

✓ Object-Oriented Programming (OOP) in Python 1

Agenda

1. Introduction to OOP

- What is OOP?
- Why OOP?

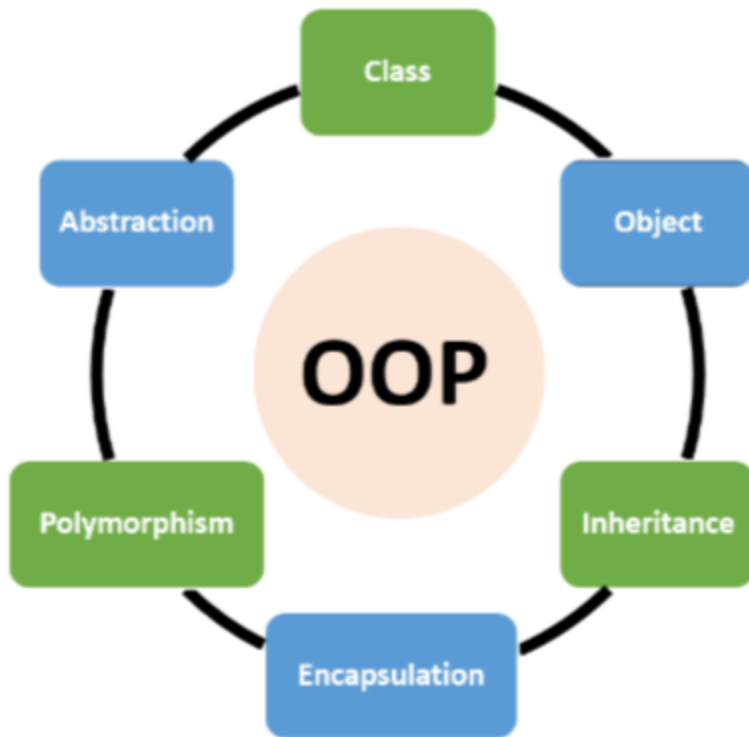
2. Core Concepts of OOP

- Classes and Objects
- Attributes and Methods
- Inheritance
- Encapsulation
- Polymorphism
- Data Abstraction

✓ Introduction to OOP

What is OOP?

- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects". It allows you to model real-world entities and their interactions in a structured and modular way.
- The main concept of object-oriented Programming (OOPs) or oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data.



Why OOP?

- OOP enhances code organization, reusability, flexibility, and maintainability, making it a valuable approach for developing complex and scalable software systems.
- Example: Imagine building a car simulation. Using OOP, you can create a Car class with attributes like color, model, and speed, and methods like start(), stop(), and accelerate(). This approach is more organized and efficient than procedural programming, where you might have a bunch of unrelated functions to handle car-related operations.

✓ Core Concepts of OOP

✓ Classes and Objects

✓ What is a Class?

- A class in Python is a blueprint or template for creating objects. It encapsulates data for the object and methods to manipulate that data. Classes are a way to bundle data and functionality together.
- Key Points:
 - A class can contain attributes (variables) and methods (functions).

- It defines the properties and behaviors of the objects that are created from it.

```
# Class example
class Dog:
    species = "Canis familiaris" # Class attribute

    def __init__(self, name, age):
        self.name = name # Instance attribute
        self.age = age

    def bark(self):
        return f"{self.name} says woof!"

# Creating an instance of Dog
my_dog = Dog("Buddy", 3)
print(my_dog.bark())
```

 Buddy says woof!

✓ What is an Object?


- An object is an instance of a class. When a class is defined, no memory is allocated until an object of that class is instantiated. Each object can have its own attributes and methods, defined by the class.
- Key Points:
 - Objects are individual instances of a class.
 - Each object can hold its own unique data and behavior, as defined by the class.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start(self):
        return f"{self.make} {self.model} of {self.year} started."

