IT_LAB_3

October 17, 2025

Name: Dwaipayan Datta
Roll No: 10200124065
Department: CSE
Semester: 3rd
Subject: IT Workshop
Subject Code: PCC CS 393
Lab Assignment: 3

1 Question 1: Dictionary Operations

This program demonstrates various dictionary operations including creation, insertion, deletion, updating, and accessing elements.

[15]:
```python
# Dictionary Operations
student = {
    "name": "Alice",
    "roll": 101,
    "grade": "A",
    "subjects": ["Math", "Physics", "Chemistry"]
}

print("Original Dictionary:")
print(student)

# Accessing elements
print("\nAccessing 'name':", student["name"])
print("Using get() method:", student.get("grade"))

# Adding new key-value pair
student["age"] = 20
print("\nAfter adding 'age':")
print(student)

# Updating existing value
student["grade"] = "A+"
print("\nAfter updating 'grade':")
print(student)
```

```python
# Removing element using pop()
removed = student.pop("subjects")
print("\nRemoved 'subjects':", removed)
print("Dictionary after pop():")
print(student)

# Dictionary methods
print("\nKeys:", list(student.keys()))
print("Values:", list(student.values()))
print("Items:", list(student.items()))

# Check if key exists
print("\nIs 'name' in dictionary?", "name" in student)

# Clear dictionary
student_copy = student.copy()
student_copy.clear()
print("\nCleared dictionary:", student_copy)
```

```
Original Dictionary:
{'name': 'Alice', 'roll': 101, 'grade': 'A', 'subjects': ['Math', 'Physics',
'Chemistry']}

Accessing 'name': Alice
Using get() method: A

After adding 'age':
{'name': 'Alice', 'roll': 101, 'grade': 'A', 'subjects': ['Math', 'Physics',
'Chemistry'], 'age': 20}

After updating 'grade':
{'name': 'Alice', 'roll': 101, 'grade': 'A+', 'subjects': ['Math', 'Physics',
'Chemistry'], 'age': 20}

Removed 'subjects': ['Math', 'Physics', 'Chemistry']
Dictionary after pop():
{'name': 'Alice', 'roll': 101, 'grade': 'A+', 'age': 20}

Keys: ['name', 'roll', 'grade', 'age']
Values: ['Alice', 101, 'A+', 20]
Items: [('name', 'Alice'), ('roll', 101), ('grade', 'A+'), ('age', 20)]

Is 'name' in dictionary? True

Cleared dictionary: {}
```

2 Question 2: Linear and Binary Search

This program implements both linear search (for unsorted arrays) and binary search (for sorted arrays).

[16]:

```python
# Linear Search
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

# Binary Search (requires sorted array)
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

# Test cases
unsorted_list = [64, 34, 25, 12, 22, 11, 90]
sorted_list = [11, 12, 22, 25, 34, 64, 90]
target = 25

print("Array for Linear Search:", unsorted_list)
result = linear_search(unsorted_list, target)
if result != -1:
    print(f"Linear Search: Element {target} found at index {result}")
else:
    print(f"Linear Search: Element {target} not found")

print("\nArray for Binary Search:", sorted_list)
result = binary_search(sorted_list, target)
if result != -1:
    print(f"Binary Search: Element {target} found at index {result}")
else:
    print(f"Binary Search: Element {target} not found")
```

```
Array for Linear Search: [64, 34, 25, 12, 22, 11, 90]
Linear Search: Element 25 found at index 2

Array for Binary Search: [11, 12, 22, 25, 34, 64, 90]
Binary Search: Element 25 found at index 3
```

3 Question 3: Stack and Queue Operations

This program implements stack (LIFO) and queue (FIFO) data structures using Python lists.

