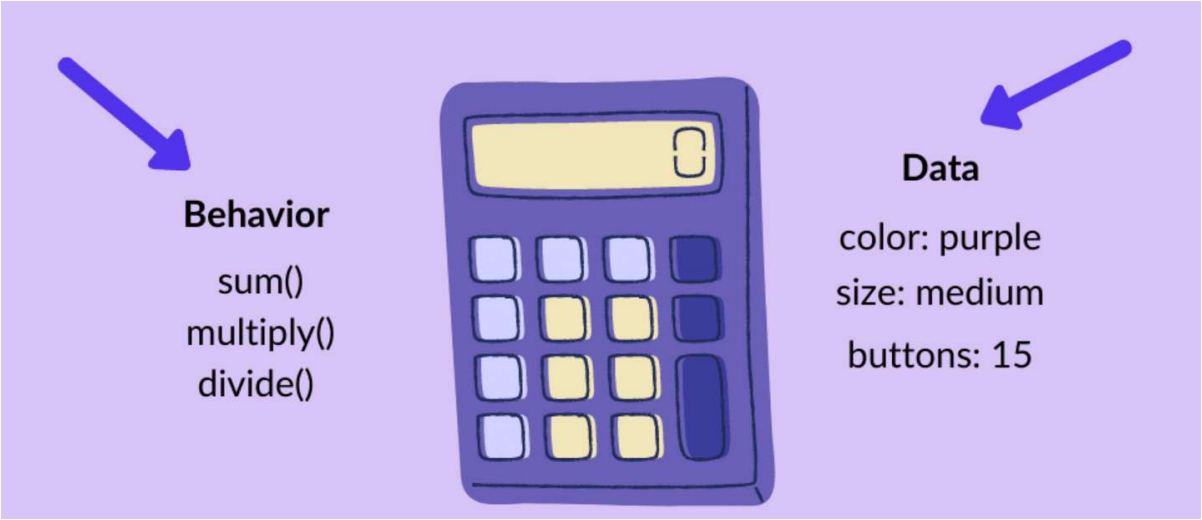
Object-Oriented Programming

Why Do We Use Object-Oriented Programming in Python?

- OOP in Python enhances code organization, promotes code reuse and modularity, simplifies maintenance and updates, and provides a scalable approach to software development.
- OOP allows us to create secure and reliable software. Many Python frameworks and libraries use this paradigm to build their codebase. Some examples are Django, Kivy, pandas, NumPy, and TensorFlow.

1- Introduction

- Object-Oriented Programming (OOP) is a programming paradigm (a pattern or model) in which we can think about complex problems as objects.
- It revolves around the concept of "objects". An object is a collection of data (variables) and methods (functions) that operate on the data.
- It allows developers to organize code into reusable, self-contained units that interact with one another
- An object is something which encapsulate data and functionality.So when we're talking about OOP, we're referring to a set of concepts and patterns we use to solve problems with objects.
- An object in Python is a single collection of data (attributes) and behavior (methods). You can think of objects as real things around you. For example, consider calculators.



- As you may notice, the data (attributes) are always nouns, while the behaviors (method) are always verbs.This compartmentalization is the central concept of Object-Oriented Programming. You build objects that store data and contain specific kinds of functionality.

2- Creating your own classes

A class is a template for objects. It contains the code for all the object's methods.

Example 1- A simple example of class

In [ ]:
```
class Example:
    def __init__(self,a, b):
        self.a = a
        self.b = b
    def add(self):
        return self.a + self.b
# Create an instance of the class
e = Example(8, 6)
# Call methods on the object
print(e.add())
# Accessing an attribute
print(e.a)
```

Points to Note:

- The class keyword:
  - To create a class, we use the class statement. Class names usually start with a capital letter.
- Attributes:
  - Attributes in a class define the properties or data associated with instances of that class. They represent the state of the objects created from the class. In Python, attributes can be accessed using dot notation.
- The constructor (**init** method):
  - Most classes will have a method called **init**. The underscores indicate that it is a special kind of method. It is called a constructor, and it is automatically called when someone creates a new object from your class. The constructor is usually used to set up the class's variables. In the above program, the constructor takes two values, a and b, and assigns the class variables a and b to those values.
- The .self argument:
  - The first argument to every method in your class is a special variable called self. Every time your class refers to one of its variables or methods, it must precede them by self. The purpose of self is to distinguish your class's variables and methods from other variables and functions in the program.
- Creating object:
  - To create a new object from the class, you call the class name along with any values that you want to send to the constructor. You will usually want to assign it to a variable

name. This is what the line e=Example(8,6) does.

