Summer of Code

Artificial Intelligence

(Machine Learning & Deep Learning)

Instructor
Wajahat Ullah
- *Research Assistant* (DIP Lab)

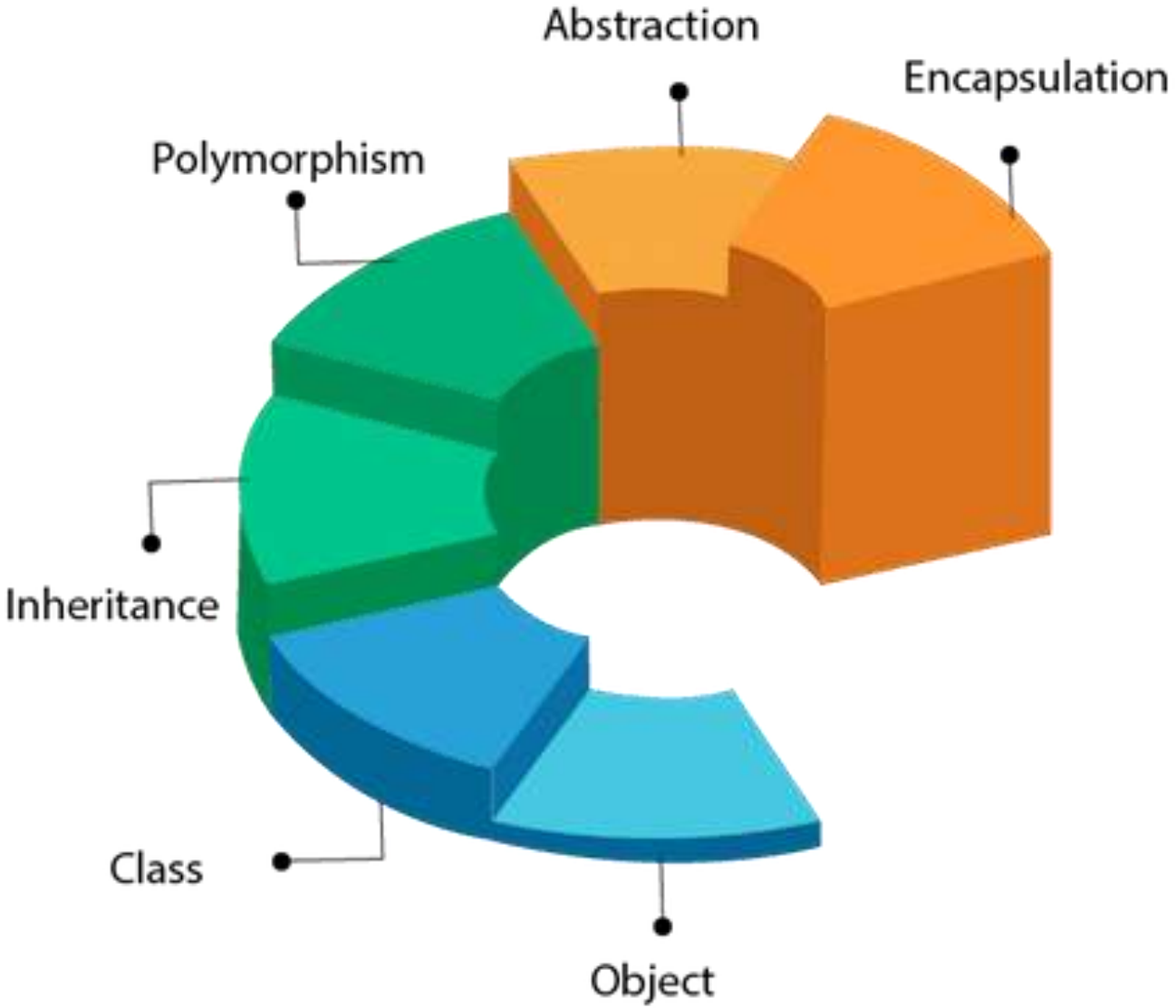
Duration
03 Months
(September – November)

Day 03 – Implementation of OOP (Classes and Objects)

Objectives:

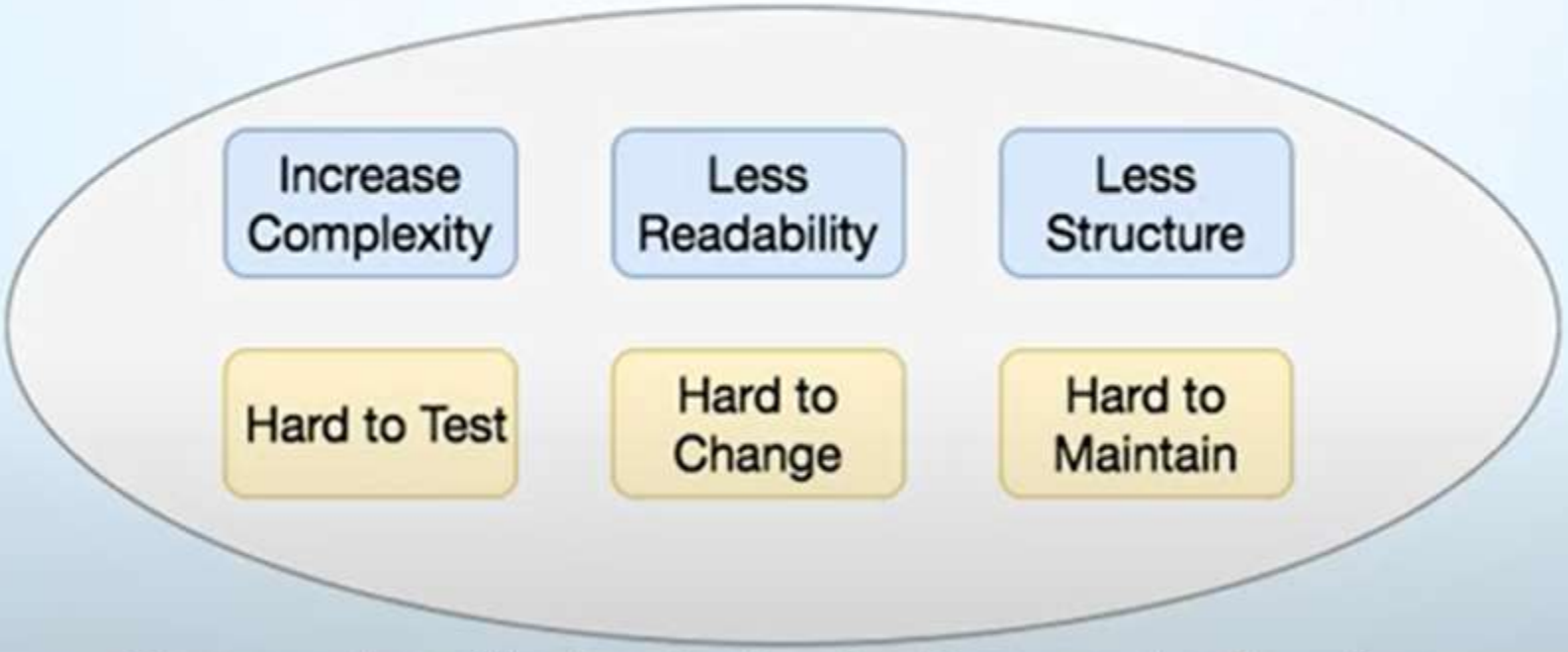
- Classes and Objects
- Instance Variables and Methods
- Class Variables
- Inheritance
- Special Methods

