Dhruv Saini (22BAI10025)

Python Programming - CSE3011: Acitivity 4

Implements inheritance for different types of users (e.g., Admin, RegularUser) sharing common authentication methods.:

```
class User:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def authenticate(self, name, passwd):
        if name == self.username and passwd == self.password:
            return True
        else:
            return False

class Admin(User):
    def __init__(self, username, password):
        super().__init__(username, password)

class RegularUser(User):
    def __init__(self, username, password):
        super().__init__(username, password)

admin = Admin("admin1", "password123")
user = RegularUser("user1", "password456")

print(f"Admin authenticated: {admin.authenticate('admin1', 'password123')}")
```

```
Admin authenticated: True
```

Demonstrates polymorphism with different payment methods (e.g., CreditCard, PayPal) using a unified interface for processing payments

```
class PaymentMethod:
    def process_payment(self, amount):
        # Function to be overloaded
        pass

class CreditCard(PaymentMethod):
    def process_payment(self, amount):
        return f"Processed payment of {amount} using CreditCard"

class PayPal(PaymentMethod):
    def process_payment(self, amount):
        return f"Processed payment of {amount} using PayPal"

def make_payment(payment_method, amount):
    print(payment_method.process_payment(amount))

credit_card = CreditCard()
paypal = PayPal()

make_payment(credit_card, 100)
make_payment(paypal, 200)
```

```
Processed payment of 100 using CreditCard
Processed payment of 200 using PayPal
```

Utilizes operator overloading to handle operations like adding items to the cart or calculating the total price.:

```
class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price

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

    def __add__(self, item):
        self.items.append(item)  #add operator is redefined
        return self

    def total_price(self):
        return sum(item.price for item in self.items)


cart = ShoppingCart()
item1 = Item("Laptop", 999.99)
item2 = Item("Headphones", 199.99)


cart = cart + item1
cart = cart + item2

print(f"Total price: {cart.total_price()}")
```

```
Total price: 1199.98
```

Using abstract classes to define a template for different types of products (PhysicalProduct and DigitalProduct)

```
from abc import ABC, abstractmethod


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

    @abstractmethod
    def get_description(self):
        pass

    def __str__(self):
        return f"{self.name} (${self.price})"


class PhysicalProduct(Product):
    def __init__(self, name, price, weight):
        super().__init__(name, price)
        self.weight = weight

    def get_description(self):
        return f"{self.name} is a physical product weighing {self.weight} kg."


class DigitalProduct(Product):
    def __init__(self, name, price, file_size):
        super().__init__(name, price)
        self.file_size = file_size

    def get_description(self):
        return f"{self.name} is a digital product with a file size of {self.file_size} MB."


book = PhysicalProduct("Book", 29.99, 1.2)
ebook = DigitalProduct("E-book", 9.99, 5)

print(book.get_description())
print(ebook.get_description())
```

```
Book is a physical product weighing 1.2 kg.
E-book is a digital product with a file size of 5 MB.
```

Python program that shows exception handling using try-except blocks to manage errors in API requests or database operations.

```
class DatabaseError(Exception):
    pass
```

```
class Database:
    def __init__(self):
        self.data = []

    def insert(self, record):
        if not isinstance(record, dict):
            raise DatabaseError("Record must be a dictionary.")
        self.data.append(record)
        print("Record inserted successfully.")

    def fetch(self, index):
        try:
            record = self.data[index]
            print("Record fetched:", record)
        except IndexError:
            print("Error: Record not found.")

    def delete(self, index):
        try:
            del self.data[index]
            print("Record deleted successfully.")
        except IndexError:
            print("Error: Cannot delete non-existent record.")


db = Database()

try:
    db.insert({"id": 1, "name": "Alice"})
    db.insert({"id": 2, "name": "Bob"})
    db.insert("Invalid record")  # This will raise an error
except DatabaseError as e:
    print(f"Database error: {e}")

db.fetch(0)
db.fetch(10)
db.delete(1)
db.delete(10)
```

```
Record inserted successfully.
Record inserted successfully.
Database error: Record must be a dictionary.
Record fetched: {'id': 1, 'name': 'Alice'}
Error: Record not found.
Record deleted successfully.
Error: Cannot delete non-existent record.
```