Module 8 - Object-Oriented Programming

Welcome to Module 8! This module introduces you to **Object-Oriented Programming (OOP)**, a powerful paradigm that allows you to structure your code in a way that models real-world entities. You'll learn how to define your own data types (classes), create objects, and understand key OOP principles like inheritance and polymorphism.

Chapter 1: Introduction to Object-Oriented Programming (OOP)

1.1 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain both data (attributes) and code (methods). It aims to organize code around data, rather than logic and functions, making it more intuitive for complex systems.

Think of it this way:

- **Procedural Programming** (what we've mostly done so far): Focuses on functions and sequences of steps. Data is passed around to functions that operate on it. (e.g., `calculate_area(length, width)`)
- **Object-Oriented Programming:** Focuses on creating "objects" that combine data and the functions that operate on that data. (e.g., a `Rectangle` object that *contains* its `length` and `width` and has an `area()` method).

Core Concepts of OOP (The Pillars of OOP):

While we'll cover some in detail, it's good to know the main pillars:

1. **Encapsulation:**
    - Bundling of data (attributes) and methods (functions) that operate on the data into a single unit (a class).
    - It also means **data hiding**, where the internal state of an object is hidden from the outside world, and can only be accessed or modified through the object's methods. This protects data integrity.
2. **Inheritance:**
    - A mechanism that allows a new class (subclass/derived class) to acquire the properties and behaviors (attributes and methods) of an existing class (superclass/base class).
    - Promotes **code reusability** and establishes an "is-a" relationship (e.g., A `Car` "is a" `Vehicle`).
3. **Polymorphism:**
    - Literally "many forms." In OOP, it refers to the ability of objects of different classes to respond to the same method call in their own specific ways.

    - Allows a single interface to be used for different underlying data types. (e.g., A `draw()` method could behave differently for a `Circle` object than for a `Square` object).
4. **Abstraction:**
    - Focusing on essential features and hiding complex implementation details.
    - Providing a simplified, high-level view to the user (or other parts of the program). (e.g., You use a remote to change TV channels without needing to know the complex electronics inside).

Benefits of OOP:

- **Modularity:** Objects are self-contained units, making it easier to understand, develop, and debug individual parts of a system.
- **Reusability:** Inheritance allows you to reuse code from existing classes, saving time and reducing errors.
- **Maintainability:** Changes in one part of the code are less likely to affect others due to encapsulation.
- **Scalability:** Easier to extend and add new features without disrupting existing code.
- **Real-World Modeling:** OOP helps in mapping real-world problems into software solutions more intuitively.

1.2 Objects and Classes

The two fundamental components of OOP are classes and objects.

1. Class: The Blueprint

- A **class** is a blueprint, a template, or a prototype for creating objects.
- It defines the common **attributes** (data/properties) and **methods** (functions/behaviors) that all objects of that type will have.
- Classes don't store data themselves; they define the structure for data.

  Analogy: A cookie cutter is a class. It defines the shape and characteristics of the cookies.

2. Object: The Instance

- An **object** (also called an **instance**) is a concrete realization of a class. It's a specific entity created based on the class's blueprint.
- Each object has its own unique set of attribute values.

  Analogy: The cookies you make using the cookie cutter are objects. Each cookie has the same shape (defined by the class), but might have different icing or sprinkles (its unique attribute values).

Relationship: You define a class once, and then you can create many objects (instances) from that single class.

Chapter 2: Defining a Class

This chapter covers the basic syntax for defining classes and how to work with their attributes and methods.

2.1 Basic Class Definition

In Python, you define a class using the `class` keyword, followed by the class name (by convention, class names are in `PascalCase` or `CamelCase`), and a colon.

Syntax:

Python
```
class ClassName:
    # Class attributes (optional)
    # Methods (functions defined within the class)
    pass # Use 'pass' if the class body is empty for now
```

Creating Objects (Instantiating a Class):

Once a class is defined, you can create objects (instances) of that class by calling the class name like a function.

Python
```
# class_definition.py

# Define a simple class
class Dog:
    # Class body can be empty initially
    pass

# Create objects (instances) of the Dog class
dog1 = Dog()
dog2 = Dog()

print(dog1)
print(dog2)
print(type(dog1))
```

Output:

```
<__main__.Dog object at 0x...>
<__main__.Dog object at 0x...>
<class '__main__.Dog'>
```

Notice that `dog1` and `dog2` are distinct objects, residing at different memory locations. They are both instances of the `Dog` class.

2.2 Instance Attributes and Methods

Objects encapsulate data and behavior.

- **Attributes:** The data associated with an object (its properties).

- **Methods:** The functions that an object can perform (its behaviours).

1. Instance Attributes:

- These are attributes that belong to a specific instance of a class. Each object will have its own unique values for instance attributes.
- They are typically defined inside the constructor method (`__init__`), which we'll cover in the next chapter.
- They are accessed using the dot notation: `object_name.attribute_name`.

2. Methods:

- Methods are functions defined inside a class that operate on the object's data.
- The first parameter of any instance method in Python **must be `self`** (by convention, though you can name it anything).
    - `self` is a reference to the instance (object) itself. Python automatically passes the object when you call a method on it.
    - It allows you to access the object's attributes and other methods within the class.

