ADVANCED PYTHON PROGRAMMING LAB MANUAL

BCA – IV Semester
Department of BCA
R.V.S. College of Engineering & Technology, Jamshedpur
Session: 2025–2028

Certificate

This is to certify that Mr./Ms. _____, Roll No. _____, of BCA IV Semester has successfully completed the Advanced Python Programming Laboratory during the academic session 2025–2028 as per the prescribed syllabus.

FacultyIn-charge:_____

Head of Department: _____

Acknowledgement

This laboratory manual has been prepared to provide students with practical exposure to advanced Python programming concepts. The experiments are designed to strengthen programming skills, develop logical thinking, and familiarize students with modern Python libraries used in software development and data analysis.

Preface

Python is one of the most widely used programming languages because of its simplicity, readability, and extensive library support. This laboratory course complements the Advanced Python Programming theory course by providing hands-on experience in advanced data structures, modular programming, object-oriented programming, exception handling, file processing, data analysis, and visualization.

Course Objectives

1. Strengthen advanced Python programming skills.
2. Develop logical thinking and problem-solving ability.
3. Apply object-oriented programming concepts in Python.
4. Work with files, exceptions, and data processing libraries.
5. Develop confidence in building small Python applications.

Course Outcomes

CO1: Write Python programs using advanced data types and functions.
CO2: Develop modular programs using modules and packages.
CO3: Implement object-oriented programming concepts in Python.
CO4: Handle files, exceptions, and text processing effectively.
CO5: Use Python libraries for data analysis, visualization, and simple applications.

CO–PO Mapping

CO	PO1	PO2	PO3	PO4	PO5	PO10	PO12
CO1	3	2	-	-	3	-	2
CO2	3	2	3	-	3	-	2
CO3	3	3	3	2	2	-	2
CO4	2	3	2	3	3	-	2
CO5	2	2	3	3	3	2	3

Software Requirements

- Python 3.12 or above
- Visual Studio Code / PyCharm / IDLE
- NumPy
- Pandas
- Matplotlib

Hardware Requirements

- Intel Core i3 or higher
- Minimum 4 GB RAM (8 GB Recommended)
- 10 GB Free Disk Space

Python Installation

1. Download Python from https://www.python.org/downloads/
2. Run the installer.
3. Select 'Add Python to PATH'.
4. Click 'Install Now'.
5. Verify installation using:

```
python --version
```

Laboratory Rules

6. Attend the laboratory with Your Lab File.
7. Complete the assigned experiment during the lab session.
8. Maintain proper documentation of all programs.
9. Do not copy programs from other students.
10. Use meaningful filenames.
11. Submit the lab record on time.
12. Maintain discipline in the laboratory.

List of Experiments

Exp. No.	Experiment	CO
1	Advanced List Operations	CO1
2	Dictionary and Set Operations	CO1
3	List, Dictionary and Set Comprehensions	CO1
4	Lambda Functions	CO1
5	Default and Keyword Arguments	CO1
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		

Experiment No. 1

Title: Advanced List Operations in Python

Course Outcome: CO1

Aim

To understand and implement advanced list operations in Python including slicing, concatenation, nested lists, insertion, deletion, sorting, searching, and list methods.

Learning Outcome

Students will be able to manipulate Python lists efficiently using built-in methods and solve basic data processing problems.

Theory

A list is an ordered, mutable collection in Python that can store heterogeneous data. Lists support indexing, slicing, insertion, deletion, searching, sorting, reversing, and nesting. Common methods include append(), extend(), insert(), remove(), pop(), sort(), reverse(), count(), and index().

Algorithm

1. Start the program.
2. Create a list of numbers.
3. Display the original list.
4. Perform append, insert, extend, remove, pop, sort, and reverse operations.
5. Demonstrate slicing and searching.
6. Display the modified list.
7. Stop the program.

Python Program

```
# Advanced List Operations
numbers=[10,20,30,40,50]
print("Original List:",numbers)

numbers.append(60)
numbers.insert(2,25)
numbers.extend([70,80])
numbers.remove(40)
numbers.pop()

print("Modified List:",numbers)
print("First Three Elements:",numbers[:3])
print("Last Two Elements:",numbers[-2:])

numbers.sort()
print("Sorted List:",numbers)
```

```
numbers.reverse()
print("Reversed List:",numbers)

print("Index of 30:",numbers.index(30))
print("Count of 20:",numbers.count(20))
```

Sample Output

```
Original List: [10, 20, 30, 40, 50]
Modified List: [10, 20, 25, 30, 50, 60, 70]
First Three Elements: [10, 20, 25]
Last Two Elements: [60, 70]
Sorted List: [10, 20, 25, 30, 50, 60, 70]
Reversed List: [70, 60, 50, 30, 25, 20, 10]
Index of 30: 3
Count of 20: 1
```

Explanation

The program demonstrates the use of common list methods for adding, deleting, searching, sorting, reversing, and slicing elements. These operations are frequently used in data processing and application development.

Result

The advanced list operations were implemented successfully, and the expected output was obtained.

Applications

- Student record management
- Inventory management
- Data preprocessing
- Menu-driven applications
- Machine learning data preparation

Precautions

- Use valid list indices.
- Check element existence before remove().
- Handle empty lists before pop().
- Use sort() only for comparable data types.

Viva Questions

1. What is a list in Python?
2. Why are lists called mutable?
3. Differentiate append() and extend().
4. What is list slicing?
5. Difference between remove() and pop()?
6. What does sort() do?

7. What is reverse()?
8. What is a nested list?

Assignment Questions

1. Write a program to find the largest and smallest element in a list.
2. Remove duplicate elements from a list.
3. Merge two sorted lists.
4. Implement a menu-driven list application.

Experiment No. 2

Title: Dictionary and Set Operations in Python

Course Outcome: CO1

Aim

To implement various dictionary and set operations in Python and understand their applications in efficient data storage and manipulation.

Learning Outcome

Students will be able to create, update, search, delete, and iterate through dictionaries and sets using built-in methods.

Theory

A dictionary is a mutable collection of key-value pairs with unique keys. A set is an unordered collection of unique elements used for membership testing and mathematical set operations such as union, intersection, difference, and symmetric difference.

