Object-Oriented Programming (OOP)

Reading Material

PW SKILLS

Topics covered in this Chapter:

1. Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming, or OOP for short, is a way of writing computer programs that's different from the old-school way of doing things. Imagine you're building a house with Lego blocks. In OOP, instead of just focusing on the steps to build the house, we create special Lego blocks (called objects) that know how to do certain things on their own.

Basic Concepts of OOP:

1. **Objects:** These are like smart Lego blocks. Each object contains both information (data) and actions it can perform (methods).

2. Classes: Think of these as blueprints for creating objects. Just like you might have a blueprint for a car before you build it, a class is a plan for making objects.

3. Encapsulation: This is like keeping your toys in a toy box. It means wrapping up data and the methods that work with that data into a single unit (the object).

4. Inheritance: Imagine if you could create a new type of Lego block that has all the features of an existing block, plus some new ones. That's what inheritance does in OOP.

5. Polymorphism: This fancy word means that different objects can respond to the same message in different ways. It's like how both a dog and a cat can "speak," but they make different sounds.

Advantages of OOP:

1. It's easier to organize and understand big programs.
2. You can reuse code more easily, saving time and effort.
3. It's simpler to fix errors and add new features.
4. It helps model real-world things in a more natural way.

OOP vs Procedural Programming:

Procedural programming is like following a recipe step by step. OOP, on the other hand, is more like setting up a kitchen where each appliance knows how to do its job. Here's a quick comparison:

Procedural:
- Focuses on writing functions and procedures
- Data and functions are separate
- Follows a top-down approach

OOP:
- Focuses on creating objects that contain both data and functions
- Combines data and functions into objects
- Follows a bottom-up approach

In the next section, we'll dive deeper into classes and objects, which are the building blocks of OOP.

2. Classes & Objects

Imagine you're creating a video game. In this game, you have many characters, and each character has certain properties and abilities. This is where classes and objects come in handy!

Defining Classes:
A class is like a blueprint for creating objects. It defines what properties and abilities an object will have.

Let's create a simple class for a game character:

```
class GameCharacter:
    def __init__(self, name, health, power):
        self.name = name
        self.health = health
        self.power = power

    def attack(self):
        print(f"{self.name} attacks with {self.power} power!")

    def take_damage(self, damage):
        self.health -= damage
        print(f"{self.name} now has {self.health} health.")
```

In this example:
- We define a class called `GameCharacter`.
- The `__init__` method is a special method called when we create a new character. It sets up the initial values for the character.
- We have two other methods: `attack` and `take_damage`.

Creating Objects:

An object is an instance of a class. It's like a real character created from our blueprint.

```
```python
hero = GameCharacter("Super Saiyan", 100, 50)
villain = GameCharacter("Evil Overlord", 120, 45)
```
```

Here, we've created two objects (characters) from our `GameCharacter` class.

Instance Variables and Methods:

- Instance variables are properties that belong to each individual object. In our example, `name`, `health`, and `power` are instance variables.
- Instance methods are functions that belong to the class and can be called on individual objects. `attack` and `take_damage` are instance methods.

Let's use these:

```
```python
hero.attack()  # Output: Super Saiyan attacks with 50 power!
villain.take_damage(20)  # Output: Evil Overlord now has 100
health.
```
```

Class Variables and Methods:

Sometimes, we want variables or methods that are shared by all instances of a class.

Let's add a class variable and method to our `GameCharacter` class:

```python
class GameCharacter:
    total_characters = 0  # This is a class variable

    def __init__(self, name, health, power):
        self.name = name
        self.health = health
        self.power = power
        GameCharacter.total_characters += 1

    @classmethod
    def show_total_characters(cls):
        print(f"Total characters created:
{cls.total_characters}")

# Now let's use it
hero = GameCharacter("Super Saiyan", 100, 50)
villain = GameCharacter("Evil Overlord", 120, 45)

GameCharacter.show_total_characters()  # Output: Total characters
created: 2
```

Constructor and Destructor:

- The constructor (`__init__` method) is called when an object is created. We've already seen this in action.
- The destructor (`__del__` method) is called when an object is about to be destroyed. It's less commonly used

in Python, but here's an example:

```python
class GameCharacter:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} enters the game!")

    def __del__(self):
        print(f"{self.name} leaves the game.")

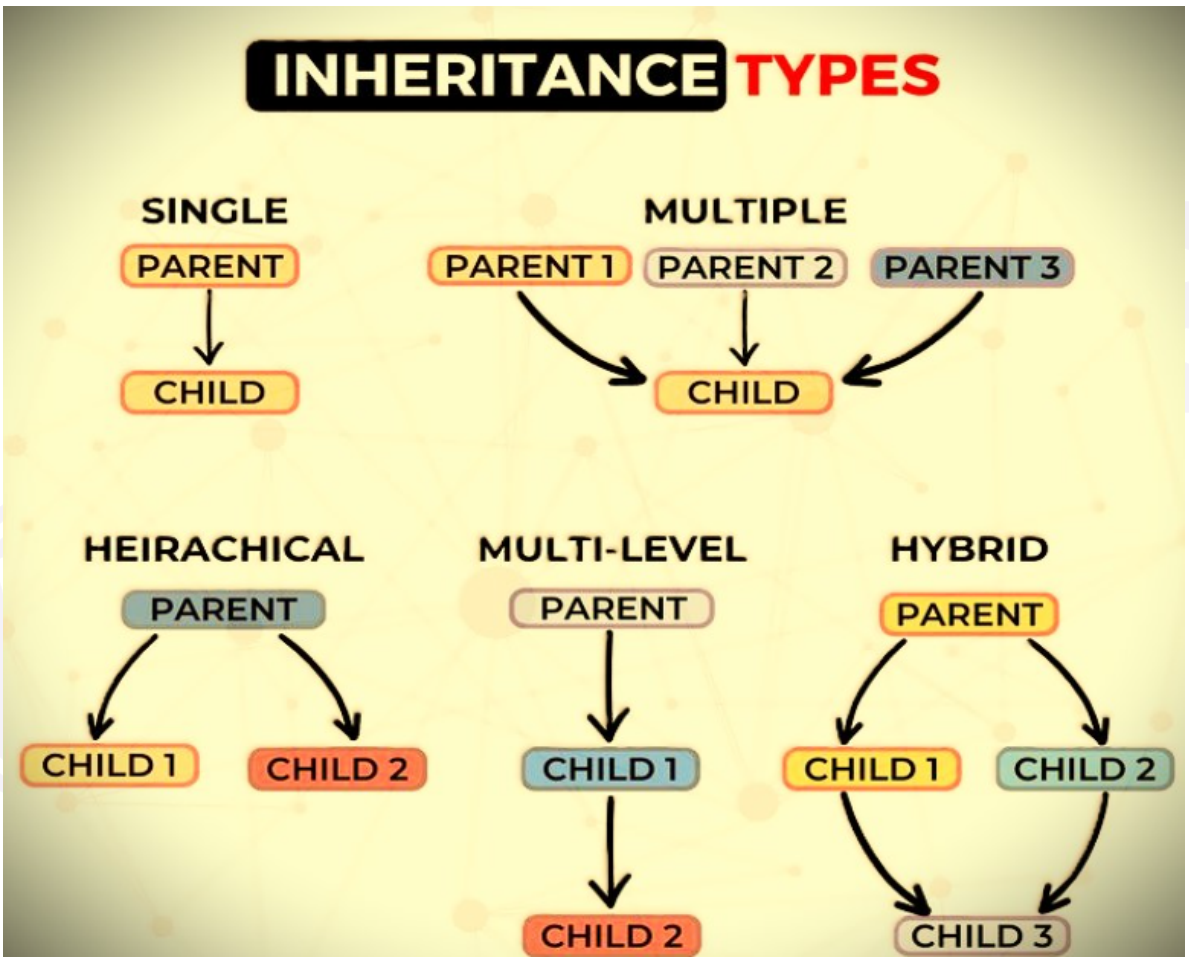
# Using it
hero = GameCharacter("Super Saiyan")  # Output: Super Saiyan
enters the game!
del hero  # Output: Super Saiyan leaves the game.
```

These concepts form the foundation of working with classes and objects in Python. They allow you to create structured, reusable code that models real-world entities or concepts in your programs. In the next section, we'll explore how we can build upon these ideas with inheritance.

