PROGRAMMING FOR AI

Lecture 4

OOP in AI

Instructor: Zafar Iqbal

Agenda

- Why OOP Matters in AI Development
- Key Differences: Procedural vs. OOP vs. Functional Programming
- Classes and Objects
- Inheritance, Polymorphism, Encapsulation and Abstraction
- Designing Modular and Reusable AI Code

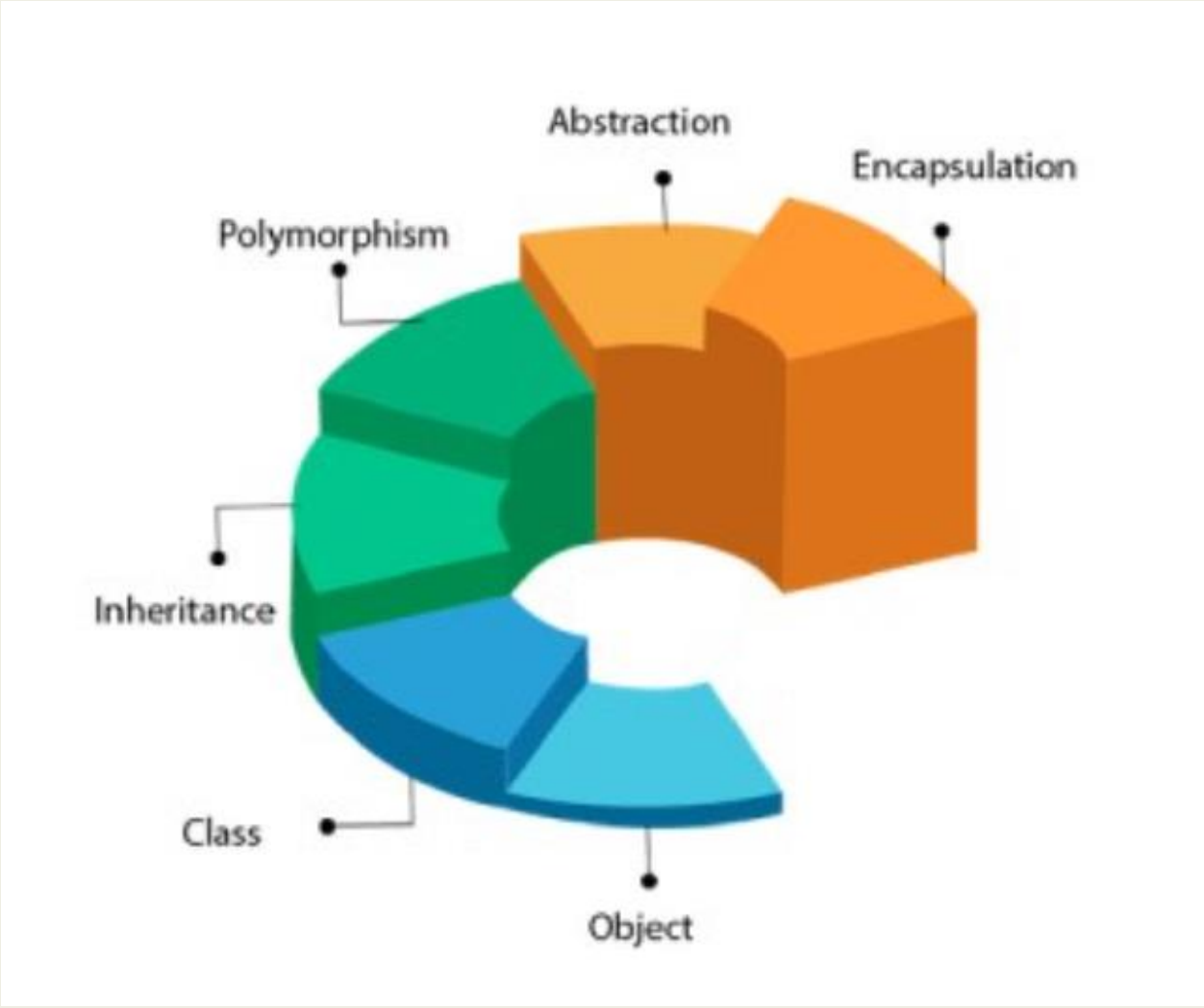
Why OOP Matters in AI Development

- Object-Oriented Programming (OOP) plays a crucial role in AI development
- It provides structure, modularity, and reusability — essential for building complex and scalable AI systems.