Object Oriented Programming (OOP)



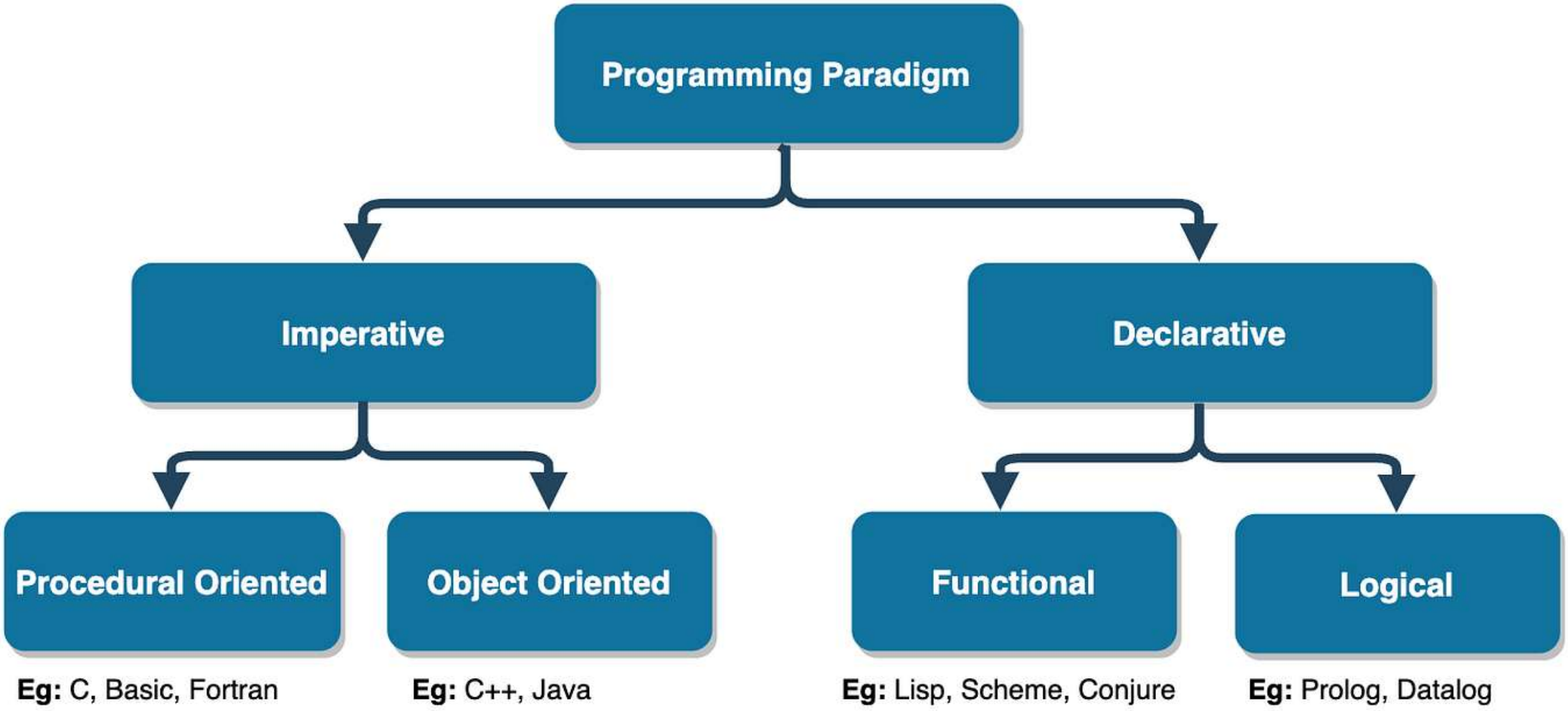
Why we need the Programming Paradigm?

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand"



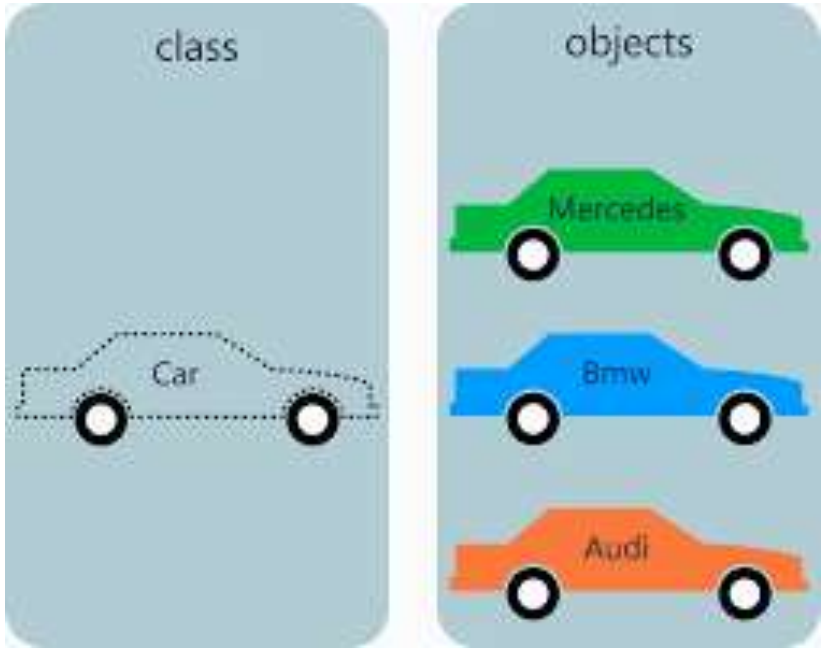
Programming without understanding Programming Paradigm

Different Programming Paradigms



Classes & Objects

- **Classes** are blueprints or templates for creating objects. Think of them as a real-world entity.
- **Objects** are instances of classes, representing a specific example of the class.
- Objects can store data (attributes) and have behavior (methods).