[17]:
```python
# Stack Implementation (LIFO - Last In First Out)
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"Pushed {item}")

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return "Stack is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return "Stack is empty"

    def is_empty(self):
        return len(self.items) == 0

    def display(self):
        print("Stack:", self.items)

# Queue Implementation (FIFO - First In First Out)
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)
        print(f"Enqueued {item}")

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
```

```python
            return "Queue is empty"

    def front(self):
        if not self.is_empty():
            return self.items[0]
        return "Queue is empty"

    def is_empty(self):
        return len(self.items) == 0

    def display(self):
        print("Queue:", self.items)

# Stack operations
print("=== Stack Operations ===")
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
stack.display()
print(f"Popped: {stack.pop()}")
print(f"Top element: {stack.peek()}")
stack.display()

print("\n=== Queue Operations ===")
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)
queue.display()
print(f"Dequeued: {queue.dequeue()}")
print(f"Front element: {queue.front()}")
queue.display()
```

```
=== Stack Operations ===
Pushed 10
Pushed 20
Pushed 30
Stack: [10, 20, 30]
Popped: 30
Top element: 20
Stack: [10, 20]

=== Queue Operations ===
Enqueued 10
Enqueued 20
Enqueued 30
Queue: [10, 20, 30]
```

```
Dequeued: 10
Front element: 20
Queue: [20, 30]
```

4 Question 4: Ascending and Descending Sort

This program demonstrates sorting arrays in both ascending and descending order using built-in methods.

[18]:
```python
# Sorting in Ascending and Descending Order
numbers = [64, 34, 25, 12, 22, 11, 90, 88]

print("Original Array:", numbers)

# Ascending order using sort()
ascending = numbers.copy()
ascending.sort()
print("\nAscending Order (using sort()):", ascending)

# Ascending order using sorted()
ascending2 = sorted(numbers)
print("Ascending Order (using sorted()):", ascending2)

# Descending order using sort()
descending = numbers.copy()
descending.sort(reverse=True)
print("\nDescending Order (using sort()):", descending)

# Descending order using sorted()
descending2 = sorted(numbers, reverse=True)
print("Descending Order (using sorted()):", descending2)

# Manual bubble sort for ascending order
def bubble_sort_ascending(arr):
    n = len(arr)
    arr_copy = arr.copy()
    for i in range(n):
        for j in range(0, n-i-1):
            if arr_copy[j] > arr_copy[j+1]:
                arr_copy[j], arr_copy[j+1] = arr_copy[j+1], arr_copy[j]
    return arr_copy

print("\nManual Bubble Sort (Ascending):", bubble_sort_ascending(numbers))
```

```
Original Array: [64, 34, 25, 12, 22, 11, 90, 88]

Ascending Order (using sort()): [11, 12, 22, 25, 34, 64, 88, 90]
Ascending Order (using sorted()): [11, 12, 22, 25, 34, 64, 88, 90]
```

```
Descending Order (using sort()): [90, 88, 64, 34, 25, 22, 12, 11]
Descending Order (using sorted()): [90, 88, 64, 34, 25, 22, 12, 11]

Manual Bubble Sort (Ascending): [11, 12, 22, 25, 34, 64, 88, 90]
```

5 Question 5: Merge Sort Implementation

This program implements the merge sort algorithm, which uses divide-and-conquer approach to sort arrays.

[19]:
```python
# Merge Sort Implementation
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursive calls
        merge_sort(left_half)
        merge_sort(right_half)

        # Merge the sorted halves
        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Copy remaining elements
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

    return arr
```

```python
# Test merge sort
arr = [38, 27, 43, 3, 9, 82, 10]
print("Original Array:", arr)

sorted_arr = arr.copy()
merge_sort(sorted_arr)
print("Sorted Array (Merge Sort):", sorted_arr)

# Step-by-step visualization
def merge_sort_verbose(arr, depth=0):
    indent = "  " * depth
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        print(f"{indent}Splitting: {arr} into {left_half} and {right_half}")

        merge_sort_verbose(left_half, depth + 1)
        merge_sort_verbose(right_half, depth + 1)

        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

        print(f"{indent}Merged: {arr}")

print("\nStep-by-step Merge Sort:")
arr2 = [38, 27, 43, 3]
merge_sort_verbose(arr2)
```

```
Original Array: [38, 27, 43, 3, 9, 82, 10]
Sorted Array (Merge Sort): [3, 9, 10, 27, 38, 43, 82]

Step-by-step Merge Sort:
Splitting: [38, 27, 43, 3] into [38, 27] and [43, 3]
  Splitting: [38, 27] into [38] and [27]
  Merged: [27, 38]
  Splitting: [43, 3] into [43] and [3]
  Merged: [3, 43]
Merged: [3, 27, 38, 43]
```

6 Question 6: Class and Objects

This program demonstrates the creation of classes and objects with attributes and methods.

