

POLYMORPHISM

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Function Polymorphism

An example of a Python function that can be used on different objects is the len() function.

Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move()

Inheritance Class Polymorphism

make a parent class called Vehicle, and make Car, Boat, Plane child classes of Vehicle, the child classes inherits the Vehicle methods, but can override them

Types of Polymorphism

Compile-time Polymorphism

- Found in statically typed languages like Java or C++, where the behavior of a function or operator is resolved during the program's compilation phase.
- Examples include method overloading and operator overloading, where multiple functions or operators can share the same name but perform different tasks based on the context.
- In Python, which is dynamically typed, compile-time polymorphism is not natively supported. Instead, Python uses techniques like dynamic typing and duck typing to achieve similar flexibility.

Runtime Polymorphism

- Occurs when the behavior of a method is determined at runtime based on the type of the object.
- In Python, this is achieved through method overriding: a child class can redefine a method from its parent class to provide its own specific implementation.
- Python's dynamic nature allows it to excel at runtime polymorphism, enabling flexible and adaptable code.

PRACTICE QUESTIONS:

1. Create a class Animal with a method speak(). Override it in Dog, Cat, Cow.

Code:

```
class animal:
    def speak(self):
        print("Animal class")

class dog:
    def speak(self):
        print("Dog class")

class cat:
    def speak(self):
        print("Cat class")

class cow:
    def speak(self):
        print("Cow class")

obj1 = animal()
obj1.speak()

obj2 = dog()
obj2.speak()

obj3 = cat()
obj3.speak()

obj4 = cow()
obj4.speak()
```

Output:

```
➡ Animal class
   Dog class
   Cat class
   Cow class
```

2. Create a base class Shape with a method area(). Override it in Circle and Rectangle.

Code:

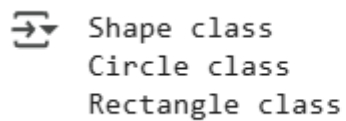
```
class shape:
    def area(self):
        print("Shape class")
class circle(shape):
    def area(self):
        print("Circle class")
class rectangle(shape):
    def area(self):
        print("Rectangle class")

obj1 = shape()
obj1.area()

obj2 = circle()
obj2.area()

obj3 = rectangle()
obj3.area()
```

Output:



```
⇒ Shape class
   Circle class
   Rectangle class
```

3. Create a function `play_sound(obj)` that accepts any object with a `sound()` method.

Code:

```
class animal:
    def sound(self):
        print("Animal class")

class dog:
    def sound(self):
        print("Dog class")

class cat:
    def sound(self):
        print("Cat class")

for play_sound in (animal(), dog(), cat()):
    play_sound.sound()
```

Output:

```
➞ Animal class  
Dog class  
Cat class
```

4. Create a class Student and override the `_str_` method to display details

Code:

```
class student:  
    def _str_(self, roll_no, name):  
        print("Roll No:", roll_no)  
        print("Name:", name)  
  
class student1:  
    def _str_(self, roll_no, name):  
        print("Roll No:", roll_no)  
        print("Name:", name)  
  
obj1= student()  
obj1._str_(1, "aman")  
  
obj2= student1()  
obj2._str_(2, "raman")
```

Output:

```
➞ Roll No: 1  
Name: aman  
Roll No: 2  
Name: raman
```