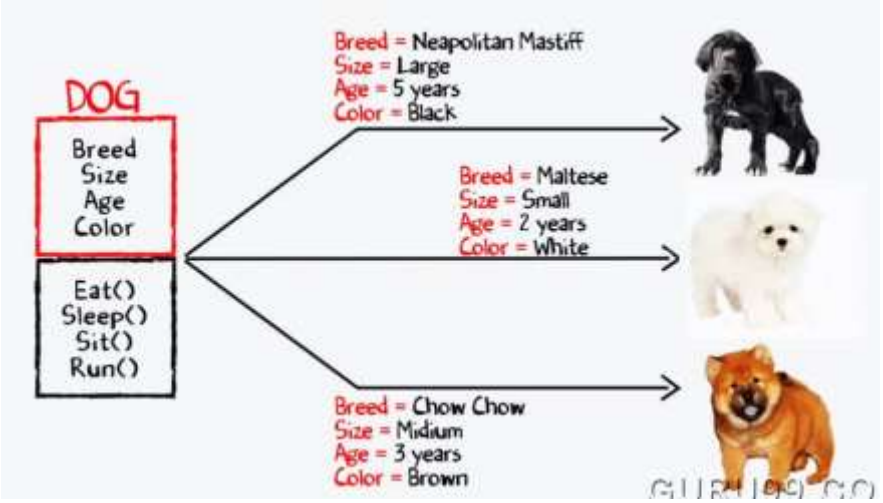


Example:

```
class Dog:
    pass
my_dog = Dog()
```

Key Concept:

- Each object shares the class structure but holds unique data.
- Classes improve modularity and reusability.
  - **Reusability:** Once a class is defined, you can create multiple objects from it.
  - **Modularity:** Encapsulation of data and methods within a class.



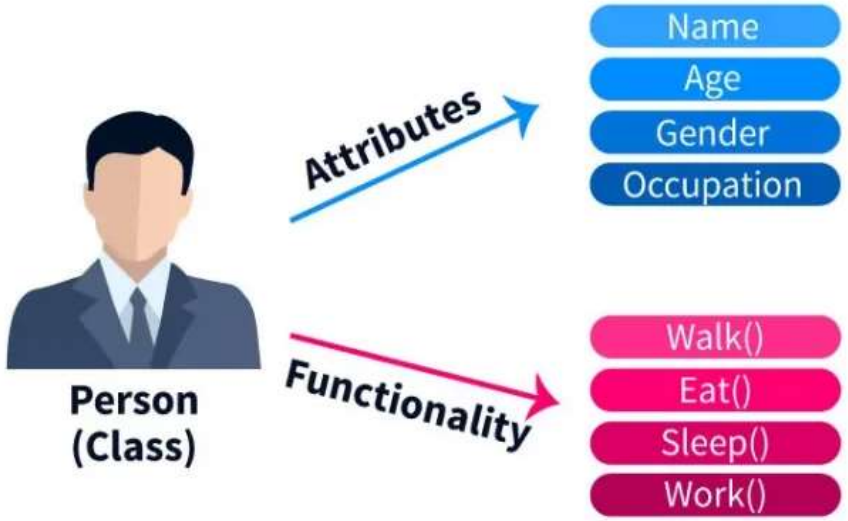
Attributes and Methods

Attributes (Properties):

- Variables that store the state of an object.

Methods:

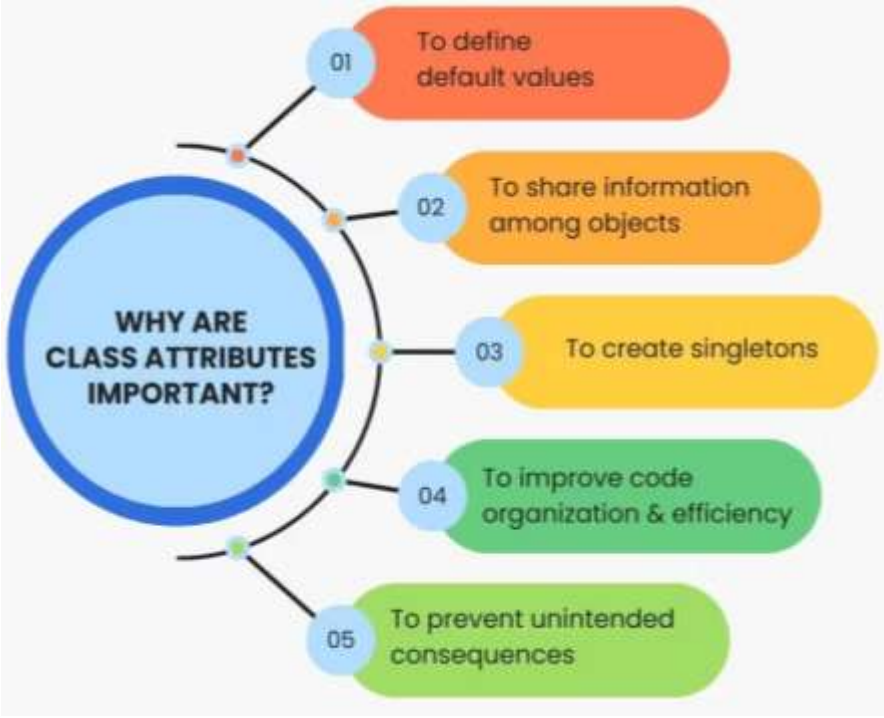
- Functions defined inside a class that describe the behaviors of the object.