[20]:
```python
# Class and Objects Implementation
class Student:
    # Class variable
    school_name = "ABC University"

    # Constructor
    def __init__(self, name, roll, marks):
        # Instance variables
        self.name = name
        self.roll = roll
        self.marks = marks

    # Instance method
    def display_info(self):
        print(f"Name: {self.name}")
        print(f"Roll No: {self.roll}")
        print(f"Marks: {self.marks}")
        print(f"School: {Student.school_name}")

    # Method to calculate grade
    def calculate_grade(self):
        if self.marks >= 90:
            return "A+"
        elif self.marks >= 80:
            return "A"
        elif self.marks >= 70:
            return "B"
        elif self.marks >= 60:
            return "C"
        else:
            return "F"
```

```python
    # Class method
    @classmethod
    def change_school(cls, new_school):
        cls.school_name = new_school

    # Static method
    @staticmethod
    def is_passing(marks):
        return marks >= 40

# Creating objects
print("=== Creating Student Objects ===")
student1 = Student("Alice", 101, 85)
student2 = Student("Bob", 102, 92)

print("\nStudent 1 Information:")
student1.display_info()
print(f"Grade: {student1.calculate_grade()}")

print("\n" + "="*30)
print("Student 2 Information:")
student2.display_info()
print(f"Grade: {student2.calculate_grade()}")

# Using static method
print("\n" + "="*30)
print(f"Is student1 passing? {Student.is_passing(student1.marks)}")
print(f"Is 35 marks passing? {Student.is_passing(35)}")

# Using class method
Student.change_school("XYZ College")
print("\nAfter changing school:")
print(f"Student 1 school: {student1.school_name}")
print(f"Student 2 school: {student2.school_name}")
```

```
=== Creating Student Objects ===

Student 1 Information:
Name: Alice
Roll No: 101
Marks: 85
School: ABC University
Grade: A

==============================
Student 2 Information:
Name: Bob
Roll No: 102
```

```
Marks: 92
School: ABC University
Grade: A+

==============================
Is student1 passing? True
Is 35 marks passing? False

After changing school:
Student 1 school: XYZ College
Student 2 school: XYZ College
```

7 Question 7: Polymorphism

This program demonstrates polymorphism through method overriding in different classes.

[21]:
```python
# Polymorphism Implementation
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

    def move(self):
        print(f"{self.name} is moving")

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

class Cat(Animal):
    def speak(self):
        return f"{self.name} says: Meow!"

class Cow(Animal):
    def speak(self):
        return f"{self.name} says: Moo!"

class Bird(Animal):
    def speak(self):
        return f"{self.name} says: Tweet! Tweet!"

    def move(self):
        print(f"{self.name} is flying")

# Polymorphic function
```

```python
def animal_sound(animal):
    print(animal.speak())
    animal.move()

# Creating objects of different classes
print("=== Polymorphism Demo ===\n")
dog = Dog("Buddy")
cat = Cat("Whiskers")
cow = Cow("Bessie")
bird = Bird("Tweety")

# Same function works differently for different objects
animals = [dog, cat, cow, bird]

for animal in animals:
    animal_sound(animal)
    print()

# Polymorphism with different return types
print("=== Method Polymorphism ===")
class Calculator:
    def calculate(self, a, b=None):
        if b is None:
            return a * a  # Square
        else:
            return a + b  # Sum

calc = Calculator()
print(f"Square of 5: {calc.calculate(5)}")
print(f"Sum of 5 and 10: {calc.calculate(5, 10)}")
```

```
=== Polymorphism Demo ===

Buddy says: Woof! Woof!
Buddy is moving

Whiskers says: Meow!
Whiskers is moving

Bessie says: Moo!
Bessie is moving

Tweety says: Tweet! Tweet!
Tweety is flying

=== Method Polymorphism ===
Square of 5: 25
Sum of 5 and 10: 15
```

8 Question 8: Function Overloading

Python doesn't support traditional function overloading, but we can simulate it using default arguments and variable-length arguments.

