SOFT40161_Lab05

October 28, 2024

1 SOFT40161 - Introduction to Computer Programming: Lab 05

Note: Try to buid up the concepts! Don't just execute the codes without knowing it's meaning.

1.1 Concepts of Object-Oriented Programming (Classes, Methods, and Constructors)

Please visit the lecture materials first!

1.2 Lab Learning Outcomes

By the end of this lab, you will be able to: 1. **Remember:** Identify key OOP concepts (e.g., classes, inheritance, polymorphism). 2. **Understand:** Explain how OOP principles support modular code. 3. **Apply:** Build simple programs using classes and methods. 4. **Analyze:** Differentiate how inheritance, encapsulation, and polymorphism solve problems. 5. **Evaluate:** Assess code for OOP principles, recommending improvements. 6. **Create:** Develop Python programs with effective use of OOP concepts.

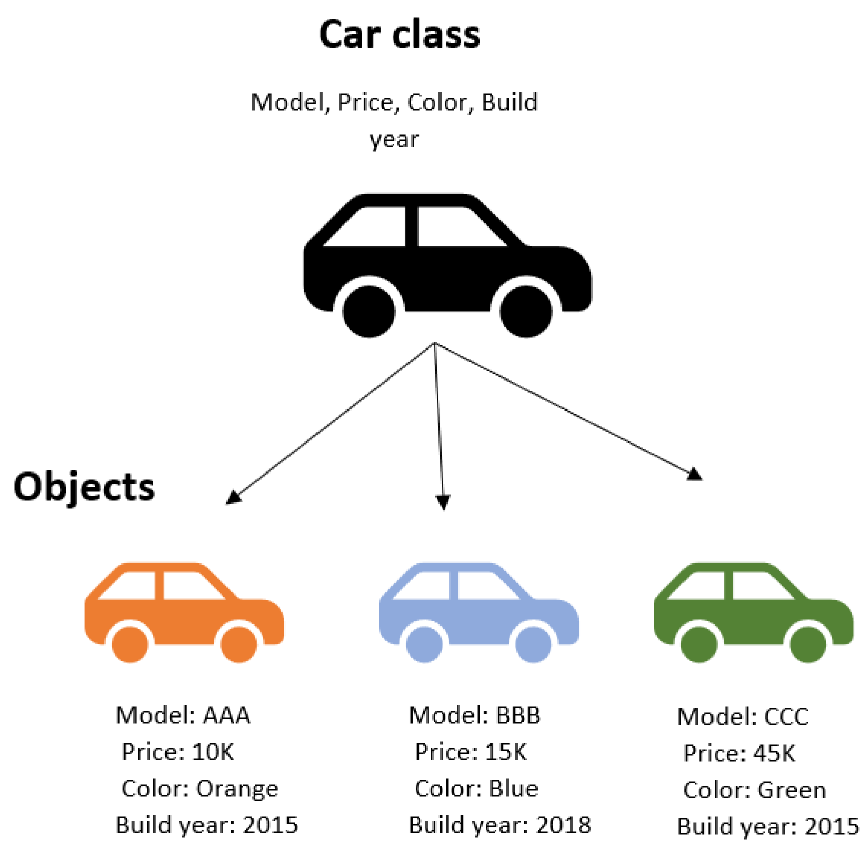
2 Example, Explanation and Exercise

2.1 Example 1: Basic Class Definition

Define a class called `Car` with attributes for `make` and `model`. Then create an instance and print the values.

[ ]:
```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
```

```python
# Creating an instance
my_car = Car("Toyota", "Corolla")
print("Car make:", my_car.make)
print("Car model:", my_car.model)
```



Explanation

The `Car` class is defined with an `__init__` method, which initializes new objects. Creating an instance of `Car` with `Car('Toyota', 'Corolla')` automatically calls `__init__`, assigning values to `make` and `model`. This is a simple example of initializing and accessing class attributes.

