Attributes, properties, fields state, variables, characteristics are all same thing

Object is a real world thing.

It has

1. Identity : Coffee mug
2. Behaviour: fill(), Empty(), clean()
3. Attributes : color, size, fullness

Nouns: Object

Verbs: Behaviours

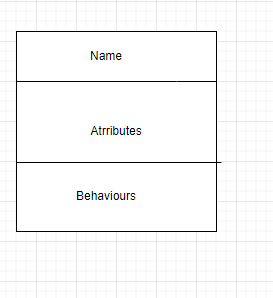
Class : it is a blueprint for creating objects

Class has:

1. Name
2. Attributes
3. Behaviours

Method: A block of code that can do some action and return some value

Methods are part of a class but functions can exist independently.



Object is an instance of a class.

Pillar of OOPS: APIE

1. Abstraction:  It’s the process of hiding the internal details of an application from the outer world and showing only the necessary features of an object. [**Abstract Classes and Methods**: In Python, we can achieve data abstraction by using abstractclasses and abstract methods1](https://www.geeksforgeeks.org/encapsulation-in-python/)[2](https://pynative.com/python-encapsulation/). [An abstract class is a class in which one or more methods are declared but not implemented1](https://www.geeksforgeeks.org/encapsulation-in-python/)[2](https://pynative.com/python-encapsulation/). [These methods are known as abstract methods and are declared using the @abstractmethod decorator](https://www.geeksforgeeks.org/encapsulation-in-python/)

**Real-life examples**: Your car and microwave are great examples of abstraction. You don’t need to know how the engine starts or how the microwave heats your food. The internal implementation and complex logic are completely hidden from you

1. Polymorphism :  [In OOP, polymorphism describes situations where you can access objects of different types through the same interface1](https://stackify.com/oop-concept-polymorphism/)[2](https://www.geeksforgeeks.org/polymorphism-in-java/)[3](https://www.codewithc.com/exploring-the-power-of-polymorphism-in-object-oriented-programming/). [Each type can provide its own independent implementation of this interface](https://stackify.com/oop-concept-polymorphism/)

* **Real-life examples**: A person can have different characteristics at the same time. For example, a man can be a father, a husband, and an employee. So the same person possesses different behaviors in different situations. [This is called polymorphis](https://www.geeksforgeeks.org/polymorphism-in-java/)m

Here are some key points about polymorphism in Python:

* [**Function Polymorphism**: Python supports function polymorphism where the same function can be used for different types1](https://www.geeksforgeeks.org/polymorphism-in-python/). [For example, the len() function can be used with many data types such as string, list, tuple, set, and dictionary1](https://www.geeksforgeeks.org/polymorphism-in-python/).
* **Operator Polymorphism**: Operators in Python can also exhibit polymorphism. [For example, the + operator performs arithmetic addition on numbers, concatenation on strings, and merge operation on lists1](https://www.geeksforgeeks.org/polymorphism-in-python/).
* [**Class Polymorphism**: Python allows different classes to have methods with the same name2](https://www.programiz.com/python-programming/polymorphism). [This allows us to use objects of different classes in the same way2](https://www.programiz.com/python-programming/polymorphism). [For example, if there is a make\_sound method in both Dog and Cat classes, we can call this method on an object without worrying about the class of the object2](https://www.programiz.com/python-programming/polymorphism).
* [**Polymorphism with Inheritance**: In Python, polymorphism lets us define methods in the child class that have the same name as the methods in the parent class1](https://www.geeksforgeeks.org/polymorphism-in-python/). [This is particularly useful when the method inherited from the parent class doesn’t quite fit the child class1](https://www.geeksforgeeks.org/polymorphism-in-python/). [In such cases, we re-implement the method in the child class1](https://www.geeksforgeeks.org/polymorphism-in-python/). [This process of re-implementing a method in the child class is known as Method Overriding](https://www.geeksforgeeks.org/polymorphism-in-python/)

1. Inheritance

Inheritance is a key concept in Object-Oriented Programming (OOP). [It allows you to create new classes that reuse, extend, and modify the behavior defined in another class1](https://www.codecademy.com/resources/blog/what-is-inheritance/)[2](https://www.geeksforgeeks.org/inheritance-in-java/)[3](https://www.geeksforgeeks.org/inheritance-in-c/). [The class whose members are inherited is called the **base class**, and the class that inherits those members is called the **derived class**1](https://www.codecademy.com/resources/blog/what-is-inheritance/)[2](https://www.geeksforgeeks.org/inheritance-in-java/)[3](https://www.geeksforgeeks.org/inheritance-in-c/).

Here are some key points about inheritance:

* [**Code Reusability**: Inheritance allows us to reuse code from the base class, reducing redundancy and making the code more manageable1](https://www.codecademy.com/resources/blog/what-is-inheritance/)[2](https://www.geeksforgeeks.org/inheritance-in-java/)[3](https://www.geeksforgeeks.org/inheritance-in-c/).
* [**Types of Inheritance**: There are several types of inheritance, including single, multiple, multilevel, and hierarchical1](https://www.codecademy.com/resources/blog/what-is-inheritance/)[2](https://www.geeksforgeeks.org/inheritance-in-java/)[3](https://www.geeksforgeeks.org/inheritance-in-c/).
* **Access Specifiers**: Access specifiers like public, private, and protected play a crucial role in inheritance. [They define the accessibility of the base class members in the derived class](https://www.codecademy.com/resources/blog/what-is-inheritance/)

1. Encapsulation

Encapsulation is a fundamental concept in Object-Oriented Programming (OOP), including C#. [It is the process of wrapping up data and methods into a single unit known as a class1](https://www.geeksforgeeks.org/c-sharp-encapsulation/)[2](https://dotnettutorials.net/lesson/encapsulation-csharp/)[3](https://www.scaler.com/topics/csharp/encapsulation-in-c-sharp/)[4](https://www.shekhali.com/csharp-encapsulation/)[5](https://www.c-sharpcorner.com/article/encapsulation-in-C-Sharp/). [This mechanism binds together the data and the functions that manipulate them1](https://www.geeksforgeeks.org/c-sharp-encapsulation/).

