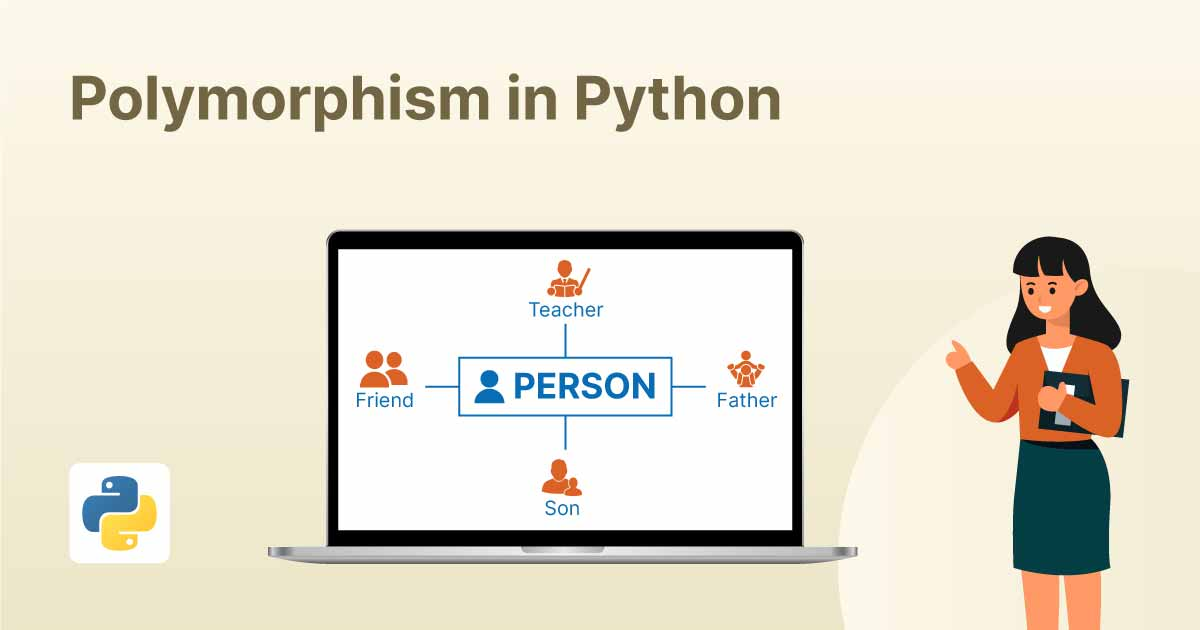
**Polymorphism in Python**

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which are instances of classes. In OOP, objects encapsulate both data (attributes) and behavior (methods). This paradigm promotes modular and reusable code, making it easier to manage and maintain large-scale software projects.And polymorphism is one of the main characteristics in OOP.

****

**Polymorphism in Python:**

**Definition:** Polymorphism in Python refers to the ability of objects to take on multiple forms. Specifically, it allows different classes to share the same method or function name, but each class can provide its own unique implementation of that method or function.

**Real-World Example:** Think about a shape class in a drawing application. Shapes like circles, squares, and triangles can all be drawn, but the specific method of drawing each shape varies. Despite this, we can have a single draw() method that behaves differently for each shape.

**Code Example:**

class Shape:

def draw(self):

pass

class Circle(Shape):

def draw(self):

return "Drawing Circle"

class Square(Shape):

def draw(self):

return "Drawing Square"

class Triangle(Shape):

def draw(self):

return "Drawing Triangle"

# Polymorphism in action

shapes = [Circle(), Square(), Triangle()]

for shape in shapes:

print(shape.draw())

**Explanation of Code:**

* We define a base class Shape with a method draw(), intended to be overridden by subclasses.
* We create subclasses Circle, Square, and Triangle, each with its own implementation of the draw() method.
* We create a list of shapes containing instances of each subclass.
* We iterate over the shapes list and call the draw() method on each object. Despite calling the same method name, the behaviour differs based on the type of shape, demonstrating polymorphism.

**How to Use Polymorphism?**

To use polymorphism in Python, follow these steps:

1. Define a base class with a method that you want to make polymorphic.
2. Create subclasses that inherit from the base class.
3. Override the method in each subclass with a unique implementation.