Example:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name  # Attribute
        self.breed = breed  # Attribute

    def bark(self):  # Method
        print(f"{self.name} is barking!")
```

- **Methods' Role**: Operate on the instance's data and can change the object's state.
- **Attributes' Role:** Represent the object's characteristics (e.g., name, breed, age).



The `__init__` Method (Constructor)

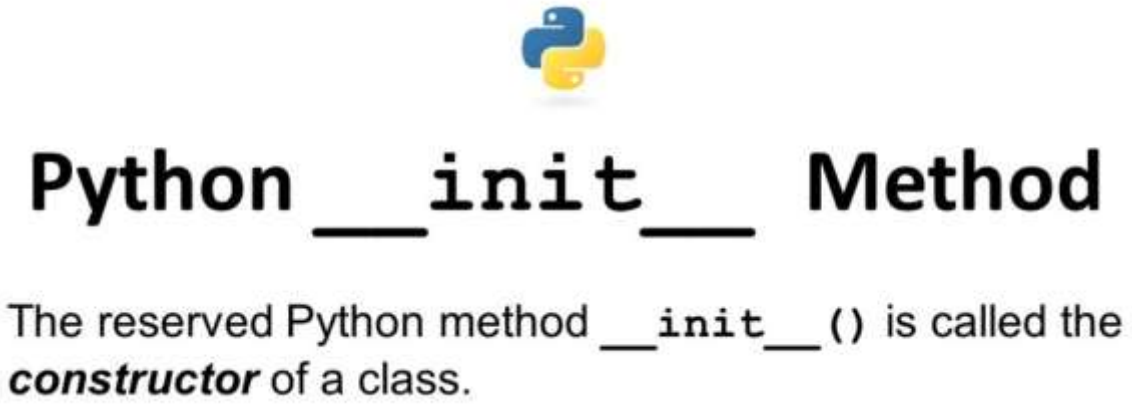
- Special method automatically called when a new object of the class is created.

Purpose:

- Used to initialize the object's attributes.
- The `__init__` method sets the initial values for attributes like name and age upon object creation.

Example:

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



- **self keyword:**
  - Refers to the instance of the class and allows access to its attributes and methods.

Class vs Instance Attributes

Instance Attributes:

- Specific to a particular instance/object, defined within `__init__`

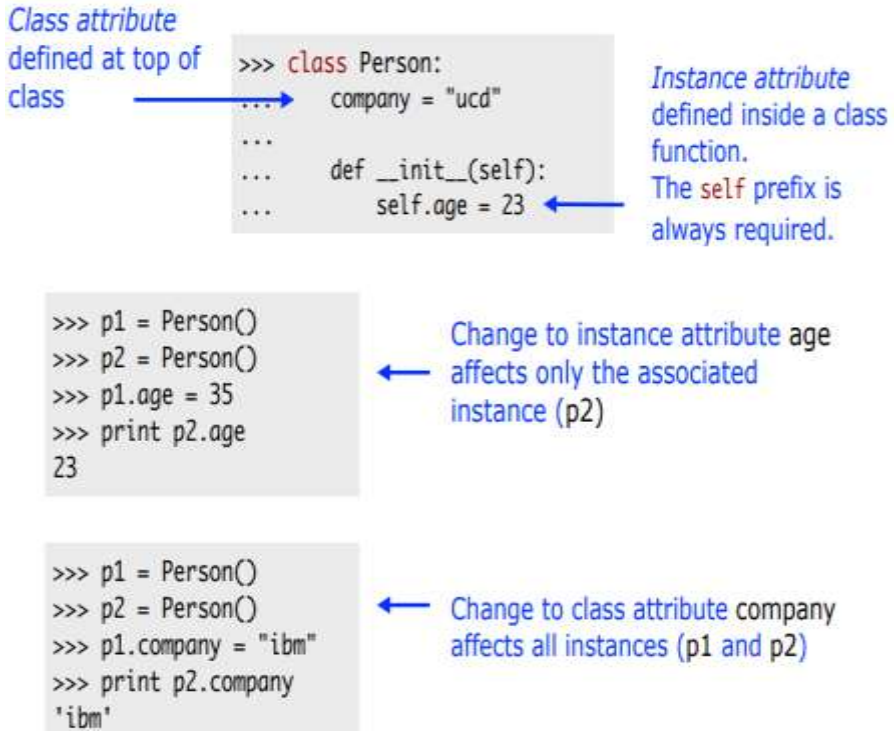
Class Attributes:

- Shared across all instances, defined directly within the class.

Example:

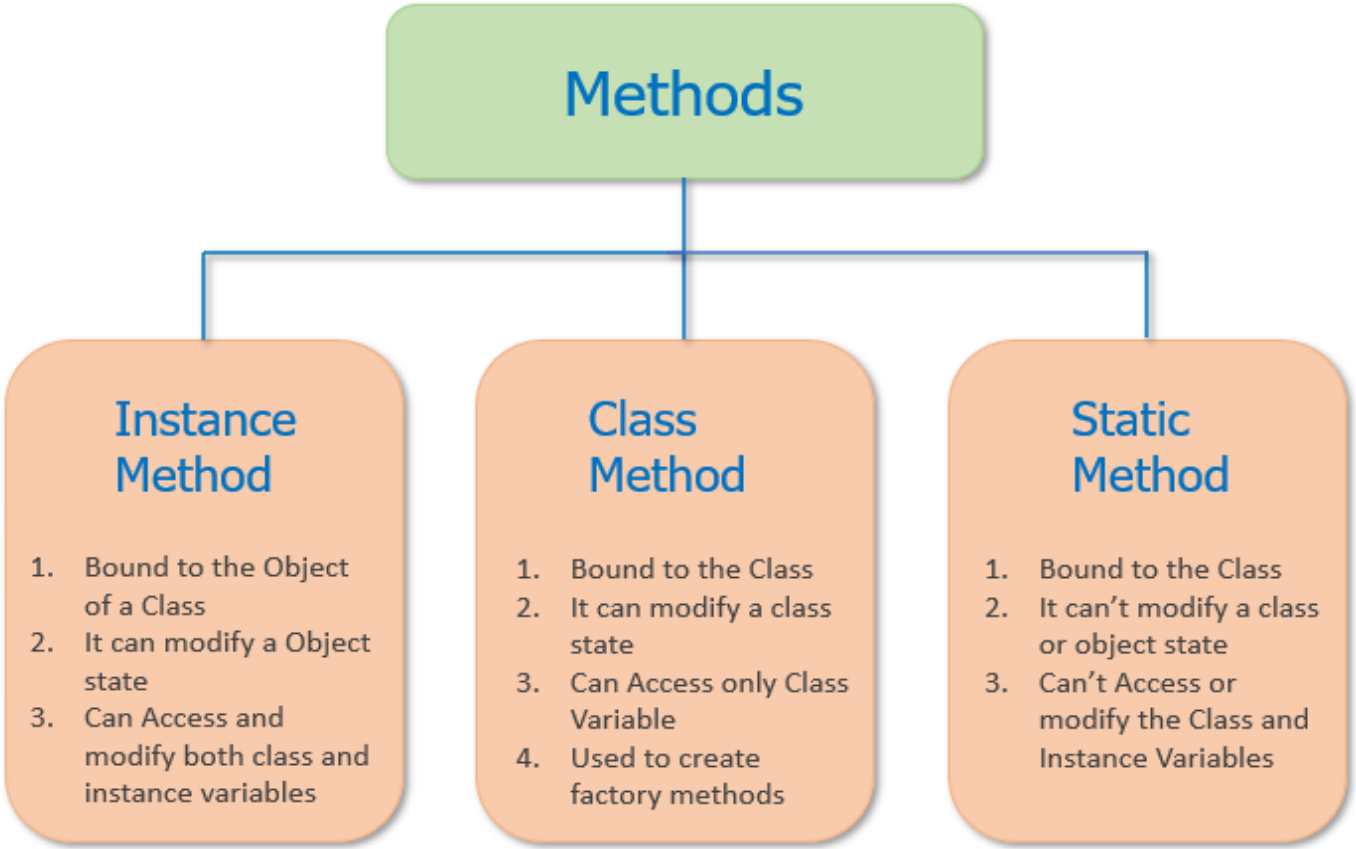
```
class Dog:
    species = 'Canine'  # Class Attribute
    def __init__ (self, name, age):
        self.name = name  # Instance Attribute
        self.age = age
```

- Class attributes are the same for all objects created from that class.
- Instance attributes vary from object to object.



Types of Methods

- **Instance Methods:** Operate on an instance of the class and access/modify instance attributes.
- **Class Methods:** Operate on the class itself and have access to class attributes.
- **Static Methods:** Standalone functions within a class. Don't operate on instances or the class itself.

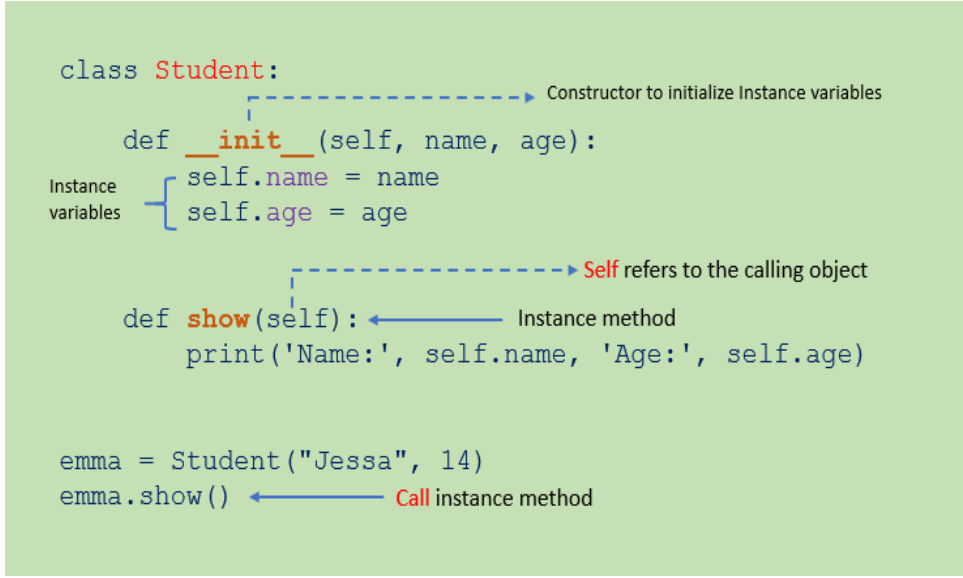


Instance Methods

- Operate on individual objects of the class and typically modify instance attributes.
- Requires `self` as the first parameter to refer to the specific instance.
- **Example:**