Algorithm

1. Start the program.
2. Create a dictionary containing student details.
3. Perform insertion, update, search, and deletion operations.
4. Create two sets.
5. Perform union, intersection, difference, and symmetric difference operations.
6. Display the results.
7. Stop the program.

Python Program

```
# Dictionary Operations
student = {"Name":"Rahul","Age":20,"Branch":"BCA"}

print("Original Dictionary:", student)

student["Age"] = 21
student["City"] = "Jamshedpur"

print("Updated Dictionary:", student)
print("Student Name:", student["Name"])

del student["Branch"]
print("After Deletion:", student)

# Set Operations
A = {1,2,3,4,5}
B = {4,5,6,7}
```

```
print("Set A:", A)
print("Set B:", B)
print("Union:", A | B)
print("Intersection:", A & B)
print("Difference:", A - B)
print("Symmetric Difference:", A ^ B)
```

Sample Output

```
Original Dictionary: {'Name': 'Rahul', 'Age': 20, 'Branch': 'BCA'}
Updated Dictionary: {'Name': 'Rahul', 'Age': 21, 'Branch': 'BCA', 'City': 'Jamshedpur'}
Student Name: Rahul
After Deletion: {'Name': 'Rahul', 'Age': 21, 'City': 'Jamshedpur'}

Set A: {1, 2, 3, 4, 5}
Set B: {4, 5, 6, 7}
Union: {1, 2, 3, 4, 5, 6, 7}
Intersection: {4, 5}
Difference: {1, 2, 3}
Symmetric Difference: {1, 2, 3, 6, 7}
```

Explanation

The program demonstrates CRUD operations on dictionaries and performs common mathematical operations on sets using Python operators.

Result

Dictionary and set operations were successfully implemented and verified.

Applications

- Student information systems
- Inventory management
- Duplicate data removal
- Membership testing
- Database indexing

Precautions

- Dictionary keys must be unique.
- Sets cannot contain duplicate values.
- Access dictionary keys safely using get() when appropriate.
- Use immutable objects as dictionary keys.

Viva Questions

1. What is a dictionary in Python?
2. How is a dictionary different from a list?
3. What are the characteristics of a set?

4. Can a set contain duplicate elements?
5. What is the difference between union and intersection?
6. What is symmetric difference?
7. What is the use of get() in dictionaries?
8. Why must dictionary keys be unique?

Assignment Questions

1. Write a program to count the frequency of words using a dictionary.
2. Write a program to merge two dictionaries.
3. Find common elements of two lists using sets.
4. Remove duplicate elements from a list using a set.

Experiment No. 3

Title: List, Dictionary and Set Comprehensions in Python

Course Outcome: CO1

Aim

To implement list, dictionary, and set comprehensions for creating collections efficiently using concise Python syntax.

Learning Outcome

Students will be able to generate and transform collections using comprehensions, leading to shorter, more readable, and efficient Python code.

Theory

Comprehensions provide a compact way to create lists, dictionaries, and sets. They combine iteration and conditional filtering into a single expression, making code more readable and often more efficient than traditional loops.

Algorithm

1. Start the program.
2. Create a list using list comprehension.
3. Create a dictionary using dictionary comprehension.
4. Create a set using set comprehension.
5. Display all generated collections.
6. Stop the program.

Python Program

```
# List Comprehension
numbers = [x*x for x in range(1,11)]
print("Squares:", numbers)

# List Comprehension with Condition
even = [x for x in range(1,21) if x % 2 == 0]
print("Even Numbers:", even)

# Dictionary Comprehension
square_dict = {x: x*x for x in range(1,6)}
print("Dictionary:", square_dict)

# Set Comprehension
square_set = {x*x for x in range(1,6)}
print("Set:", square_set)
```

Sample Output

Squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Even Numbers: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Dictionary: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
Set: {1, 4, 9, 16, 25}

Explanation

The program demonstrates how comprehensions can replace multiple lines of loop-based code. List comprehensions create lists, dictionary comprehensions generate key-value pairs, and set comprehensions produce unique elements.

Result

The program successfully demonstrated list, dictionary, and set comprehensions in Python.

Applications

- Data filtering and transformation
- Generating lookup tables
- Removing duplicate values
- Data preprocessing for machine learning
- Efficient collection creation

Precautions

- Avoid overly complex comprehensions that reduce readability.
- Use conditions carefully to filter data correctly.
- Prefer normal loops for very complex logic.
- Ensure dictionary keys remain unique.

Viva Questions

1. What is a list comprehension?
2. How is list comprehension different from a for loop?
3. What is dictionary comprehension?
4. What is set comprehension?
5. Can conditions be used in comprehensions?
6. What are the advantages of comprehensions?
7. When should comprehensions be avoided?
8. How do comprehensions improve performance?

Assignment Questions

1. Create a list of cubes from 1 to 20 using list comprehension.
2. Create a dictionary of numbers and their factorial values.
3. Create a set of vowels from a given string.
4. Filter all prime numbers between 1 and 100 using comprehension.

Experiment No. 4

Title: Lambda Functions in Python

Course Outcome: CO1

Aim

To understand and implement lambda (anonymous) functions in Python for writing concise and efficient programs.

Learning Outcome

Students will be able to create and use lambda functions with built-in functions such as map(), filter(), and sorted().

Theory

A lambda function is an anonymous function defined using the 'lambda' keyword. It can take any number of arguments but contains only one expression. Lambda functions are commonly used for short operations and as arguments to higher-order functions like map(), filter(), and sorted().

Algorithm

1. Start the program.
2. Define a lambda function for addition.
3. Use map() to square list elements.
4. Use filter() to obtain even numbers.
5. Sort a list of tuples using a lambda expression.
6. Display all results.
7. Stop the program.

