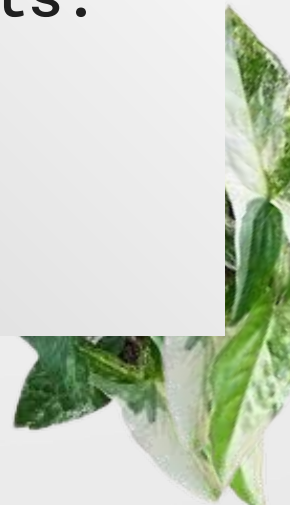
# Python Object Oriented Programming

*Pillars of OOP : A Beginner's Guide*

BS. Information Technology Department
Prof. Leonard Andrew Mesiera

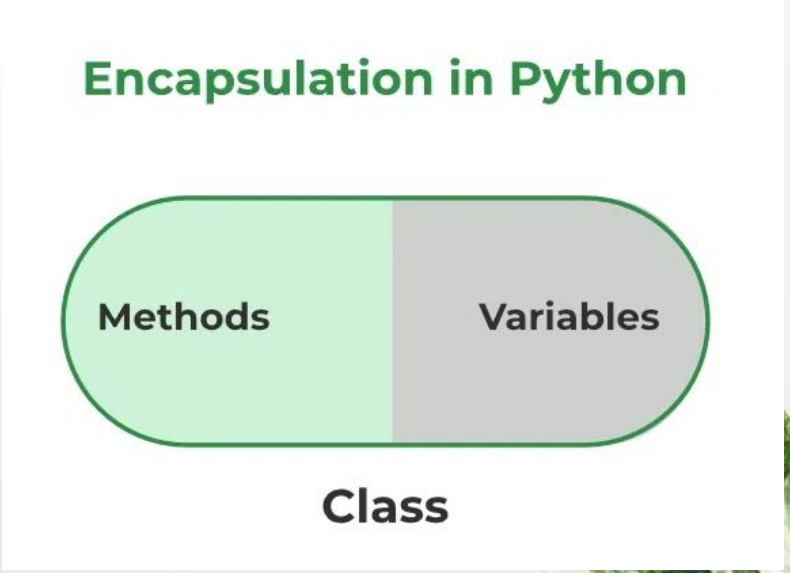
# What is OOP?

- OOP is a programming paradigm that organizes code around objects rather than functions.
- Objects are entities that encapsulate data (attributes) and behavior (methods).
- Classes are blueprints for creating objects.

# Encapsulation

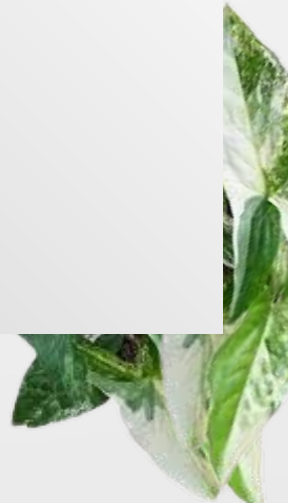
- Encapsulation is the process of hiding data and methods within an object.
- It protects data from external interference and promotes code modularity.
- Python's use of access modifiers (public, private, and protected) supports encapsulation.



# Why Encapsulation is Important in OOP?

Benefits:

- Data Hiding: Restricting access to the internal details of an object.
- Modularity: Easier maintenance and updates since changes are localized.
- Security: Controlling access to sensitive data.

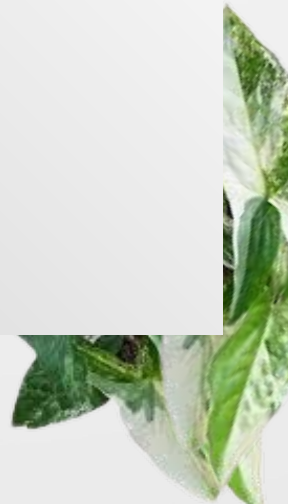



# Encapsulation Components

- Attributes (Properties):
  - Data members that store information.
- Methods (Functions):
  - Operations or behaviors that can be performed on the object.