```
class Dog:
    def __init__ (self, name):
        self.name = name
    def bark(self):
        print(f"{self.name} barks!")
```

- **Explanation:** `self` allows each instance to access its own data.
- **Key Points:**
  - Used to manipulate or retrieve instance-specific data.
  - Can modify object state.



Class Methods

- Methods that operate on the class itself and typically use `@classmethod` decorator.
- Uses cls as the first parameter to refer to the class.
- **Example:**

```
class Dog:
    species = ‘Canine’
    @classmethod
    def show_species(cls):
        print(f”All dogs are {cls.species}.”)
```

- **Explanation:** Class methods are often used for operations that affect the class as a whole, like modifying class-level attributes.

Syntax of a Python Class:

[ ]:

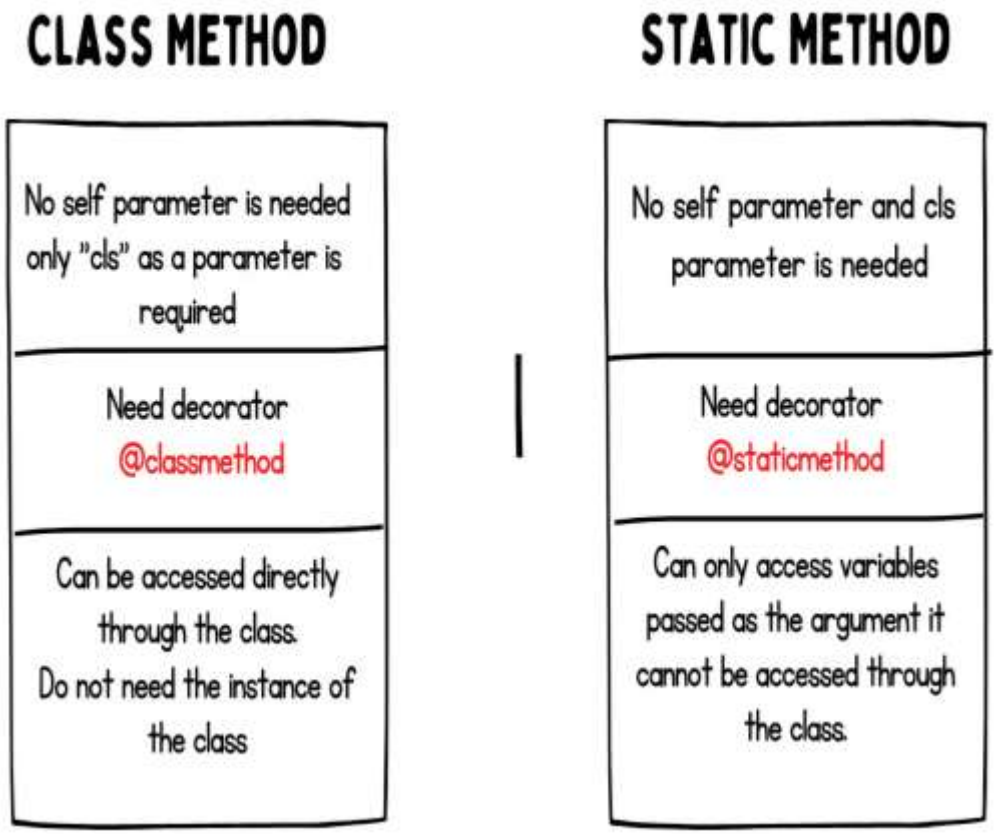
```
class ClassName:
    # Class variables
    class_variable = value