Example: Defining Attributes and Methods

Python

```
# dog_class.py

class Dog:
    # This is a Class Attribute (shared by all instances)
    species = "Canis familiaris"

    # The constructor method (more on this in Chapter 3)
    # It initializes instance attributes when a new object is created
    def __init__(self, name, breed):
        self.name = name  # Instance attribute
        self.breed = breed # Instance attribute
        self.is_hungry = True # Default instance attribute

    # Instance Method 1
    def bark(self):
        print(f"{self.name} says Woof!")

    # Instance Method 2
    def eat(self):
        if self.is_hungry:
            print(f"{self.name} is eating...")
            self.is_hungry = False
        else:
            print(f"{self.name} is not hungry.")

# Create objects
my_dog = Dog("Buddy", "Golden Retriever")
your_dog = Dog("Lucy", "Labrador")

# Access instance attributes
print(f"My dog's name: {my_dog.name}")
print(f"Your dog's breed: {your_dog.breed}")
```

```
# Access class attribute
print(f"My dog's species: {my_dog.species}")
print(f"Your dog's species: {your_dog.species}")
print(f"Dog species from class: {Dog.species}")

# Call methods
my_dog.bark()
your_dog.eat()
your_dog.eat() # Call again to see hungry status change

# Change an instance attribute directly (possible, but methods are
preferred for complex logic)
my_dog.is_hungry = False
my_dog.eat()
```

Output:

```
My dog's name: Buddy
Your dog's breed: Labrador
My dog's species: Canis familiaris
Your dog's species: Canis familiaris
Dog species from class: Canis familiaris
Buddy says Woof!
Lucy is eating...
Lucy is not hungry.
Buddy is not hungry.
```

Key Takeaways:

- `self` is crucial for distinguishing instance attributes/methods from local variables within a method.
- Class attributes are shared across all instances. Instance attributes are unique to each instance.

Chapter 3: Constructors and Destructors

Special methods in Python classes, known as "magic methods" or "dunder methods" (due to their double underscores), provide powerful functionalities. Constructors and destructors are among them.

3.1 Constructors (`__init__`)

- **Purpose:** The `__init__` method is the **constructor** in Python classes. It's a special method that gets automatically called immediately after an object is created.
- Its primary role is to **initialize the state** (attributes) of the newly created object.
- It takes `self` as its first argument, followed by any other parameters needed to set up the object's initial state.

Syntax:

Python

```
class ClassName:
    def __init__(self, param1, param2, ...):
        self.attribute1 = param1
        self.attribute2 = param2
        # ... and so on
```

Example: Using `__init__`

Python

```
# student_class.py

class Student:
    def __init__(self, name, student_id, major="Undeclared"):
        """
        Constructor for the Student class.
        Initializes a new Student object with name, student_id, and an
optional major.
        """
        self.name = name          # Instance attribute
        self.student_id = student_id # Instance attribute
        self.major = major        # Instance attribute
        self.courses_enrolled = [] # Default empty list for courses

    def enroll_course(self, course_name):
        """Adds a course to the student's enrolled courses."""
        if course_name not in self.courses_enrolled:
            self.courses_enrolled.append(course_name)
            print(f"{self.name} enrolled in {course_name}.")
        else:
            print(f"{self.name} is already enrolled in {course_name}.")

    def get_details(self):
        """Returns student details as a formatted string."""
        return (f"Name: {self.name}, ID: {self.student_id}, "
                f"Major: {self.major}, Courses: {',
'.join(self.courses_enrolled)}")

# Creating Student objects
s1 = Student("Alice Smith", "S1001", "Computer Science")
s2 = Student("Bob Johnson", "S1002") # Using default major

print(s1.get_details())
s1.enroll_course("Python Basics")
s1.enroll_course("Data Structures")
print(s1.get_details())

print("\n" + s2.get_details())
s2.enroll_course("Calculus I")
print(s2.get_details())
```

Output:

```
Name: Alice Smith, ID: S1001, Major: Computer Science, Courses:
Alice enrolled in Python Basics.
Alice enrolled in Data Structures.
Name: Alice Smith, ID: S1001, Major: Computer Science, Courses: Python
Basics, Data Structures

Name: Bob Johnson, ID: S1002, Major: Undeclared, Courses:
```

```
Bob enrolled in Calculus I.
Name: Bob Johnson, ID: S1002, Major: Undeclared, Courses: Calculus I
```

3.2 Destructors (`__del__`)

- **Purpose:** The `__del__` method, also known as the **destructor** or finalizer, is a special method that gets called when an object is about to be destroyed (i.e., its reference count drops to zero, and it's being garbage collected).
- Its primary role is to perform any necessary cleanup operations, such as closing file handles, database connections, or releasing other external resources.

Syntax:

Python
```
class ClassName:
    def __del__(self):
        # Cleanup operations here
        print(f"{self.__class__.__name__} object is being destroyed.")
```

Example: Using `__del__`

Python
```
# file_handler_class.py

class FileHandler:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        try:
            self.file = open(filename, mode)
            print(f"File '{self.filename}' opened in mode '{self.mode}'.")
        except IOError as e:
            print(f"Error opening file {filename}: {e}")
            self.file = None

    def write_line(self, line):
        if self.file:
            self.file.write(line + "\n")
            print(f"Wrote: '{line}' to {self.filename}")
        else:
            print(f"Cannot write, file '{self.filename}' is not open.")

    def __del__(self):
        """
        Destructor: Ensures the file is closed when the object is
destroyed.
        """
        if self.file and not self.file.closed:
            self.file.close()
            print(f"File '{self.filename}' closed automatically by
destructor.")
        else:
            print(f"File '{self.filename}' was already closed or not
opened.")

print("Creating FileHandler object...")
handler = FileHandler("my_log.txt", "w")
```

```
handler.write_line("First log entry.")
handler.write_line("Second log entry.")

print("\nDeleting reference to object (triggering __del__)...")
del handler # Explicitly delete the reference. __del__ will be called.
# Or just let the program end, and garbage collection will eventually run
__del__
print("Reference deleted. __del__ should have been called.")

# Try to write after deletion (will fail because object is gone)
# handler.write_line("This won't work.") # NameError: name 'handler' is not
defined

# Creating another handler
print("\nCreating another FileHandler object...")
another_handler = FileHandler("another_log.txt", "w")
another_handler.write_line("Third log entry.")
# When the program finishes, another_handler's __del__ will be called
```