3. Inheritance

Inheritance is like passing down traits from parents to children. In programming, it allows us to create new classes based on existing ones, inheriting their attributes and methods.

Types of Inheritance:



1. Single Inheritance:

This is when a class inherits from one parent class. It's the simplest form of inheritance.

Example:

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

    def speak(self):
        pass

class Dog(Animal):
```
```

```
def speak(self):
        return f"{self.name} says Woof!"

fido = Dog("Fido")
print(fido.speak())  # Output: Fido says Woof!
```

Here, `Dog` inherits from `Animal`. It gets the `name` attribute from `Animal` and overrides the `speak` method.

2. Multiple Inheritance:

This is when a class inherits from more than one parent class.

Example:

```
```python
class Flying:
    def fly(self):
        return "I can fly!"

class Swimming:
    def swim(self):
        return "I can swim!"

class Duck(Flying, Swimming):
    pass

donald = Duck()
print(donald.fly())   # Output: I can fly!
print(donald.swim())  # Output: I can swim!
```
```

Here, `Duck` inherits from both `Flying` and `Swimming`, gaining abilities from both.

3. Multilevel Inheritance:

This is when we have a chain of inheritance.

Example:

```
```python
class Grandparent:
    def speak(self):
        return "I am a grandparent"

class Parent(Grandparent):
    def talk(self):
        return "I am a parent"

class Child(Parent):
    def babble(self):
        return "I am a child"

baby = Child()
print(baby.babble())  # Output: I am a child
print(baby.talk())    # Output: I am a parent
print(baby.speak())   # Output: I am a grandparent
```
```

Here, `Child` inherits from `Parent`, which inherits from `Grandparent`.

Method Overriding:

This happens when a child class provides a specific implementation for a method that is already defined in its parent class.

Example:

```
```python
class Animal:
    def make_sound(self):
        return "Some generic animal sound"

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

generic_animal = Animal()
kitty = Cat()
print(generic_animal.make_sound())  # Output: Some generic animal
sound
print(kitty.make_sound())           # Output: Meow!
```
```

Super() Function:

The `super()` function is used to call methods from the parent class. It's especially useful when you want to extend the functionality of a parent method rather than completely replacing it.

Example:

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

class Student(Person):
    def __init__(self, name, age, grade):
        super().__init__(name, age)  # Call the parent's __init__
method
        self.grade = grade

john = Student("John", 15, 9)
print(f"{john.name} is {john.age} years old and in grade
{john.grade}")
# Output: John is 15 years old and in grade 9
```
```

Here, `Student` uses `super()` to call the `__init__` method of `Person`, then adds its own initialization for `grade`.

Multiple Inheritance and Method Resolution Order (MRO):
When using multiple inheritance, Python needs to decide which method to call if the same method name exists in multiple parent classes. It uses the Method Resolution Order (MRO) to make this decision.

Example:

```python
class A:
    def greet(self):
        return "Hello from A"

class B(A):
    def greet(self):
        return "Hello from B"

class C(A):
    def greet(self):
        return "Hello from C"
class D(B, C):
    pass

d = D()
print(d.greet())  # Output: Hello from B
```

In this case, even though `D` inherits from both `B` and `C`, it uses the `greet` method from `B` because `B` comes first in the inheritance list.

Inheritance is a powerful feature that promotes code reuse and allows for the creation of flexible and modular code structures. It's a fundamental concept in OOP that helps in building complex systems by extending and specializing existing code.

Let's move on to the next topic: Polymorphism.

4. Polymorphism

Polymorphism is a fancy word that simply means "many forms." In programming, it allows objects of different types to be treated as objects of a common base type. This concept is crucial for writing flexible and reusable code.

There are several ways polymorphism manifests in Python:

1. Method Overloading:

While Python doesn't support traditional method overloading like some other languages, we can simulate it using default arguments or variable-length arguments.

Example using default arguments:

```python
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c

calc = Calculator()
print(calc.add(5))        # Output: 5
print(calc.add(5, 3))     # Output: 8
print(calc.add(5, 3, 2))  # Output: 10
```

In this example, the `add` method can handle different numbers of arguments.