[22]:
```python
# Function Overloading Simulation in Python

# Method 1: Using default arguments
class Calculator:
    def add(self, a, b=None, c=None):
        if b is None and c is None:
            return a
        elif c is None:
            return a + b
        else:
            return a + b + c

# Method 2: Using *args
class MathOperations:
    def multiply(self, *args):
        result = 1
        for num in args:
            result *= num
        return result

    def display(self, *args, **kwargs):
        print(f"Positional arguments: {args}")
        print(f"Keyword arguments: {kwargs}")

# Method 3: Using type checking
class Processor:
    def process(self, data):
        if isinstance(data, int):
            return f"Processing integer: {data * 2}"
        elif isinstance(data, str):
            return f"Processing string: {data.upper()}"
        elif isinstance(data, list):
            return f"Processing list: Sum = {sum(data)}"
        else:
            return "Unknown type"

# Testing function overloading
print("=== Method 1: Default Arguments ===")
calc = Calculator()
print(f"add(5): {calc.add(5)}")
print(f"add(5, 10): {calc.add(5, 10)}")
print(f"add(5, 10, 15): {calc.add(5, 10, 15)}")
```

```python
print("\n=== Method 2: Variable Arguments ===")
math_ops = MathOperations()
print(f"multiply(2): {math_ops.multiply(2)}")
print(f"multiply(2, 3): {math_ops.multiply(2, 3)}")
print(f"multiply(2, 3, 4): {math_ops.multiply(2, 3, 4)}")

print("\nUsing *args and **kwargs:")
math_ops.display(1, 2, 3, name="Python", version=3.9)

print("\n=== Method 3: Type Checking ===")
processor = Processor()
print(processor.process(10))
print(processor.process("hello"))
print(processor.process([1, 2, 3, 4, 5]))

# Using functools.singledispatch for true overloading
from functools import singledispatch

@singledispatch
def process_data(data):
    return f"Default: {data}"

@process_data.register(int)
def _(data):
    return f"Integer: {data ** 2}"

@process_data.register(str)
def _(data):
    return f"String: {data.lower()}"

@process_data.register(list)
def _(data):
    return f"List length: {len(data)}"

print("\n=== Using singledispatch ===")
print(process_data(5))
print(process_data("HELLO"))
print(process_data([1, 2, 3]))
```

```
=== Method 1: Default Arguments ===
add(5): 5
add(5, 10): 15
add(5, 10, 15): 30

=== Method 2: Variable Arguments ===
multiply(2): 2
multiply(2, 3): 6
multiply(2, 3, 4): 24
```

```
Using *args and **kwargs:
Positional arguments: (1, 2, 3)
Keyword arguments: {'name': 'Python', 'version': 3.9}

=== Method 3: Type Checking ===
Processing integer: 20
Processing string: HELLO
Processing list: Sum = 15

=== Using singledispatch ===
Integer: 25
String: hello
List length: 3
```

9 Question 9: Types of Inheritance

This program demonstrates single, multiple, and multilevel inheritance in Python.