- Using object's methods:
  - To use the object's methods, use the dot operator, as in e.addmod()

Example 2- A more practical example

In [ ]:
```
class Analyzer:
    def __init__(self, s):
        s = s.lower()
        self.words = s.split()

    def number_of_words(self):
        return len(self.words)
```

In [ ]:
```
s = "This is a test of how our class Analyzer will work. Let's see."
# Create an instance of the class
analyzer = Analyzer(s)
```

In [ ]:
```
# accessing the "words" attribute of the Analyzer object represented by the
analyzer.words
```

In [ ]:
```
# Call method on the object
analyzer.number_of_words()
```

Example 3- Define a class having methods:

* print name and age of a student.
* print Branch name.

In [ ]:
```
class Person:
    def __init__(self, name, age, branch):
        self.name = name
        self.age = age
        self.branch= branch

    def greet(self):
        print("Hello, my name is {} and I am {} years old.".format(self.nam

    def edu_info(self):
        print(" I have completed my B.Tech in {}.".format(self.branch))
```

In [ ]:
```
# Create an instance of the Person class
person1 = Person("Alice", 30, 'CSE')
```

In [ ]:
```
# Call the greet method on the instance
person1.greet()
```

In [ ]:
```
# Call the edu_info method on the instance
person1.edu_info()
```

Example 4 - :Create a class Car with the properties brand, model and year. Create two objects of the class Car and print all the details of the two objects.

In [ ]:
```
# Define a class named 'Car'
class Car:
    # Constructor method to initialize attributes
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self.speed = 0  # Initial speed is 0

    # Method to display car information
    def display_info(self):
        print(f"{self.year} {self.brand} {self.model}")
```

In [ ]:
```
# Create an instance of the Car class
my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info()
```

In [ ]:

In [ ]:

3- Key Concepts of OOP

3.1- Class:

- A class is a collection of objects. It is a logical entity or blueprint of the object which contains some attributes and methods. if you have a Car class, then it should contain an attribute and method, i.e. type, color, model year, company name, etc. Because every car has its own type either it can be an SUV, hatchback, or sedan .
- A blueprint or template for creating objects. It defines the attributes (data) and methods (functions) that all objects of that class will have.

In [ ]:
```
class Car:
    pass
```

In [ ]:

3.2- Objects

- It is an instance of the class or we can say an entity that has behavior and state with it. It is the real implementation of a class that has collections of data(variables) and methods (functions) that access the data.
- An object consists of State, Behaviour, and Identity.
- State: We can represent the state with help of attributes of an object and it also reflects the properties of an object.
- Behavior: we can express the behavior with help of methods of an object and also reflects the response of an object to other objects.

In [ ]:
```
obj=Car()
```

3.3- Constructors and Destructors:

- A constructor is a special method that is automatically called when an instance of a class is created. In Python, the constructor method is called **init**().
- A destructor, unlike constructors, is not commonly used in Python. Python has a garbage collection mechanism that automatically handles memory management and cleanup. However, you can define a destructor method, **del**().

In [ ]:
```
# Example without using constructor
class Counter:
    def set_count(self, count):
        self.count = count

# Creating an object without using the __init__ method
counter_without_init = Counter()
counter_without_init.set_count(10)
print("Counter without init method:", counter_without_init.count)
```

In [ ]:
```
# Example  using constructor
class Counter:
    def __init__(self, count):
        self.count = count

# Creating an object using the __init__ method
counter_with_init = Counter(5)
print("Counter with init method:", counter_with_init.count)  # Output: 5
```

In [ ]:
```python
# Example of destructor
class MyClass:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} created.")

    def __del__(self):
        print(f"{self.name} destroyed.")

# Create objects
obj1 = MyClass("Object 1")
obj2 = MyClass("Object 2")

# Delete objects
del obj1
del obj2
```

In [ ]:
```python

```

3.4- Encapsulation: (meaning:- to enclose in)

- The bundling of data (attributes) and methods that operate on that data within a single unit (a class).
- Encapsulation hides the internal state of the object from the outside world and only exposes the necessary functionality.This helps to protect the integrity of the data and prevent unintended manipulation.

