```
 #1. Shape Hierarchy:
 # Create an abstract class Shape with abstract methods area() and perimeter().
 # Implement derived classes like Circle, Rectangle, and Triangle that calculate
 # these values.
 #  Calculate the area and perimeter of various shapes and display the results.

from abc import ABC, abstractmethod
import math

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

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

    def area(self):
        return math.pi * self.radius**2

    def perimeter(self):
        return 2 * math.pi * self.radius

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

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

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

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

    def area(self):
        s = (self.side1 + self.side2 + self.side3) / 2
        return math.sqrt(s * (s - self.side1) * (s - self.side2) * (s - self.side3))

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

# Example usage:
circle = Circle(5)
print("Circle - Area:", circle.area())
print("Circle - Perimeter:", circle.perimeter())

rectangle = Rectangle(4, 6)
print("\nRectangle - Area:", rectangle.area())
print("Rectangle - Perimeter:", rectangle.perimeter())

triangle = Triangle(3, 4, 5)
print("\nTriangle - Area:", triangle.area())
print("Triangle - Perimeter:", triangle.perimeter())
```

```
Circle - Area: 78.53981633974483
Circle - Perimeter: 31.41592653589793

Rectangle - Area: 24
Rectangle - Perimeter: 20

Triangle - Area: 6.0
Triangle - Perimeter: 12
```

```
#2. Library Management System:
# Design a library management system using abstraction.
# Create abstract classes for LibraryItem and LibraryMember.
# Implement derived classes for books, DVDs, and library members.
# Use encapsulation to hide internal details and maintain data integrity.
```

```
from abc import ABC, abstractmethod

class LibraryItem(ABC):
    def __init__(self, title, author, item_id):
        self.title = title
        self.author = author
        self.item_id = item_id
        self.checked_out = False

    @abstractmethod
    def check_out(self):
        pass

    @abstractmethod
    def check_in(self):
        pass

class Book(LibraryItem):
    def __init__(self, title, author, item_id, genre):
        super().__init__(title, author, item_id)
        self.genre = genre

    def check_out(self):
        if not self.checked_out:
            self.checked_out = True
            print(f"Book '{self.title}' by {self.author} checked out successfully.")
        else:
            print("This book is already checked out.")

    def check_in(self):
        if self.checked_out:
            self.checked_out = False
            print(f"Book '{self.title}' by {self.author} checked in successfully.")
        else:
            print("This book is already checked in.")

class DVD(LibraryItem):
    def __init__(self, title, director, item_id, duration):
        super().__init__(title, director, item_id)
        self.director = director
        self.duration = duration

    def check_out(self):
        if not self.checked_out:
            self.checked_out = True
            print(f"DVD '{self.title}' directed by {self.director} checked out successfully.")
        else:
            print("This DVD is already checked out.")

    def check_in(self):
        if self.checked_out:
            self.checked_out = False
            print(f"DVD '{self.title}' directed by {self.director} checked in successfully.")
        else:
            print("This DVD is already checked in.")

class LibraryMember(ABC):
    def __init__(self, name, member_id):
        self.name = name
        self.member_id = member_id
        self.checked_out_items = []

    def check_out_item(self, item):
        if not item.checked_out:
            item.check_out()
            self.checked_out_items.append(item)
        else:
            print("Sorry, this item is already checked out.")

    def check_in_item(self, item):
        if item in self.checked_out_items:
            item.check_in()
            self.checked_out_items.remove(item)
        else:
            print("This item was not checked out by this member.")

# Example usage:
book1 = Book("Python Programming", "John Doe", "B001", "Programming")
dvd1 = DVD("Inception", "Christopher Nolan", "D001", 120)

member1 = LibraryMember("Alice", "M001")
member1.check_out_item(book1)
```

```
member1.check_out_item(dvd1)

member2 = LibraryMember("Bob", "M002")
member2.check_out_item(dvd1)
member2.check_in_item(dvd1)
```

```
Book 'Python Programming' by John Doe checked out successfully.
DVD 'Inception' directed by Christopher Nolan checked out successfully.
Sorry, this item is already checked out.
This item was not checked out by this member.
```

```
#3. Game Development:
#  Develop a simple game using Python.
# Create abstract classes for game objects like Player, Enemy, and PowerUp.
# Implement concrete classes for specific game characters.
# Use abstraction to manage game object interactions and behaviors.

from abc import ABC, abstractmethod
import random

class GameObject(ABC):
    @abstractmethod
    def move(self):
        pass

class Player(GameObject):
    def move(self):
        print("Player moves.")

