Object-Oriented Programming (OOP) in Python

Programming languages are built upon certain models to ensure the code behaves predictably. Python primarily follows what is known as an **object-oriented paradigm** or model. As you'll soon discover, **object-oriented programming (OOP)** relies heavily on simplicity and reusability to improve workflow. By the end of this video, you'll be familiar with the **object-oriented programming paradigm**, and you'll also be able to identify the four main concepts that define OOP.

Programming Paradigms

Programming paradigms are strategies for reducing code complexity and determining the flow of execution. There are several different paradigms such as:

- **Declarative**
- **Procedural**
- **Object-oriented**
- **Functional**
- **Logic**
- **Event-driven**
- **Flow-driven**

These paradigms are not mutually exclusive. Programs and programming languages can opt for multiple paradigms. For example, **Python** is primarily object-oriented, but it's also procedural and functional.

In simple terms, a paradigm can be defined as a style of writing a program. **OOP** is one of the most widely used paradigms today due to the growing popularity of languages that use it, such as **Java, Python, C++**, and more. But the ability of OOP to translate real-world problems into code is arguably the biggest factor in its success. OOP has high modularity, which makes code easier to understand, reuse, and allows for layers of abstraction. This enables code blocks to be moved between projects.

Key Components of OOP

To help you better understand OOP, let's first clarify some of its key components:

1. **Classes**: A class is a logical code block that contains **attributes** (variables) and **behavior** (functions). In Python, a class is defined with the `class` keyword.

   For example:

```
class Employee:
    def __init__(self, position, status):
        self.position = position
```

```
            self.status = status

        def intro(self):
            return f"My position is {self.position} and I am {self.status}."
```

2. **Objects**: An object is an instance of a class. You can create any number of objects from the same class. The state of an object is defined by its attributes and behavior.

   Example of instantiating an object:

```
emp1 = Employee("Shift Lead", "Full-time")
print(emp1.intro())  # Output: "My position is Shift Lead and I am Full-time."
```

3. **Methods**: Methods are functions defined inside a class that determine the behavior of an object instance. In the example above, `intro` is a method.

Core Concepts of OOP

Now that you know about classes, objects, and methods, let's explore the concepts that OOP hinges upon:

1. Inheritance

Inheritance is the creation of a new class by deriving from an existing one. The original class is called the **parent** or **superclass**, and the new class is called the **subclass** or **child class**.

Example:

```
class Manager(Employee):  # Manager is a subclass of Employee
    def __init__(self, position, status, department):
        super().__init__(position, status)
        self.department = department

    def intro(self):
        return f"I am a {self.position} in the {self.department} department and I am {self.sta
```

2. Polymorphism

Polymorphism means "many forms." In Python, polymorphism allows the same method to behave differently depending on the object.

For example, the `+` operator behaves differently depending on the datatype:

```
print(3 + 5)  # Output: 8 (addition for integers)
```

```
print("Hello" + " World")  # Output: "Hello World" (concatenation for strings)
```

Here, the `+` operator has multiple forms, depending on whether it's used with numbers or strings.

3. Encapsulation

Encapsulation means bundling the methods and variables within a class and restricting direct access to some of the object's components. This helps in preventing unwanted modifications and reducing errors.

Example:

```
class BankAccount:
    def __init__(self, balance=0):
        self.__balance = balance  # Private attribute

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

    def get_balance(self):
        return self.__balance
```

In this example, `__balance` is private, and can only be accessed or modified through the `deposit` and `get_balance` methods.

4. Abstraction

Abstraction refers to hiding the implementation details and showing only the essential features of an object. Python uses inheritance to achieve abstraction.

Example:

```
from abc import ABC, abstractmethod

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

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

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

In this example, the `Animal` class is abstract and cannot be instantiated. The `sound` method is abstract and must be implemented by any subclass (like `Dog` or `Cat`).

Conclusion

In this video, you became familiar with the **OOP paradigm** and the four key concepts that support it:

1. **Inheritance**
2. **Polymorphism**
3. **Encapsulation**
4. **Abstraction**

See you next time!

This format organizes the content into clear sections and provides code examples for better understanding. Let me know if you'd like further clarification or additional examples!