    # Constructor
    def __init__(self, parameters):
        self.instance_variable = parameters

    # Instance method
    def method_name(self, parameters):
        # method body

#clcoding.com
```

Static Methods

- A static method does not depend on instance or class; behaves like a regular function, but it is inside a class.
- Uses the `@staticmethod` decorator.
- **Example:**

```
class Dog:
    @staticmethod
    def info():
        print("Dogs are loyal animals.")
```

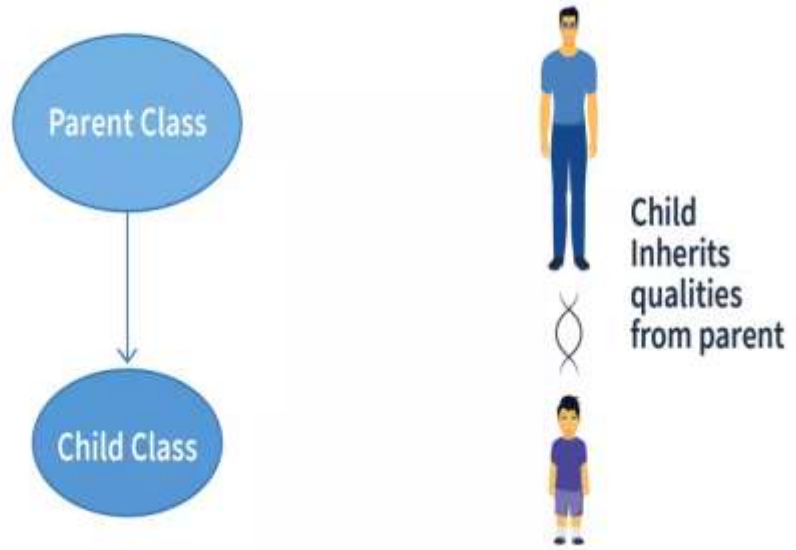
- **Explanation**: Static methods are used for utility functions within the class that don't need access to class or instance attributes.
- **Key Points:**
  - No self or cls arguments required.
  - Useful for grouping logically related functions with the class, even if they don't modify object or class state.



Object-Oriented Programming (OOP)
Inheritance

Inheritance

- Inheritance is a fundamental concept in **OOP** that allows one class (**the child class**) to inherit the properties and behaviors (methods) of another class (**the parent class**).

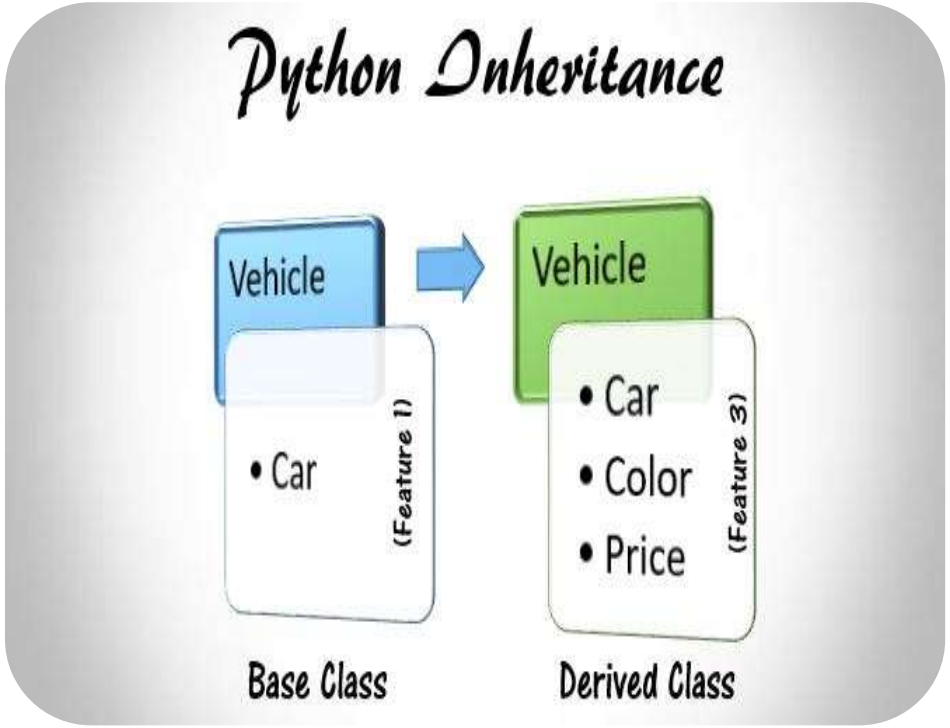


Analogy:

- Consider inheritance as a family tree, where traits and behaviors are passed down from parents to children.

Purpose:

- Code Reusability
- Extend properties and functionality of an existing class in new class.
- No need to repeat code in sub classes.
- It allows for a more organized and maintainable code structure.

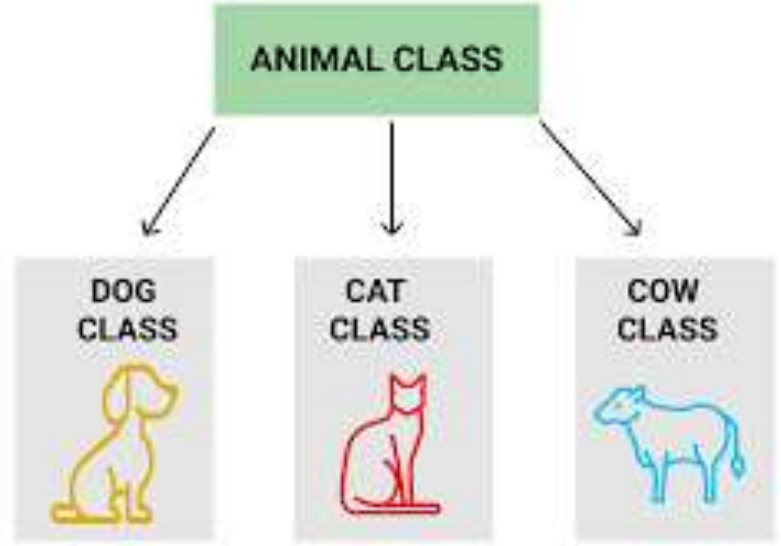


Parent Class (Super Class)

- Serves as a base class for other classes to inherit from.
- Can define:
  - **Attributes** (e.g., *species, age*)
  - **Methods**: (e.g., make_sound())

Example

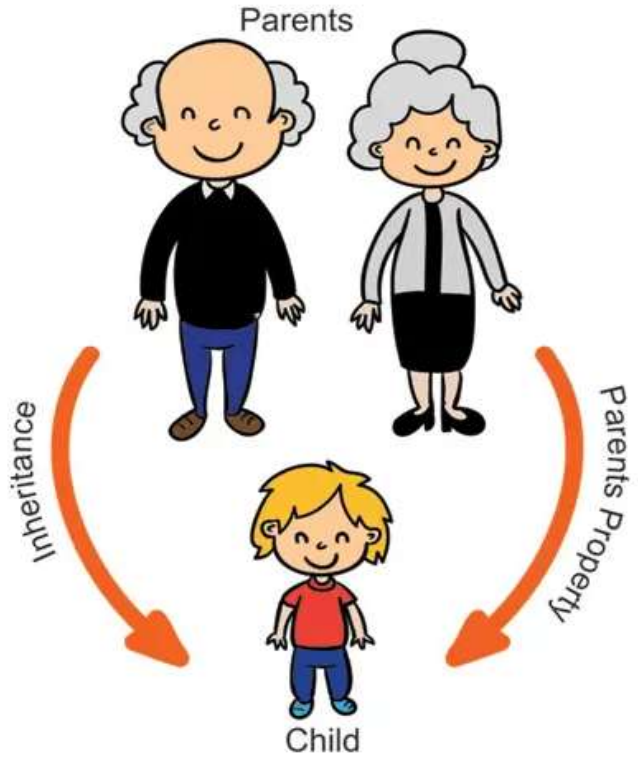
```
class Animal:  # Parent Class
    def __init__(self, species, age):
        self.species = species
        self.age = age
    def make_sound(self):
        return “Some sound”
```