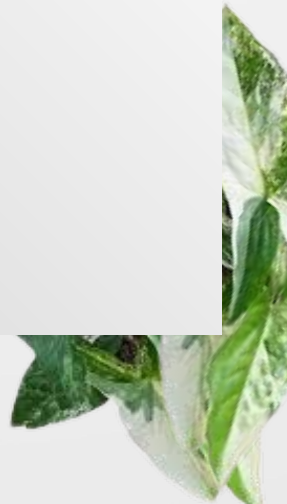
# Access Modifiers

- Public:
  - Accessible from outside the class.
- Private:
  - Accessible only within the class.
- Protected:
  - Accessible within the class and its subclasses.

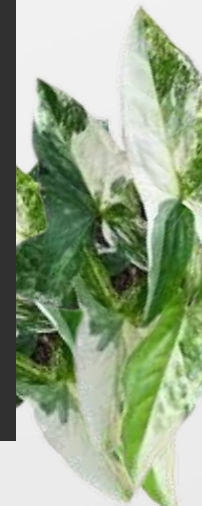
# How to implement encapsulation in Python?

- Private Attributes:
  - Prefix attributes with double underscores (__) to make them private.
- Getter and Setter Methods:
  - Provide controlled access to private attributes.

# How to implement encapsulation in Python?

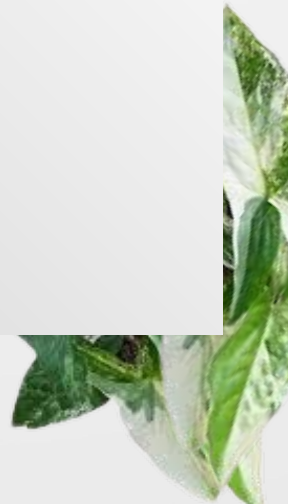
```
class MyClass:
    def __init__(self):
        self.__private_attribute = 0

    def get_private_attribute(self):
        return self.__private_attribute

    def set_private_attribute(self, value):
        if value >= 0:
            self.__private_attribute = value
        else:
            print("Invalid value. Must be non-negative.")
```



# Examples of Encapsulation

- Creating an Object:
  - Instantiate the class.
- Accessing Attributes:
  - Use getter and setter methods.

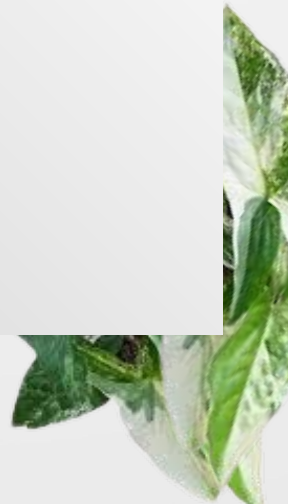
```
obj = MyClass()

obj.set_private_attribute(42)

print(obj.get_private_attribute())

# Output: 42
```

# Encapsulation and Property Decorators

Property Decorators:

- An alternative to getter and setter methods.
- Provides a more Pythonic way to encapsulate attributes.

# Encapsulation and Property Decorators

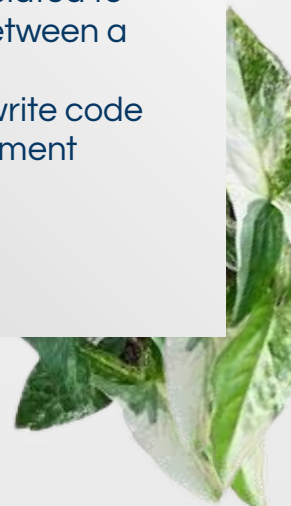
```
class MyClass:
    def __init__(self):
        self.__private_attribute = 0

    @property
    def private_attribute(self):
        return self.__private_attribute

    @private_attribute.setter
    def private_attribute(self, value):
        if value >= 0:
            self.__private_attribute = value
        else:
            print("Invalid value. Must be non-negative.")
```

# Inheritance

Inheritance is a feature of object-oriented programming (OOP) that allows you to create new classes based on existing classes. This allows you to reuse code and create hierarchies of classes that are related to each other.

