**Polymorphism** in Python lets you use the same function name for different types of objects.

* This means one function or method can act in many forms.
* It makes your code flexible and easy to manage.
* It helps you write code that can work with various objects.
* You do not need to know their exact type.
* This saves time.
* It also makes your programs more adaptable.

**3 Ways to Use Polymorphism in Python**

You can use polymorphism in Python in a few ways:

* **With Built-in Functions:** Python’s own functions often use polymorphism.
* **With Class Methods:** Different classes can have methods that share the same time.
* **With Inheritance (Method Overriding):** Child classes can change methods they get from their parent classes.

**Polymorphism with Built-in Function:**

Many Built-in Python functions work in different ways. Their actions changes based on the type of data you give them. The len() function is good example.

* When you use len() on a string, it counts the characters.
* When you use len() on a list, it counts the items.
* When you use len() on a dictionary, it counts the keys.

**Here’s how len () works:**

**#**Use len() with a string

my\_string = “hello”

print(len(my\_string))

#Use len() with a list

my\_list = (1,2,3,4)

print(len(my\_list))

#Use len() with a dictionary

my\_dict = {“a”:1,”b”:2}

print(len(my\_dict))

**Polymorphism using Class Methods:**

You can define methods with the same name in different classes. These classes do not need to be related though inheritance. You can still call the same method on objects created from these different classes.

Think about these two classes: Dog and Cat. Both have a sound() method.

class Dog:

def sound(self):

print(“Woof!”)

class Cat:

def sound(self):

print(“Meow!”)

#Make objects (instances) from these classes

my\_dod=Dog()

my\_cat=Cat()

#Call the same method on different objects

my\_dog.sound()

my\_cat.sound()