Procedural Vs. OOP Vs. Functional Programming

Feature	Procedural Programming	Object-Oriented Programming (OOP)	Functional Programming
Paradigm	Procedure/Step-by-step approach	Object-based, modeling real-world entities	Function-based, treating computation as evaluation of functions
Structure	Organized into functions	Organized into classes and objects	Organized into pure functions
Data Handling	Data and functions are separate	Data and methods are bundled in objects	Emphasizes immutability; data is not modified directly
Reusability	Limited reusability through functions	High reusability via inheritance and polymorphism	High reusability through higher-order functions
Modularity	Functions provide modularity	Classes and objects offer better modularity and encapsulation	Functions are self-contained and modular
State Management	Uses global and local variables	Manages state within objects	Avoids state, prefers stateless functions
Ease of Debugging	Can be harder with complex code	Easier due to encapsulation and modularity	Easier with pure functions and no side effects
Suitability for AI	Good for simple scripts and algorithms	Great for building scalable, complex AI models	Ideal for parallel processing and data transformation tasks
Example in AI	Writing simple machine learning scripts	Designing models as objects (e.g., `NeuralNetwork` class)	Preprocessing data with map, reduce, and lambda functions

Object-Oriented Programming in Python



Classes and Objects

- Class: simply an abstraction of something (e.g. a desk on which your laptop is laying is an object whereas a representation of all desks is a class)

- Object: An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

Python Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

- To create a class, use the keyword class:
- Example
  - *Create a class named MyClass, with a property named x:*

```
class MyClass:
  x = 5

print(MyClass)
```

```
# define a class
class  Dog:
      sound  =  "bark"  # class attribute
```

Create Object

- Now we can use the class named MyClass to create objects:
- Example
  - *Create an object named p1, and print the value of x:*

```
class MyClass:
  x = 5

p1 = MyClass()
print(p1.x)
```

```
class Dog:

        sound = "bark"

# Create an object from the class

dog1 = Dog()

# Access the class attribute

(dog1.sound)
```

The __init__() Function

- To understand the meaning of classes we have to understand the built-in __init__() function.
- All classes have a function called __init__(), which is always executed when the class is being initiated.
- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:
- Example
  - *Create a class named Person, use the __init__() function to assign values for name and age:*

The __init__() Function

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

The __init__() Function

```
class Dog:
        species = "Canine" # Class attribute

def __init__(self, name, age):
        self.name = name # Instance attribute
        self.age = age # Instance attribute
```

```
class Dog:
        species = "Canine" # Class attribute
        def __init__(self, name, age):
                self.name = name # Instance attribute
                self.age = age # Instance attribute
# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)
(dog1.name) # Output: Buddy
(dog1.species) # Output: Canine
```

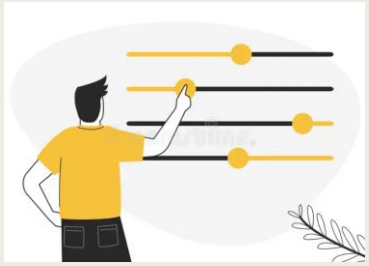
Self Parameter

- self parameter is a reference to the current instance of the class. It allows us to access the attributes and methods of the object.

```
class Dog:
        def __init__(self, name, age):
            self.name = name
            self.age = age
        def bark(self):
            (f"{self.name} is barking!")
# Creating an instance of Dog
dog1 = Dog("Buddy", 3)

dog1.bark()
```

The self Parameter



- It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class:
- Example
  - *Use the words mysillyobject and abc instead of self:*