# Creating objects of the Car class
car1 = Car("Toyota", "Camry", 2020)
car2 = Car("Honda", "Civic", 2019)

print(car1.start()) # Output: Toyota Camry of 2020 started.
print(car2.start())
```

 Toyota Camry of 2020 started.
Honda Civic of 2019 started.

✓ Attributes and Methods

✓ Attributes

- Attributes are variables that are associated with a class or an instance of a class. They represent the state or properties of an object.
- Types of Attributes:-

1. Class Attributes:

- Shared across all instances of a class.
- Defined directly within the class, outside of any methods.

#Example of Class Attributes

```
class Dog:
    species = "Canis familiaris" # Class attribute
```

2. Instance Attributes:

- Unique to each instance of a class.
- Defined within methods, typically inside the **init** method, using self.

Example of Instance Attributes

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

✓ Methods

- Methods are functions defined within a class that describe the behaviors of an object. They can operate on the data contained in the instance attributes.
- Types of Methods:
 - **Instance Methods:** Operate on an instance of the class and can access and modify instance attributes. Always take self as the first parameter.
 - **Class Methods:** Operate on the class itself rather than any instance. Take cls as the first parameter and are decorated with @classmethod.
 - **Static Methods:** Do not modify the class or instance state. Decorated with @staticmethod and do not take self or cls as a parameter.

```
# Example of Instance method
class Dog:
    def bark(self):
        return f"{self.name} says woof!"
```

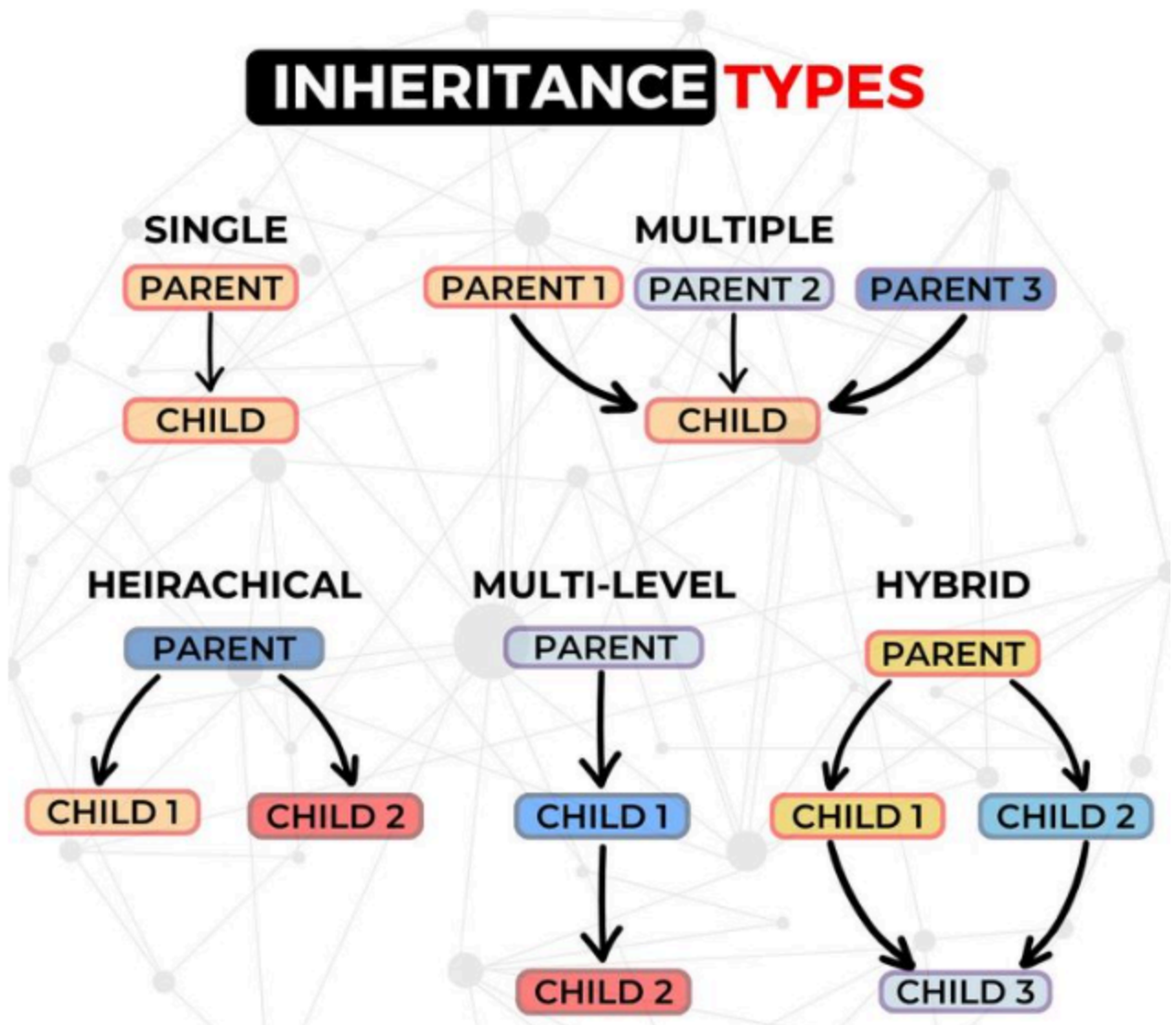
```
# Example of Class method
class Dog:
    species = "Canis familiaris"