Python Program

```
# Lambda Function
add = lambda a, b: a + b
print("Sum:", add(10, 20))

# map() with Lambda
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x*x, numbers))
print("Squares:", squares)

# filter() with Lambda
even = list(filter(lambda x: x % 2 == 0, numbers))
print("Even Numbers:", even)

# sorted() with Lambda
students = [("Rahul", 75), ("Anita", 90), ("Vikram", 82)]
sorted_students = sorted(students, key=lambda x: x[1])
```

```
print("Sorted by Marks:", sorted_students)
```

Sample Output

Sum: 30
Squares: [1, 4, 9, 16, 25]
Even Numbers: [2, 4]
Sorted by Marks: [('Rahul', 75), ('Vikram', 82), ('Anita', 90)]

Explanation

The program demonstrates anonymous functions using lambda expressions. It shows how lambda simplifies simple computations and works effectively with map(), filter(), and sorted().

Result

The program successfully demonstrated the use of lambda functions and their applications in Python.

Applications

- Data transformation
- Sorting custom objects
- Filtering datasets
- Functional programming
- Quick mathematical operations

Precautions

- Use lambda only for simple expressions.
- Prefer normal functions for complex logic.
- Ensure map() and filter() receive iterable objects.
- Keep lambda expressions readable.

Viva Questions

1. What is a lambda function?
2. Why are lambda functions called anonymous functions?
3. What is the syntax of a lambda function?
4. When should lambda functions be used?
5. What is the difference between lambda and def?
6. How does map() work with lambda?
7. How does filter() work with lambda?
8. Can a lambda function contain multiple expressions?

Assignment Questions

1. Write a lambda function to calculate the cube of a number.
2. Use map() to convert a list of temperatures from Celsius to Fahrenheit.
3. Use filter() to extract names longer than five characters.
4. Sort a list of dictionaries based on age using lambda.

Experiment No. 5

Title: Default and Keyword Arguments in Python

Course Outcome: CO1

Aim

To understand and implement default arguments and keyword arguments in Python functions.

Learning Outcome

Students will be able to write flexible and reusable functions using default parameter values and keyword arguments.

Theory

Default arguments provide predefined values to function parameters, while keyword arguments allow values to be passed by parameter name rather than position. These features improve code readability, flexibility, and reusability.

Algorithm

1. Start the program.
2. Create a function with default arguments.
3. Call the function with and without optional arguments.
4. Create another function using keyword arguments.
5. Display the results.
6. Stop the program.

Python Program

```
# Function with Default Arguments
def student(name, course="BCA"):
    print("Name :", name)
    print("Course :", course)

student("Rahul")
student("Anita", "MCA")

print()

# Function with Keyword Arguments
def employee(emp_id, name, department):
    print("Employee ID :", emp_id)
    print("Name :", name)
    print("Department :", department)

employee(emp_id=101, name="Deepak", department="Computer Science")
print()
```

employee(department="IT", name="Riya", emp_id=102)

Sample Output

Name : Rahul
Course : BCA

Name : Anita
Course : MCA

Employee ID : 101
Name : Deepak
Department : Computer Science

Employee ID : 102
Name : Riya
Department : IT

Explanation

The program demonstrates how default arguments supply predefined values when arguments are omitted and how keyword arguments improve readability by allowing arguments to be passed in any order.

Result

The program was executed successfully and demonstrated the use of default and keyword arguments.

Applications

- Reusable functions
- Menu-driven applications
- API development
- Code readability
- Software development

Precautions

- Place default parameters after required parameters.
- Avoid mutable objects as default values.
- Use correct parameter names in keyword arguments.
- Keep functions simple and reusable.

Viva Questions

1. What are default arguments?
2. What are keyword arguments?
3. What are the advantages of keyword arguments?
4. Can default values be overridden?
5. Why should default parameters come after required parameters?

6. Can keyword arguments be passed in any order?
7. What happens if a required argument is omitted?
8. Differentiate positional and keyword arguments.

Assignment Questions

1. Write a function to calculate simple interest using default values.
2. Create a function to display employee details using keyword arguments.
3. Write a function using positional, default and keyword arguments.
4. Develop a menu-driven calculator using functions.

Experiment No. 6

Title: Create a User-Defined Module and Import it into Another Program

Course Outcome: CO2

Aim

To create a user-defined Python module and import it into another program to promote modular and reusable programming.

Learning Outcome

Students will be able to create reusable modules, import them into Python programs, and organize code efficiently.

Theory

A module is a Python file containing functions, variables, and classes that can be reused in multiple programs. Using modules reduces code duplication, improves maintainability, and supports modular software development. Modules can be imported using the import statement or by importing specific functions.

Algorithm

1. Create a Python file named 'mymath.py'.
2. Define arithmetic functions inside the module.
3. Save the module.
4. Create another Python program.
5. Import the module.
6. Call the module functions and display the results.
7. Stop the program.

Module: mymath.py

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    return a / b
```

Main Program

```
import mymath

print("Addition =", mymath.add(20, 10))
```

```
print("Subtraction =", mymath.subtract(20, 10))
print("Multiplication =", mymath.multiply(20, 10))
print("Division =", mymath.divide(20, 10))
```

Sample Output

Addition = 30
Subtraction = 10
Multiplication = 200
Division = 2.0

Explanation

The arithmetic functions are stored in the user-defined module 'mymath.py'. The main program imports the module and invokes its functions, demonstrating code reuse and modular programming.

Result

The user-defined module was created successfully and imported into another Python program.

Applications

- Large-scale software projects
- Reusable utility libraries
- Enterprise applications
- Scientific computing
- Package development

Precautions

- Save the module in the same directory as the main program or configure the Python path.
- Use meaningful module names.
- Avoid naming modules the same as Python's built-in modules.
- Check for syntax errors before importing.

Viva Questions

1. What is a Python module?
2. What are the advantages of modules?
3. How do you import a module?
4. What is the difference between a module and a package?
5. What is a user-defined module?
6. Can one module import another module?
7. Where should a module be stored?
8. Why is modular programming important?

Assignment Questions

1. Create a module to perform area calculations for different shapes.
2. Create a module for temperature conversion.
3. Develop a module containing mathematical utility functions.
4. Write a program that imports only specific functions from a module.

Experiment No. 7

Title: Develop a Simple Package and Demonstrate its Usage

Course Outcome: CO2

Aim

To create a simple Python package containing multiple modules and demonstrate how to import and use them in an application.

Learning Outcome

Students will be able to organize Python programs into packages, improve code reusability, and manage larger projects effectively.

Theory

A package is a collection of related Python modules organized in a directory. A package usually contains an __init__.py file and one or more modules. Packages improve project organization, modularity, and maintainability.

Package Structure

