**Polymorphism in Python**

Allows entities like functions, methods or operators to behave differently based on the type of data they are handling.

**Polymorphism in Built-in Functions**

print(len("Hello")) *# String length*

print(len([1, 2, 3])) *# List length*

**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**

Polymorphism allows methods in different classes to share the same name but perform distinct tasks.

**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.

**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.

**Inheritance Class Polymorphism**

Inheritance-based polymorphism occurs when a subclass overrides a method from its parent class, providing a specific implementation. This process of re-implementing a method in the child class is known as **method overriding**

**Method Overloading in Python**

In many programming languages like C++ or Java, you can define multiple methods with the same name but different parameter lists. This concept is called **method overloading**.

Python does not support method overloading by default. If you define multiple methods with the same name, only the latest definition will be used.

**1. Using Variable Arguments (\*args)**

In this approach, we use variable-length arguments to accept any number of arguments and handle them inside the function.

**def** add(datatype, \*args):

**if** datatype == 'int':

res = 0

**elif** datatype == 'str':

res = ''

**for** item **in** args:

res += item

print(res)

add('int', 5, 6)

add('str', 'Hi ', 'Geeks')

**2. Using Default Arguments (None as default value)**

**def** add(a=**None**, b=**None**):

**if** a **is** **not** **None** **and** b **is** **None**:

print(a)

**else**:

print(a + b)

add(2, 3)

add(2)

**3. Using Multiple Dispatch**

**from** **multipledispatch** **import** dispatch

@dispatch(int, int)

**def** product(first, second):

result = first \* second

print(result)

@dispatch(int, int, int)

**def** product(first, second, third):

result = first \* second \* third

print(result)