    @classmethod
    def get_species(cls):
        return cls.species
```

```
# Example of Static method
class Math:
    @staticmethod
    def add(x, y):
        return x + y
```

✓ Inheritance

- What is Inheritance?
 - Inheritance allows a new class (child or subclass) to inherit properties and behaviors (attributes and methods) from an existing class (parent or superclass). This mechanism enables the creation of a more specialized class that builds upon the general functionality of the parent class.
 - Key Concepts:
 1. Parent Class (Superclass): The class whose attributes and methods are inherited.
 2. Child Class (Subclass): The class that inherits from the parent class.
- Types of Inheritance:-
 - Single Inheritance
 - Multiple Inheritance
 - Multilevel Inheritance
 - Hierarchical Inheritance
 - Hybrid Inheritance



1. Single Inheritance:


- When a child class inherits from only one parent class, it is called single inheritance.

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

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal): # Single Inheritance
    def speak(self):
        return f"{self.name} barks."

# Usage
dog = Dog("Buddy")
print(dog.speak())
```

 Buddy barks.

2. Multiple Inheritance:


- When a class is derived from more than one base class it is called multiple Inheritance.

```
class Walker:
    def walk(self):
        return "Walking on the ground."

class Swimmer:
    def swim(self):
        return "Swimming in the water."

class Amphibian(Walker, Swimmer): # Multiple Inheritance
    pass

# Usage
frog = Amphibian()
print(frog.walk())
print(frog.swim())
```

 Walking on the ground.
Swimming in the water.

3. Multilevel Inheritance:

- Multilevel inheritance in Python allows a class to inherit properties and methods from a class that is already inherited from another class.

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

    def speak(self):
        return f"{self.name} makes a sound."

class Mammal(Animal): # Intermediate Class
    def walk(self):
        return f"{self.name} walks on four legs."

class Dog(Mammal): # Multilevel Inheritance
    def speak(self):
        return f"{self.name} barks."

# Usage
dog = Dog("Buddy")
```

```
print(dog.speak())
print(dog.walk())
```



Buddy barks.
Buddy walks on four legs.

4. Hierarchical Inheritance:

- Hierarchical Inheritance is a specific form of inheritance in Python that involves a single base class with multiple derived classes.

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

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal): # Child class 1
    def speak(self):
        return f"{self.name} barks."

class Cat(Animal): # Child class 2
    def speak(self):
        return f"{self.name} meows."

# Usage
dog = Dog("Buddy")
cat = Cat("Whiskers")
print(dog.speak())
print(cat.speak())
```



Buddy barks.
Whiskers meows.