Exercise 1

Create a Person class with attributes `name` and `age`. Instantiate the class with a `name` and `age`, then print out the values.

Exercise 1: Solution

[ ]:

```

```

2.2 Adding Methods to a Class

Enhance the `Car` class by adding a method called `display_info` that prints out the car's details.

[ ]:
```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        print(f"Car make: {self.make}, Model: {self.model}")

# Creating an instance and calling the method
my_car = Car("Toyota", "Corolla")
my_car.display_info()
```

Explanation

This example adds a method `display_info` to the `Car` class. Methods are defined like functions within the class and use `self` to refer to the instance they belong to. Calling `my_car.display_info()` executes this method, accessing the `make` and `model` attributes through `self.make` and `self.model`. This demonstrates encapsulation, as the data (attributes) and behavior (methods) are encapsulated within a single `Car` class.

Exercise 2

Add a greet method to the `Person` class that prints a greeting with the person's name. Create an instance and call `greet()` to verify it works.

Exercise 2: Solution

[ ]:
```

```

2.3 Example 3: Using Default Constructor Parameters

Create a `Book` class with `title` and `author` attributes. If no `author` is provided, default to "Unknown".

[ ]:
```
class Book:
    def __init__(self, title, author="Unknown"):
        self.title = title
        self.author = author

    def display_info(self):
        print(f"Book: {self.title}, Author: {self.author}")

# Creating instances with and without specifying the author
book1 = Book("1984", "George Orwell")
book2 = Book("The Great Gatsby")

book1.display_info()
```

```
book2.display_info()
```

Explanation:

Here, we introduce a default value for the author parameter in the `__init__` method. If an `author` is not specified, the constructor assigns `"Unknown"` to `author`. The `display_info` method then uses these attributes when called. In this example, `book2` is created without providing an author, so `self.author` defaults to `"Unknown"`. This is useful when optional information may or may not be provided during instantiation.

Exercise 3

Create a `Movie` class with attributes `title` and `rating`. Set a default value of `"Unrated"` for rating. Create two instances, one with and one without specifying a rating, then print the details.

Exercise 3: Solution

[ ]:
```

```

2.4 Example 4: Adding Class Methods with Parameters

Add a `drive` method to the `Car` class that takes a parameter `distance` and prints the car's distance traveled.

[ ]:
```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        print(f"Car make: {self.make}, Model: {self.model}")

    def drive(self, distance):
        print(f"The {self.make} {self.model} drove {distance} miles.")

# Using the method
my_car = Car("Toyota", "Corolla")
my_car.drive(100)
```

[ ]:
```

```

Explanation

The `drive` method accepts a parameter `distance`, demonstrating how methods can take arguments to modify behavior or calculations. Here, `distance` is passed when drive is called. Inside the method, it prints a message using the instance's `make`, `model`, and `distance` values. This example shows how methods can be customized with additional information when called.

Exercise 4

Add a method calculate_age to the Person class, which takes a birth_year parameter and calculates the person's age based on the current year. Print the calculated age..

Exercise 4: Solution

```
[ ]:
```

2.5 Example 5: Modifying Class Attributes Using Methods

Add a method to the 'Car' class to update the 'make' of the 'car'.

```python
[ ]: class Car:
        def __init__(self, make, model):
            self.make = make
            self.model = model

        def update_make(self, new_make):
            self.make = new_make
            print(f"Car make updated to: {self.make}")

    # Updating the attribute
    my_car = Car("Toyota", "Corolla")
    my_car.update_make("Honda")
```

Explanation:

In this example, `update_make` is a method that modifies the `make` attribute of a `Car` object. Calling `my_car.update_make("Honda")` changes the `make` from `"Toyota"` to `"Honda"`. This illustrates how instance methods can change the internal state of an object, providing controlled ways to update attributes.

Exercise 5:

Add an `update_rating` method to the `Movie` class that takes a new `rating` and updates the `rating` attribute. Create an instance of `Movie` and change its `rating`.