In [ ]:
```python
# Not protected mode
class Example:
    def __init__(self,a, b):
        self.a = a
        self.b = b
    def add(self):
        return self.a + self.b
# Create an instance of the class
e = Example(8, 6)
# Call methods on the object
print(e.add())
# Accessing an attribute
print(e.a)
```

In [ ]:
```python
# Protected mode (Single underscore befroe attributes)
class Example:
    def __init__(self,a, b):
        self._a = a
        self._b = b
    def add(self):
        return self._a + self._b
# Create an instance of the class
e = Example(8, 6)
# Call methods on the object
print(e.add())
# Accessing an attribute
print(e._a)
```

In [ ]:
```python
# Private mode (Double underscore befroe attributes)
class Example:
    def __init__(self,a, b):
        self.__a = a
        self.__b = b
    def add(self):
        return self.__a + self.__b
# Create an instance of the class
e = Example(8, 6)
# Call methods on the object
print(e.add())
# Accessing an attribute
print(e.__a)
```

Example- A class to represent a simple bank account and use it.

In [ ]:
```python
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.account_number = account_number  # Encapsulated attribute
        self.balance = balance  # Encapsulated attribute

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount

    def get_balance(self):
        return self.balance

    def get_account_number(self):
        return self.account_number
```

In this example:

- We define a class BankAccount to represent a simple bank account.

- The attributes account_number and balance are encapsulated with a leading underscore (_), indicating that they are intended to be private and should not be accessed directly from outside the class.
- The class provides methods (deposit, withdraw, get_balance, and get_account_number) to interact with the bank account's data.
- Users of the BankAccount class can deposit money, withdraw money (if sufficient balance is available), and retrieve the account balance and number using the provided methods.
- The encapsulation ensures that the internal state of the BankAccount object is protected from direct manipulation, and all interactions with the object are performed through well-defined methods.

Here's how you would use the BankAccount class:

In [ ]:
```
# Create an instance of BankAccount
account = BankAccount("123456", 1000)

# Deposit and withdraw money
account.deposit(500)
account.withdraw(200)

# Get account balance and number
print("Account balance:", account.get_balance())
print("Account number:", account.get_account_number())
```

In [ ]:
```

```

3.5- Abstraction: (meaning:- the action of seperating)

- It involves hiding the complex implementation details of a class and exposing only the necessary features of an object to the outside world. In simpler terms, abstraction allows you to focus on what an object does rather than how it does it.
- Emphasizing the "what" rather than the "how".
- It allows developers to interact with objects at a high level, without needing to understand the intricate inner workings of those objects.
- It's like using a remote control to operate a TV without needing to know the intricate electrical engineering behind it; you simply press the power button and the TV turns on, abstracting away the complexities involved.

In [ ]:
```python
class Dog:
    def speak(self):
        return "Woof!"

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

# Creating instances of animals
dog = Dog()
cat = Cat()

# Using abstraction to make animals speak
print("Dog says:", dog.speak())  # Output: Dog says: Woof!
print("Cat says:", cat.speak())  # Output: Cat says: Meow!
```

In [ ]:
```python
# Example 1
class Animal:
    def speak(self):
        pass

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

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

# Creating instances of animals
dog = Dog()
cat = Cat()

# Using abstraction to make animals speak
print("Dog says:", dog.speak())  # Output: Dog says: Woof!
print("Cat says:", cat.speak())  # Output: Cat says: Meow!
```

In above example:

- There's an abstract class Animal with a method speak() that acts as a placeholder.
- Subclasses Dog and Cat inherit from the Animal class and provide their own implementations for the speak() method.
- Instances of Dog and Cat can be created, and by calling the speak() method on each instance, we can make them "speak".
- This example demonstrates abstraction by providing a common interface (speak()) for different types of animals. Users of the Dog and Cat classes don't need to know the internal details of how each animal speaks; they can simply use the speak() method to interact with the objects at a high level.
- The Animal class can still act as an abstraction layer, hiding the implementation details of specific animal types while providing a standardized interface for interacting with them.