5. Hybrid Inheritance:

- Hybrid Inheritance is a blend of more than one type of inheritance. The class is derived from the two classes as in the multiple inheritance.

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

    def speak(self):
        return f"{self.name} makes a sound."

class Walker:
```



```

def walk(self):
    return "Walking on the ground."

class Dog(Animal, Walker): # Hybrid Inheritance
    def speak(self):
        return f"{self.name} barks."

# Usage
dog = Dog("Buddy")
print(dog.speak()) # Output: Buddy barks.
print(dog.walk())

```

➞ Buddy barks.
Walking on the ground.

✓ The super() Function

- The `super()` function is used to call a method from the parent class within the child class. This is particularly useful when you want to extend or modify the behavior of an inherited method.
- Commonly used within the **init** method to ensure the parent class is properly initialized.

```

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

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name) # Call to the parent class's __init__ method
        self.breed = breed

    def speak(self):
        return f"{self.name}, a {self.breed}, barks."

# Usage
dog = Dog("Buddy", "Golden Retriever")
print(dog.speak())

```

➞ Buddy, a Golden Retriever, barks.

✓ Method Overriding

- Method overriding occurs when a method in a child class has the same name as a method in the parent class. The method in the child class overrides the one in the parent class.

- Used to provide specific implementation in the child class that is different from the parent class.

```
class Animal:
    def speak(self):
        return "Animal makes a sound."
```

```
class Dog(Animal):
    def speak(self):
        return "Dog barks."
```

```
# Usage
animal = Animal()
dog = Dog()
```

```
print(animal.speak())
print(dog.speak())
```

```
➞ Animal makes a sound.
   Dog barks.
```

✓ The **init** Method in Inheritance

- In an inheritance hierarchy, the **init** method of the child class can call the **init** method of the parent class using `super()`. This ensures that the base class is properly initialized.
- Useful when the child class needs to add more attributes on top of those in the parent class.

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

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name) # Call to the parent class's __init__ method
        self.breed = breed
```

```
# Usage
dog = Dog("Buddy", "Golden Retriever")
print(dog.name) # Output: Buddy
print(dog.breed)
```

```
➞ Buddy
   Golden Retriever
```

✓ Inheritance of Class Attributes and Methods

- Child classes inherit all the attributes and methods of the parent class. This means that a child class can access methods and attributes defined in the parent class.
- Enables code reuse and can reduce redundancy.

```
class Animal:
    species = "Animal"

    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

# Usage
dog = Dog("Buddy", "Golden Retriever")
print(dog.species)
print(dog.speak())
```