```
# Parent class
class Animals:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Child Class (Sub class)

- Inherits properties and methods from the parent class.
- It can also have additional attributes and methods or override existing ones.



Characteristics:

- **Inheritance:** Inherits attributes and methods from the parent class.
- **Overriding**: Can redefine methods to provide specific implementations.

Example:

- **Class Name**: Dog (inherits from Animal)
- **Additional Attribute**: breed
- **Overridden Method**: make_sound() to return "Bark!"

```
class Dog(Animal):
  def __init__(self, species, age, breed
    super().__init__(species, age)
    self.breed = breed
  def make_sound(self):
    return "Bark!"
```

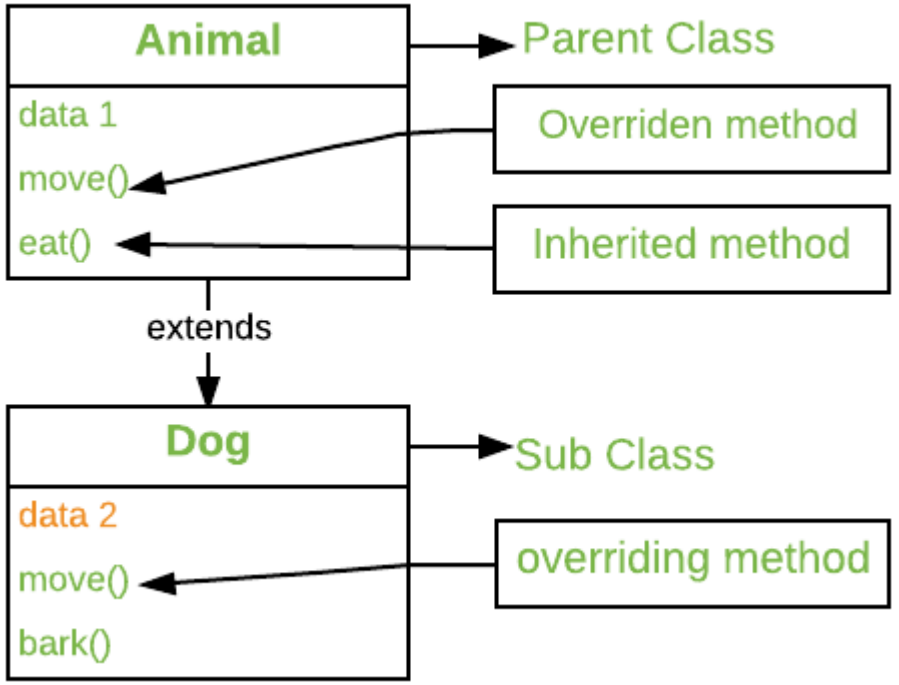
```
# Child class that inherits from the parent class
class Dog(Animals):
    def __init__(self, name, age, color):
        # Using super to access methods of the parent class
        super().__init__(name, age)
        self.color = color
```

Overriding Methods

- A child class can override a method from the parent class by redefining it.
- Enables polymorphism: Same method name behaves differently across classes.

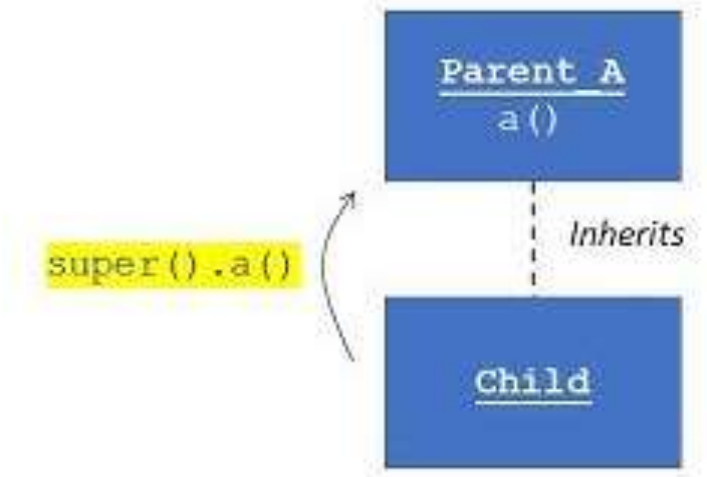
Example:

- In the Dog class, the `make_sound()` method is overridden to return "Bark!" instead of the generic "Some sound" from the Animal class.



Using `super()`:

- `super()` lets the child class access methods of the parent class, especially useful in constructors.
- Keeps the code clean and maintainable by avoiding hardcoding the parent class name.