```
calculator/
│
├── __init__.py
├── arithmetic.py
└── geometry.py
```

main.py

Algorithm

1. Create a package directory named 'calculator'.
2. Add an __init__.py file.
3. Create arithmetic.py and geometry.py modules.
4. Write functions inside both modules.
5. Create a main.py program.
6. Import the package modules and execute their functions.
7. Display the output.

Module: arithmetic.py

```
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

Module: geometry.py

```
def area_square(side):
    return side * side

def area_rectangle(length, breadth):
    return length * breadth
```

Main Program

```
from calculator import arithmetic, geometry

print("Addition =", arithmetic.add(10, 20))
print("Multiplication =", arithmetic.multiply(5, 6))
print("Area of Square =", geometry.area_square(4))
print("Area of Rectangle =", geometry.area_rectangle(8, 5))
```

Sample Output

Addition = 30
Multiplication = 30
Area of Square = 16
Area of Rectangle = 40

Explanation

The package contains two related modules. The main program imports these modules and calls their functions, demonstrating package creation and modular software development.

Result

The Python package was created successfully and its modules were imported and executed correctly.

Applications

- Enterprise software
- Reusable libraries
- Web development
- Data science projects
- Large Python applications

Precautions

- Keep the package directory structure organized.
- Include the __init__.py file.
- Avoid naming conflicts with built-in packages.
- Import only required modules when possible.

Viva Questions

1. What is a Python package?

2. What is the purpose of __init__.py?
3. How is a package different from a module?
4. Why are packages used?
5. How do you import a module from a package?
6. What are the advantages of packages?
7. Can a package contain sub-packages?
8. How do packages improve code organization?

Assignment Questions

1. Create a package for banking operations.
2. Develop a package containing trigonometric functions.
3. Create a package with separate modules for student and employee management.
4. Import only specific functions from a package and demonstrate their usage.

Experiment No. 8

Title: Write a Python Program to Create a Class and Object

Course Outcome: CO3

Aim

To understand the concepts of classes and objects by creating a user-defined class and instantiating objects in Python.

Learning Outcome

Students will be able to create classes, define attributes and methods, create objects, and access class members.

Theory

Object-Oriented Programming (OOP) is a programming paradigm based on objects. A class is a blueprint for creating objects, while an object is an instance of a class. Classes contain attributes (data members) and methods (member functions) that define the behavior of objects.

Algorithm

1. Start the program.
2. Define a class named Student.
3. Create a constructor to initialize student details.
4. Define a method to display student information.
5. Create an object of the Student class.
6. Call the display method using the object.
7. Stop the program.

Python Program

```
class Student:
    def __init__(self, name, roll, course):
        self.name = name
        self.roll = roll
        self.course = course

    def display(self):
        print("Student Name :", self.name)
        print("Roll Number  :", self.roll)
        print("Course       :", self.course)

# Creating Object
s1 = Student("Rahul", 101, "BCA")
s1.display()
```

Sample Output

Student Name : Rahul
Roll Number : 101
Course : BCA

Explanation

The Student class contains a constructor (__init__) to initialize object attributes and a display() method to print student details. An object named s1 is created, and the display() method is invoked using that object.

Result

The program successfully demonstrated the creation of a class, object, constructor, and member method in Python.

Applications

- Student Management System
- Library Management System
- Banking Applications
- Employee Management System
- Inventory Management Software

Precautions

- Follow proper indentation while defining classes and methods.
- Use meaningful class and object names.
- Initialize all required attributes inside the constructor.
- Access instance variables using the self keyword.

Viva Questions

1. What is Object-Oriented Programming?
2. What is a class?
3. What is an object?
4. What is a constructor?
5. What is the purpose of the self keyword?
6. How is an object created in Python?
7. Can multiple objects be created from the same class?
8. Differentiate between a class and an object.

Assignment Questions

1. Create a class Employee with attributes and methods.
2. Write a Python program to create multiple objects of a class.
3. Develop a class Book to store book details and display them.
4. Create a class Circle to calculate area and circumference.

Experiment No. 9

Title: Implement Constructor and Instance Variables in Python

Course Outcome: CO3

Aim

To understand the use of constructors and instance variables for initializing and managing object data in Python.

Learning Outcome

Students will be able to define constructors, initialize instance variables, and access them through object methods.

Theory

A constructor is a special method named __init__() that is automatically called when an object is created. It is used to initialize the instance variables of an object. Instance variables are unique to each object and store object-specific data.

Algorithm

1. Start the program.
2. Define a class Employee.
3. Create a constructor (__init__) to initialize employee details.
4. Store values in instance variables.
5. Create a method to display employee information.
6. Create multiple objects and invoke the display method.
7. Stop the program.

Python Program

```
class Employee:
    def __init__(self, emp_id, name, department, salary):
        self.emp_id = emp_id
        self.name = name
        self.department = department
        self.salary = salary

    def display(self):
        print("Employee ID :", self.emp_id)
        print("Name       :", self.name)
        print("Department  :", self.department)
        print("Salary      :", self.salary)
        print("-" * 30)

emp1 = Employee(101, "Rahul", "IT", 45000)
emp2 = Employee(102, "Anita", "HR", 40000)
```

```
emp1.display()
emp2.display()
```

Sample Output

```
Employee ID : 101
Name        : Rahul
Department  : IT
Salary      : 45000
------------------------------

Employee ID : 102
Name        : Anita
Department  : HR
Salary      : 40000
------------------------------
```

Explanation

The constructor initializes the instance variables when an object is created. Each Employee object stores its own data, and the display() method prints the details.

Result

The program successfully demonstrated the implementation of constructors and instance variables in Python.

Applications

- Employee Management Systems
- Student Information Systems
- Banking Applications
- Hospital Management Systems
- Inventory and Billing Software

Precautions

- Always define the constructor with the self parameter.
- Initialize all required instance variables inside the constructor.
- Access instance variables using self.
- Use meaningful variable names for better readability.