- In the given simple example, the Animal class might seem redundant, but in more complex systems with diverse types of animals or objects, using such a base class can provide

In [ ]:
```python
# Example 2
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

    def perimeter(self):
        return 2 * (self.width + self.height)
```

In [ ]:
```python
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

    def perimeter(self):
        return 2 * 3.14 * self.radius
```

In [ ]:
```python
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

    def perimeter(self):
        # Assuming it's an equilateral triangle
        return 3 * self.base
```

In [ ]:
```python
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass
```

In [ ]:
```python
# Creating instances of different shapes
rectangle = Rectangle(4, 5)
circle = Circle(3)
triangle = Triangle(4, 5)

# Using abstraction to calculate areas and perimeters
print("Rectangle Area:", rectangle.area())
print("Rectangle Perimeter:", rectangle.perimeter())

print("Circle Area:", circle.area())
print("Circle Perimeter:", circle.perimeter())

print("Triangle Area:", triangle.area())
print("Triangle Perimeter:", triangle.perimeter())
```

In above example:

- The Shape class serves as an abstraction, providing common methods area() and perimeter() that all shapes should have.
- Subclasses (Rectangle, Circle, Triangle) inherit from Shape and provide their own implementations for these methods.
- Users of these classes interact with the shapes using the common interface (area() and perimeter() methods) without needing to know the internal details of how each shape calculates its area and perimeter.
- This example demonstrates abstraction by providing a simplified and unified way to work with different shapes, hiding the complexities of their calculations behind a common interface.

3.6- Inheritance

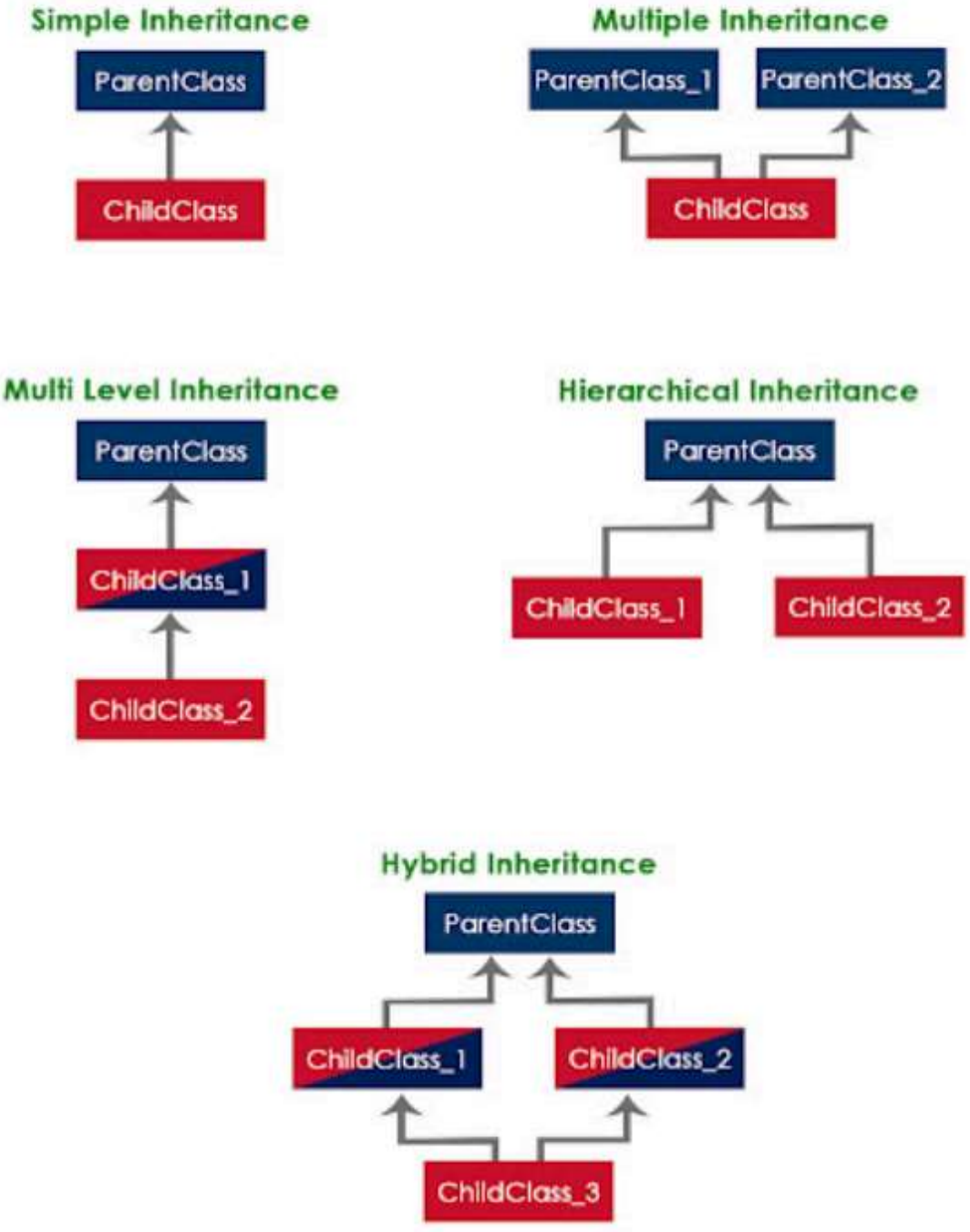
Inheritance, in general, is the passing of properties to someone. In programming languages, the concept of inheritance comes with classes.