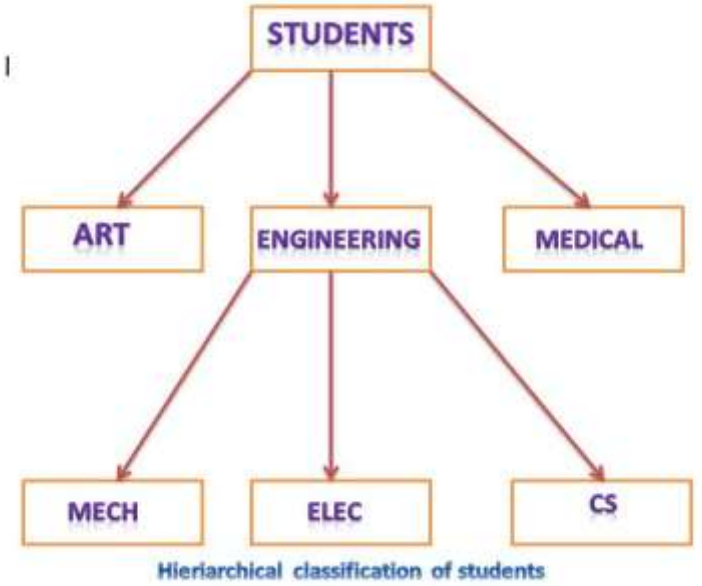


Hierarchical Inheritance

- Multiple child classes inherit from the same parent class.

Characteristics:

- Allows for shared attributes and methods from a common parent class.
- Promotes code reusability.
- Different child classes can have additional, unique functionality.

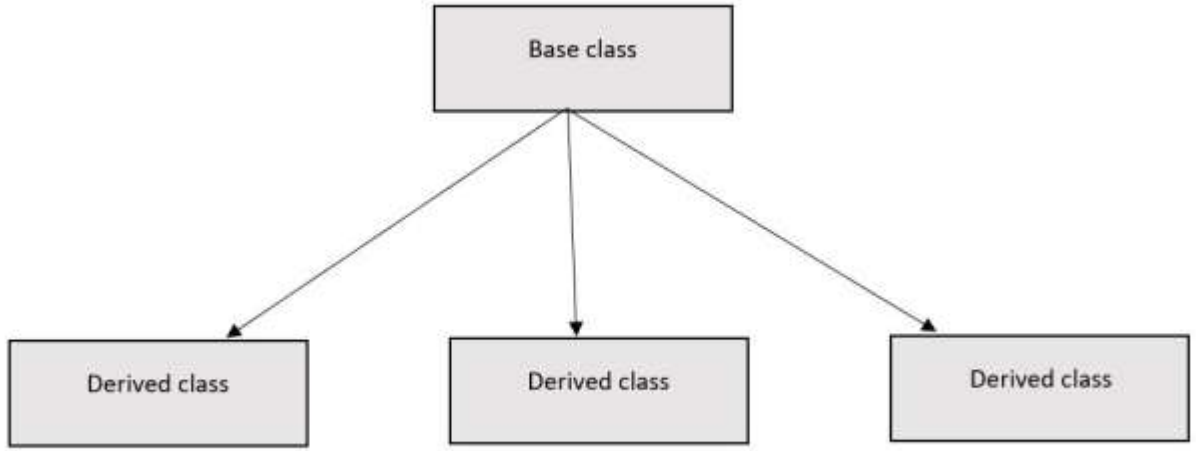


Hierarchical classification of students

Example:

- **Parent Class**: Animal
- **Child Classes**: Dog, Cat

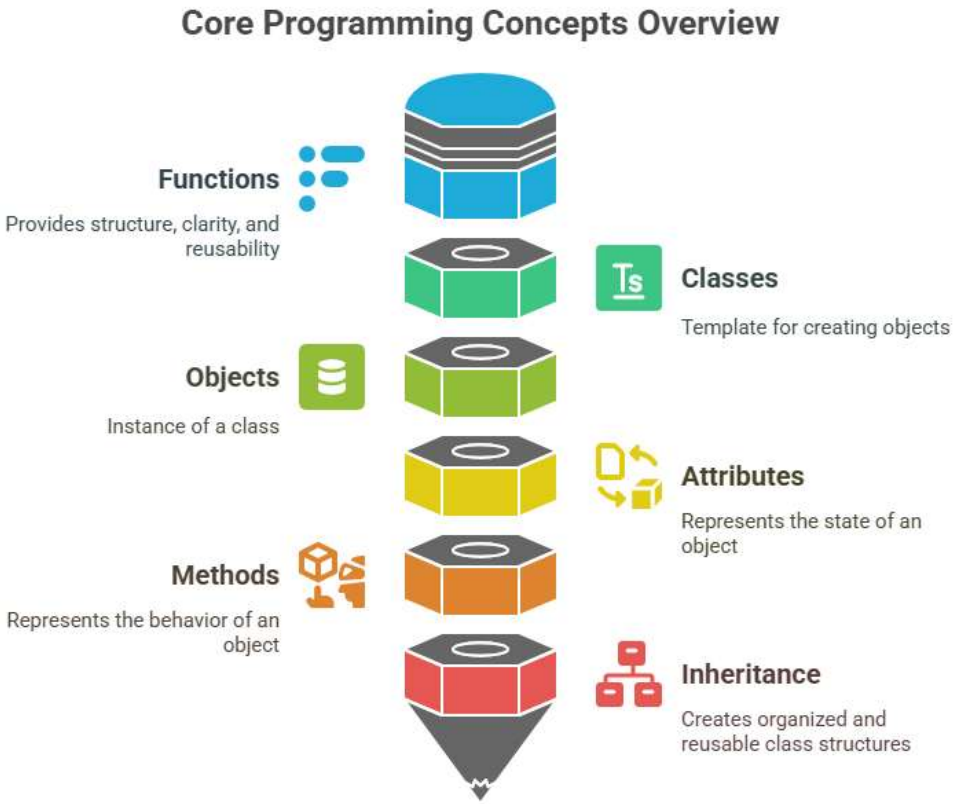
```
class Animal:
   def make_sound(self):
      return “Some sound”
class Dog(Animal):
   def bark(self):
      return “Woof!”
class Cat(Animal):
   def meow(self):
      return “Meow!”
```



Key Takeaways

- **Functions** provide structure, clarity, and reusability.
- Learn to use `def`, *return*, *arguments*, and *scope* effectively.
- Use *positional*, *keyword*, and *variable-length* arguments where appropriate.

- **Class:** Template for objects.
- **Object:** Instance of a class.
- **Attributes:** Represent state.
- **Methods:** Represent behavior.
- **Types of Methods:** Instance, Class, Static.
- **`__init__`:** Constructor for initialization.



- **Inheritance** helps create an organized, reusable, and extendable class structure.
- Use inheritance to model "is-a" relationships (e.g., A Dog is an Animal).
- Choose the right type of inheritance based on design needs.
- Be cautious with multiple and hybrid inheritance due to increased complexity.

Happy Coding