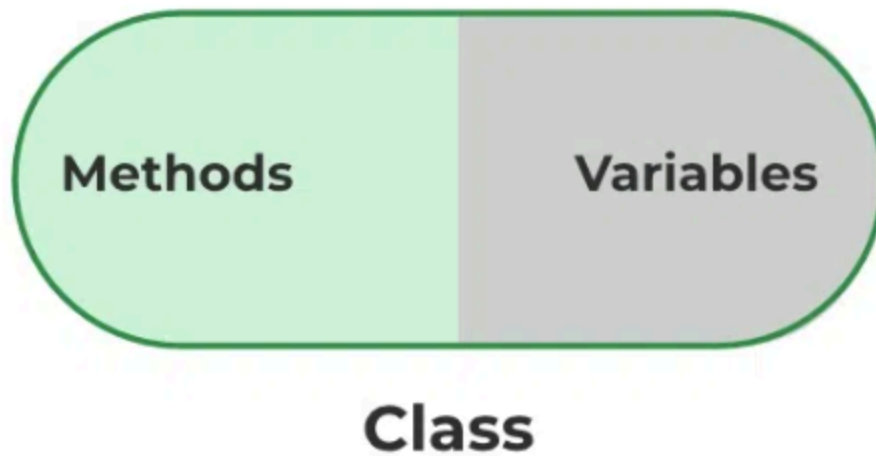
```
Animal
Buddy makes a sound.
```

✓ Encapsulation

1. What is Encapsulation?

- Encapsulation is the mechanism of wrapping data (variables) and methods (functions) together as a single unit. In Python, encapsulation is achieved using classes, which act as a container for data and methods.
- Key Concepts:
 - Public Access Modifier: Allows attributes and methods to be accessed from outside the class.
 - Private Access Modifier: Restricts access to attributes and methods from outside the class.
 - Protected Access Modifier: Suggests that an attribute or method is intended for internal use by the class or its subclasses, but not from outside.

Encapsulation in Python



✓ Public Access Modifier

- Public members are accessible from outside the class. In Python, all members are public by default unless explicitly specified otherwise.
- Suitable for attributes and methods that need to be accessed from outside the class.

```
class Car:
    def __init__(self, make, model):
        self.make = make # Public attribute
        self.model = model # Public attribute

    def display(self):
        return f"Car: {self.make} {self.model}" # Public method
```

```
# Creating an instance
car = Car("Toyota", "Camry")
```

```
# Accessing public attributes and method
print(car.make)
print(car.model)
print(car.display())
```

```
⇒ Toyota
   Camry
   Car: Toyota Camry
```

✓ Private Access Modifier

- Private members are accessible only within the class where they are defined. In Python, private members are indicated by prefixing the attribute or method name with two underscores (`__`).
- Used for internal attributes and methods that should not be exposed outside the class.

```
class Car:
    def __init__(self, make, model):
        self.__make = make # Private attribute
        self.__model = model # Private attribute


    def __display(self): # Private method
        return f"Car: {self.__make} {self.__model}"

    def public_display(self):
        return self.__display()

# Creating an instance
car = Car("Honda", "Accord")

# Accessing private attributes and methods
# print(car.__make) # This will raise an AttributeError
# print(car.__display()) # This will raise an AttributeError

# Accessing private members via a public method
print(car.public_display())
```

 Car: Honda Accord

✓ Protected Access Modifier

- Protected members are accessible within the class and its subclasses but are not intended for public use. In Python, protected members are indicated by prefixing the attribute or method name with a single underscore (`_`).
- Used when an attribute or method should be accessible to subclasses but not to external code.

```
class Car:
    def __init__(self, make, model):
        self._make = make # Protected attribute
        self._model = model # Protected attribute

    def _display(self): # Protected method
        return f"Car: {self._make} {self._model}"
```


```

class ElectricCar(Car):
    def battery_info(self):
        return f"{self._make} {self._model} has a battery."

# Creating an instance
car = ElectricCar("Tesla", "Model S")

# Accessing protected attributes and methods
print(car._make)
print(car._display())
print(car.battery_info())

```

 Tesla
 Car: Tesla Model S
 Tesla Model S has a battery.

✓ Getter and Setter Methods

1. Getter:-

- Methods that retrieve or "get" the value of a private attribute.
- Allows controlled access to private attributes.

```

class Car:
    def __init__(self, make, model):
        self.__make = make
        self.__model = model


    def get_make(self): # Getter method
        return self.__make

    def get_model(self): # Getter method
        return self.__model

# Creating an instance
car = Car("Toyota", "Corolla")

# Accessing private attributes via getter methods
print(car.get_make())
print(car.get_model())

```

 Toyota
 Corolla

2. Setters:-

- Methods that modify or "set" the value of a private attribute.