2. Operator Overloading:

This allows us to define how operators work for our custom objects.

Example:

```
```python
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)
 def __str__(self):
        return f"({self.x}, {self.y})"

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3)  # Output: (4, 6)
```
```

Here, we've defined how the `+` operator should work with our `Point` objects.

3. Duck Typing:

This is a concept in Python where the type or class of an object is less important than the methods it defines. "If it walks like a duck and quacks like a duck, it's a duck."

Example:

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

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

class Human:
    def speak(self):
        return "Hello!"

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

dog = Dog()
cat = Cat()
human = Human()
make_speak(dog)    # Output: Woof!
make_speak(cat)    # Output: Meow!
make_speak(human)  # Output: Hello!
```
```

In this example, `make_speak` doesn't care about the type of `animal`. It only cares that `animal` has a `speak` method.

4. Method Overriding (Runtime Polymorphism):

This is when a subclass provides a specific implementation for a method that is already defined in its superclass.

Example:

```python
class Shape:
    def area(self):
        return 0

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

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

    def area(self):
        return 3.14 * self.radius ** 2

shapes = [Square(5), Circle(3)]
for shape in shapes:
    print(f"Area: {shape.area()}")
# Output:
# Area: 25
# Area: 28.26
```

Here, `Square` and `Circle` both override the `area` method of `Shape`. When we call `area()` on each shape, the appropriate method is called based on the actual type of the object.

Note: Static polymorphism (static binding) is a kind of polymorphism that occurs at compile time. An example of compile-time polymorphism is method overloading.

Runtime polymorphism or dynamic polymorphism (dynamic binding) is a type of polymorphism which is resolved during runtime. An example of runtime polymorphism is method overriding.

Polymorphism is a powerful concept that allows for more flexible and extensible code. It enables us to write code that can work with objects of multiple types, as long as they support the operations we're using. This leads to more modular and reusable code, as we can write functions that operate on abstractions rather than specific implementations.

In the next section, we'll explore encapsulation, another fundamental principle of OOP that helps in organizing and protecting our data.
Let's move on to the next topic: Encapsulation.

5. Encapsulation

Encapsulation is like wrapping up your data and methods in a neat package. It's about bundling the data (attributes) and the methods that work on that data within a single unit (class). This concept helps in hiding the internal details of how an object works and protects the data from unauthorized access.

Key aspects of encapsulation:

1. Access Modifiers:
In Python, we use naming conventions to indicate the access level of attributes and methods:

- **Public:** No special prefix (e.g., `name`)
- **Protected:** Single underscore prefix (e.g., `_age`)
- **Private:** Double underscore prefix (e.g., `__salary`)

Example:

```python
class Employee:
    def __init__(self, name, age, salary):
        self.name = name        # Public
        self._age = age         # Protected
        self.__salary = salary  # Private

    def display_info(self):
        print(f"Name: {self.name}, Age: {self._age}")

    def __calculate_bonus(self):
        return self.__salary * 0.1

emp = Employee("Alice", 30, 50000)
print(emp.name)        # Output: Alice
print(emp._age)        # Output: 30 (but we shouldn't access this
directly)
# print(emp.__salary)  # This would raise an AttributeError
emp.display_info()     # Output: Name: Alice, Age: 30
```

2. Getters and Setters:
These are methods used to get and set the values of private attributes. They allow us to add validation or computation when accessing or modifying data.

Example:

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def set_balance(self, balance):
        if balance >= 0:
            self.__balance = balance
        else:
            print("Balance cannot be negative")

account = BankAccount(1000)
print(account.get_balance())  # Output: 1000
account.set_balance(1500)
print(account.get_balance())  # Output: 1500
account.set_balance(-500)     # Output: Balance cannot be
negative
```

3. Property Decorators:
Python provides a more elegant way to implement getters and setters using the `@property` decorator.

Example:

```python
class Circle:
    def __init__(self, radius):
        self.__radius = radius

    @property
    def radius(self):
        return self.__radius

    @radius.setter
    def radius(self, value):
        if value > 0:
            self.__radius = value
        else:
            print("Radius must be positive")