class Enemy(GameObject):
    def move(self):
        print("Enemy moves.")

class PowerUp(GameObject):
    def move(self):
        print("PowerUp moves.")

# Game Logic
player = Player()
enemy = Enemy()
power_up = PowerUp()

game_objects = [player, enemy, power_up]

for _ in range(5):
    random_object = random.choice(game_objects)
    random_object.move()
```

```
Player moves.
Player moves.
PowerUp moves.
Enemy moves.
Player moves.
```

```
# 4. E-commerce Cart:
# Build a shopping cart system for an e-commerce website.
# Create abstract classes for Cart and Product.
# Implement derived classes for different product types.
# Use encapsulation to manage cart operations securely.

from abc import ABC, abstractmethod

class Product(ABC):
    def __init__(self, name, price):
        self.name = name
        self.price = price

    @abstractmethod
    def display_info(self):
        pass

class Cart:
    def __init__(self):
        self.items = []

    def add_item(self, product, quantity=1):
        self.items.append({'product': product, 'quantity': quantity})
        print(f"{quantity} {product.name}(s) added to the cart.")

    def remove_item(self, product, quantity=1):
        for item in self.items:
            if item['product'] == product:
                if item['quantity'] <= quantity:
                    self.items.remove(item)
                else:
                    item['quantity'] -= quantity
                print(f"{quantity} {product.name}(s) removed from the cart.")
                return
        print(f"{product.name} not found in the cart.")

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item['product'].price * item['quantity']
        return total

class Clothing(Product):
    def __init__(self, name, price, size, color):
        super().__init__(name, price)
        self.size = size
        self.color = color

    def display_info(self):
        print(f"{self.name} - ${self.price} | Size: {self.size}, Color: {self.color}")

class Electronics(Product):
    def __init__(self, name, price, brand):
        super().__init__(name, price)
        self.brand = brand

    def display_info(self):
        print(f"{self.name} - ${self.price} | Brand: {self.brand}")

cart = Cart()

shirt = Clothing("T-shirt", 15.99, "M", "Blue")
laptop = Electronics("Laptop", 899.99, "Dell")

cart.add_item(shirt, 2)
cart.add_item(laptop)

cart.remove_item(shirt)
cart.remove_item(laptop)

print("Total:", cart.calculate_total())
```

```
 2 T-shirt(s) added to the cart.
 1 Laptop(s) added to the cart.
 1 T-shirt(s) removed from the cart.
 1 Laptop(s) removed from the cart.
 Total: 15.99
```

```
# 5. Financial Portfolio Management:
# Develop a financial portfolio management system.
# Create abstract classes for Investment and Portfolio.
# Implement derived classes for various types of investments (stocks, bonds,
# real estate).
# Use abstraction to calculate portfolio returns and diversify investments.

from abc import ABC, abstractmethod

class Investment(ABC):
    def __init__(self, name, amount, return_rate):
        self.name = name
        self.amount = amount
        self.return_rate = return_rate

    @abstractmethod
    def calculate_return(self):
        pass

class Stock(Investment):
    def __init__(self, name, amount, return_rate, volatility):
        super().__init__(name, amount, return_rate)
        self.volatility = volatility

    def calculate_return(self):
        return self.amount * (1 + self.return_rate)

class Bond(Investment):
    def __init__(self, name, amount, return_rate, maturity):
        super().__init__(name, amount, return_rate)
        self.maturity = maturity

    def calculate_return(self):
        return self.amount * (1 + self.return_rate)

class RealEstate(Investment):
    def __init__(self, name, amount, return_rate, location):
        super().__init__(name, amount, return_rate)
        self.location = location

    def calculate_return(self):
        return self.amount * (1 + self.return_rate)

class Portfolio:
    def __init__(self):
        self.investments = []

    def add_investment(self, investment):
        self.investments.append(investment)

    def calculate_portfolio_return(self):
        total_return = 0
        for investment in self.investments:
            total_return += investment.calculate_return()
        return total_return

# Usage example
portfolio = Portfolio()

stock1 = Stock("TechStock", 10000, 0.05, 0.2)
bond1 = Bond("GovernmentBond", 5000, 0.03, 5)
real_estate1 = RealEstate("CityProperty", 200000, 0.08, "Metropolis")

portfolio.add_investment(stock1)
portfolio.add_investment(bond1)
portfolio.add_investment(real_estate1)

total_portfolio_return = portfolio.calculate_portfolio_return()
print(f"Total Portfolio Return: ${total_portfolio_return:.2f}")
```

```
Total Portfolio Return: $231650.00
```