Viva Questions

1. What is a constructor in Python?
2. When is the __init__() method executed?
3. What are instance variables?
4. What is the purpose of the self keyword?
5. Can a class have multiple constructors in Python?
6. How are instance variables different from class variables?
7. Can constructors accept parameters?

8. Why are constructors important in object-oriented programming?

Assignment Questions

1. Create a Student class with a constructor to initialize student details.
2. Write a program to manage book information using constructors.
3. Develop a BankAccount class with account details initialized through a constructor.
4. Create a Product class to display product information using instance variables.

Experiment No. 10

Title: Write a Python Program to Demonstrate Inheritance

Course Outcome: CO3

Aim

To understand and implement inheritance in Python for code reusability and hierarchical class design.

Learning Outcome

Students will be able to create base and derived classes, inherit properties and methods, and demonstrate code reuse.

Theory

Inheritance is an Object-Oriented Programming concept that allows a new class (derived/child class) to acquire the properties and methods of an existing class (base/parent class). It promotes code reusability, extensibility, and hierarchical relationships among classes.

Algorithm

1. Start the program.
2. Create a base class Person.
3. Define attributes and a display() method in the base class.
4. Create a derived class Student inheriting from Person.
5. Add additional attributes and methods in Student.
6. Create an object of the derived class and call inherited methods.
7. Stop the program.

Python Program

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

    def display_person(self):
        print("Name :", self.name)
        print("Age  :", self.age)

class Student(Person):
    def __init__(self, name, age, course):
        super().__init__(name, age)
        self.course = course

    def display_student(self):
        self.display_person()
        print("Course :", self.course)
```

```
s = Student("Rahul", 20, "BCA")
s.display_student()
```

Sample Output

```
Name : Rahul
Age  : 20
Course : BCA
```

Explanation

The Student class inherits the attributes and methods of the Person class using inheritance. The super() function is used to invoke the constructor of the parent class. The derived class extends the functionality by adding the course attribute.

Result

The program successfully demonstrated inheritance and code reusability in Python.

Applications

- Student Management Systems
- Employee Management Systems
- Banking Applications
- Hospital Information Systems
- Enterprise Software Development

Precautions

- Call the parent constructor using super() when required.
- Maintain proper indentation.
- Avoid unnecessary inheritance hierarchies.
- Use meaningful class names.

Viva Questions

1. What is inheritance?
2. What are the advantages of inheritance?
3. Differentiate parent and child classes.
4. What is the purpose of super()?
5. Can Python support multiple inheritance?
6. What is code reusability?
7. What are the different types of inheritance?
8. How does inheritance support OOP?

Assignment Questions

1. Create a Vehicle class and derive Car from it.
2. Implement multilevel inheritance using Animal, Mammal, and Dog classes.
3. Write a program demonstrating multiple inheritance.
4. Create an Employee class and derive Manager with additional features.

Experiment No. 11

Title: Write a Python Program to Demonstrate Method Overriding

Course Outcome: CO3

Aim

To understand and implement method overriding in Python using inheritance.

Learning Outcome

Students will be able to redefine inherited methods in a derived class and understand runtime polymorphism.

Theory

Method overriding occurs when a derived class provides its own implementation of a method already defined in the base class. The overridden method in the child class is executed when called through an object of the child class, enabling runtime polymorphism.

Algorithm

1. Start the program.
2. Create a base class Animal with a sound() method.
3. Create derived classes Dog and Cat.
4. Override the sound() method in each derived class.
5. Create objects of the derived classes.
6. Call the overridden methods.
7. Stop the program.

Python Program

```
class Animal:
    def sound(self):
        print("Animals make different sounds.")

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

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

d = Dog()
c = Cat()

d.sound()
c.sound()
```

Sample Output

Dog barks.
Cat meows.

Explanation

The Dog and Cat classes inherit from Animal and override the sound() method. When sound() is invoked using Dog and Cat objects, the child class implementations are executed instead of the parent implementation.

Result

The program successfully demonstrated method overriding and runtime polymorphism in Python.

Applications

- GUI application development
- Game development
- Framework and library design
- Enterprise software systems
- Object-oriented application development

Precautions

- Ensure the method name matches the parent method exactly.
- Use inheritance correctly before overriding methods.
- Maintain proper indentation.
- Override methods only when behavior needs to change.

Viva Questions

1. What is method overriding?
2. How is method overriding different from method overloading?
3. Why is method overriding used?
4. What is runtime polymorphism?
5. Can a child class access the parent method?
6. What is the role of inheritance in method overriding?
7. What happens if a method is not overridden?
8. Can super() be used with overridden methods?

Assignment Questions

1. Create a Shape class and override area() in Circle and Rectangle classes.
2. Implement method overriding for Vehicle and Bike classes.
3. Create an Employee class and override display() in Manager.
4. Use super() to call the parent method before executing the child method.

Experiment No. 12

Title: Implement Polymorphism Using Python Classes

Course Outcome: CO3

Aim

To understand and implement polymorphism in Python using classes and methods.

Learning Outcome

Students will be able to apply runtime polymorphism by invoking the same method on different class objects.

Theory

Polymorphism is one of the core principles of Object-Oriented Programming (OOP). It allows objects of different classes to be treated through a common interface. In Python, runtime polymorphism is achieved through method overriding, where different classes implement the same method in their own way.

Algorithm

1. Start the program.
2. Create a base class Shape with an area() method.
3. Create derived classes Rectangle and Circle.
4. Override the area() method in each derived class.
5. Create objects of the derived classes.
6. Call the area() method using different objects.
7. Display the results.
8. Stop the program.

Python Program