Here are some key points about encapsulation:

* [**Data Hiding**: Encapsulation hides the internal state and functionality of an object and only allows access through a public set of functions1](https://www.geeksforgeeks.org/c-sharp-encapsulation/)[2](https://dotnettutorials.net/lesson/encapsulation-csharp/)[3](https://www.scaler.com/topics/csharp/encapsulation-in-c-sharp/)[4](https://www.shekhali.com/csharp-encapsulation/)[5](https://www.c-sharpcorner.com/article/encapsulation-in-C-Sharp/). [The variables or data of a class are hidden from any other class and can be accessed only through any member function of its own class in which they are declared1](https://www.geeksforgeeks.org/c-sharp-encapsulation/)[2](https://dotnettutorials.net/lesson/encapsulation-csharp/)[3](https://www.scaler.com/topics/csharp/encapsulation-in-c-sharp/)[4](https://www.shekhali.com/csharp-encapsulation/)[5](https://www.c-sharpcorner.com/article/encapsulation-in-C-Sharp/).

Composition is often preferred over inheritance in object-oriented programming for several reasons:

**1. Flexibility and Reusability**

* **Composition** allows you to build complex objects by combining simpler ones. [This “has-a” relationship promotes flexibility and reusability, as you can easily swap out components without affecting the entire system1](https://en.wikipedia.org/wiki/Composition_over_inheritance).
* [**Inheritance**, on the other hand, creates a tight coupling between the parent and child classes, making it harder to change one without impacting the other2](https://www.digitalocean.com/community/tutorials/composition-vs-inheritance).

**2. Avoiding the Fragile Base Class Problem**

* [When using **inheritance**, changes in the base class can inadvertently affect all derived classes, potentially introducing bugs2](https://www.digitalocean.com/community/tutorials/composition-vs-inheritance).
* [**Composition** avoids this issue by containing instances of other classes, thus changes in one class do not directly impact others1](https://en.wikipedia.org/wiki/Composition_over_inheritance).

**3. Better Encapsulation**

* **Composition** keeps the internal details of each class hidden, promoting better encapsulation. [Each class manages its own behavior and state3](https://www.infoworld.com/article/2338602/composition-vs-inheritance-in-oop-and-c-sharp.html).
* [**Inheritance** can expose the internal details of the parent class to the child class, which can lead to tighter coupling and less encapsulation2](https://www.digitalocean.com/community/tutorials/composition-vs-inheritance).

**4. Simpler Hierarchies**

* **Composition** leads to simpler and more maintainable class hierarchies. [You can create complex behaviors by combining simple objects rather than creating deep inheritance trees4](https://www.prepbytes.com/blog/general/difference-between-composition-and-inheritance/).
* [**Inheritance** can lead to complex and deep hierarchies, which are harder to understand and maintain](https://en.wikipedia.org/wiki/Composition_over_inheritance)[2](https://www.digitalocean.com/community/tutorials/composition-vs-inheritance).

**5. Dynamic Behavior**

* [**Composition** allows for dynamic behavior changes at runtime by swapping out components1](https://en.wikipedia.org/wiki/Composition_over_inheritance).
* [**Inheritance** is static and determined at compile time, making it less flexible for dynamic behavior changes](https://en.wikipedia.org/wiki/Composition_over_inheritance)

In object-oriented programming, association represents a relationship between two classes where one class uses or interacts with another. It indicates how objects of one class are related to objects of another class, but it doesn’t necessarily imply ownership or hierarchical relationships like inheritance does.

**Explanation**

1. **Book Class**:
   * Represents a book with attributes like title and author.
   * Provides a \_\_str\_\_ method for a readable string representation of the book.
2. **Library Class**:
   * Manages a collection of Book objects using the books list.
   * Provides methods to add\_book to the library and list\_books to list all books.
3. **Association**:
   * **Library and Book**: The Library class has a list of Book objects, demonstrating an association. The Library does not own the Book class, but it interacts with Book objects.

Aggregation is a concept in object-oriented programming where one class (the container or aggregate) contains or uses objects of another class (the parts or components) to build a more complex structure. Unlike composition, aggregation represents a relationship where the lifetime of the contained objects is not strictly tied to the lifetime of the container. In other words, the contained objects can exist independently of the container.

**Example of Aggregation in Python**

Let's create a real-world example of aggregation using a Department and Employee. A Department can have multiple Employee objects, but employees are not exclusively tied to a specific department—they can potentially move between departments or exist independently.

**Explanation**

1. **Employee Class**:
   * Represents an employee with attributes like name and position.
   * Provides a \_\_str\_\_ method for a readable string representation of the employee.
2. **Department Class**:
   * Represents a department with a name and a list of Employee objects.
   * Provides methods to add\_employee to add employees to the department and list\_employees to list all employees in the department.
3. **Aggregation**:
   * **Department and Employee**: The Department class aggregates Employee objects. Employees can be added to multiple departments if needed, and their existence is not strictly dependent on the department.