Output (approximate, order might vary slightly due to garbage collection):

```
Creating FileHandler object...
File 'my_log.txt' opened in mode 'w'.
Wrote: 'First log entry.' to my_log.txt
Wrote: 'Second log entry.' to my_log.txt

Deleting reference to object (triggering __del__)...
File 'my_log.txt' closed automatically by destructor.
Reference deleted. __del__ should have been called.

Creating another FileHandler object...
File 'another_log.txt' opened in mode 'w'.
Wrote: 'Third log entry.' to another_log.txt
File 'another_log.txt' closed automatically by destructor.
```

Cautionary Notes on `__del__`:

- `__del__` is **not guaranteed** to be called immediately when an object is no longer referenced. Python's garbage collector determines when to destroy objects. This can be unpredictable.
- For managing resources like files, it's generally better to use **context managers (`with` statement)**, as they guarantee cleanup even if errors occur. (We'll cover this in advanced topics).
- Use `__del__` primarily for debugging or very specific resource management where context managers are not suitable.

Chapter 4: Functions vs. Object-Oriented Programming

Understanding the differences between a functional/procedural approach and an object-oriented approach is crucial for choosing the right design for your program.

4.1 Procedural/Functional vs. OOP Approach

Let's illustrate the difference with a simple example: managing a `Book`.

Procedural/Functional Approach:

- **Separation of Data and Logic:** Data is stored in basic data structures (lists, dictionaries, variables), and functions are written to operate on this data.
- Focus is on a sequence of operations.

Python

```
# procedural_book.py

# Data representation (e.g., a dictionary)
book1_data = {
    "title": "The Hitchhiker's Guide to the Galaxy",
    "author": "Douglas Adams",
    "pages": 193,
    "is_open": False
}

book2_data = {
    "title": "Pride and Prejudice",
    "author": "Jane Austen",
    "pages": 279,
    "is_open": False
}

# Functions that operate on book data
def open_book(book):
    if not book["is_open"]:
        book["is_open"] = True
        print(f"'{book['title']}' is now open.")
    else:
        print(f"'{book['title']}' is already open.")

def close_book(book):
    if book["is_open"]:
        book["is_open"] = False
        print(f"'{book['title']}' is now closed.")
    else:
        print(f"'{book['title']}' is already closed.")

def get_book_info(book):
    return (f"Title: {book['title']}, Author: {book['author']}, "
            f"Pages: {book['pages']}, Open: {book['is_open']}")

# Usage
print(get_book_info(book1_data))
open_book(book1_data)
print(get_book_info(book1_data))
close_book(book1_data)
print(get_book_info(book1_data))

print("\n" + get_book_info(book2_data))
open_book(book2_data)
```

Observations for Procedural:

- You need to pass the `book` dictionary to every function.

- The data structure (`book_data`) is separate from the functions (`open_book`, `close_book`, `get_book_info`).
- If the structure of `book_data` changes, you might need to modify multiple functions.

Object-Oriented Programming Approach:

- **Bundling of Data and Logic:** Data (attributes) and the functions that operate on that data (methods) are encapsulated within a single unit: an object.
- Focus is on creating self-contained objects that manage their own state.

Python
```
# oop_book.py

class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
        self.is_open = False # Initial state

    def open(self):
        """Opens the book."""
        if not self.is_open:
            self.is_open = True
            print(f"'{self.title}' is now open.")
        else:
            print(f"'{self.title}' is already open.")

    def close(self):
        """Closes the book."""
        if self.is_open:
            self.is_open = False
            print(f"'{self.title}' is now closed.")
        else:
            print(f"'{self.title}' is already closed.")

    def get_info(self):
        """Returns book details."""
        return (f"Title: {self.title}, Author: {self.author}, "
                f"Pages: {self.pages}, Open: {self.is_open}")

# Usage
book1 = Book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams", 193)
book2 = Book("Pride and Prejudice", "Jane Austen", 279)

print(book1.get_info())
book1.open()
print(book1.get_info())
book1.close()
print(book1.get_info())

print("\n" + book2.get_info())
book2.open()
```

Observations for OOP:

- Each `Book` object holds its own `title`, `author`, `pages`, and `is_open` status.

- Methods like `open()`, `close()`, and `get_info()` belong directly to the `Book` object and implicitly operate on *its own* data (`self.title`, `self.is_open`, etc.). You don't pass `book1` or `book2` explicitly as an argument.
- If you add a new attribute (e.g., `publisher`), you only need to modify the `Book` class, not every function that uses book data.

