Lecture 05 - Object-Oriented Programming

Overview

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of `class` and `objects` which can contain

- Data (`attributes`)
- Code (`methods`)

Why does OOP matter?

OOP allows for organizing complex programs into manageable, modular components, making the code more:

- **Reusable**: Once a `class` is created, it can be used across different parts of the program without rewriting.
- **Scalable**: Capacity to build on existing structures without rewriting them.
- **Maintainable**: Since the code is organized into discrete components, fixing bugs and adding new features is straightforward.

This notebook covers:

- The basics of OOP: **`classes`** and **`objects`**
- **Defining** `classes` and **creating** `objects`
- **`Attributes`** and **`methods`**
- **Inheritance** and **polymorphism**

1. The Basics of OOP: `Classes` and `Objects`

1.1 Definitions

What is a `Class`?

A `class` is like a **blueprint** or **template** for creating `objects`.

It defines the `attributes` (variables) and `methods` (functions) that `objects` created from the `class` will have.

What is an `Object`?

An object is an **instance** of a `class`. It has its own specific values for the attributes defined by the `class`.

Why use `Classes` and `Objects`?

`Classes` are like cookie cutters and `objects` are the actual cookies.

The `class` defines the shape and structure, while each `object` is an independent **instance** that can have different values but shares the same structure.

1.2 Syntax

General

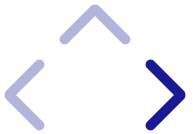
In general, a `class` is alway defined by a **`constructor`** and a set of **`methods`** **and** **`attributes`**.

```
class NAME:
        CONSTRUCTOR
        OTHER METHODS
        ATTRIBUTES
```

In general, a `class` is alway defined by a **`constructor`** and a set of **`methods`** **and** **`attributes`**.

```
class NAME:
        CONSTRUCTOR
        OTHER METHODS
        ATTRIBUTES
```

- The **`Constructor`** sets up initial values for attributes when an `object` is created. This ensures that each `object` starts in a well-defined state.
- The **`Methods`** are callable functions within and outside the `class object`
- The **`Attributes`** are callable variables withing and outside the `class object`

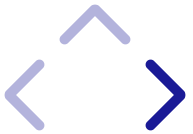


In `Python`

```
class NAME:
        def __init__(self, ...):
            ACTIONS

        def METHODS (self,...):
            ACTIONS

        self.ATTRIBUTES
```

```
class NAME:
        def __init__(self, ...):
            ACTIONS

        def METHODS (self,...):
            ACTIONS

        self.ATTRIBUTES
```

- `Constructor` is defined by the function `init()` where `self` is a reference to the `class object` itself.
- `Methods` are defined like standard functions.
    - Reference to `attributes` that belong to the current class object in the constructor and methods are specified using the prefix `self.`



1.3 Example

Consider a simple example of a **bank account**. Each bank account will have:

1. An **account holder**.
2. A **balance** that tracks the amount of money in the account.

In [ ]:

```python
class BankAccount:
    # constructor
    def __init__(self, account_holder, balance=0):
        self.account_holder = account_holder  # Assigning account holde
        self.balance = balance  # Setting initial balance (default is 0
```

In [ ]:
```python
# Creating an instance of BankAccount
account1 = BankAccount("John Doe", 500)
```

In [ ]:
```python
# Accessing attributes of the object
print(f"Account Holder: {account1.account_holder}")  # Output: John Doe
print(f"Balance: {account1.balance}")  # Output: 500
```

Step-by-step Explanation:

1. `Class` **Definition:** We define the `BankAccount` class.
2. **Constructor (`__init__` method):** This method initializes the attributes `account_holder` and `balance` when a new `BankAccount` object is created.
3. **Creating an Object:** We create an object `account1` of the `BankAccount` class.
4. **Accessing Object Attributes:** We print the values of the object's attributes using the dot notation (e.g., `account1.balance`).

2. `Attributes` and `Methods`

2.1 `Attributes`

`Attributes` are variables that hold data specific to an object.

They are defined within a `class` and belong to each instance of that `class`.

Why are `attributes` important?

`Attributes` store the **state** of the `object`.

In the bank account example, the balance attribute holds the current state of the account.

2.2 `Methods`

`Methods` are functions defined inside a `class` that operate on `objects` of that `class`.

`Methods` can read or modify the `object`'s `attributes`.

Why use `Methods`?

`Methods` define behaviors specific to the `class`.

2.3 Example

Let's add some functionality to our `BankAccount` class.

We want to be able to:

1. **Deposit** money into the account.
2. **Withdraw** money from the account.

