating salary
emp1 = Employee(employee_id=101, name="Alice", department="IT", basic_salary=50000, bonus=5000, overtime_hours=10)
emp2 = Employee(employee_id=102, name="Bob", department="HR", basic_salary=40000)  # No bonus or overtime

# Displaying employee details
print("Employee 1 Details:")
emp1.display_employee_details()
print("\nEmployee 2 Details:")
emp2.display_employee_details()
```

Sample Output:

```
Employee 1 Details:
Employee ID: 101
Name: Alice
Department: IT
Basic Salary: INR 50000
Bonus: INR 5000
Overtime Hours: 10
Total Salary: INR 57500

Employee 2 Details:
```

Employee ID: 102

Name: Bob

Department: HR

Basic Salary: INR 40000

Bonus: INR 0

Overtime Hours: 0

Total Salary: INR 40000

Experiment No : 13

Name of the Experiment : Create a base class Shape and derived classes Circle and Rectangle. The Shape class should have a method to calculate area . The Circle class should inherit from Shape and have an attribute for the radius. The Rectangle class should inherit from Shape and have attributes for length and width. Formula: circle: 3.14 X radius X radius rectangle: length X breadth.

Source Code:

```
# Base class Shape
class Shape:
    def calculate_area(self):
        pass  # This will be overridden in the derived classes


# Derived class Circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius * self.radius  # Area of circle: π * r^2


# Derived class Rectangle
class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        return self.length * self.width  # Area of rectangle: length * breadth
```

```
# Creating instances of Circle and Rectangle
circle = Circle(radius=5)
rectangle = Rectangle(length=4, width=6)

# Calculating and displaying the area of the shapes
print(f"Area of Circle: {circle.calculate_area()} square units")
print(f"Area of Rectangle: {rectangle.calculate_area()} square units")
```

Sample Output:

```
Area of Circle: 78.5 square units
Area of Rectangle: 24 square units
```

Experiment No : 14

Name of the Experiment : Demonstrate a program using a class named Person where:

a. An attribute name is initialized with the value "Raja".
b. Attempt is made to access another attribute age which is not defined.
c. Handle the resulting AttributeError using a try-except block and provide an appropriate message.

Source Code:

```
class Person:
    def __init__(self):
        # Initializing the name attribute
        self.name = "Raja"

# Creating an instance of Person
person = Person()

try:
    # Attempting to access the 'age' attribute which is not defined
    print(f"Person's age: {person.age}")
except AttributeError as e:
    # Handling the AttributeError
    print(f"Error: {e}")
    print("The attribute 'age' is not defined for this person.")
```

Sample Output:

Error: 'Person' object has no attribute 'age'

The attribute 'age' is not defined for this person.

Experiment No : 15

Name of the Experiment : Demonstrate a program using a class Math where:

a. Two numbers n1 and n2 are passed during object creation and stored as attributes.
b. The class includes two methods: cal_add (intended for addition) and cal_sub (intended for subtraction). c. Due to a logical error, cal_add performs subtraction and cal_sub performs addition.

Source Code:

```
class Math:
    def __init__(self, n1, n2):
        # Initializing the attributes n1 and n2
        self.n1 = n1
        self.n2 = n2

    # Method intended for addition, but due to the logical error, it performs subtraction
    def cal_add(self):
        return self.n1 - self.n2  # Logical error: should be addition but performs subtraction

    # Method intended for subtraction, but due to the logical error, it performs addition
    def cal_sub(self):
        return self.n1 + self.n2  # Logical error: should be subtraction but performs addition

# Creating an instance of Math with two numbers
math_obj = Math(10, 5)

# Demonstrating the logical errors
print(f"cal_add (should add but subtracts): {math_obj.cal_add()}")  # This performs subtraction
print(f"cal_sub (should subtract but adds): {math_obj.cal_sub()}")  # This performs addition
```

Sample Output:

```
cal_add (should add but subtracts): 5
cal_sub (should subtract but adds): 15
```

Experiment No : 16

Name of the Experiment : Demonstrate a program where: a. The user is prompted to input a number. b. An attempt is made to divide a fixed value (10) by the entered number. c. Use a try-except block to handle multiple exceptions: ZeroDivisionError: Raised when the user enters 0. ValueError: Raised when the input is not a valid integer. It should be satisfied with that following inputs: • A valid positive integer (e.g., 5) • 0 • An invalid input (e.g., "abc")

Source Code:

```
# Program to demonstrate handling multiple exceptions


try:
    # Prompting the user to input a number
    user_input = input("Please enter a number to divide 10: ")

    # Trying to convert the user input to an integer
    num = int(user_input)

    # Attempting to divide 10 by the entered number
    result = 10 / num
    print(f"Result of division: 10 / {num} = {result}")

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

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

Sample Output:

Case 1: Valid Input (e.g., 5)
Please enter a number to divide 10: 5

Result of division: 10 / 5 = 2.0

Case 2: Zero Input (e.g., 0)

Please enter a number to divide 10: 0

Error: Cannot divide by zero.

Case 3: Invalid Input (e.g., "abc")

Please enter a number to divide 10: abc

Error: Invalid input. Please enter a valid integer.

Experiment No : 17

Name of the Experiment : Demonstrate a program to understand File open and closing in Python

Source Code:

```
# Program to demonstrate file opening and closing in Python


# Open a file in write mode ('w') and write some text to it
with open("example_file.txt", "w") as file:
    file.write("Hello, this is a simple file.\n")
    file.write("This demonstrates file open and closing in Python.\n")
    print("Data has been written to the file.")


# Open the file again in read mode ('r') to read its contents
with open("example_file.txt", "r") as file:
    content = file.read()  # Read the entire content of the file
    print("Contents of the file:")
    print(content)


# The file is automatically closed after the 'with' block ends, no need to explicitly call file.close()
```

Sample Output:

Data has been written to the file.