[23]:
```python
# Inheritance Implementation

# 1. Single Inheritance (One parent, one child)
print("=== 1. Single Inheritance ===")
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def show_brand(self):
        print(f"Brand: {self.brand}")

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

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

car = Car("Toyota", "Camry")
car.show_brand()
car.show_details()

# 2. Multiple Inheritance (Multiple parents, one child)
print("\n=== 2. Multiple Inheritance ===")
class Father:
    def skills_father(self):
        print("Father's skills: Business, Management")
```

```python
class Mother:
    def skills_mother(self):
        print("Mother's skills: Cooking, Teaching")

class Child(Father, Mother):
    def skills_child(self):
        print("Child's skills: Programming, Gaming")

child = Child()
child.skills_father()
child.skills_mother()
child.skills_child()

# 3. Multilevel Inheritance (Chain of inheritance)
print("\n=== 3. Multilevel Inheritance ===")
class GrandParent:
    def __init__(self, surname):
        self.surname = surname

    def show_surname(self):
        print(f"Surname: {self.surname}")

class Parent(GrandParent):
    def __init__(self, surname, parent_name):
        super().__init__(surname)
        self.parent_name = parent_name

    def show_parent(self):
        print(f"Parent: {self.parent_name} {self.surname}")

class Child(Parent):
    def __init__(self, surname, parent_name, child_name):
        super().__init__(surname, parent_name)
        self.child_name = child_name

    def show_child(self):
        print(f"Child: {self.child_name} {self.surname}")

    def show_family(self):
        self.show_surname()
        self.show_parent()
        self.show_child()

child2 = Child("Smith", "John", "Mike")
child2.show_family()
```

```python
# Comprehensive Example with all three types
print("\n=== Comprehensive Example ===")
class Animal:
    def __init__(self, species):
        self.species = species

    def show_species(self):
        return f"Species: {self.species}"

class Mammal(Animal):
    def __init__(self, species, sound):
        super().__init__(species)
        self.sound = sound

    def make_sound(self):
        return f"{self.species} makes sound: {self.sound}"

class CanFly:
    def fly(self):
        return "Can fly"

class CanSwim:
    def swim(self):
        return "Can swim"

class Bat(Mammal, CanFly):
    def __init__(self):
        super().__init__("Bat", "Screech")

class Whale(Mammal, CanSwim):
    def __init__(self):
        super().__init__("Whale", "Whale song")

bat = Bat()
print(bat.show_species())
print(bat.make_sound())
print(bat.fly())

print()

whale = Whale()
print(whale.show_species())
print(whale.make_sound())
print(whale.swim())
```

```
=== 1. Single Inheritance ===
Brand: Toyota
Car: Toyota Camry
```

```
=== 2. Multiple Inheritance ===
Father's skills: Business, Management
Mother's skills: Cooking, Teaching
Child's skills: Programming, Gaming

=== 3. Multilevel Inheritance ===
Surname: Smith
Parent: John Smith
Child: Mike Smith

=== Comprehensive Example ===
Species: Bat
Bat makes sound: Screech
Can fly

Species: Whale
Whale makes sound: Whale song
Can swim
```

10 Question 10: Operator Overloading

This program demonstrates operator overloading by defining special methods for custom classes.

[24]:
```python
# Operator Overloading Implementation
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Overload + operator
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    # Overload - operator
    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)

    # Overload * operator (scalar multiplication)
    def __mul__(self, scalar):
        return Point(self.x * scalar, self.y * scalar)

    # Overload == operator
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # Overload < operator
```

```python
    def __lt__(self, other):
        return (self.x**2 + self.y**2) < (other.x**2 + other.y**2)

    # Overload str() function
    def __str__(self):
        return f"Point({self.x}, {self.y})"

    # Overload repr() function
    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

# Testing operator overloading
print("=== Operator Overloading Demo ===\n")
p1 = Point(3, 4)
p2 = Point(1, 2)

print(f"Point 1: {p1}")
print(f"Point 2: {p2}")

# Addition
p3 = p1 + p2
print(f"\np1 + p2 = {p3}")

# Subtraction
p4 = p1 - p2
print(f"p1 - p2 = {p4}")

# Multiplication
p5 = p1 * 2
print(f"p1 * 2 = {p5}")

# Comparison
print(f"\np1 == p2: {p1 == p2}")
print(f"p1 < p2: {p1 < p2}")

# Complex number example
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexNumber(self.real + other.real, self.imag + other.imag)

    def __sub__(self, other):
        return ComplexNumber(self.real - other.real, self.imag - other.imag)
```

```python
    def __mul__(self, other):
        real = self.real * other.real - self.imag * other.imag
        imag = self.real * other.imag + self.imag * other.real
        return ComplexNumber(real, imag)

    def __str__(self):
        if self.imag >= 0:
            return f"{self.real} + {self.imag}i"
        else:
            return f"{self.real} - {abs(self.imag)}i"

print("\n=== Complex Number Operations ===\n")
c1 = ComplexNumber(3, 2)
c2 = ComplexNumber(1, 4)

print(f"c1 = {c1}")
print(f"c2 = {c2}")
print(f"c1 + c2 = {c1 + c2}")
print(f"c1 - c2 = {c1 - c2}")
print(f"c1 * c2 = {c1 * c2}")
```

```
=== Operator Overloading Demo ===

Point 1: Point(3, 4)
Point 2: Point(1, 2)

p1 + p2 = Point(4, 6)
p1 - p2 = Point(2, 2)
p1 * 2 = Point(6, 8)

p1 == p2: False
p1 < p2: False

=== Complex Number Operations ===

c1 = 3 + 2i
c2 = 1 + 4i
c1 + c2 = 4 + 6i
c1 - c2 = 2 - 2i
c1 * c2 = -5 + 14i
```

11 Question 11: Method Overriding

This program demonstrates method overriding where child classes provide specific implementation of parent class methods.