4.2 When to Choose Which?

- **Choose Procedural/Functional:**
  - For **simple, small scripts** where the complexity of defining classes is overkill.
  - When the problem is primarily about **transforming data** from one form to another (functional programming excels here).
  - When you have a set of independent utilities that don't naturally group into objects.
- **Choose OOP:**
  - For **larger, more complex applications** where structuring your code around distinct entities improves readability and manageability.
  - When your program involves many similar "things" that have both data and behavior (e.g., users, products, vehicles, sensors).
  - When you need to model **real-world entities** and their interactions.
  - When you want to leverage OOP principles like inheritance (for code reuse) and polymorphism (for flexible interfaces).
  - When you anticipate that your code will need to be **extended** or **maintained** by multiple developers over time.

In essence, OOP helps manage complexity by breaking down a large problem into smaller, self-contained, and interacting objects.

Chapter 5: Inheritance

Inheritance is one of the most powerful concepts in OOP, allowing you to build new classes based on existing ones.

5.1 Introduction to Inheritance

- **Concept:** Inheritance is a mechanism where a new class (called the **derived class**, **subclass**, or **child class**) inherits properties (attributes) and behaviors (methods) from an existing class (called the **base class**, **superclass**, or **parent class**).
- **"Is-a" Relationship:** Inheritance models an "is-a" relationship. For example, a `Car` "is a" `Vehicle`, a `Dog` "is an" `Animal`.
- **Benefits:**
  - **Code Reusability:** You don't have to rewrite common attributes and methods in new classes. They are inherited from the parent.
  - **Extensibility:** You can extend the functionality of existing classes without modifying them.
  - **Maintainability:** Changes in the base class are automatically reflected in all derived classes.

- **Polymorphism:** A key enabler for polymorphism, allowing child class objects to be treated as parent class objects.

Syntax:

Python
```
class BaseClass:
    # ... attributes and methods ...
    pass

class DerivedClass(BaseClass): # DerivedClass inherits from BaseClass
    # ... additional attributes and methods specific to DerivedClass ...
    pass
```

Example: `Animal` and `Dog`

Python
```
# animal_inheritance.py

class Animal: # Base Class
    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.is_alive = True

    def make_sound(self):
        print("Generic animal sound")

    def eat(self, food):
        print(f"{self.name} is eating {food}.")

    def describe(self):
        return f"{self.name} is a {self.species}."

class Dog(Animal): # Derived Class - Dog inherits from Animal
    def __init__(self, name, breed):
        # Call the constructor of the parent class (Animal)
        # to initialize inherited attributes (name, species)
        super().__init__(name, species="Dog") # Species is always "Dog" for
a Dog
        self.breed = breed # New attribute specific to Dog

    # Dog can have its own specific method
    def bark(self):
        print(f"{self.name} says Woof!")

    # Dog can also have inherited methods
    # (implicitly available: make_sound, eat, describe)

# Create objects
my_animal = Animal("Leo", "Lion")
my_dog = Dog("Buddy", "Golden Retriever")

print(my_animal.describe())
my_animal.make_sound()

print(my_dog.describe()) # Inherited from Animal
my_dog.make_sound()      # Inherited from Animal
my_dog.eat("kibble")     # Inherited from Animal
```

```
my_dog.bark()            # Specific to Dog
```

Output:

```
Leo is a Lion.
Generic animal sound
Buddy is a Dog.
Generic animal sound
Buddy is eating kibble.
Buddy says Woof!
```

5.2 Overriding Methods

- A derived class can provide its own specific implementation for a method that is already defined in its base class. This is called **method overriding**.
- When an overridden method is called on an object of the derived class, the derived class's version is executed. If called on a base class object, the base class's version is executed.

Example: Overriding `make_sound`

Python

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
    def make_sound(self):
        print("Generic animal sound")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species="Dog")
        self.breed = breed
    def make_sound(self): # Overriding make_sound
        print(f"{self.name} barks loudly!")

class Cat(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species="Cat")
        self.breed = breed
    def make_sound(self): # Overriding make_sound
        print(f"{self.name} says Meow!")

animal1 = Animal("Leo", "Lion")
dog1 = Dog("Buddy", "Golden Retriever")
cat1 = Cat("Whiskers", "Siamese")

animal1.make_sound() # Calls Animal's make_sound
dog1.make_sound()    # Calls Dog's make_sound
cat1.make_sound()    # Calls Cat's make_sound
```

Output:

```
Generic animal sound
Buddy barks loudly!
Whiskers says Meow!
```

5.3 `super()` Function

- The `super()` function allows you to refer to the parent or sibling class dynamically.
- Its most common use is to call a method from the parent class, especially the `__init__` constructor, to ensure that inherited attributes are properly initialized.
- It automatically refers to the correct parent class, even in complex inheritance hierarchies.

Usage with `__init__`:

Python
```
class Animal:
    def __init__(self, name, species):
        print(f"Initializing Animal: {name}, {species}")
        self.name = name
        self.species = species

class Dog(Animal):
    def __init__(self, name, breed):
        # Call Animal's __init__ to initialize 'name' and 'species'
        super().__init__(name, "Dog")
        print(f"Initializing Dog: {name}, {breed}")
        self.breed = breed

my_dog = Dog("Max", "German Shepherd")
# Output:
# Initializing Animal: Max, Dog
# Initializing Dog: Max, German Shepherd
```

Usage with Overridden Methods:

You can use `super()` to call the parent's version of an overridden method, and then add specific logic in the child.

Python
```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand
    def start_engine(self):
        print(f"{self.brand} engine starts.")
    def display_info(self):
        print(f"Brand: {self.brand}")

class Car(Vehicle):
    def __init__(self, brand, model):
        super().__init__(brand) # Call parent's __init__
        self.model = model
    def start_engine(self):
        # Call parent's start_engine and then add car-specific logic
        super().start_engine()
        print(f"{self.model} specific car checks complete.")
    def display_info(self):
        super().display_info() # Call parent's display_info
        print(f"Model: {self.model}")

my_car = Car("Toyota", "Camry")
my_car.start_engine()
```

```
print("-" * 20)
my_car.display_info()
```

Output:

```
Toyota engine starts.
Camry specific car checks complete.
--------------------
Brand: Toyota
Model: Camry
```

Chapter 6: Types of Inheritance

Python supports several types of inheritance, allowing flexible class hierarchies.

6.1 Single-Level Inheritance

- **Definition:** The simplest form of inheritance, where a derived class inherits from only **one base class**.
- **Relationship:** Parent-Child.
- **Diagram:**
- `    Base Class`
- `        ^`
- `        |`
- `    Derived Class`
- **Example:** (Revisiting the `Animal` and `Dog` example)

  Python

  ```
  class Vehicle:
      def __init__(self, make, model):
          self.make = make
          self.model = model
      def get_details(self):
          return f"{self.make} {self.model}"

  class Car(Vehicle): # Car inherits from Vehicle
      def __init__(self, make, model, num_doors):
          super().__init__(make, model)
          self.num_doors = num_doors
      def get_details(self): # Overriding
          return f"{super().get_details()} (Doors: {self.num_doors})"

  my_car = Car("Honda", "Civic", 4)
  print(my_car.get_details())
  ```

  Output: `Honda Civic (Doors: 4)`

6.2 Multi-Level Inheritance

- **Definition:** A class inherits from a derived class, forming a chain. Class C inherits from Class B, and Class B inherits from Class A (A -> B -> C).

- **Relationship:** Grandparent -> Parent -> Child.
- **Diagram:**

```
    Grandparent Class (A)
            ^
            |
        Parent Class (B)
            ^
            |
        Child Class (C)
```

- **Example:** `Animal` -> `Mammal` -> `Dog`

Python

```
class Animal: # Grandparent
    def __init__(self, name):
        self.name = name
    def breathe(self):
        print(f"{self.name} is breathing.")

class Mammal(Animal): # Parent, inherits from Animal
    def __init__(self, name, fur_color):
        super().__init__(name)
        self.fur_color = fur_color
    def give_birth(self):
        print(f"{self.name} gives birth to live young.")

class Dog(Mammal): # Child, inherits from Mammal
    def __init__(self, name, fur_color, breed):
        super().__init__(name, fur_color)
        self.breed = breed
    def bark(self):
        print(f"{self.name} ({self.breed}) says Woof!")

my_dog = Dog("Buddy", "Golden", "Golden Retriever")
my_dog.breathe()     # Inherited from Animal
my_dog.give_birth()  # Inherited from Mammal
my_dog.bark()        # Specific to Dog
print(f"My dog's fur color: {my_dog.fur_color}")
```

6.3 Multiple Inheritance

- **Definition:** A class inherits from **more than one base class**. The derived class gets attributes and methods from all its parent classes.
- **Diagram:**

```
    Class A       Class B
        ^             ^
        |             |
        +------+------+
               |
            Class C
```

- **Example:** `Flyer` + `Swimmer` -> `Duck`

Python

```
class Flyer:
```

```
    def fly(self):
        print("I can fly!")

class Swimmer:
    def swim(self):
        print("I can swim!")

class Duck(Flyer, Swimmer): # Duck inherits from both Flyer and
Swimmer
    def quack(self):
        print("Quack! Quack!")

my_duck = Duck()
my_duck.fly()
my_duck.swim()
my_duck.quack()
```

Method Resolution Order (MRO) in Multiple Inheritance:

- **The Diamond Problem:** When multiple inheritance leads to a class inheriting from two classes that both inherit from a common ancestor, conflicts can arise if methods with the same name exist in multiple parent classes. Python needs a clear order to resolve which method to call.
- **Python's MRO:** Python uses a specific algorithm called **C3 linearization** to determine the Method Resolution Order (MRO). This order specifies the sequence in which Python searches for a method in the inheritance hierarchy.
- You can view the MRO of any class using its `.__mro__` attribute or the `help()` function.

  Python

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

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

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

class D(B, C): # Inherits from B first, then C
    pass

obj_d = D()
obj_d.method() # Which method() will be called?

print(D.__mro__)
```

  Output:

```
Method from B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

In this case, `D` inherits from `B` first, so `B`'s `method` is found and called before `C`'s or `A`'s. MRO ensures a consistent and predictable search order.

Chapter 7: Operator Overloading

Operator overloading allows you to define how standard Python operators (like `+`, `-`, `*`, `==`, etc.) behave when applied to instances of your custom classes.

7.1 What is Operator Overloading?

- **Concept:** Giving extended meaning beyond their predefined operational meaning. For example, the `+` operator usually performs addition for numbers and concatenation for strings. With operator overloading, you can define what `+` means when applied to objects of your own class.
- **How it's Achieved:** Python achieves operator overloading through special methods (also known as **dunder methods** or **magic methods**) that begin and end with double underscores (e.g., `__add__`, `__sub__`, `__eq__`). When you use an operator on an object, Python internally calls the corresponding dunder method.

7.2 Common Operators and Their Dunder Methods

Operator	Dunder Method	Description
`+`	`__add__(self, other)`	Addition
`-`	`__sub__(self, other)`	Subtraction
`*`	`__mul__(self, other)`	Multiplication
`/`	`__truediv__(self, other)`	True division
`//`	`__floordiv__(self, other)`	Floor division
`%`	`__mod__(self, other)`	Modulo
`**`	`__pow__(self, other)`	Exponentiation
`==`	`__eq__(self, other)`	Equality (`self == other`)
`!=`	`__ne__(self, other)`	Inequality (`self != other`)
`<`	`__lt__(self, other)`	Less than (`self < other`)
`<=`	`__le__(self, other)`	Less than or equal to (`self <= other`)
`>`	`__gt__(self, other)`	Greater than (`self > other`)
`>=`	`__ge__(self, other)`	Greater than or equal to (`self >= other`)
`str()`	`__str__(self)`	String representation for users (e.g., `print()`)
`repr()`	`__repr__(self)`	Official string representation for developers
`len()`	`__len__(self)`	Length of the object

`[]`	`__getitem__(self, key)`, `__setitem__(self, key, value)`, `__delitem__(self, key)`	Indexing/slicing (e.g., `obj[key]`)

Export to Sheets

7.3 Examples

1. Overloading `+` for a `Vector` Class:

Let's say we want to add two `Vector` objects.

Python