- Creating new classes from already existing classes is called inheritance.
- The existing class is called a super class or base class or parent class.
- The new class is called a subclass or derived class or child class.
- Inheritance allows sub classes to inherit the variables, methods and constructors of their super class.

Advantages of Inheritance:

- The main advantage of inheritance is code re-usability.
- Time taken for application development will be less.
- Redundancy (repetition) of the code can be reduce

Types of Inheritance in Python



3.6.1. Simple Inheritance

In [ ]:

```
class Parent:
    def m1(self):
        print("Parent class m1 method")

class Child(Parent):
    def m2(self):
        print("Child class m2 method")

c = Child()
c.m1()
c.m2()
```

3.6.2. Multiple Inheritance

In [ ]:

```
class Parent1:
    def m1(self):
        print("Parent1 Method")
class Parent2:
    def m2(self):
        print("Parent2 Method")
class Child(Parent1, Parent2):
    def m3(self):
        print("Child Method")
c=Child()
c.m1()
c.m2()
c.m3()
```

3.6.3. Multilevel Inheritance

In [ ]:

```
class A:
    def m1(self):
        print("Parent class A: m1 Method")
class B(A):
    def m2(self):
        print("Child class B derived from A: m2 Method")
class C(B):
    def m3(self):
        print("Child class C derived from B: m3 Method")
obj=C()
obj.m1()
obj.m2()
obj.m3()
```

3.6.4. Hierarichal Inheritance

In [ ]:
```
#Python program to show hierarchical inheritance

# The base class
class Parent:
    def funA(self):
        print("This function is in the parent class.")

# Derived class one
class Child_one(Parent):
    def funB(self):
        print("This function is in class child_one.")

# Derived class two
class Child_two(Parent):
    def funC(self):
        print("This function is in class child_two.")
```

3.6.5. Hybrid Inheritance

Python even supports Hybrid inheritance in which we implement more than one type of inheritance in one code.

In [ ]:
```
class A:
    def funcA(self):
        print("Hello, you are now in class A")
class B(A):
    def funcB(self):
        print("Hello, you are now in class B")
class C(A):
    def funcC(self):
        print("Hello, you are now in class C")
class D(C,A):
    def funcD(self):
        print("Hello, you are now in class D")

obj = D()
obj.funcD()
obj.funcC()
obj.funcA()
```

3.7- Polymorphism

- The word ‘Poly’ means many and ‘Morphs’ means forms. The process of representing “one form in many forms” is called a polymorphism.
- Polymorphism may be used in one of the following ways in an object-oriented language:
  - Overloading of operators
  - Class Polymorphism in Python
  - Method overriding, also referred to as Run time Polymorphism
  - Method overloading, also known as Compile time Polymorphism

3.7.1- Polymorphism in Python through Operator Overloading

- Operator overloading is a type of polymorphism in which the same operator performs various operations depending on the operands. Python allows for operator overloading.