Contents of the file:

Hello, this is a simple file.

This demonstrates file open and closing in Python.

Experiment No : 18

Name of the Experiment : Demonstrate Student Data Management System where the following operations need to be implemented:

7. **Store Student Information:**
    - Create a text file (students.txt) to store student information in the format:
        - Student Name
        - Age
        - Course
    - Store multiple students' data in this text file.
8. **Update Student Information:**

- Append additional information (like new students) to the existing students.txt file.
- You must be able to update a student's course or age, based on the student name, by editing the respective record.

9. **Search for Student by Name:**
   - Read the students.txt file and search for a student's information by their name.
   - If the student is found, display their information.
   - If the student is not found, print an appropriate message.
10. **Student Information Backup (Serialization):**
   - Serialize the student data to a binary file (students_backup.pickle) using pickle to create a backup of the current student records.
11. **Restore Student Data from Backup:**
   - Unpickle the backup file (students_backup.pickle) and restore the data back to the system. This will allow you to read the students' information and display it.
12. **Seek and Tell Usage:**
   - Demonstrate how to use the seek() and tell() methods while reading the student data file.
   - Use seek() to navigate to specific positions in the file and use tell() to get the current file pointer position.

Source Code:

```
import os

import pickle


# File paths

text_file = "students.txt"

pickle_file = "students_backup.pickle"


# Student class to represent student information

class Student:

    def __init__(self, name, age, course):

        self.name = name

        self.age = age

        self.course = course


    def __str__(self):

        return f"{self.name}, Age: {self.age}, Course: {self.course}"


# Function to store student information in the text file

def store_student_data():

    with open(text_file, "a") as file:
```

```
        while True:
            name = input("Enter student name (or 'exit' to stop): ")
            if name.lower() == 'exit':
                break
            age = int(input(f"Enter age for {name}: "))
            course = input(f"Enter course for {name}: ")
            student = Student(name, age, course)
            file.write(f"{student.name}, {student.age}, {student.course}\n")
            print(f"Student {name} added successfully.")

# Function to update student information
def update_student_data():
    name_to_update = input("Enter the name of the student to update: ")
    updated = False
    with open(text_file, "r") as file:
        lines = file.readlines()

    with open(text_file, "w") as file:
        for line in lines:
            if line.startswith(name_to_update):
                new_age = int(input(f"Enter new age for {name_to_update}: "))
                new_course = input(f"Enter new course for {name_to_update}: ")
                file.write(f"{name_to_update}, {new_age}, {new_course}\n")
                updated = True
            else:
                file.write(line)

    if updated:
        print(f"Student {name_to_update} updated successfully.")
    else:
        print(f"Student {name_to_update} not found.")

# Function to search for a student by name
def search_student_by_name():
```

```
    search_name = input("Enter the name of the student to search for: ")
    found = False
    with open(text_file, "r") as file:
        for line in file:
            if line.startswith(search_name):
                print("Student found:", line.strip())
                found = True
                break
    if not found:
        print(f"Student {search_name} not found.")

# Function to backup student data using pickle
def backup_student_data():
    students = []
    with open(text_file, "r") as file:
        for line in file:
            name, age, course = line.strip().split(", ")
            student = Student(name, int(age), course)
            students.append(student)

    with open(pickle_file, "wb") as file:
        pickle.dump(students, file)
    print("Student data backed up successfully.")

# Function to restore student data from pickle file
def restore_student_data():
    try:
        with open(pickle_file, "rb") as file:
            students = pickle.load(file)
            print("\nRestored Student Data:")
            for student in students:
                print(student)
    except FileNotFoundError:
        print("No backup found!")
```

```
# Demonstrate seek() and tell() methods
def demonstrate_seek_tell():
    with open(text_file, "r") as file:
        print("\nDemonstrating seek() and tell():")
        file.seek(0)  # Go to the start of the file
        print(f"Current position after seek: {file.tell()}")

        # Read first 20 characters
        data = file.read(20)
        print(f"First 20 characters: {data}")
        print(f"Current position after read: {file.tell()}")

        # Move back to the start and read again
        file.seek(0)
        print(f"File position after seek to start: {file.tell()}")

# Main function to run the student management system
def main():
    # Check if the student file exists
    if not os.path.exists(text_file):
        print(f"{text_file} does not exist. Creating a new file.")
        open(text_file, "w").close()  # Create the file if it doesn't exist

    # Menu for student data management
    while True:
        print("\nStudent Data Management System")
        print("1. Store Student Data")
        print("2. Update Student Data")
        print("3. Search Student by Name")
        print("4. Backup Student Data")
        print("5. Restore Student Data from Backup")
        print("6. Demonstrate seek() and tell() Methods")
        print("7. Exit")
```

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

        if choice == 1:
            store_student_data()
        elif choice == 2:
            update_student_data()
        elif choice == 3:
            search_student_by_name()
        elif choice == 4:
            backup_student_data()
        elif choice == 5:
            restore_student_data()
        elif choice == 6:
            demonstrate_seek_tell()
        elif choice == 7:
            print("Exiting the program.")
            break
        else:
            print("Invalid choice! Please try again.")

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

Sample Output:

```
Student Data Management System
1. Store Student Data
2. Update Student Data
3. Search Student by Name
4. Backup Student Data
5. Restore Student Data from Backup
6. Demonstrate seek() and tell() Methods
7. Exit
Enter your choice: 1
Enter student name (or 'exit' to stop): John
```

Enter age for John: 20

Enter course for John: Computer Science

Student John added successfully.

Enter student name (or 'exit' to stop): exit

...