Inheritance is important in programming because it can help you to:

- Write more reusable code. By creating new classes that inherit from existing classes, you can reuse the code that is already written in the existing classes. This can save you a lot of time and effort, and it can also help to make your code more maintainable.
- Create hierarchies of classes. Inheritance allows you to create hierarchies of classes that are related to each other. This can be useful for modeling real-world relationships, such as the relationship between a parent class and a child class.
- Implement polymorphism. Polymorphism is another important feature of OOP. It allows you to write code that can handle different types of objects in a consistent way. Inheritance can be used to implement polymorphism by allowing you to create classes that inherit from a common base class.

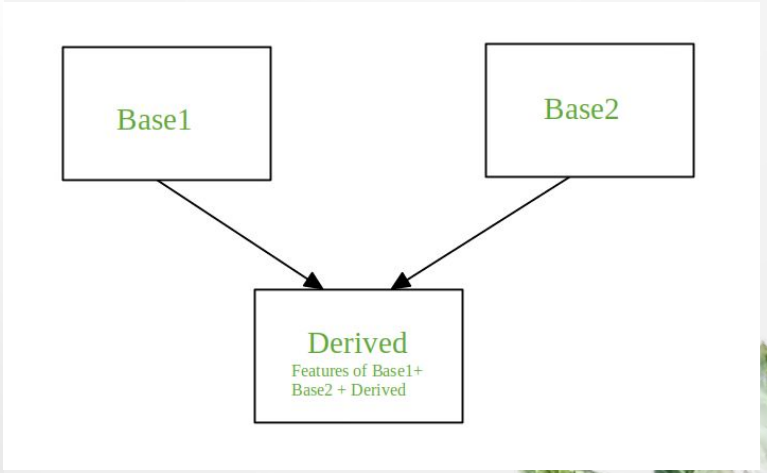
# Code Demo : Inheritance

```
class Animal:
    def make_sound(self):
        pass

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

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

# Create a new Dog object
dog = Dog()

# Create a new Cat object
cat = Cat()

# Call the make_sound() method on each object
dog.make_sound()
cat.make_sound()
```

# OOP Pillar #2: INHERITANCE

# Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that enables code reuse and organization. It allows programmers to create new classes based on existing classes, inheriting their attributes and methods. This powerful feature promotes code efficiency and maintainability, making it widely used in software development.




Benefits of Inheritance

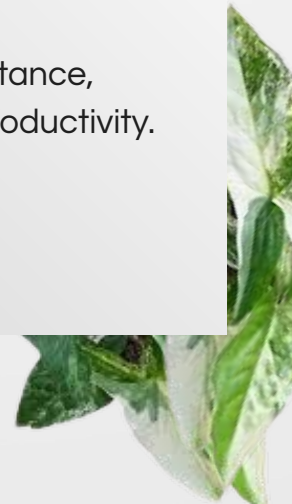
- Code Reuse: Inheritance eliminates the need to rewrite common code, reducing development time and effort.
- Organized Code Structure: Inheritance promotes a hierarchical organization of classes, making code easier to understand and maintain.
- Flexibility: Inheritance enables the creation of specialized classes tailored to specific needs without duplicating code.

# Inheritance

Real-World Applications of Inheritance

- **Graphical User Interfaces (GUIs):** Buttons, menus, and windows inherit common properties and behaviors from a base class.
- **Data Structures:** Lists, arrays, and trees inherit basic operations like insertion, deletion, and searching from a parent class.
- **Game Development:** Characters, objects, and environments inherit attributes and behaviors from their respective base classes.

Inheritance is an essential tool for building robust and efficient software systems. By leveraging inheritance, programmers can create reusable and maintainable code, enhancing overall software quality and productivity.