```
# vector_overload.py

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

    def __str__(self): # For user-friendly string representation
        return f"Vector({self.x}, {self.y})"

    def __repr__(self): # For developer-friendly string representation
(often same as __str__)
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        """
        Overloads the '+' operator.
        Adds two Vector objects by adding their corresponding components.
        """
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Can only add a Vector object to another Vector
object.")

    def __mul__(self, scalar):
        """
        Overloads the '*' operator for scalar multiplication.
        Multiplies a Vector by a scalar (number).
        """
        if isinstance(scalar, (int, float)):
            return Vector(self.x * scalar, self.y * scalar)
        else:
            raise TypeError("Can only multiply a Vector by a scalar
(int/float).")

    def __eq__(self, other):
        """
        Overloads the '==' operator.
        Compares two Vector objects for equality.
        """
        if isinstance(other, Vector):
            return self.x == other.x and self.y == other.y
        return NotImplemented # Indicates that comparison is not
implemented for other types

# Create Vector objects
```

```
v1 = Vector(2, 3)
v2 = Vector(1, 5)

print(f"v1: {v1}") # Uses __str__
print(f"v2: {v2}")

# Use the overloaded '+' operator
v_sum = v1 + v2
print(f"v1 + v2 = {v_sum}") # Output: Vector(3, 8)

# Use the overloaded '*' operator
v_scaled = v1 * 3
print(f"v1 * 3 = {v_scaled}") # Output: Vector(6, 9)

# Use the overloaded '==' operator
v3 = Vector(2, 3)
print(f"v1 == v2: {v1 == v2}") # Output: False
print(f"v1 == v3: {v1 == v3}") # Output: True

# This will raise a TypeError because we didn't implement addition with int
# print(v1 + 5)
```

2. Overloading `len()` and Indexing (`__len__`, `__getitem__`):

Python

```
class MyListLike:
    def __init__(self, data):
        self.data = list(data) # Ensure it's a list internally

    def __len__(self):
        """Overloads len() to return the length of the internal data."""
        return len(self.data)

    def __getitem__(self, index):
        """Overloads [] for indexing (e.g., obj[0])."""
        return self.data[index]

    def __str__(self):
        return str(self.data)

my_collection = MyListLike([10, 20, 30, 40])
print(f"Collection: {my_collection}")
print(f"Length of collection: {len(my_collection)}") # Calls __len__

print(f"First element: {my_collection[0]}") # Calls __getitem__
print(f"Last element: {my_collection[3]}")
print(f"Slice: {my_collection[1:3]}") # __getitem__ also handles slicing
```

Chapter 8: Data Hiding / Encapsulation

Encapsulation is a core OOP principle that involves bundling data (attributes) and the methods that operate on that data within a single unit (a class). It also implies **data hiding**, which restricts direct access to some of an object's components.

8.1 Concept of Encapsulation

- **Bundling:** Combining attributes and methods into a single class. This is achieved naturally by defining attributes in `__init__` and methods within the class.
- **Data Hiding (Information Hiding):** The idea of protecting the internal state of an object from direct external access. This prevents accidental corruption of data and ensures that the object's internal logic is consistent.
    - Instead of direct access (`object.attribute = value`), data should ideally be accessed or modified through **public methods** (often called getters and setters).
    - This provides a controlled interface to the object's data, allowing for validation, logging, or other logic to be executed when data is accessed or changed.

8.2 Data Hiding in Python (by Convention)

Unlike languages like Java or C++ that have keywords like `private` or `protected`, Python does not have strict access modifiers. Instead, it relies on conventions and a mechanism called "name mangling."

1. Public Attributes (No Underscore):

- Attributes without any leading underscores are considered **public**. They can be accessed and modified directly from anywhere.
- This is the default and most common in Python.

Python

```
class PublicExample:
    def __init__(self, value):
        self.public_attribute = value

obj = PublicExample(10)
print(obj.public_attribute) # Accessible
obj.public_attribute = 20   # Modifiable
print(obj.public_attribute)
```

2. "Protected" Attributes (Single Underscore `_name`):

- Attributes prefixed with a **single underscore** (e.g., `_protected_attribute`) are a **convention** to indicate that they are "protected" or "internal use only."
- Python does *not* prevent direct access or modification of these attributes. It's a hint to other developers: "Don't touch this directly unless you know what you're doing."

Python