3.7.1.1.Polymorphism in + operator:

We already know that the '+' operator is frequently used in Python programs. However, it does not have a single application. When you wish to add two integers, concatenate two strings, or extend two lists, you use the same + sign. The + operator behaves differently depending on the type of object on which it is used.

In [ ]:
```
# Example
a = 10
b = 20
print('Addition of 2 numbers:', a + b)

str1 = 'Hello '
str2 = 'Python'
print('Concatenation of 2 strings:', str1 + str2)

list1 = [1, 2, 3]
list2 = ['A', 'B']
print('Concatenation of 2 lists:', list1 + list2)
```

*3.7.1.2. Polymorphism in * operator:*

The * operator is used to multiply 2 numbers if the data elements are numeric values.

If one of the data types is a string, and the other is numeric, the string is printed that many times as that of the 2nd variable in the multiplication process.

In [ ]:
```
# Example
a = 10
b = 5
print('Multiplication of 2 numbers:', a * b)

num = 3
mystr = 'Python'
print('Multiplication of string:', num * mystr)
```

3.7.2- Function Polymorphism in Python

- There are certain Python functions that can be used with different data types. The len() function is one example of such a function. Python allows it to work with a wide range of data types.

- The built-in function len() estimates an object's length based on its type. If an object is a string, it returns the number of characters; or if an object is a list, it returns the number of

In [ ]:
```
# Example
mystr = 'Programming'
print('Length of string:', len(mystr))

mylist = [1, 2, 3, 4, 5]
print('Length of list:', len(mylist))

mydict = {1: 'One', 2: 'Two'}
print('Length of dict:', len(mydict))
```

3.7.3- Polymorphism in classes & objects

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility and extensibility in code by allowing objects to be manipulated in a uniform way regardless of their individual types.

In [ ]:
```
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 to make any animal speak
def make_speak(animal):
    print(animal.speak())

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

# Using polymorphism to make animals speak
make_speak(dog)
make_speak(cat)
make_speak(duck)
```

In above example:

- We have a common superclass Animal with a method speak() that acts as a placeholder for animal sounds.
- Subclasses (Dog, Cat, Duck) inherit from Animal and provide their own implementations for the speak() method.
- The make_speak() function accepts any object of type Animal (or its subclasses) and calls its speak() method.
- We create instances of different animal types (dog, cat, duck) and pass them to the make_speak() function.
- Despite the objects being of different types, they are all treated uniformly as Animal objects due to polymorphism, and their respective speak() methods are invoked appropriately.
- Polymorphism allows for a more flexible and modular design, where different classes can share a common interface (speak() method in this case) and be used interchangeably in

3.8- Method Overriding (Polymorphism and Inheritance)

- Polymorphism is most commonly associated with inheritance.
- In Python, child classes, like other programming languages, inherit methods and attributes from the parent class. Method Overriding is the process of redefining certain methods and attributes to fit the child class.
- This is especially handy when the method inherited from the parent class does not exactly fit the child class. In such circumstances, the method is re-implemented in the child class. Method Overriding refers to the technique of re-implementing a method in a child class.

In [ ]:

```
class Vehicle:
    def __init__(self, brand, model, price):
        self.brand = brand
        self.model = model
        self.price = price

    def show(self):
        print('Details:', self.brand, self.model, 'Price:', self.price)

    def max_speed(self):
        print('Vehicle max speed is 160')

    def gear_system(self):
        print('Vehicle has 6 shifter gearbox')

# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 260')

    def gear_system(self):
        print('Car has Automatic Transmission')

# Car Object
car = Car('Audi', 'R8', 9000000)
car.show()
# call methods from Car class
car.max_speed()
car.gear_system()

# Vehicle Object
vehicle = Vehicle('Nissan', 'Magnite', 550000)
vehicle.show()
# call method from a Vehicle class
vehicle.max_speed()
vehicle.gear_system()
```

3.9- Method Overloading (Compile-Time Polymorphism)

Method overloading occurs when a class contains many methods with the same name. The types and amount of arguments passed by these overloaded methods vary. Python does not support method overloading or compile-time polymorphism. If there are multiple methods with the same name in a class or Python script, the method specified in the latter one will override the earlier one.

****