    @property
    def area(self):
        return 3.14 * self.__radius ** 2

circle = Circle(5)
print(circle.radius)  # Output: 5
circle.radius = 7
print(circle.radius)  # Output: 7
print(circle.area)    # Output: 153.86
circle.radius = -2    # Output: Radius must be positive
```

In this example:
- `radius` is a property that allows getting and setting the private `__radius` attribute.
- `area` is a read-only property that calculates the area based on the radius.

Benefits of Encapsulation:

1. Data Protection: It prevents accidental modification of data from outside the class.
2. Flexibility: We can change the internal implementation without affecting the code that uses the class.
3. Data Validation: We can add checks when setting values to ensure data integrity.
4. Abstraction: It hides the complex implementation details, showing only what's necessary.

Encapsulation is crucial for creating robust and maintainable code. It allows us to control how data is accessed and modified, preventing unintended side effects and making our code more modular and easier to understand.

In the next section, we'll explore abstraction, which is closely related to encapsulation but focuses more on hiding complexity and providing a simpler interface to work with objects.

Let's move on to our final topic: Abstraction.

6. Abstraction

Abstraction is about simplifying complex systems by modeling classes based on the essential properties and behaviors they need to have, while hiding the unnecessary details. It's like using a TV remote - you don't need to know how it works internally, you just need to know which buttons to press.

Key points of Abstraction:

1. Abstract Classes:
An abstract class is a class that is meant to be inherited from, but not instantiated directly. It often contains one or more abstract methods - methods that are declared but don't have an implementation.
In Python, we use the `abc` module (Abstract Base Classes) to create abstract classes.

Example:

```
```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# shape = Shape()  # This would raise an error
rect = Rectangle(5, 3)
print(f"Area: {rect.area()}")        # Output: Area: 15
print(f"Perimeter: {rect.perimeter()}")  # Output: Perimeter: 16
```
```

In this example:
- `Shape` is an abstract class with two abstract methods.
- We can't create an instance of `Shape` directly.
- `Rectangle` inherits from `Shape` and must implement all its abstract methods.

2. Interfaces:
Python doesn't have a formal interface keyword, but we can use abstract classes with only abstract methods to create interface-like structures.

Example:

```python
from abc import ABC, abstractmethod

class Drawable(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Drawable):
    def draw(self):
        return "Drawing a circle"

class Square(Drawable):
    def draw(self):
        return "Drawing a square"

def render(item):
    print(item.draw())
circle = Circle()
square = Square()

render(circle)  # Output: Drawing a circle
render(square)  # Output: Drawing a square
```

Here, `Drawable` acts like an interface, defining a contract that classes must follow.

3. Abstract Base Classes (ABC):
Python's `abc` module provides infrastructure for defining abstract base classes.

Example:

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def move(self):
        pass

    def breathe(self):
        return "Inhale... Exhale..."

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

    def move(self):
        return "Running on four legs"

dog = Dog()
print(dog.make_sound())  # Output: Woof!
print(dog.move())        # Output: Running on four legs
print(dog.breathe())     # Output: Inhale... Exhale...
```

In this example:

- `Animal` is an abstract base class with two abstract methods and one concrete method.
- `Dog` must implement all abstract methods from `Animal`.
- The `breathe` method is inherited as-is.

Benefits of Abstraction:

1. Simplicity: It hides complex implementation details and provides a simple interface.
2. Code Reusability: Abstract classes can be used as a base for multiple concrete classes.
3. Flexibility: You can change the implementation of derived classes without affecting the abstract class.
4. Security: It helps in hiding certain details and only showing the important details of an object.

Abstraction is a powerful concept that allows you to create more organized and manageable code. It helps in dealing with the complexity of large systems by breaking them down into more manageable pieces. By focusing on what an object does rather than how it does it, abstraction supports better code organization and easier maintenance.

This concludes our overview of the main concepts of Object-Oriented Programming. These principles - Encapsulation, Inheritance, Polymorphism, and Abstraction - form the foundation of OOP and are crucial for writing clean, efficient, and maintainable code.
For Practice Purpose: **Colab**