```
class ProtectedExample:
    def __init__(self, value):
        self._protected_attribute = value # Convention for
"protected"

obj = ProtectedExample(10)
print(obj._protected_attribute) # Still accessible, but convention
says don't
obj._protected_attribute = 20   # Still modifiable
```

```
print(obj._protected_attribute)
```

3. "Private" Attributes (Double Underscore `__name` - Name Mangling):

- Attributes prefixed with **two leading underscores** (e.g., `__private_attribute`) trigger **name mangling**.
- Python *internally renames* these attributes to `_ClassName__private_attribute`. This makes them harder, but not impossible, to access directly from outside the class. It's a way to avoid naming conflicts in inheritance.
- They are still technically accessible by their mangled name, but it's strongly discouraged.

Python

```
class PrivateExample:
    def __init__(self, value):
        self.__private_attribute = value # Triggers name mangling

    def get_private(self):
        return self.__private_attribute

obj = PrivateExample(10)
# print(obj.__private_attribute) # This will cause an AttributeError

print(obj.get_private()) # Accessible via method
# Output: 10

# Accessing via mangled name (not recommended)
print(obj._PrivateExample__private_attribute)
# Output: 10
obj._PrivateExample__private_attribute = 20
print(obj.get_private())
# Output: 20
```

Summary on Data Hiding in Python: Python's philosophy is "we're all consenting adults." It provides mechanisms to indicate intent (like _ and __), but ultimately trusts developers to respect conventions.

8.3 Getters and Setters (Accessor and Mutator Methods)

- **Purpose:** To provide a controlled way to access (get) and modify (set) an object's attributes. This aligns with the encapsulation principle, as it allows you to add validation, logging, or other logic whenever an attribute is read or written.
- **Getter (Accessor):** A method that returns the value of an attribute.
- **Setter (Mutator):** A method that sets or modifies the value of an attribute.

1. Using Simple Methods:

Python

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius # Convention: internal use only

    def get_celsius(self):
```

```
        return self._celsius

    def set_celsius(self, value):
        if value < -273.15: # Basic validation
            print("Error: Temperature cannot be below absolute zero.")
        else:
            self._celsius = value
            print(f"Temperature set to {self._celsius}°C.")

    def get_fahrenheit(self):
        return (self._celsius * 9/5) + 32

t = Temperature(25)
print(f"Current temp: {t.get_celsius()}°C ({t.get_fahrenheit()}°F)")

t.set_celsius(30)
print(f"New temp: {t.get_celsius()}°C")

t.set_celsius(-300) # This will trigger the validation
print(f"Temp after invalid set attempt: {t.get_celsius()}°C")
```

2. Using the `@property` Decorator (Pythonic Way):

The `@property` decorator allows you to define methods as if they were attributes, providing a cleaner syntax while still enforcing encapsulation logic.

Python

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property # This decorator makes the 'celsius' method a getter for an
attribute 'celsius'
    def celsius(self):
        """The temperature in Celsius."""
        return self._celsius

    @celsius.setter # This decorator makes the 'celsius' method a setter
for the 'celsius' attribute
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero.")
        self._celsius = value
        print(f"Temperature set to {self._celsius}°C using property.")

    @property
    def fahrenheit(self):
        """The temperature in Fahrenheit."""
        return (self.celsius * 9/5) + 32 # Access celsius via property, not
_celsius directly

# Usage
t = Temperature(25)
print(f"Current temp (getter): {t.celsius}°C") # Calls the celsius getter
method
print(f"Fahrenheit (getter): {t.fahrenheit}°F") # Calls the fahrenheit
getter method

t.celsius = 30 # Calls the celsius setter method
```

```
print(f"New temp: {t.celsius}°C")

try:
    t.celsius = -300 # This will raise a ValueError
except ValueError as e:
    print(e)
```

The `@property` decorator is the preferred Pythonic way to implement getters and setters, as it allows you to maintain the convenience of attribute access while still implementing validation or other logic behind the scenes.

Chapter 9: Coding Challenges for Object-Oriented Programming

Let's apply your knowledge of OOP by building some practical examples!

Challenge 1: Basic Class & Object - Car Simulation

Goal: Create a `Car` class to represent car objects with basic attributes and behaviors.

Concepts Covered: Class definition, instance attributes, methods, `__init__`.

Requirements:

1. Define a class named `Car`.
2. The `Car` class should have an `__init__` method that takes `make`, `model`, and `year` as arguments and initializes them as instance attributes.
3. Add an instance attribute `is_running` initialized to `False`.
4. Add a method `start()` that changes `is_running` to `True` and prints a message like "The [make] [model] engine started."
5. Add a method `stop()` that changes `is_running` to `False` and prints "The [make] [model] engine stopped."
6. Add a method `display_info()` that prints all the car's details (make, model, year, and running status).
7. Create at least two `Car` objects.
8. Call their methods to demonstrate functionality.

Example Interaction:

```
--- My Cars ---
Car 1: Toyota Camry (2020), Running: False
Car 2: Honda Civic (2022), Running: False

Starting Car 1...
The Toyota Camry engine started.
Car 1: Toyota Camry (2020), Running: True