```
import math

class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    def area(self):
        return self.length * self.breadth

class Circle(Shape):
    def __init__(self, radius):
```

```
        self.radius = radius

    def area(self):
        return math.pi * self.radius * self.radius

shapes = [
    Rectangle(10, 5),
    Circle(7)
]

for shape in shapes:
    print("Area =", round(shape.area(), 2))
```

Sample Output

Area = 50
Area = 153.94

Explanation

The Rectangle and Circle classes override the area() method of the Shape class. Although the same method name is called, each object executes its own implementation. This demonstrates runtime polymorphism.

Result

The program successfully demonstrated polymorphism using Python classes.

Applications

- Graphics and drawing applications
- Game development
- Scientific and engineering software
- Framework and library development
- Enterprise object-oriented applications

Precautions

- Ensure methods have the same name in all related classes.
- Use inheritance where appropriate.
- Override methods carefully to maintain functionality.
- Test each implementation independently.

Viva Questions

1. What is polymorphism?
2. What are the different types of polymorphism?
3. How is runtime polymorphism implemented in Python?
4. How is polymorphism related to inheritance?
5. What is the advantage of polymorphism?
6. Can polymorphism exist without inheritance?

7. Differentiate between method overriding and polymorphism.
8. Give a real-world example of polymorphism.

Assignment Questions

1. Create a class hierarchy for different vehicles and demonstrate polymorphism.
2. Implement polymorphism using different payment methods.
3. Develop a program using Animal, Dog, and Cat classes with a common sound() method.
4. Create a banking application demonstrating polymorphism for different account types.

Experiment No. 13

Title: Demonstrate Encapsulation Using Access Methods

Course Outcome: CO3

Aim

To understand and implement encapsulation in Python using private data members and access methods.

Learning Outcome

Students will be able to protect object data using encapsulation and access or modify it through getter and setter methods.

Theory

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It combines data and methods into a single unit (class) and restricts direct access to data. In Python, encapsulation is achieved using private members (prefixed with double underscores) and public getter and setter methods.

Algorithm

1. Start the program.
2. Create a class BankAccount.
3. Declare a private data member for account balance.
4. Create getter and setter methods.
5. Create an object of the class.
6. Access and update the balance using access methods.
7. Display the updated balance.
8. Stop the program.

Python Program

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid Balance!")

account = BankAccount(5000)
```

```
print("Current Balance:", account.get_balance())

account.set_balance(7500)

print("Updated Balance:", account.get_balance())
```

Sample Output
Current Balance: 5000
Updated Balance: 7500

Explanation
The BankAccount class hides the balance variable by declaring it as private using double underscores (__balance). The get_balance() and set_balance() methods provide controlled access to the private data, ensuring data security and validation.

Result
The program successfully demonstrated encapsulation using private variables and access methods.

Applications
- Banking applications
- Student information systems
- Hospital management systems
- Payroll management software
- Secure enterprise applications

Precautions
- Declare sensitive data as private.
- Use getter and setter methods for controlled access.
- Validate input values inside setter methods.
- Avoid exposing private variables directly.

Viva Questions
1. What is encapsulation?
2. Why is encapsulation important in OOP?
3. How are private variables declared in Python?
4. What is the purpose of getter and setter methods?
5. Can private variables be accessed directly?
6. What is name mangling in Python?
7. How does encapsulation improve security?
8. Differentiate between encapsulation and abstraction.

Assignment Questions
1. Create a Student class using private variables and getter/setter methods.
2. Develop an Employee class with encapsulated salary details.

3. Implement encapsulation in a Library Management System.
4. Create a Product class with validation for product price.

Experiment No. 14

Title: Implement Exception Handling Using try, except, else, and finally

Course Outcome: CO4

Aim

To understand and implement exception handling in Python using try, except, else, and finally blocks.

Learning Outcome

Students will be able to handle runtime errors gracefully and write robust Python programs.

Theory

Exception handling is a mechanism for detecting and handling runtime errors without terminating the program unexpectedly. The try block contains code that may raise an exception. The except block handles the exception, the else block executes when no exception occurs, and the finally block executes regardless of whether an exception occurs.

Algorithm

1. Start the program.
2. Read two numbers from the user.
3. Place the division operation inside the try block.
4. Handle division by zero using the except block.
5. Display the result using the else block if no exception occurs.
6. Execute the finally block.
7. Stop the program.

Python Program

```
try:
    a = int(input("Enter first number: "))
    b = int(input("Enter second number: "))
    result = a / b

except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")

except ValueError:
    print("Error: Please enter valid integers.")

else:
    print("Result =", result)

finally:
    print("Program execution completed.")
```

Sample Output

```
Enter first number: 20
Enter second number: 5
Result = 4.0
Program execution completed.
```

Explanation

The try block performs the division operation. If the user enters zero as the divisor, a ZeroDivisionError is caught. Invalid input is handled using ValueError. The else block executes only when no exception occurs, and the finally block executes in every case.

Result

The program successfully demonstrated exception handling using try, except, else, and finally blocks.

Applications

- Banking and financial applications
- File processing systems
- Database applications
- Web applications
- Enterprise software development

Precautions

- Handle only expected exceptions.
- Avoid using a generic except block unless necessary.
- Use finally for cleanup operations such as closing files.
- Provide meaningful error messages.

Viva Questions

1. What is an exception?
2. What is the purpose of the try block?
3. What is the difference between except and finally?
4. When is the else block executed?
5. What is ZeroDivisionError?
6. What is ValueError?
7. Why is exception handling important?
8. Can multiple except blocks be used in a single try statement?

Assignment Questions

1. Write a program to handle FileNotFoundError.
2. Create a menu-driven calculator using exception handling.
3. Implement a program that validates user input using exceptions.
4. Demonstrate multiple exception handling in a single program.

Experiment No. 15

Title: Perform File Handling Operations (Read, Write, and Append) in Python

Course Outcome: CO4

Aim

To understand file handling operations in Python by creating, writing, reading, and appending data to text files.

Learning Outcome

Students will be able to perform basic file operations using Python and manage persistent data efficiently.

Theory

File handling allows programs to store and retrieve data permanently. Python provides the open() function to work with files in different modes such as read (r), write (w), append (a), and read/write (r+). The with statement is recommended because it automatically closes the file after use.

Algorithm

1. Start the program.
2. Open a text file in write mode and write sample data.
3. Open the same file in append mode and add more data.
4. Open the file in read mode.
5. Display the contents of the file.
6. Close the file automatically using the with statement.
7. Stop the program.