- Provides controlled modification of private attributes, allowing for validation or transformation of input data.

```
class Car:
    def __init__(self, make, model):
        self.__make = make
        self.__model = model

    def set_make(self, make): # Setter method
        self.__make = make


    def set_model(self, model): # Setter method
        self.__model = model

    def get_details(self):
        return f"Car: {self.__make} {self.__model}"

# Creating an instance
car = Car("Honda", "Civic")

# Modifying private attributes via setter methods
car.set_make("Nissan")
car.set_model("Altima")

# Accessing modified attributes via a public method
print(car.get_details())
```

 Car: Nissan Altima

✓ Name Mangling

- Python applies name mangling to private attributes and methods to avoid accidental access and modification of private members in subclasses. This means that the names of private members are internally changed to include the class name as a prefix.
- Ensures that private members are not easily overridden by subclasses.

```
class Car:
    def __init__(self, make, model):
        self.__make = make

    def __display(self):
        return f"Car make: {self.__make}"

# Accessing the mangled name
car = Car("BMW", "X5")
print(car._Car__make)
print(car._Car__display())
```

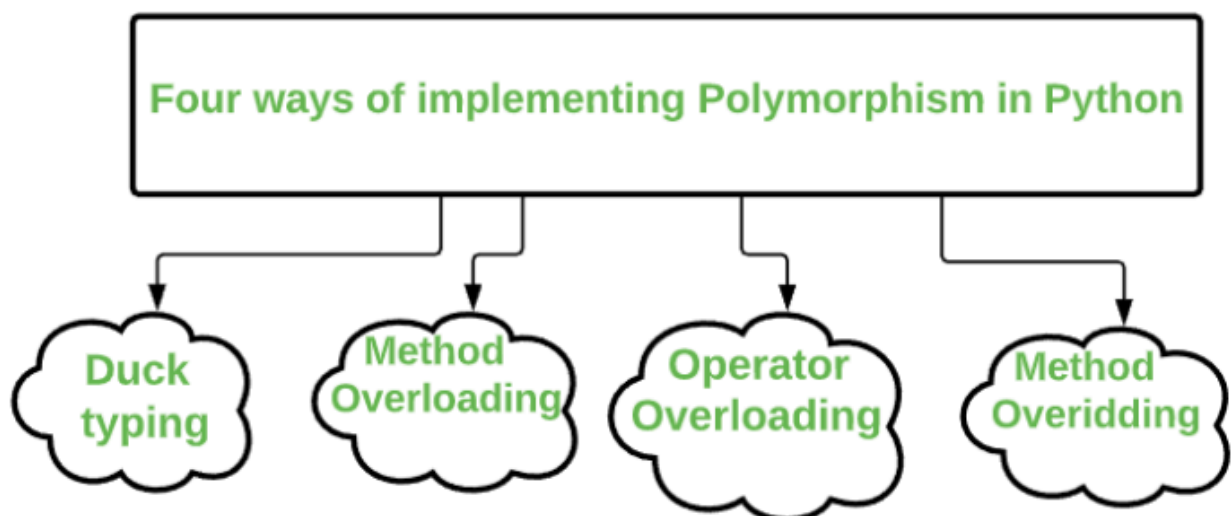


BMW

Car make: BMW

✓ Polymorphism

- What is polymorphism?
 - Polymorphism means "many forms," and it allows the same function or method to be used on different types of objects, leading to different behaviors. The concept is used to perform a single action in different ways.
- Key Concepts:
 - Method Overriding: A subclass provides a specific implementation of a method that is already defined in its superclass.
 - Method Overloading: A single method can have multiple signatures (different parameter lists). Python doesn't support method overloading explicitly but it can be mimicked using default arguments.
 - Duck Typing: Python's dynamic typing nature where the type of an object is determined by its behavior (methods and properties), not by its inheritance from a specific class.
 - Operator Overloading: Python allows operators like `+`, `-`, `*`, etc., to be used in a polymorphic manner through operator overloading. This is done by defining special methods in the class, such as **`add`**, **`sub`**, etc.