In [ ]:

```python
class BankAccount:
    # Constructor
    def __init__(self, account_holder, balance=0):
        self.account_holder = account_holder
        self.balance = balance

    # Method for depositing money
    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount}. New balance is {self.balance}")

    # Method for withdrawing money
    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds.")
        else:
            self.balance -= amount
            print(f"Withdrew {amount}. New balance is {self.balance}")
```

In [ ]:

```
# Creating a new BankAccount object
account2 = BankAccount("Jane Doe", 1000)
account2.deposit(200)  # Depositing 200
account2.withdraw(300)  # Withdrawing 300
```

2.4 Additional examples

Define a `Student` class with the following attributes:

- `name`
- `student_id`
- `grade`

Add methods to:

1. Update the grade.
2. Print a summary of the student's information.

In [ ]:

```python
# Defining the Student class
class Student:
    # Constructor to initialize the attributes of the class
    def __init__(self, name, student_id, grade):
        self.name = name            # Student's name
        self.student_id = student_id  # Unique student ID
        self.grade = grade          # Current grade of the student

    # Method to update the student's grade
    def update_grade(self, new_grade):
        self.grade = new_grade
        print(f"Grade updated to: {self.grade}")

    # Method to print a summary of the student's information
    def print_summary(self):
        print(f"Student Name: {self.name}")
        print(f"Student ID: {self.student_id}")
        print(f"Current Grade: {self.grade}")
```

In [ ]:
```
# Creating an instance of the Student class
student1 = Student("Alice", 12345, "80")
```

In [ ]:
```
# Calling the methods to test the functionality
student1.print_summary()  # Printing the initial student details
```

In [ ]:
```
# Updating the grade and printing the updated details
student1.update_grade("90")
student1.print_summary()  # Should show the updated grade
```

Additional methods to implement

- `has_failed()`
- `can_retake()` -- cannot fail more than twice

In [ ]:
```
# Your code
```

Global exercise

From a list of objects `Student`, order students from highest to lowest grade

In [ ]:

```
# Creating a list of multiple Student objects with varying grades
students = [
    Student("Alice", 12345, "80"),
    Student("Bob", 67890, "40"),
    Student("Charlie", 54321, "60"),
    Student("David", 98765, "99"),
    Student("Eve", 45678, "21")
]

# Function to sort the list of students by their grades
def sort_students_by_grade(student_list):
    # Sort by grade in ascending order (A > B > C)
    sorted_students = sorted(student_list, key=lambda student: student.
    return sorted_students

# Sorting the students and printing the result
sorted_students = sort_students_by_grade(students)

# Displaying the sorted list of students
print("Students sorted by grades:")
for student in sorted_students:
    student.print_summary()
```

3. Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a new `class` to **inherit** **`attributes`** **and** **`methods`** from an existing `class`.

This existing `class` is known as the **parent class** (or `base class`), while the new `class` is referred to as the **child class** (or `derived class`).

Why use inheritance?

1. **Code Reusability**: Inheritance helps reduce code duplication by allowing new `classes` to use the features of existing `classes`.

2. **Logical Hierarchies**: Model real-world relationships, such as “A Car is-a Vehicle.”

3. **Extendability**: Add new features to the `child class` without modifying the `parent class`, making it easier to maintain the code.

How it works

When creating a `child class`, the class **automatically inherits** all the attributes and methods of the `parent class`.

However, the functionality of the `parent class` can also be **extended** or **overridden** within the `child class`.

3.1 Syntax

```
class CHILD (PARENT):
        def __init__(self,...):
            super().__init__(...):
            self....
```

3.2 Step-by-step example

Step 1: Create a `parent class`

Start with a `Person` class that has basic attributes like `name` and `age`, and a method to print a summary.

In [ ]:

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

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")
```

- The `Person` class has two attributes: `name` and `age`.
- It includes a method `display()` that prints the name and age.

Step 2: Create a child class that inherits from `Person`

Now let's create a `Student` class that **inherits** from the `Person` class.

The `Student` class will have an additional attribute `grade` and its own method to print details specific to students.

In [ ]:

```python
class Student(Person):
    def __init__(self, name, age, grade):
        # Call the constructor of the parent class
        super().__init__(name, age)
        self.grade = grade  # New attribute specific to Student

    def display_student(self):
        print(f"Student Name: {self.name}, Age: {self.age}, Grade: {sel
```

- The `Student` class **inherits** from the `Person` class using `class Student(Person):`.
- The `Student` class has its own constructor (`__init__`) that **calls the constructor of the parent class** using `super().__init__(name, age)`.
    - This is necessary to properly initialize attributes from the parent class.
- The `Student` class has a new attribute `grade` and a new method `display_student()`.

Step 3: Creating objects and using inheritance

Now that we have both classes, let's create a `Student` object and use the inherited and new methods.

In [ ]:

