



**Department of Computer Science and Engineering  
PES University, Bangalore, India**

# **Lecture Notes Python for Computational Problem Solving UE23CS151A**

**Lecture #102  
*Polymorphism***

**By,  
Prof. Sindhu R Pai,  
Anchor, PCPS - 2023  
Assistant Professor  
Dept. of CSE, PESU  
&  
Dr. Ramya C,  
Associate Professor  
Dept. of CSE, PESU**

**Many Thanks to  
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former  
Chairperson, CSE, PES University)  
Prof. Chitra G M (Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)**

# Introduction

The term “**POLYMORPHISM**” is derived from two distinct Greek terms: **poly**, which means **numerous** or **many**, and **morphs**, which means **forms**. This term is used in Python to refer to an **object's [functions, operators, classes] ability to take on multiple forms with the same name**.

## Consider the builtin function – len()

```
x = "Hello World!"  
print(len(x))
```

```
>>> -- - - -  
= RESTART: C:/Users/c1  
py  
12  
>>>
```

```
mytuple = ("apple", "banana", "cherry")  
print(len(mytuple))
```

```
>>>  
= RESTART: C:/Users/  
py  
3  
>>>
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(len(thisdict))
```

```
>>>  
= RESTART: C:/Users/  
py  
3  
>>>
```

The same function len() is working on different types of objects. Hence this is **function polymorphism**

## Polymorphism with function

To induce polymorphism using a function, we need to create such a function that can take any object.

```
class Dog:  
    def animal_kingdom(self):  
        print("Mammal")  
    def legs(self):  
        print("Four")  
class Lizard:
```

```
def animal_kingdom(self):
    print("Not Mammal")
def legs(self):
    print("Two+Two")
def function1(obj):
    obj.animal_kingdom()
    obj.legs()
d = Dog()
li = Lizard()
function1(d)
function1(li)
```

```
C:\Users\Dell>python hello.py
Mammal
Four
Not Mammal
Two+Two
```

## Polymorphism with methods of the class

Python uses different class types in the same way. Create a for loop that iterates through a tuple of objects. Call the methods without being concerned about which class type each object belongs to. Assumption is that these methods actually exist in each class. In the above example code, driver code is as below

```
for obj in (Dog(), Lizard()):
    obj.animal_kingdom()
    obj.legs()
```

```
C:\Users\Dell>python hello.py
Mammal
Four
Not Mammal
Two+Two
```

## Polymorphism with Operators

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two sequences. It is achievable because '+' operator is overloaded by int class, str class and other classes of sequences by using the concept of duck typing. **The same built-in operator shows different behavior for objects of different classes, this is called Operator Overloading.** Same way, \* works differently on different objects. It is a polymorphic nature of any object oriented programming.

**Note: There is no function/method overloading in python**

When we use + operator, special method `__add__` is automatically invoked in which the operation for + operator is defined. Using + on types or using `__add__` is same.

```
>>> 3+5
8
>>> int.__add__(3,5)
8
>>>
```

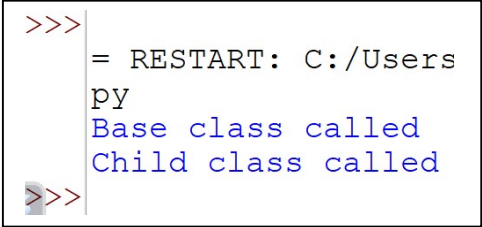
There by changing the definition of this method, we can give extra meaning to the + operator for our type. This is known as **overriding of methods**

### Method Overriding in Polymorphism (Run-time polymorphism)

We know that the child class inherits all the methods from the parent class .However, there will be situations where the definitions in parent class wont be applicable to child class. In such cases we need to reform the method in the child class. This process is known as **method overriding**. This is an ability of a language that **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**. When a method in a subclass has the **same name, same parameters or signature and same return type(or sub-type)** as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

#### Example\_code\_1: Demoing overriding of methods

```
class A:
    def display(self):
        print("Base class called")
class B(A):
    def display(self):
        print("Child class called")
a1=A()
b1=B()
a1.display()
b1.display()
```



```
>>>
= RESTART: C:/Users
py
Base class called
Child class called
>>>
```

The question that arises is as to which method will get executed when the function is invoked. The answer to this is it **depends on the object that is invoking the function(method)**. If the object of child class invokes, then the method in child class will be executed and if the object of parent class invokes, then the method in parent class gets executed.

**Example\_code\_6:**

```
class A:
    def explore(self):
        print("explore() method from class A")
class B(A):
    def explore(self):
        print("explore() method from class B")
b = B() #object of child class created
a = A()
b.explore() # override the baseclass explore()
a.explore()
```

```
>>>
= RESTART: C:/Users/cramy/AppDa
py
explore() method from class B
explore() method from class A
>>>
```

Now let us create our own type called Person and perform addition of two Person objects by concatenating two names of these persons

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
p1 = Person("SINDHU",30)
p2 = Person("SHYAMA",35)
p3 = Person("JAYA", 45)
print(p1+p2)#addition of two person objects
print("ended")
```

```
C:\Users\Dell>python classtesting.py
Traceback (most recent call last):
  File "C:\Users\Dell\classtesting.py", line 8, in <module>
    print(p1+p2)
    ~~~~
TypeError: unsupported operand type(s) for +: 'Person' and 'Person'
```

Note that the + operator not supported between Person objects. We need to override `__add__` in Person.

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def __add__(self, p): #overridden method
        return self.name + " "+p.name
```

```
p1 = Person("SINDHU",30)
p2 = Person("SHYAMA",35)
p3 = Person("JAYA", 45)
print(p1+p2)
print("ended")
```

```
C:\Users\Dell>python classtesting.py
SINDHU SHYAMA
ended
```

Now can we compare two person objects based on the age of them?

`print(p1 > p2)` # this statement throws `TypeError` saying `>` operator not supported.

To customize the behavior of the greater than operator `x > y`, you can override the `__gt__()` method in your class definition. Python internally calls `x.__gt__(y)` to obtain a return value when comparing two objects using `x > y`. The return value can be of any data type because any value can automatically be converted to a Boolean.

**Syntax: `__gt__(self, other)`**

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def __add__(self, p):
        return self.name + " "+p.name
    def __gt__(self,p):
        return self.age > p.age
```

```
p1 = Person("SINDHU",30)
p2 = Person("SHYAMA",35)
p3 = Person("JAYA", 45)
print(p1+p2)
if (p1 > p2):
    print(p1.name, "is older with age",p1.age)
else:
    print(p1.name, "is younger or same as as",p2.name)
print("ended")
```

```
C:\Users\Dell>python classtesting.py
SINDHU SHYAMA
SINDHU is younger or same as as SHYAMA
ended
```

Like this, operator for which you want the special meaning in your own type, you need to override those respective functions in the type created.

Operator	Special Function in Python	Operator	Special Function in Python
+	<code>__add__()</code>	<code>&lt;=</code>	<code>__le__()</code>
-	<code>__sub__()</code>	<code>&gt;=</code>	<code>__ge__()</code>
*	<code>__mul__()</code>	<code>==</code>	<code>__eq__()</code>
/	<code>__truediv__()</code>	<code>!=</code>	<code>__ne__()</code>
%	<code>__mod__()</code>	<code>in</code>	<code>__contains__()</code>
<code>&lt;</code>	<code>__lt__()</code>	<code>len</code>	<code>__len__()</code>
<code>&gt;</code>	<code>__gt__()</code>	<code>str</code>	<code>__str__()</code>

**-END-**