Python Program

```
# Writing to a file
with open("student.txt", "w") as file:
    file.write("Name: Rahul\n")
    file.write("Course: BCA\n")

# Appending data
with open("student.txt", "a") as file:
    file.write("Semester: 4\n")

# Reading data
with open("student.txt", "r") as file:
    content = file.read()

print("File Contents:")
print(content)
```

Sample Output

File Contents:
Name: Rahul
Course: BCA
Semester: 4

Explanation

The program creates a text file, writes initial data, appends additional information, and finally reads and displays the complete file contents. The with statement ensures that the file is properly closed.

Result

The program successfully demonstrated file creation, writing, appending, and reading operations in Python.

Applications

- Student record management
- Log file generation
- Configuration file handling
- Report generation
- Data storage for desktop applications

Precautions

- Always close files or use the with statement.
- Use the correct file mode (r, w, a).
- Handle file-related exceptions when necessary.
- Verify the file path before accessing a file.

Viva Questions

1. What is file handling?
2. What is the difference between write and append mode?
3. Why is the with statement preferred?
4. What does read() do?
5. What are common file modes in Python?
6. How do you check if a file exists?
7. What exception is raised when a file is not found?
8. Why is persistent storage important?

Assignment Questions

1. Write a program to copy the contents of one file to another.
2. Count the number of lines, words, and characters in a text file.
3. Create a student record file and search for a specific student.
4. Handle FileNotFoundError while reading a file.

Experiment No. 16

Title: Handle CSV Files Using the pandas Library

Course Outcome: CO4

Aim

To learn how to create, read, analyze, and save CSV files using the pandas library.

Learning Outcome

Students will be able to manipulate tabular data using pandas DataFrames and perform basic data analysis.

Theory

Pandas is a powerful Python library for data manipulation and analysis. A DataFrame is a two-dimensional data structure that stores data in rows and columns. CSV (Comma-Separated Values) files are widely used for storing and exchanging tabular data.

Algorithm

1. Import the pandas library.
2. Create a dictionary containing student records.
3. Convert the dictionary into a DataFrame.
4. Save the DataFrame as a CSV file.
5. Read the CSV file.
6. Display the contents of the DataFrame.
7. Stop the program.

Python Program

```
import pandas as pd

data = {
   "Roll No": [101, 102, 103],
   "Name": ["Rahul", "Anita", "Riya"],
   "Marks": [85, 90, 88]
}

df = pd.DataFrame(data)

df.to_csv("students.csv", index=False)

new_df = pd.read_csv("students.csv")

print(new_df)
```

Sample Output

	Roll No	Name	Marks
0	101	Rahul	85
1	102	Anita	90
2	103	Riya	88

Explanation

The program creates a DataFrame from a dictionary, stores it in a CSV file using to_csv(), and reads the file back using read_csv(). The DataFrame is then displayed.

Result

The program successfully demonstrated reading and writing CSV files using the pandas library.

Applications

- Data analysis
- Business reporting
- Machine learning data preprocessing
- Student and employee record management
- Data migration between applications

Precautions

- Install pandas before running the program.
- Verify the CSV file path.
- Use index=False when row numbers are not required.
- Handle missing values appropriately.

Viva Questions

1. What is pandas?
2. What is a DataFrame?
3. What is a CSV file?
4. What is the purpose of read_csv()?
5. What does to_csv() do?
6. Why is index=False used?
7. How do you display the first five rows of a DataFrame?
8. What are the advantages of pandas?

Assignment Questions

1. Create a CSV file containing employee records and display it using pandas.
2. Find the average marks of students from a CSV file.
3. Filter students with marks greater than 85.
4. Add a new column named Grade and save the updated CSV file.

Experiment No. 17

Title: Data Visualization Using Matplotlib

Course Outcome: CO5

Aim

To create and visualize data using different types of charts with the Matplotlib library.

Learning Outcome

Students will be able to create basic graphs such as line charts, bar charts, and pie charts for data visualization.

Theory

Matplotlib is a popular Python library used for creating static, animated, and interactive visualizations. It helps present numerical data graphically, making trends, comparisons, and patterns easier to understand.

Algorithm

1. Import the matplotlib.pyplot module.
2. Create sample data for months and sales.
3. Plot a line graph using the plot() function.
4. Add title and axis labels.
5. Display the graph using show().
6. Stop the program.

Python Program

```
import matplotlib.pyplot as plt

months = ["Jan", "Feb", "Mar", "Apr", "May"]
sales = [120, 150, 180, 170, 210]

plt.plot(months, sales, marker="o")
plt.title("Monthly Sales")
plt.xlabel("Month")
plt.ylabel("Sales")
plt.grid(True)

plt.show()
```

Sample Output

A line graph is displayed showing monthly sales from January to May with appropriate titles and axis labels.

Explanation

The plot() function draws a line graph using the supplied data. The title(), xlabel(), and ylabel() functions improve readability, while grid(True) adds grid lines.

Result

The program successfully demonstrated data visualization using a line chart in Matplotlib.

Applications

- Business analytics
- Scientific research
- Financial reporting
- Machine learning visualization
- Academic data analysis

Precautions

- Install Matplotlib before executing the program.
- Ensure x-axis and y-axis data have the same length.
- Use meaningful chart titles and labels.
- Select the appropriate chart type for the dataset.

Viva Questions

1. What is Matplotlib?
2. What is the purpose of pyplot?
3. What does plt.plot() do?
4. How do you display a graph in Python?
5. Name different chart types available in Matplotlib.
6. What is the purpose of plt.grid()?
7. Why are axis labels important?
8. What is data visualization?

Assignment Questions

1. Create a bar chart showing the marks of five students.
2. Create a pie chart representing departmental student strength.
3. Plot the monthly temperature of your city using a line graph.
4. Compare two products' sales using a multiple-line chart.

Experiment No. 18

Title: Data Analysis and Visualization Using pandas and Matplotlib

Course Outcome: CO5

Aim

To perform basic data analysis using pandas and visualize the analyzed data using Matplotlib.

Learning Outcome

Students will be able to load datasets, perform simple statistical analysis, and create charts for effective data interpretation.

Theory

Pandas provides powerful tools for reading, organizing, filtering, and analyzing data, while Matplotlib is used to present the analyzed data graphically. Combining these libraries enables efficient exploratory data analysis and visualization.

