OOPS

Object Oriented Programming using Python

A simple Analogy of OOPS

- Imagine you own a restaurant. Running a restaurant involves many different activities, like cooking food, serving customers, managing orders, and keeping the place clean. Now, instead of handling everything chaotically, you organize your restaurant by dividing these tasks into different roles—like chefs, waiters, and cleaners. Each role knows exactly what to do, making the restaurant run smoothly.

- **Objects are like staff members in the restaurant:**
- In OOP, everything is an object. Think of each staff member in your restaurant as an object. For example, the `Chef`, `Waiter`, and `Cleaner` are objects.
- **Classes are like job roles:**
- A class in OOP is like a job role in your restaurant. The `Chef` class defines what a chef is and what they can do, like cooking food.

Why Do We Need OOP?

- **Organization:** Just as organizing staff by roles makes your restaurant run smoothly, OOP helps organize your code, making it more manageable and efficient.
- **Reusability:** In your restaurant, you can hire multiple chefs using the same Chef job role template. Similarly, OOP allows you to reuse code by creating multiple objects from the same class, reducing redundancy.
- **Maintainability:** If you decide to change how something is done in your restaurant, like updating a recipe, you only need to update it in one place (the chef's instructions). In OOP, changes are easier to manage because related code is grouped together in classes.
- **Scalability:** As your restaurant grows and you hire more staff, the roles and responsibilities remain clear and organized. Similarly, OOP makes it easier to scale your program by keeping the code modular and well-structured.

Object-Oriented Programming System (OOPS)

- **Object-Oriented Programming System (OOPS)** is a programming paradigm that uses the concept of "objects" to design and structure software.
- It allows developers to model real-world entities and their interactions within a program by bundling data and the methods (functions) that operate on that data into a single unit called an "object."

Classes and Objects:

- **Analogy:** Think of a **class** as a recipe, and an **object** as the actual dish you cook using that recipe.
- **Class:** A recipe for making a chocolate cake. It tells you what ingredients you need and the steps to follow.
- **Object:** The chocolate cake you bake using the recipe. You can make many cakes (objects) from the same recipe (class), and each cake is an instance of that recipe.
- **Example:**
- **Class:** A blueprint for a house.
- **Object:** The actual house built using that blueprint. Different houses (objects) can be built from the same blueprint (class).

What Do Classes Do?

- *Classes* ***bundle data and functionality together*****.** This means that within a class, you can define attributes (data) and methods (functions) that operate on that data.
- When you create an object from a class (this process is called "instantiating" a class), the object can have its own unique set of attributes (like the color or size of a house built from a blueprint) and can perform actions defined by the methods in the class

Namespaces in OOP

- In OOP, you can think of a **namespace** as a place where names (like variable names, function names, class names) are stored. Every object, function, and class in Python has its own namespace.
- This is important in OOP because it allows you to organize and manage your code in a way that prevents naming conflicts.

Class as a Namespace:

- When you define a class, you are creating a new namespace. Inside this class, you can have attributes and methods. These are names that are specific to that class's namespace, meaning they can't be directly accessed from outside the class unless you use the proper syntax (e.g., `object.attribute`).

Scopes in OOP

- **Scopes** determine where a variable or function is accessible. In OOP, scope management is crucial for ensuring that variables are accessed and modified in the right context.
- **Local Scope in Methods:**
- When you define a method inside a class, it has its own local scope. Variables defined inside this method are only accessible within that method, not outside of it.
- **Global Scope in Modules:**
- Variables defined at the top level of a module are in the global scope. These are accessible throughout the module but are not directly accessible within a class or method unless you specifically import them or use the `global` keyword.

Scope and Namespace Resolution in OOP

- When you access an attribute or method in a class, Python follows a specific order to find where it's defined. This is similar to how Python handles scopes and namespaces:
- **Local Scope (inside the method):**
- Python first checks if the name exists in the local scope of the current method or function.
- **Enclosing Scope (inside the class):**
- If the name isn't found in the local scope, Python checks the enclosing scope, such as the containing class.
- **Global Scope (inside the module):**
- If it's not found in the class's namespace, Python looks in the global scope of the module where the class is defined.
- **Built-in Scope:**
- If the name is still not found, Python finally checks the built-in scope, where built-in functions like `print()` and `len()` are defined.