✓ Types of Polymorphism

1. Compile-Time Polymorphism (Method Overloading):

- The ability to define multiple methods with the same name but different signatures (parameter lists). This type of polymorphism is determined at compile-time.
- Note: Python does not support method overloading directly as it allows only the last defined method with the same name to be used. However, similar behavior can be achieved using default parameters.

```
class MathOperations:
    def add(self, a, b, c=0):
        return a + b + c

# Usage
math_op = MathOperations()

# Calling add method with two arguments
print(math_op.add(5, 10))

# Calling add method with three arguments
print(math_op.add(5, 10, 15))
```

⇒ 15
30

2. Run-Time Polymorphism (Method Overriding):

- The ability of a subclass to provide a specific implementation of a method that is already defined in its superclass. The method to be executed is determined at runtime based on the object's type.
- This allows for dynamic method resolution at runtime, enabling more flexible and maintainable code.

```
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Usage
animals = [Dog(), Cat(), Animal()]
```

```
for animal in animals:
    print(animal.sound())
```

```
⇒ Bark
    Meow
    Some sound
```

3. Duck Typing:

- Duck typing in Python is a type of polymorphism where the object's compatibility is determined by the presence of certain methods and properties, rather than the actual type of the object. The term comes from the saying, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."
- Enables polymorphic behavior without requiring a strict type hierarchy.

```
class Dog:
    def speak(self):
        return "Bark"
```

```
class Cat:
    def speak(self):
        return "Meow"
```

```
class Bird:
    def speak(self):
        return "Chirp"
```

```
# Usage
def animal_sound(animal):
    return animal.speak()

animals = [Dog(), Cat(), Bird()]

for animal in animals:
    print(animal_sound(animal))
```

```
⇒ Bark
    Meow
    Chirp
```

4. Operator Overloading (Special Methods)

- The + operator can be overloaded to perform addition on custom objects, depending on how the **add()** method is implemented.
- Allows for intuitive use of operators on user-defined objects.

```

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

# Usage
v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2 # Calls the __add__ method
print(v3)

→ Vector(7, 10)

```

✓ Polymorphism with Class Methods

- Polymorphism is commonly seen when class methods can be called on instances of different classes, each responding in their own way to the same method call.

```

class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius * self.radius

# Usage
shapes = [Rectangle(3, 4), Circle(5)]

for shape in shapes:

```

```
print(shape.area())
```

```
→ 12  
78.53975
```

✓ Polymorphism in Inheritance

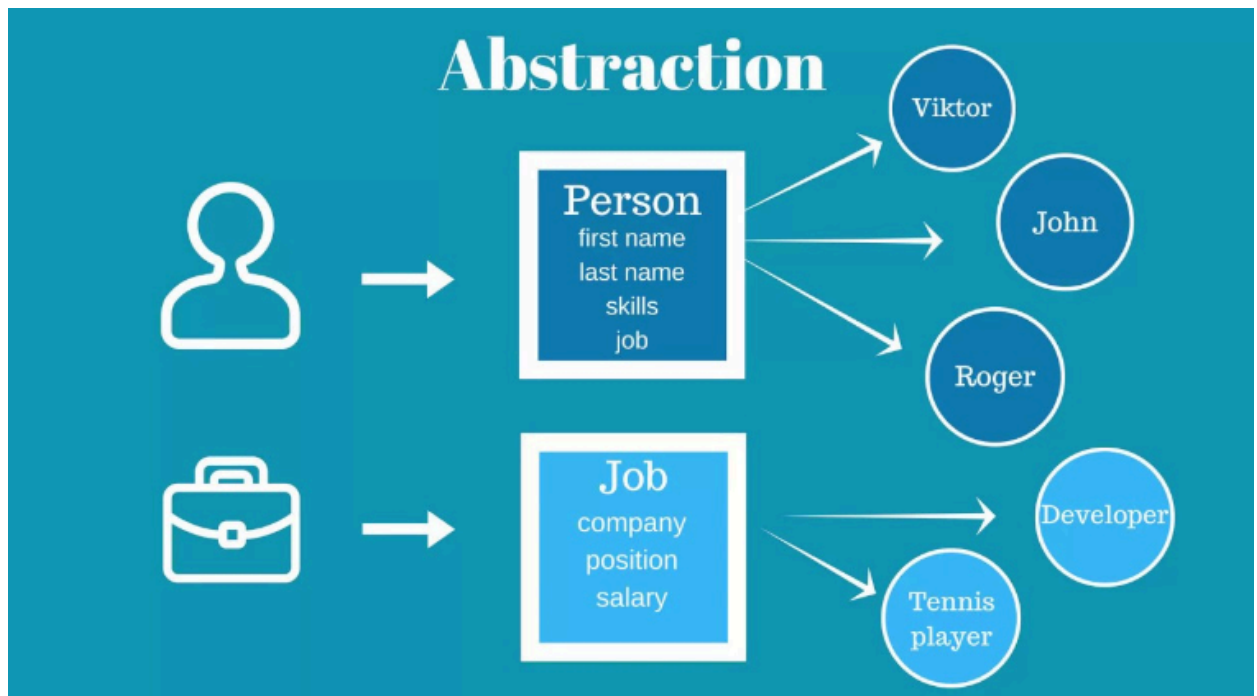
- In an inheritance hierarchy, polymorphism enables objects of different classes to be treated as objects of a common superclass. This is especially useful when working with collections of objects that share a common interface or base class.

```
class Animal:  
    def sound(self):  
        raise NotImplementedError("Subclasses must implement this method")  
  
class Dog(Animal):  
    def sound(self):  
        return "Bark"  
  
class Cat(Animal):  
    def sound(self):  
        return "Meow"  
  
# Using polymorphism in a function  
def make_sound(animal):  
    print(animal.sound())  
  
# Usage  
dog = Dog()  
cat = Cat()  
  
make_sound(dog)  
make_sound(cat)  
  
→ Bark  
Meow
```

✓ Data Abstraction

- What is Data Abstraction?
 - Data Abstraction is the process of exposing only the essential features of an object while hiding the implementation details. It is a way to manage complexity by presenting a simplified model of a system.

- Key Concepts:
 - Abstract Classes: Classes that cannot be instantiated and are designed to be subclassed.
 - Abstract Methods: Methods declared in an abstract class that do not have a body and must be implemented by subclasses.
 - Interfaces: In some languages, interfaces are used to define a contract for classes without specifying how they should be implemented. Python achieves similar functionality through abstract classes.



✓ Advantages of Data Abstraction

1. Simplification of Complex Systems:

- By hiding unnecessary details, abstraction allows users to interact with a simplified model of the system, reducing cognitive load and making the system easier to understand.

2. Code Reusability:

- Abstract classes can define a common interface for multiple classes, allowing for code reuse. Subclasses can share common behavior defined in the abstract class while still providing their specific implementations.

3. Enhanced Code Maintenance:

- Abstraction leads to better code organization and separation of concerns. It allows developers to focus on high-level design without getting bogged down by implementation details.

details.

```
from abc import ABC, abstractmethod
```

```
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass
```

```
    @abstractmethod
    def stop(self):
        pass
```

```
class Car(Vehicle):
    def start_engine(self):
        return "Car engine started"
```

```
    def stop(self):
        return "Car stopped"
```

```
class Bike(Vehicle):
    def start_engine(self):
        return "Bike engine started"
```

```
    def stop(self):
        return "Bike stopped"
```

Usage

```
car = Car()
```

```
bike = Bike()
```

```
print(car.start_engine())
print(bike.start_engine())
```



```
Car engine started
Bike engine started
```

✓ Limitations of Data Abstraction

While data abstraction provides significant benefits, it also has some limitations:

1 Performance Overhead