Python

Polymorphism:

allows entities like functions, methods or operators to behave differently based on the type of data they are handling. Derived from Greek, the term literally means “many forms”.

Polymorphism in built-in functions:

```
print(len("Hello"))        # String length
print(len([1, 2, 3]))      # List length
print(max(1, 3, 2))        # Maximum of integers
print(max("a", "z", "m"))  # Maximum in strings
```

Polymorphism in functions:

```
def add(a, b):
return a + b
print(add(3, 4))                  # Integer addition
print(add("Hello, ", "World!"))   # String concatenation
print(add([1, 2], [3, 4]))        # List concatenation
```

Polymorphism in Operators:

```
print(5 + 10)             # Integer addition
print("Hello " + "World!")  # String concatenation
```

```
print([1, 2] + [3, 4])      # List concatenation
```

Polymorphism in OOPS:

In OOPS polymorphism all methods in different classes to share the same name but perform distinct tasks.

```
class Shape:
    def area(self):
        return "Undefined"

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

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

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

    def area(self):
        return 3.14 * self.radius ** 2

shapes = [Rectangle(2, 3), Circle(5)]
for shape in shapes:
    print(f"Area: {shape.area()}")
```

Types of Polymorphism:

Compile Time:

- 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:

- 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.

Method Overloading

Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

python does not support method overloading by default. But there are different ways to achieve method overloading in Python.

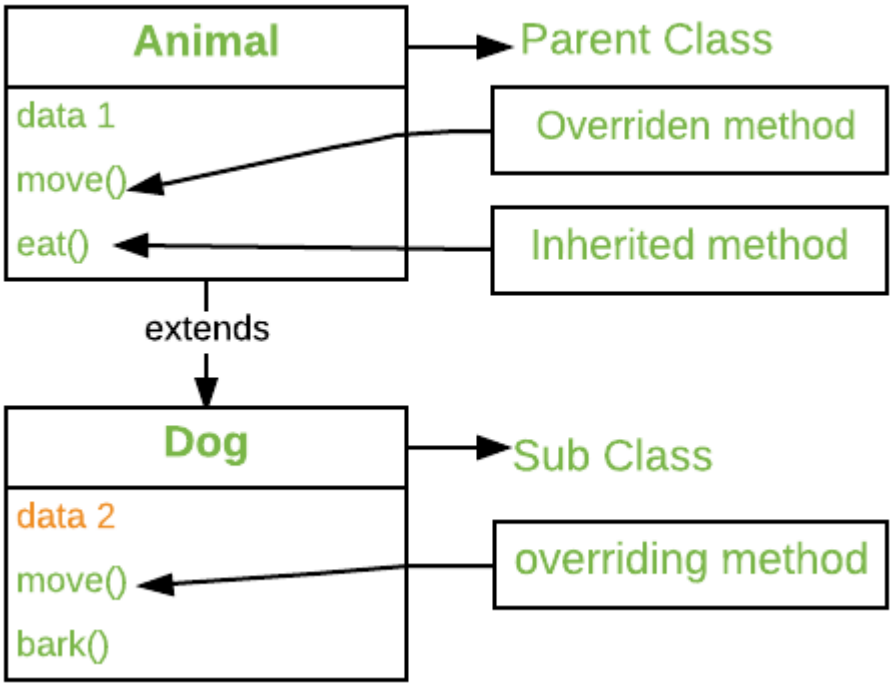
```
# First product method.
# Takes two argument and print their
# product


def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product


def product(a, b, c):
    p = a * b*c
```

```
    print(p)

# Uncommenting the below line shows an error
# product(4, 5)


# This line will call the second product method
product(4, 5, 5)
```

Method Overriding:

OOPS language allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.

example: in 2011 you didn't have a phone but your father had a nokia then in 2020 you bought a oneplus. In 2011 you used father's nokia to communicate now that in 2024 you have overriden his phone with oneplus you can use your own phone.




```
# Python program to demonstrate
# Defining parent class
```

```
class Parent():

    # Constructor
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
        super().__init__()  # Call parent constructor
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)

# Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()  # Should print "Inside Parent"
obj2.show()  # Should print "Inside Child"
```

Method Resolution Order

It is the order in which Python looks for a method in a hierarchy of classes. MRO follows a specific sequence to determine which method to invoke when it encounters multiple classes with the same method name.

languages that support multiple inheritance method resolution order plays a very crucial role.

```
# Python program showing
# how MRO works

class A:
    def rk(self):
        print("In class A")

class B(A):
    def rk(self):
        print("In class B")

r = B()
r.rk()
```

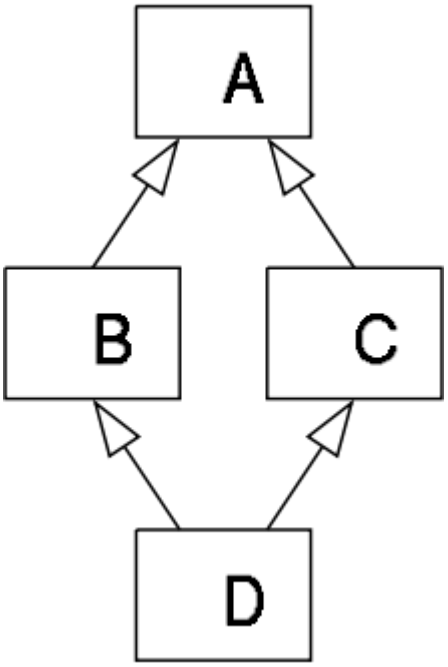
In the above example the methods that are invoked is from class B but not from class A, and this is due to Method Resolution Order(MRO).

The order that follows in the above code is- class B – > class A .

Example: Diamond Inheritance

```
class A:
    def rk(self):
        print("In class A")

class B(A):
    def rk(self):
        print("In class B")

class C(A):
    def rk(self):
        print("In class C")

# classes ordering
class D(B, C):
    pass
```

```
r = D()
r.rk()
```



Python follows a depth-first lookup order and hence ends up calling the method from class A. By following the method resolution order, the lookup order as follows.

Class D → Class B → Class C → Class A

Python follows depth-first order to resolve the methods and attributes. So in the above example, it executes the method in class B.