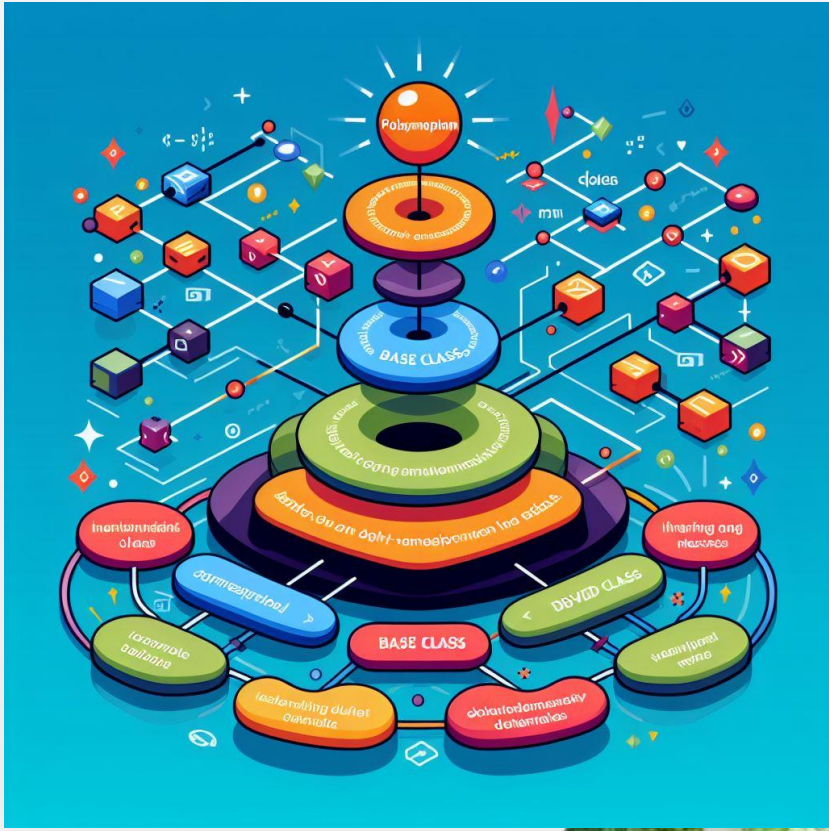
# Day5_activity5.py part 1

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

    def speak(self):
        print(f"{self.name} is an {self.species} and makes a sound.")

# Subclass inheriting from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        # Call the constructor of the base class (Animal) using super()
        super().__init__(name, species="Dog")
        self.breed = breed

    def speak(self):
        print(f"{self.name} is a {self.breed} dog and barks.")
```

# Day5_activity5_2.py part 2

```
# Subclass inheriting from Animal
class Cat(Animal):
    def __init__(self, name, color):
        # Call the constructor of the base class (Animal) using super()
        super().__init__(name, species="Cat")
        self.color = color

    def speak(self):
        print(f"{self.name} is a {self.color} cat and meows.")