[25]:

```python
# Method Overriding Implementation
class Bank:
    def __init__(self, name):
        self.name = name

    def interest_rate(self):
        return 5.0  # Base interest rate

    def show_details(self):
        print(f"Bank: {self.name}")
        print(f"Interest Rate: {self.interest_rate()}%")

class SBI(Bank):
    def __init__(self):
        super().__init__("State Bank of India")

    # Overriding interest_rate method
    def interest_rate(self):
        return 6.5

class HDFC(Bank):
    def __init__(self):
        super().__init__("HDFC Bank")

    # Overriding interest_rate method
    def interest_rate(self):
        return 7.0

class ICICI(Bank):
    def __init__(self):
        super().__init__("ICICI Bank")

    # Overriding interest_rate method
    def interest_rate(self):
        return 6.8

print("=== Method Overriding Demo ===\n")
banks = [SBI(), HDFC(), ICICI()]

for bank in banks:
    bank.show_details()
    print()

# Another example with shapes
class Shape:
    def __init__(self, name):
        self.name = name
```

```python
    def area(self):
        return 0

    def perimeter(self):
        return 0

    def display(self):
        print(f"Shape: {self.name}")
        print(f"Area: {self.area()}")
        print(f"Perimeter: {self.perimeter()}")

class Rectangle(Shape):
    def __init__(self, length, width):
        super().__init__("Rectangle")
        self.length = length
        self.width = width

    # Overriding area method
    def area(self):
        return self.length * self.width

    # Overriding perimeter method
    def perimeter(self):
        return 2 * (self.length + self.width)

class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    # Overriding area method
    def area(self):
        return 3.14159 * self.radius ** 2

    # Overriding perimeter method
    def perimeter(self):
        return 2 * 3.14159 * self.radius

class Triangle(Shape):
    def __init__(self, side1, side2, side3):
        super().__init__("Triangle")
        self.side1 = side1
        self.side2 = side2
        self.side3 = side3

    # Overriding area method (Heron's formula)
```

```
    def area(self):
        s = (self.side1 + self.side2 + self.side3) / 2
        return (s * (s - self.side1) * (s - self.side2) * (s - self.side3)) **
 ↪0.5

    # Overriding perimeter method
    def perimeter(self):
        return self.side1 + self.side2 + self.side3

print("=== Shape Overriding Example ===\n")
shapes = [
    Rectangle(5, 3),
    Circle(4),
    Triangle(3, 4, 5)
]

for shape in shapes:
    shape.display()
    print()

# Example with super() to extend parent method
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def display(self):
        print(f"Employee: {self.name}")
        print(f"Salary: ${self.salary}")

class Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

    # Overriding and extending parent method
    def display(self):
        super().display()  # Call parent method
        print(f"Department: {self.department}")
        print(f"Total Compensation: ${self.salary * 1.2:.2f}")

print("=== Extending Parent Method ===\n")
emp = Employee("John", 50000)
emp.display()

print()
```

```python
mgr = Manager("Alice", 80000, "IT")
mgr.display()
```

```
=== Method Overriding Demo ===

Bank: State Bank of India
Interest Rate: 6.5%

Bank: HDFC Bank
Interest Rate: 7.0%

Bank: ICICI Bank
Interest Rate: 6.8%

=== Shape Overriding Example ===

Shape: Rectangle
Area: 15
Perimeter: 16

Shape: Circle
Area: 50.26544
Perimeter: 25.13272

Shape: Triangle
Area: 6.0
Perimeter: 12

=== Extending Parent Method ===

Employee: John
Salary: $50000

Employee: Alice
Salary: $80000
Department: IT
Total Compensation: $96000.00
```