```
# 6. Social Media Profile:
# Design a social media profile system.
# Create an abstract class UserProfile with abstract methods
# like post(), comment(), and like().
# Implement derived classes for different user roles (e.g., regular user, admin).
# Use encapsulation to protect user data and interactions.

from abc import ABC, abstractmethod
```

```
class UserProfile(ABC):
    def __init__(self, username):
        self.username = username
        self.posts = []
        self.comments = []
        self.likes = []

    @abstractmethod
    def post(self, content):
        pass

    @abstractmethod
    def comment(self, post, text):
        pass

    @abstractmethod
    def like(self, post):
        pass

class RegularUser(UserProfile):
    def post(self, content):
        self.posts.append(content)
        print(f"{self.username} posted: '{content}'")

    def comment(self, post, text):
        self.comments.append((post, text))
        print(f"{self.username} commented on post {post}: '{text}'")

    def like(self, post):
        self.likes.append(post)
        print(f"{self.username} liked post {post}")

class Admin(UserProfile):
    def post(self, content):
        self.posts.append(content)
        print(f"Admin {self.username} posted: '{content}'")

    def comment(self, post, text):
        self.comments.append((post, text))
        print(f"Admin {self.username} commented on post {post}: '{text}'")

    def like(self, post):
        self.likes.append(post)
        print(f"Admin {self.username} liked post {post}")

# Usage example
user1 = RegularUser("Alice")
user2 = Admin("Admin1")

user1.post("Hello, everyone!")
user2.post("Welcome to the community!")

user1.comment(1, "Nice post!")
user2.comment(2, "Thank you!")

user1.like(2)
user2.like(1)
```

```
Alice posted: 'Hello, everyone!'
Admin Admin1 posted: 'Welcome to the community!'
Alice commented on post 1: 'Nice post!'
Admin Admin1 commented on post 2: 'Thank you!'
Alice liked post 2
Admin Admin1 liked post 1
```

```
#7. Inventory Management:
# Build an inventory management system for a store.
# Create abstract classes for Product and Store.
# Implement derived classes for various product categories (e.g., electronics,
# clothing).
# Use abstraction to manage inventory operations and stock levels.

from abc import ABC, abstractmethod

class UserProfile(ABC):
    def __init__(self, username):
        self.username = username
        self.posts = []
        self.comments = []
        self.likes = []

    @abstractmethod
```

```
        @abstractmethod
        def post(self, content):
            pass

        @abstractmethod
        def comment(self, post_id, text):
            pass

        @abstractmethod
        def like(self, post_id):
            pass

    class RegularUser(UserProfile):
        def post(self, content):
            self.posts.append(content)
            print(f"{self.username} posted: '{content}'")

        def comment(self, post_id, text):
            if post_id < len(self.posts):
                self.comments.append((post_id, text))
                print(f"{self.username} commented on post {post_id}: '{text}'")
            else:
                print(f"Post with ID {post_id} does not exist.")

        def like(self, post_id):
            if post_id < len(self.posts):
                self.likes.append(post_id)
                print(f"{self.username} liked post {post_id}")
            else:
                print(f"Post with ID {post_id} does not exist.")

    class Admin(UserProfile):
        def post(self, content):
            self.posts.append(content)
            print(f"Admin {self.username} posted: '{content}'")

        def comment(self, post_id, text):
            if post_id < len(self.posts):
                self.comments.append((post_id, text))
                print(f"Admin {self.username} commented on post {post_id}: '{text}'")
            else:
                print(f"Post with ID {post_id} does not exist.")

        def like(self, post_id):
            if post_id < len(self.posts):
                self.likes.append(post_id)
                print(f"Admin {self.username} liked post {post_id}")
            else:
                print(f"Post with ID {post_id} does not exist.")

    # Usage example
    user1 = RegularUser("Alice")
    admin1 = Admin("Admin1")

    user1.post("Hello, everyone!")
    admin1.post("Welcome to the community!")
```