```python
# Creating an instance of the Student class
student1 = Student("Alice", 20, "A")

# Calling methods
student1.display()  # Inherited method from Person class
student1.display_student()  # Method specific to Student class
```

- The `student1.display()` call demonstrates **inherited behavior**:
    - `display()` is defined in `Person`, but `student1` (an instance of `Student`) can use it because `Student` inherits from `Person`.
- The `student1.display_student()` call demonstrates **extended behavior**:
    - `display_student()` is defined only in the `Student` class.

3.3 Other Example

Let's create a new class called `SavingsAccount` that inherits from `BankAccount`.

A savings account might have an **interest rate** and a method to apply interest.

In [ ]:

```python
# Inheriting from BankAccount class
class SavingsAccount(BankAccount):
    def __init__(self, account_holder, balance=0, interest_rate=0.02):
        # Call the parent class constructor
        super().__init__(account_holder, balance)
        self.interest_rate = interest_rate  # Additional attribute for

    # New method to apply interest
    def apply_interest(self):
        interest = self.balance * self.interest_rate
        self.deposit(interest)  # Use deposit method to add interest
        print(f"Interest applied. New balance: {self.balance}")
```

In [ ]:

```
# Creating a SavingsAccount object
savings = SavingsAccount("Alice", 2000)
savings.apply_interest()
```

4. Method Overriding and Polymorphism

4.1 Method Overriding

Method overriding occurs when a **`child class`** provides a **specific implementation** for a `method` that is already defined in **its `parent class`**.

This allows the `child class` to define its own behavior for the inherited `method`.

- **Purpose**: To **customize** or **extend** the behavior of a `method` in the `child class`.
- **Syntax**: A `method` in the `child class` has the **same name, parameters**, and **return type** as a `method` in the `parent class`.

In [ ]:

```python
# Parent class
class Vehicle:
    def move(self):
        print("The vehicle is moving")

# Child class overriding the move method
class Car(Vehicle):
    def move(self):
        print("The car is driving on the road")

# Creating instances
v = Vehicle()
c = Car()

# Calling the move method
v.move()
c.move()
```

Example

Let's override the `display()` method in the `Student` class to show the grade as well:

In [ ]:
```python
class Student(Person):
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

    # Overriding the display method from Person class
    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}
```

In [ ]:
```python
student1 = Student("Bob", 21, "B")
student1.display()
```

Other Example:

Let's override the `withdraw()` method from `BankAccount` to `SavingsAccount`.

If a `SavingsAccount` has different withdrawal rules (e.g., no withdrawals below a minimum balance), we can override the `withdraw` method.

In [ ]:

```python
class SavingsAccount(BankAccount):
    def __init__(self, account_holder, balance=0, interest_rate=0.02):
        super().__init__(account_holder, balance)
        self.interest_rate = interest_rate

    def withdraw(self, amount):
        if self.balance - amount < 500:  # Minimum balance of 500
            print("Withdrawal denied: Balance cannot go below 500.")
        else:
            super().withdraw(amount)  # Call the parent class method
```

4.2 Polymorphism

Polymorphism means "many forms" and refers to the ability of different `classes` to **respond to the same method call in different ways**.

It allows the same `method` name to be used for different types of `objects`.

Purpose: To enable `objects` of different `classes` to be treated as objects of a **common parent class**.

In [ ]:

```python
# Parent class
class Animal:
    def sound(self):
        raise NotImplementedError("Subclasses must implement this metho

# Child classes
class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Polymorphism: List of different objects
animals = [Dog(), Cat()]

# Using polymorphism to call the same method
for animal in animals:
    print(animal.sound())  # Output: Bark, Meow
```

In the above example:

1. Both `Dog` and `Cat` classes override the `sound()` method of the `Animal` parent class.
2. When we call `sound()` on each object, **polymorphism** ensures that the correct implementation is executed based on the type of the object (`Dog` or `Cat`).

4.3 Take-away

- **Method overriding** is a technique that allows **polymorphism** to occur.
    - When a `child class` overrides a `method`, it enables polymorphism because `objects` of different `classes` can respond to the same `method` call in their own unique way.
- **Polymorphism** is a broader concept that encompasses method overriding as a way to implement it.

Visual Analogy:

- **Method Overriding** ~ customizing a basic recipe.
    - If the `parent class` provides a recipe for a "generic cake," the `child class` can **override** this recipe to make a "chocolate cake."
- **Polymorphism** ~ ability to treat both the "generic cake" and "chocolate cake" as simply "cakes" when needed.
    - Call `bake()` on both and get the right result, even if the recipes are different.