**Example:** *(Refer to the previous example for implementation)*

**Explanation:**

* Define a base class with a method that serves as the interface for polymorphism.
* Create subclasses that inherit from the base class and provide their own implementations of the method.
* Use instances of these subclasses interchangeably, calling the method on each object as needed.

**Operator Overloading:**

Operator overloading in Python allows operators like +, -, \*, etc., to behave differently based on the operands they are applied to. It enables objects of a class to define their own behavior with respect to operators.

**Real-World Example:** Consider a Vector class representing mathematical vectors. We can overload the + operator to perform vector addition.

**Code Example:**

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)

# Operator overloading in action

v1 = Vector(2, 3)

v2 = Vector(4, 5)

result = v1 + v2

print(f"Result: ({result.x}, {result.y})")

**Explanation of Code:**

* We define a Vector class with attributes x and y representing coordinates.
* We overload the + operator by implementing the \_\_add\_\_() method, which performs vector addition.
* We create instances v1 and v2 of the Vector class and add them together using the + operator, resulting in a new Vector object result.

**Method Overloading:**

Method overloading in Python allows a class to define multiple methods with the same name but different signatures (number or types of parameters). The interpreter determines which method to call based on the arguments provided.

**Real-World Example:** Consider a Calculator class with an add() method. We can overload the add() method to support addition of integers, floats, or even strings.

**Code Example:**

class Calculator:

def add(self, a, b):

return a + b

# Method overloading in action

calc = Calculator()

print("Sum of integers:", calc.add(2, 3))

print("Sum of floats:", calc.add(2.5, 3.5))

print("Concatenation of strings:", calc.add("Hello", " World"))

**Explanation of Code:**

* We define a Calculator class with an add() method that accepts two parameters.
* We call the add() method with different types of arguments: integers, floats, and strings.
* Python determines which version of the add() method to invoke based on the types of arguments provided, demonstrating method overloading.

**Polymorphism with Inheritance:**

Polymorphism with inheritance in Python allows subclasses to override methods inherited from their parent class, providing their own implementations. This enables different subclasses to share a common interface while exhibiting different behaviors.

**Real-World Example:** Consider a Shape class with a draw() method. Subclasses like Circle, Square, and Triangle can inherit from Shape and override the draw() method to implement shape-specific drawing logic.

**Code Example:** *(Refer to the polymorphism example for implementation)*

**Explanation of Code:**

* We define a base class Shape with a method draw() intended to be overridden by subclasses.
* We create subclasses like Circle, Square, and Triangle, each with its own implementation of the draw() method.
* The subclasses inherit the common interface from the Shape class but provide their own behavior, showcasing polymorphism with inheritance.

**Duck Typing:**

Duck typing is a concept in Python that focuses on an object's behavior rather than its type. It allows objects of different classes to be used interchangeably if they support the same set of methods or attributes, regardless of their actual types.

**Real-World Example:** Imagine you have a function that takes an object and calls a quack() method on it. If the object has a quack() method, it can be treated as a duck, regardless of whether it's an actual Duck object or not.

**Code Example:**

class Duck:

def quack(self):

return "Quack!"

class Person:

def quack(self):

return "I'm not a duck!"

# Duck typing in action

def make\_quack(obj):

return obj.quack()

duck = Duck()

person = Person()

print("Duck says:", make\_quack(duck))

print("Person says:", make\_quack(person))

**Explanation of Code:**

* We define a Duck class and a Person class, both with a quack() method.
* We create instances of both classes and pass them to a function make\_quack().
* Despite Duck and Person being different classes, as long as they have a quack() method, they can be treated as ducks, demonstrating duck typing.

**Need for Polymorphism:**

The need for polymorphism arises from the desire to write flexible and reusable code

. By using polymorphism, we can design programs that can work with objects of various types without needing to know their specific classes. This promotes code scalability, maintainability, and extensibility.

**Real-World:** Imagine a sorting algorithm that can sort a list of objects regardless of their types (integers, floats, strings, etc.). By leveraging polymorphism, the sorting algorithm can operate on a wide range of data types without modification.

Polymorphism is a powerful concept in Python that enables flexibility and code reuse in object-oriented programming. By understanding and implementing polymorphism through mechanisms like operator overloading, method overloading, inheritance, and duck typing, developers can create more adaptable and maintainable software systems.