Programming for Data Science

Python Object-Oriented Programming Language

Say OL

say_ol@yahoo.com

September 28, 2024

Introduction to OOP

- Programming paradigm based on "objects"
- Objects contain data and methods
- Promotes code reuse and modularity

Key Concepts of OOP

- Classes and Objects
- Encapsulation
- Inheritance
- Polymorphism

Classes and Objects

- A class is a blueprint for creating objects.
- An object is an instance of a class.
- Example:

```
class MyClass:
    pass
```

Defining Classes in Python

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

Creating Objects

- Creating an object:
  ```
  my_dog = Dog("Buddy")
  ```
- Accessing attributes:
  ```
  print(my_dog.name)  # Output: Buddy
  ```

Attributes and Methods

- Attributes: Variables that belong to the object.
- Methods: Functions defined inside a class.

```
class Dog:
    def bark(self):
        return "Woof!"
```

The __init__ Method

- Constructor method to initialize attributes.
- Example:

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

my_dog = Dog("Buddy")
print(my_dog.name)  # Output: Buddy
```

Instance vs. Class Variables

- Instance Variables: Unique to each object.
- Class Variables: Shared among all instances.

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

    def __init__(self, name):
        self.name = name  # Instance variable

dog1 = Dog("Buddy")
dog2 = Dog("Max")
print(dog1.species)  # Output: Canine
print(dog2.species)  # Output: Canine
```

Encapsulation

- Bundling data and methods together.
- Hides internal state using private attributes.
- Example:

```
class Account:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = Account(100)
account.deposit(50)
print(account.get_balance())  # Output: 150
```

Getters and Setters

- Accessor and mutator methods.

```
class Dog:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

my_dog = Dog("Buddy")
print(my_dog.get_name())  # Output: Buddy
my_dog.set_name("Max")
print(my_dog.get_name())  # Output: Max
```

Inheritance

- Mechanism for creating a new class from an existing class.
- Allows reuse of existing code.

Base and Derived Classes

- Base class: Parent class.
- Derived class: Child class that inherits from the base class.

```
class Dog:
    def bark(self):
        return "Woof!"

class Puppy(Dog):
    def bark(self):
        return "Yip!"

puppy = Puppy()
print(puppy.bark())  # Output: Yip!
```

Method Overriding

- Redefining a method in a derived class.

```
class Dog:
    def bark(self):
        return "Woof!"

class Puppy(Dog):
    def bark(self):
        return "Yip!"

my_dog = Dog()
my_puppy = Puppy()
print(my_dog.bark())  # Output: Woof!
print(my_puppy.bark())  # Output: Yip!
```

Polymorphism

- Ability to present the same interface for different data types.
- Example:

```
class Cat:
    def speak(self):
        return "Meow!"

def animal_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()
animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!
```

Abstract Classes

- Classes that cannot be instantiated.
- Used to define interfaces.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

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

dog = Dog()
print(dog.sound())  # Output: Woof!
```

Interfaces

- Define methods that must be created within any child classes.
- Achieved through abstract classes.
- Example:

```
class Animal(ABC):
    @abstractmethod
    def move(self):
        pass

class Bird(Animal):
    def move(self):
        return "Flies"

class Fish(Animal):
    def move(self):
        return "Swims"

bird = Bird()
fish = Fish()
print(bird.move())  # Output: Flies
print(fish.move())  # Output: Swims
```

Composition vs. Inheritance

- Composition: Using classes as attributes of other classes.
- Inheritance: Extending a class's behavior.
- Example:

```python
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return self.engine.start()

car = Car()
print(car.start())  # Output: Engine started
```

Multiple Inheritance

- Inheriting from multiple base classes.
- Can lead to complexity (e.g., Diamond Problem).

```
class A:
    def method(self):
        return "Method from A"

class B(A):
    def method(self):
        return "Method from B"

class C(A):
    def method(self):
        return "Method from C"

class D(B, C):
    pass

d = D()
print(d.method())  # Output: Method from B (MRO)
```

Method Resolution Order (MRO)

- The order in which base classes are looked up.
- Use ‘super()‘ for method calls.
- Example:

```
class A:
    def method(self):
        return "Method from A"

class B(A):
    def method(self):
        return super().method() + " and Method from B"

b = B()
print(b.method())  # Output: Method from A and Method from B
```

Operator Overloading

- Defining how operators behave for custom objects.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

point1 = Point(1, 2)
point2 = Point(3, 4)
result = point1 + point2
print(result.x, result.y)  # Output: 4 6
```

Design Patterns Overview

- Solutions to common software design problems.
- Promotes best practices and reusable code.

Singleton Pattern

- Ensures a class has only one instance.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
          cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2)  # Output: True
```

Factory Pattern

- Method for creating objects without specifying the exact class.

```
class AnimalFactory:
    @staticmethod
    def create_animal(type):
        if type == "dog":
            return Dog()
        elif type == "cat":
            return Cat()
        return None

animal = AnimalFactory.create_animal("dog")
print(animal.bark())  # Output: Woof!
```

Decorator Pattern

- Allows behavior to be added to individual objects.

```
def decorator_function(original_function):
    def wrapper_function():
      print("Wrapper executed before the original function")
        return original_function()
    return wrapper_function

@decorator_function
def display():
    return "Display function executed"

print(display())
# Output: Wrapper executed before the original function
#          Display function executed
```

Observer Pattern

- A one-to-many dependency between objects.
- Example: Event handling in GUI applications.

```
class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

class Observer:
    def update(self, message):
        print(f"Received message: {message}")

subject = Subject()
observer = Observer()
subject.attach(observer)
subject.notify("Hello, Observers!")
# Output: Received message: Hello, Observers!
```

Advantages of OOP

- Code reusability.
- Improved maintainability.
- Clear structure.

Disadvantages of OOP

- Increased complexity.
- Performance overhead.

Future of OOP in Python

- Continued evolution with new features.
- Integration with functional programming concepts.

Q&A Slide

- Questions and discussions.