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

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

The __str__() Function

- __str__ method in Python allows us to define a custom string representation of an object.
- The __str__() function controls what should be returned when the class object is represented as a string.

```
class Dog:
            def __init__(self, name, age):
                self.name = name
                 self.age = age
            def __str__(self):
                return f"{self.name} is {self.age} years old. # Correct: Returning a string
dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)
print(dog1)
print(dog2)
```

The __str__() Function

- The string representation of an object WITH the __str__() function:

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

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

p1 = Person("John", 36)

print(p1)
```

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.
- Let us create a method in the Person class:
- Example
  - *Insert a function that prints a greeting, and execute it on the p1 object:*

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

  def myfunc(self):    #This is the method
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()     #Calling the method
```

Modify/Delete Object Properties

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)

p1.age = 40

print(p1.age)
```

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)

del p1.age

print(p1.age)
```

Delete Objects

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

  def myfunc(self):
print("Hello my name is " + self.name)

p1 = Person("John", 36)

del p1

print(p1)
```

The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

```
class Person:
  pass
```

Python Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

- Create a class named Person, with *firstname* and *lastname* properties, and a *printname* method

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then
execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class
- Example
  - *Create a class named Student, which will inherit the properties and methods from the Person class*

- Note: Use the pass keyword when you do not want to add any other properties or methods to the class.

```
class Student(Person):
  pass
```

Create a Child Class

- Use the Student class to create an object, and then execute the printname method

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  pass

x = Student("Mike", "Olsen")
x.printname()
```

Add Properties

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019

x = Student("Mike", "Olsen")
print(x.graduationyear)
```

Add Methods

- Add a method called welcome to the Student class

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

x = Student("Mike", "Olsen", 2024)
x.welcome()
```

Python Polymorphism

- An example of a Python function that can be used on different objects is the len() function.

```
x = "Hello World!"

print(len(x))
```

For tuples len() returns the number of items in the tuple

```
mytuple = ("apple", "banana", "cherry")

print(len(mytuple))
```

Python Polymorphism

- For dictionaries len() returns the number of key/value pairs in the dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

print(len(thisdict))
```

Python Polymorphism

- Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.
- For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move()

Python Polymorphism

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Drive!")
class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Sail!")
class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747")     #Create a Plane object

for x in (car1, boat1, plane1):
  x.move()
```

Task

- Modify the program to add a new class Bicycle,
- which has the same structure as the other vehicle classes (Car, Boat, Plane).
- The **move**() method of Bicycle should print "**Pedal**!"

Task

```
# New Bicycle class
class Bicycle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    def move(self):
        print("Pedal!")

# Creating objects
car1 = Car("Ford", "Mustang")
boat1 = Boat("Ibiza", "Touring 20")
plane1 = Plane("Boeing", "747")
bike1 = Bicycle("Giant", "Escape 3")  # Creating a Bicycle object

# Loop through all objects and call move()
for x in (car1, boat1, plane1, bike1):
    x.move()
```

Encapsulation

```
class Student:
    def __init__(self, name, grade):
        self.name = name          # Public attribute
        self.__grade = grade      # Private attribute

    def get_grade(self):       #Getter Method to access private attributes
        return self.__grade       # Accessing private attribute

# Creating an object
s1 = Student("Alice", "A")

# Accessing public attribute
print(s1.name)  # Output: Alice

# Accessing private attribute (will cause an error)
# print(s1.__grade)  # Uncommenting this line will cause an AttributeError

# Using method to access private attribute
print(s1.get_grade())  # Output: A
```

Task

- Create another student:
- Instantiate a new student with the name "Emma" and grade "C".
- Print their details using get_grade()

Task

```
class Student:
    def __init__(self, name, grade):
        self.name = name          # Public attribute
        self.__grade = grade      # Private attribute

    def get_grade(self):
        return self.__grade       # Accessing private attribute

# Creating first student
s1 = Student("Alice", "A")
print(f"Student: {s1.name}, Grade: {s1.get_grade()}")

# Creating another student (Emma)
s2 = Student("Emma", "C")
print(f"Student: {s2.name}, Grade: {s2.get_grade()}")
```