Exercise 5: Solution

```
[ ]:
```

2.6 Example: 6 Implementing Inheritance

Create a class `ElectricCar` that inherits from `Car` and adds a `battery_capacity` attribute.

```python
[ ]: class Car:
        def __init__(self, make, model):
            self.make = make
            self.model = model

        def display_info(self):
            print(f"Car make: {self.make}, Model: {self.model}")

    class ElectricCar(Car):
```

```python
    def __init__(self, make, model, battery_capacity):
        super().__init__(make, model)
        self.battery_capacity = battery_capacity

    def display_battery_info(self):
        print(f"Battery Capacity: {self.battery_capacity} kWh")

# Creating an ElectricCar instance
my_electric_car = ElectricCar("Tesla", "Model S", 100)
my_electric_car.display_info()
my_electric_car.display_battery_info()
```

Explanation:

This example demonstrates inheritance. The `ElectricCar` class inherits from `Car`, meaning it gets all `Car`'s attributes and methods, but also has its own additional attribute `battery_capacity`. The `super().__init__(make, model)` call initializes the parent class (`Car`) so that `make` and `model` are properly set up for `ElectricCar.display_battery_info` is a unique method in `ElectricCar`, and `my_electric_car.display_battery_info()` uses this method to show the battery capacity. Inheritance allows us to extend existing classes with new functionality.

Exercise 6:

Create a SmartPhone class that inherits from a Device class. Device should have attributes brand and model. SmartPhone should add an attribute for storage_capacity. Instantiate SmartPhone and print out all attributes.

Exercise 6: Solution

[ ]:

```python

```

2.7 Example: 7 Implementing Polymorphism

Create a program where different types of animals make sounds, demonstrating polymorphism through a common method, make_sound(), that behaves differently based on the animal type.

[ ]:

```python
class Animal:
    def make_sound(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

class Cow(Animal):
    def make_sound(self):
```

```
        return "Moo!"

# Create instances of each animal
animals = [Dog(), Cat(), Cow()]

# Use polymorphism to call make_sound on each animal
for animal in animals:
    print(animal.make_sound())
```

Explanation:

1. `Base Class (Animal)`: We define a base class Animal with an abstract method make_sound() using raise NotImplementedError(), which makes it clear that subclasses must implement this method.
2. `Subclasses (Dog, Cat, Cow)`: Each subclass (Dog, Cat, and Cow) inherits from Animal and provides its own implementation of make_sound(). Each subclass has a unique sound string in its make_sound() method.
3. Polymorphism in Action: We create a list of different animal objects and loop over them. Thanks to polymorphism, the make_sound() method called on each object in the animals list produces a different output based on the object's specific class.

2.8 Example: 8 Implementing Polymorphism (2nd Example)

Create a program that calculates the area of different shapes (e.g., Rectangle, Circle, and Triangle) using polymorphism.

[ ]:
```
import math

# Base class for Shape
class Shape:
    def calculate_area(self):
        raise NotImplementedError("Subclass must implement calculate_area()")

# Rectangle class inheriting Shape
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

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

    def calculate_area(self):
```

```
        return math.pi * (self.radius ** 2)

# Triangle class inheriting Shape
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def calculate_area(self):
        return 0.5 * self.base * self.height

# List of shapes with various dimensions
shapes = [
    Rectangle(5, 10),
    Circle(7),
    Triangle(6, 8)
]

# Calculating and printing the area of each shape
for shape in shapes:
    print(f"The area of the {shape.__class__.__name__} is: {shape.
 ↪calculate_area():.2f}")
```

Explanation

1. `Base Class (Shape)`: The Shape class defines an abstract calculate_area() method. Each subclass of Shape will implement its own version of calculate_area().
2. `Rectangle Class`: The Rectangle class inherits from Shape and has a calculate_area() method that multiplies width and height.
3. `Circle Class`: The Circle class also inherits from Shape and calculates the area as multiplied by the square of radius.
4. `Triangle Class`: The Triangle class inherits from Shape and calculates the area as 0.5 multiplied by the base and height.