Algorithm

1. Import pandas and matplotlib.pyplot.
2. Create a DataFrame containing student marks.
3. Display the DataFrame.
4. Calculate the average marks.
5. Plot a bar chart of student marks.
6. Display the graph.
7. Stop the program.

Python Program

```
import pandas as pd
import matplotlib.pyplot as plt

data = {
   "Student": ["Amit", "Riya", "Rahul", "Neha", "Karan"],
   "Marks": [78, 92, 85, 88, 74]
}

df = pd.DataFrame(data)

print(df)
print("Average Marks:", df["Marks"].mean())

plt.bar(df["Student"], df["Marks"])
plt.title("Student Marks")
plt.xlabel("Students")
plt.ylabel("Marks")
```

```
plt.show()
```

Sample Output

```
  Student  Marks
0    Amit     78
1    Riya     92
2   Rahul     85
3    Neha     88
4   Karan     74

Average Marks: 83.4
```

A bar chart displaying the marks of all students is generated.

Explanation

The program creates a pandas DataFrame, computes the average marks using the mean() function, and visualizes the marks using a bar chart in Matplotlib.

Result

The program successfully demonstrated basic data analysis and visualization using pandas and Matplotlib.

Applications

- Educational analytics
- Business intelligence dashboards
- Research data analysis
- Performance reporting
- Machine learning data exploration

Precautions

- Install pandas and Matplotlib before execution.
- Ensure DataFrame columns are referenced correctly.
- Verify that chart labels match the dataset.
- Handle missing values before analysis.

Viva Questions

1. What is exploratory data analysis (EDA)?
2. What is a DataFrame in pandas?
3. What is the purpose of mean()?
4. Why are charts used in data analysis?
5. Differentiate between a line chart and a bar chart.
6. How do pandas and Matplotlib work together?
7. What are the advantages of data visualization?
8. Name some real-world applications of pandas.

Assignment Questions

1. Read student data from a CSV file and calculate the highest and lowest marks.
2. Create a pie chart showing grade distribution.
3. Filter students scoring more than 80 marks and display the results.
4. Create a line chart comparing marks in two different subjects.

Mini Project

Title: Student Record Management System Using Python

Course Outcome: CO1, CO2, CO3, CO4, CO5

Aim

To develop a menu-driven Student Record Management System integrating Python programming concepts such as functions, modules, file handling, exception handling, object-oriented programming, and data processing.

Learning Outcome

Students will be able to design, develop, test, and document a complete Python application by integrating concepts learned throughout the course.

Problem Statement

Develop a Student Record Management System that allows users to add, search, update, delete, and display student records. The records should be stored permanently using CSV files.

Software Requirements

- Python 3.10 or above
- Visual Studio Code / PyCharm
- pandas Library
- Operating System: Windows/Linux/macOS

Functional Requirements

- Add a new student record
- Display all student records
- Search a student by Roll Number
- Update student details
- Delete a student record
- Store records in a CSV file
- Handle invalid inputs using exception handling
- Generate basic statistics (average marks, highest marks)

Algorithm

1. Start the program.
2. Display the main menu.
3. Accept the user's choice.
4. Perform the selected operation.
5. Store updated records in a CSV file.
6. Repeat until the user exits.
7. Stop the program.

Suggested Python Modules

csv, pandas, os, datetime (optional), matplotlib (optional for graphical reports).

Expected Output

A menu-driven application that successfully manages student records with permanent storage and user-friendly interaction.

Evaluation Rubric

Criteria	Marks
Program Functionality	30
Code Quality & Documentation	20
Object-Oriented Design	15
File Handling	15
Exception Handling	10
User Interface & Output	10

Viva Questions

1. Which Python concepts were used in your project?

2. Why did you choose CSV for data storage?

3. How is exception handling implemented?

4. What improvements can be made to this project?

5. How can this project be converted into a GUI application?

Submission Guidelines

- Submit the complete source code.
- Include sample CSV data files.
- Prepare a project report with screenshots.
- Demonstrate the project during the viva.
- Upload the project to GitHub with a README file.

Appendix

Python Coding Standards

- Use meaningful variable and function names.
- Follow PEP 8 coding style guidelines.
- Write comments for complex logic.
- Use proper indentation (4 spaces).
- Keep functions small and reusable.

Common Python Libraries

- math – Mathematical operations
- os – Operating system utilities
- sys – System-specific parameters
- csv – CSV file processing
- pandas – Data analysis
- matplotlib.pyplot – Data visualization

Common Python Exceptions

- SyntaxError – Invalid syntax
- NameError – Undefined variable
- TypeError – Invalid operation on a data type
- ValueError – Invalid value
- IndexError – Invalid list index
- KeyError – Missing dictionary key
- ZeroDivisionError – Division by zero
- FileNotFoundError – File does not exist

Useful Keyboard Shortcuts (VS Code)

- Ctrl + S : Save file
- Ctrl + / : Toggle comment
- Ctrl + Shift + P : Command Palette
- F5 : Run with debugger
- Ctrl + ` : Open integrated terminal

Laboratory Assessment Rubric

- Attendance and Preparation – 10 Marks
- Program Development – 30 Marks
- Output and Correctness – 25 Marks
- Viva Voce – 20 Marks
- Record Book – 15 Marks

Recommended References

- Learning Python by Mark Lutz
- Python Crash Course by Eric Matthes

- Think Python by Allen B. Downey
- Python Documentation: https://docs.python.org/3/
- pandas Documentation: https://pandas.pydata.org/
- Matplotlib Documentation: https://matplotlib.org/

Learning Outcomes Achieved

After completing all experiments and the mini project, students should be able to:

- Develop modular Python applications.
- Apply object-oriented programming concepts.
- Perform file handling and exception handling.
- Analyze data using pandas.
- Visualize data using Matplotlib.
- Design and implement complete Python-based projects.

Conclusion

This laboratory manual provides hands-on experience in advanced Python programming. The experiments progress from core language features to object-oriented programming, file handling, data analysis, visualization, and a comprehensive mini project, preparing students for academic, research, and industry applications.