# Create instances of the subclasses
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "Gray")

# Call methods on the objects
dog.speak()  # Output: Buddy is a Golden Retriever dog and barks.
cat.speak()  # Output: Whiskers is a Gray cat and meows.
```

# Polymorphism



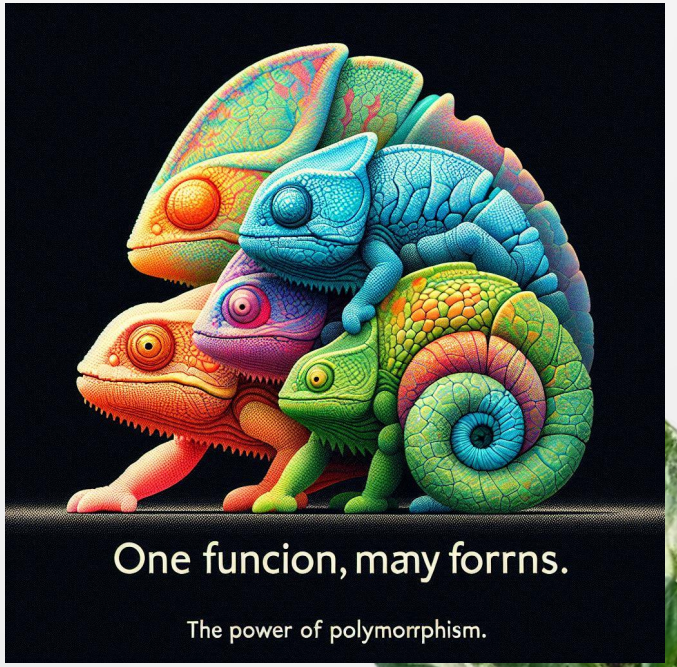
# What is Polymorphism

- Imagine a chef with one tool for everything. Not practical, right?
- Polymorphism is like having a magic utensil that adapts to different ingredients.
- In programming, it's the ability of one thing to take on different forms or behaviors.
- One Class, Many Forms

# One Function, Many Types

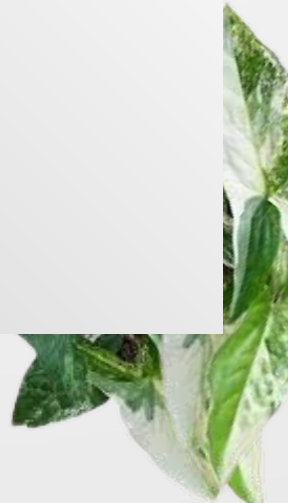
- Think of a function as a recipe, and the ingredients are the data it works with.
- **Polymorphism** lets you use the same recipe with different ingredients (numbers, strings, lists), and it adapts automatically.
- Example:

  print function works with all data types, displaying each one correctly.



# One Interface, Many Objects

- Imagine a universal kitchen tool for stirring, scooping, and spreading.
- An interface is like a set of instructions for different objects to follow.
- Polymorphism lets objects of different types implement the same interface, but each with its own twist.
- Example: Shape interface with area and perimeter methods. Circles, squares, and triangles implement it differently.

# Polymorphism - Additional Analogy

- Imagine one car key unlocking different car models (sedan, SUV, truck).
- The key works the same, but driving each car is different. That's polymorphism!

# Benefits of Polymorphism

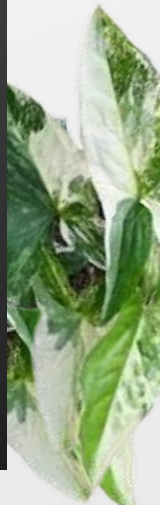
- Code Reusability: Write generic code that works with different data types or objects.
- Flexibility and Maintainability: Easy to adapt and change code as your program evolves.
- Cleaner Code: Avoids repetitive if-else statements, making code more concise and readable.

# Get Polymorphic !

- Polymorphism is a powerful tool
- Remember the kitchen analogy and keep practicing!
- It's like a magic chef's trick that makes your code sing!

```
class Animal:
    def speak(self):
        pass

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

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

class Duck(Animal):
    def speak(self):
        return "Quack!"
```

```
# Function that demonstrates polymorphism
def animal_says(animal):
    return animal.speak()

# Create instances of different animals
dog = Dog()
cat = Cat()
duck = Duck()

# Call the function with different types of animals
print(animal_says(dog))  # Output: Woof!
print(animal_says(cat))  # Output: Meow!
print(animal_says(duck))  # Output: Quack!
```



# Explanation

- We define a base class Animal with a method speak().
- We create three derived classes (Dog, Cat, and Duck) that inherit from the Animal class and override the speak() method.
- The animal_says() function takes an object of type Animal and calls its speak() method.
- When we call animal_says() with different types of animals, it demonstrates polymorphism. Each animal object responds to the speak() method in its own way.

Polymorphism allows us to treat objects of different classes uniformly when they share a common interface (in this case, the speak() method). This flexibility makes the code more extensible and easier to maintain.

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

    def speak(self):
        pass

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

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

class Duck(Animal):
    def speak(self):
        return "Quack!"
```

```python
# Function that demonstrates polymorphism with a list of animals
def animal_info(animals):
    for animal in animals:
        print(f"{animal.name} says: {animal.speak()}")

# Create instances of different animals
dog = Dog("Buddy")
cat = Cat("Whiskers")
duck = Duck("Daffy")

# Create a list of animals
animal_list = [dog, cat, duck]

# Call the function with the list of animals
animal_info(animal_list)
```

# Explanation

- Each `Animal` has a `name` attribute, and we initialize it through the `__init__` method.
- The `animal_info()` function takes a list of animals and uses polymorphism to call the `speak()` method for each animal in the list.
- We create instances of `Dog`, `Cat`, and `Duck` with specific names.
- These instances are added to a list called `animal_list`, which includes animals of different types.
- The `animal_info()` function is then called with the list, and it prints out what each animal says.

This example demonstrates how polymorphism allows us to work with objects of different types through a common interface, making the code more flexible and scalable.