Polymorphism in Action: We create a list of shapes, each of which can be a different type (Rectangle, Circle, or Triangle). By looping through the `list` and calling `calculate_area()`, we see different calculations based on the shape type, demonstrating polymorphic behavior.

2.9 Example: 9 Passing object as an argument

Create a program where one class interacts with another class by passing an object as an argument.

Our objective is to create two classes, BankAccount and Bank, where:

-`BankAccount` handles individual account details, like the owner's name and balance, and allows for deposits and withdrawals. - `Bank` is responsible for transferring funds between two bank accounts.

The main goal is to demonstrate object passing: passing instances of `BankAccount` to the `Bank` class to perform a transfer. This allows classes to interact with each other by using the passed object's methods and attributes.

[ ]:

```python
# Define a class for BankAccount
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount} into {self.owner}'s account. New balance:␣
 ↪{self.balance}")

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew {amount} from {self.owner}'s account. New balance:␣
 ↪{self.balance}")
        else:
            print("Insufficient balance!")

# Define a class for Bank, which will interact with BankAccount objects
class Bank:
    def transfer(self, sender, receiver, amount):
        print(f"Initiating transfer of {amount} from {sender.owner} to␣
 ↪{receiver.owner}")
        if sender.balance >= amount:
            sender.withdraw(amount)
            receiver.deposit(amount)
            print(f"Transfer successful! {sender.owner}'s new balance: {sender.
 ↪balance}, {receiver.owner}'s new balance: {receiver.balance}")
        else:
            print("Transfer failed: insufficient funds in sender's account")

# Create two BankAccount objects
alice_account = BankAccount("Alice", 500)
bob_account = BankAccount("Bob", 300)

# Create a Bank object and initiate a transfer between accounts
bank = Bank()
bank.transfer(alice_account, bob_account, 200)
```

Explanation

1. **Classes:** >- `BankAccount` represents a bank account with deposit and withdrawal functionalities. >- `Bank` contains a method for transferring funds, which requires two BankAccount objects as sender and receiver.
2. **Object Passing:** We create instances `alice_account` and `bob_account` of `BankAccount`.
   - The `transfer` method of the `Bank` class receives these `BankAccount` objects as

arguments.
  - It accesses and modifies their balances using the methods defined in `BankAccount`.

3. **Execution:**

   When the `transfer` method is called, it checks if the sender has enough funds. If so, it initiates a withdrawal from the sender's account and a deposit to the receiver's account, effectively completing the transfer.

This example illustrates object passing in OOP, where a class uses another class's objects to perform specific operations. It promotes modular design and shows how different components (like bank accounts and bank operations) can be designed to interact smoothly.

3 Challenge (H/W): Data Aggregator for Weather Stations

Create a simple data aggregator for weather stations that can store, update, and analyze temperature data collected from multiple locations.

Requirements:

Define a WeatherStation class with:

- station_id, location, and temperature_records (a list of temperatures) as attributes.
- Methods to add a temperature reading, calculate the average temperature, and display temperature records for each station.

Program Flow: - Add Station: Prompt the user to enter a unique station ID and location to create a new station. - Add Temperature Reading: Prompt the user to select a station and input a new temperature reading. - Calculate Average Temperature: Allow the user to select a station and view the average temperature based on its records. - View All Records: Display all recorded temperatures for a selected station.

Guidelines: - Store weather stations in a dictionary with station_id as the key for easy access. - Implement error handling (e.g., check if a station ID exists when adding a temperature).