Stopping Car 2...
The Honda Civic engine stopped.
Car 2: Honda Civic (2022), Running: False
```

Challenge 2: Bank Account (Constructors and Methods)

Goal: Develop a `BankAccount` class that models a simple bank account.

Concepts Covered: `__init__`, instance attributes, methods, basic error handling.

Requirements:

1. Create a class `BankAccount`.
2. Its `__init__` method should take `account_number` and `account_holder_name` as required arguments, and an optional `initial_balance` (default to 0).
3. Implement methods:
    - `deposit(amount)`: Adds `amount` to the balance. Print a confirmation. Ensure amount is positive.
    - `withdraw(amount)`: Subtracts `amount` from the balance. Print a confirmation. Ensure amount is positive and there are sufficient funds.
    - `get_balance()`: Returns the current account balance.
    - `display_account_info()`: Prints all account details.
4. Create an account object and perform a series of deposits and withdrawals, demonstrating error handling for invalid operations.

Example Interaction:

```
Account created for Alice Smith (12345) with initial balance $1000.00

Depositing $200.00...
Deposit successful. Current balance: $1200.00

Withdrawing $500.00...
Withdrawal successful. Current balance: $700.00

Withdrawing $1000.00...
Insufficient funds. Withdrawal failed. Current balance: $700.00

Withdrawing $-50.00...
Withdrawal amount must be positive.

Account Information:
Account Number: 12345
Account Holder: Alice Smith
Current Balance: $700.00
```

Challenge 3: Shape Inheritance (Multi-level Inheritance)

Goal: Design a class hierarchy for shapes demonstrating multi-level inheritance.

Concepts Covered: Multi-level inheritance, `super()`, method overriding.

Requirements:

1. **`Shape` (Base Class):**
    - `__init__(self, name)`: Initializes `name`.
    - `display_name()`: Prints `f"This is a {self.name}."`

2. **`TwoDShape` (Intermediate Class):**
    - Inherits from `Shape`.
    - `__init__(self, name, sides)`: Initializes `name` (via `super()`) and `sides`.
    - `display_sides()`: Prints `f"It has {self.sides} sides."`
3. **`Square` (Derived Class):**
    - Inherits from `TwoDShape`.
    - `__init__(self, side_length)`: Initializes `name` as "Square" (via `super()`), `sides` as 4 (via `super()`), and `side_length`.
    - `calculate_area()`: Returns `side_length * side_length`.
    - `display_info()`: Calls `super().display_name()`, `super().display_sides()`, and then prints `f"Area: {self.calculate_area()}"`.
4. Create a `Square` object and call `display_info()`.

Example Output:

```
This is a Square.
It has 4 sides.
Area: 25
```

Challenge 4: Simple Fraction Calculator (Operator Overloading)

Goal: Create a `Fraction` class and overload the `+` and `==` operators.

Concepts Covered: Operator overloading (`__add__`, `__eq__`), `__str__`, `__repr__`, `gcd` (optional for simplification).

Requirements:

1. Create a class `Fraction` with an `__init__(self, numerator, denominator)` method.
2. Add a simple `__str__` method to represent the fraction (e.g., "1/2").
3. **Overload `+` (`__add__`)**: Define how two `Fraction` objects are added.
    - `a/b + c/d = (ad + bc) / bd`
    - Return a *new* `Fraction` object.
4. **Overload `==` (`__eq__`)**: Define how two `Fraction` objects are compared for equality.
    - Two fractions `a/b` and `c/d` are equal if `a*d == b*c`.
    - Consider simplifying fractions before comparison for more robust equality (e.g., 1/2 == 2/4). (Hint: `math.gcd` can help for simplification).
5. Create `Fraction` objects and demonstrate addition and equality checks.

Example Interaction:

```
Fraction 1: 1/2
Fraction 2: 1/3
Sum: 5/6

Fraction A: 1/2
Fraction B: 2/4
Fraction C: 3/5
```

```
A == B: True
A == C: False
```

Challenge 5: Employee Management (Inheritance and Data Hiding)

Goal: Design an `Employee` and `Manager` class demonstrating inheritance and simple data hiding concepts.

Concepts Covered: Inheritance, `super()`, "protected" attributes (_), simple methods for controlled access.

Requirements:

1. **`Employee` (Base Class):**
    - `__init__(self, employee_id, name, salary)`: Initialize these attributes. Use `_salary` (with a single underscore) to indicate it's intended for internal use/controlled access.
    - `get_employee_info()`: Returns a string with ID, Name, Salary.
    - `get_salary()`: Returns the `_salary` attribute.
    - `set_salary(new_salary)`: Updates `_salary` if `new_salary` is positive. Prints success/failure.
2. **`Manager` (Derived Class):**
    - Inherits from `Employee`.
    - `__init__(self, employee_id, name, salary, department)`: Initialize `employee_id`, `name`, `salary` using `super().__init__()`, and add `department`.
    - `assign_task(employee, task_description)`: Takes an `Employee` object and a task. Prints `f"Manager {self.name} assigned '{task_description}' to {employee.name}."`
    - `get_manager_info()`: Overrides `get_employee_info()` to include department. (Call `super().get_employee_info()` and append department).
3. Create an `Employee` and a `Manager` object.
4. Demonstrate:
    - Accessing employee info.
    - Setting manager's salary using the `set_salary` method.
    - Assigning a task from the manager to the employee.

Example Interaction:

```
--- Employee & Manager System ---
Employee 1: ID: E101, Name: John Doe, Salary: $50000.00
Manager 1: ID: M001, Name: Jane Smith, Salary: $80000.00, Department: Sales

Setting Manager Jane's salary to $85000.00...
Salary updated for Jane Smith. New salary: $85000.00

Manager Jane Smith assigned 'Complete Q3 Sales Report' to John Doe.

Updated Manager Info: ID: M001, Name: Jane Smith, Salary: $85000.00,
Department: Sales
```