Example interaction: Weather Station Data Aggregator 1. Add a new station 2. Add temperature reading 3. Calculate average temperature 4. View temperature records 5. Exit

Choose an option:

3.1 Solution:

[ ]:
```python
# Step 1: Define the WeatherStation Class
class WeatherStation:
    def __init__(self, station_id, location):
        self.station_id = station_id
        self.location = location
        self.temperature_records = []

    def add_temperature(self, temperature):
        """Add a new temperature reading to the records."""
        self.temperature_records.append(temperature)
```

```
        print(f"Temperature {temperature}°C added for station {self.station_id}.
↪")

    def calculate_average_temperature(self):
        """Calculate and return the average temperature."""
        if not self.temperature_records:
            return "No temperature records available."
        average_temp = sum(self.temperature_records) / len(self.
↪temperature_records)
        return f"Average Temperature for station {self.station_id} is:␣
↪{average_temp:.2f}°C."

    def display_records(self):
        """Display all temperature records for this station."""
        if not self.temperature_records:
            print("No temperature records available.")
        else:
            print(f"Temperature records for {self.station_id} in {self.
↪location}: {self.temperature_records}")

# Step 2: Set up the Data Aggregator (Dictionary to store stations)
weather_stations = {}

# Step 3: Define functions for each operation
def add_new_station():
    """Prompt user to add a new weather station."""
    station_id = input("Enter Station ID: ")
    location = input("Enter Location: ")
    if station_id in weather_stations:
        print("Station ID already exists. Try a different ID.")
    else:
        station = WeatherStation(station_id, location)
        weather_stations[station_id] = station
        print(f"Station {station_id} at {location} added successfully.")

def add_temperature_reading():
    """Add temperature reading to an existing station."""
    station_id = input("Enter Station ID to add temperature: ")
    if station_id not in weather_stations:
        print("Station ID not found.")
    else:
        try:
            temperature = float(input("Enter temperature in °C: "))
            weather_stations[station_id].add_temperature(temperature)
        except ValueError:
            print("Invalid temperature. Please enter a number.")
```

```python
def calculate_average_temperature():
    """Calculate and display the average temperature for a station."""
    station_id = input("Enter Station ID to calculate average temperature: ")
    if station_id not in weather_stations:
        print("Station ID not found.")
    else:
        print(weather_stations[station_id].calculate_average_temperature())

def view_temperature_records():
    """View all temperature records for a station."""
    station_id = input("Enter Station ID to view temperature records: ")
    if station_id not in weather_stations:
        print("Station ID not found.")
    else:
        weather_stations[station_id].display_records()

# Step 4: Main Program Loop
def main():
    while True:
        print("\nWeather Station Data Aggregator")
        print("1. Add a new station")
        print("2. Add temperature reading")
        print("3. Calculate average temperature")
        print("4. View temperature records")
        print("5. Exit")
        choice = input("Choose an option: ")

        if choice == "1":
            add_new_station()
        elif choice == "2":
            add_temperature_reading()
        elif choice == "3":
            calculate_average_temperature()
        elif choice == "4":
            view_temperature_records()
        elif choice == "5":
            print("Exiting the program.")
            break
        else:
            print("Invalid option. Please select from 1 to 5.")

# Run the program
main()
```

Explanation of the Solution

1. Class Definition (`WeatherStation`):

- The `WeatherStation` class has attributes for `station_id`, `location`, and a list called `temperature_records` to store temperature data. Methods:
- `add_temperature()` appends a new temperature to `temperature_records`.
- `calculate_average_temperature()` computes the average of all temperatures in `temperature_records`.
- `display_records()` prints all temperature records stored for the station.

2. Functions for Operations:

- `add_new_station()` prompts the user to enter a new `station_id` and `location`. It checks if the `station_id` is unique before adding.
- `add_temperature_reading()` prompts for a temperature value and adds it to the specified station's record.
- `calculate_average_temperature()` prints the average temperature for a given station.
- `view_temperature_records()` displays all recorded temperatures for a station.

3. Main Program Loop (`main()`): Displays a menu with options for the user. Each option triggers the relevant function. The loop continues until the user chooses to exit.