**Polymorphism in Python**

Updated on: October 21, 2021 | [15 Comments](https://pynative.com/python-polymorphism/#llc_comments)

[Object-Oriented Programming](https://pynative.com/python/object-oriented-programming/) (OOP) has four essential characteristics: abstraction, encapsulation, inheritance, and polymorphism.

This lesson will cover what polymorphism is and how to implement them in Python. Also, you’ll learn how to implement polymorphism using function overloading, method overriding, and operator overloading.

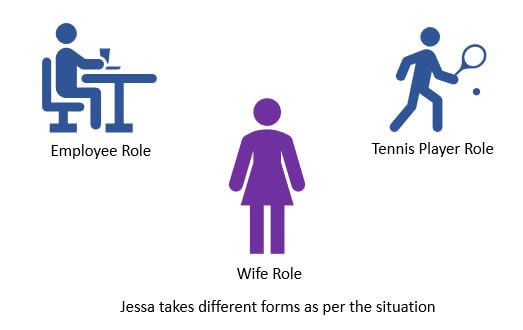
**Table of contents**

* [What is Polymorphism in Python?](https://pynative.com/python-polymorphism/#h-what-is-polymorphism-in-python)
  + [Polymorphism in Built-in function len()](https://pynative.com/python-polymorphism/#h-polymorphism-in-built-in-function-len)
* [Polymorphism With Inheritance](https://pynative.com/python-polymorphism/#h-polymorphism-with-inheritance)
  + [Example: Method Overriding](https://pynative.com/python-polymorphism/#h-example-method-overriding)
  + [Overrride Built-in Functions](https://pynative.com/python-polymorphism/#h-overrride-built-in-functions)
* [Polymorphism In Class methods](https://pynative.com/python-polymorphism/#h-polymorphism-in-class-methods)
  + [Polymorphism with Function and Objects](https://pynative.com/python-polymorphism/#h-polymorphism-with-function-and-objects)
* [Polymorphism In Built-in Methods](https://pynative.com/python-polymorphism/#h-polymorphism-in-built-in-methods)
* [Method Overloading](https://pynative.com/python-polymorphism/#h-method-overloading)
* [Operator Overloading in Python](https://pynative.com/python-polymorphism/#h-operator-overloading-in-python)
  + [Overloading + operator for custom objects](https://pynative.com/python-polymorphism/#h-overloading-operator-for-custom-objects)
  + [Overloading the \* Operator](https://pynative.com/python-polymorphism/#h-overloading-the-operator)
  + [Magic Methods](https://pynative.com/python-polymorphism/#h-magic-methods)

**What is Polymorphism in Python?**

Polymorphism in Python is the ability of an [object](https://pynative.com/python-classes-and-objects/) to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.

For example, Jessa acts as an employee when she is at the office. However, when she is at home, she acts like a wife. Also, she represents herself differently in different places. Therefore, the same person takes different forms as per the situation.

A person takes different forms

In polymorphism, a method can **process objects differently depending on the class type or data type**. Let’s see simple examples to understand it better.

**Polymorphism in Built-in function len()**

The built-in function len() calculates the length of an object depending upon its type. If an object is a string, it returns the count of characters, and If an object is a [list](https://pynative.com/python-lists/), it returns the count of items in a list.

The len() method treats an object as per its class type.

**Example**:

students = ['Emma', 'Jessa', 'Kelly']

school = 'ABC School'

# calculate count

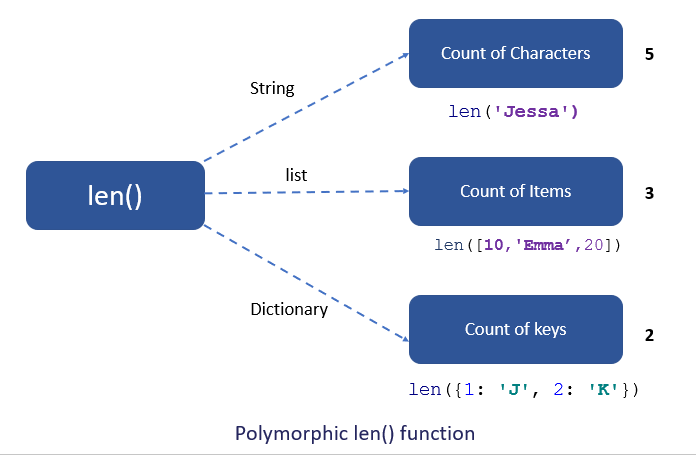
**print**(**len**(students))

**print**(**len**(school))

**Output**

3

10

Polymorphic len() function

**Polymorphism With Inheritance**

Polymorphism is mainly used with inheritance. In [**inheritance**](https://pynative.com/python-inheritance/), child class inherits the attributes and methods of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

Using **method overriding** polymorphism allows us to defines methods in the child [class](https://pynative.com/python-classes-and-objects/) that have the same name as the methods in the parent class. This **process of re-implementing the inherited method in the child class** is known as Method Overriding.

**Advantage of method overriding**

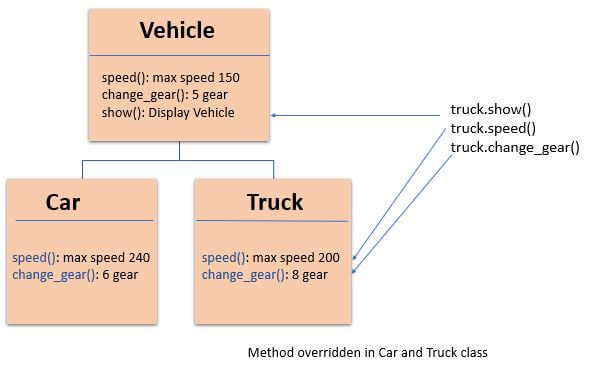
* It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn’t fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
* Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don’t need to modification the parent class code

In polymorphism, **Python first checks the object’s class type and executes the appropriate method** when we call the method. For example, If you create the Car object, then Python calls the speed() method from a Car class.

Let’s see how it works with the help of an example.

**Example: Method Overriding**

In this example, we have a vehicle class as a parent and a ‘Car’ and ‘Truck’ as its sub-class. But each vehicle can have a different seating capacity, speed, etc., so we can have the same [instance method](https://pynative.com/python-instance-methods/) name in each class but with a different implementation. Using this code can be extended and easily maintained over time.

Polymorphism with Inheritance

**class** Vehicle:

**def** \_\_init\_\_(self, name, color, price):

self.name = name

self.color = color

self.price = price

**def** show(self):

**print**('Details:', self.name, self.color, self.price)

**def** max\_speed(self):

**print**('Vehicle max speed is 150')

**def** change\_gear(self):

**print**('Vehicle change 6 gear')

# inherit from vehicle class

**class** Car(Vehicle):

**def** max\_speed(self):

**print**('Car max speed is 240')

**def** change\_gear(self):

**print**('Car change 7 gear')

# Car Object

car = Car('Car x1', 'Red', 20000)

car.show()

# calls methods from Car class

car.max\_speed()

car.change\_gear()

# Vehicle Object

vehicle = Vehicle('Truck x1', 'white', 75000)

vehicle.show()

# calls method from a Vehicle class

vehicle.max\_speed()

vehicle.change\_gear()

**Output**:

Details: Car x1 Red 20000

Car max speed is 240

Car change 7 gear

Details: Truck x1 white 75000

Vehicle max speed is 150

Vehicle change 6 gear

As you can see, due to polymorphism, the Python interpreter recognizes that the max\_speed() and change\_gear() methods are overridden for the car object. So, it uses the one defined in the child class (Car)

On the other hand, the show() method isn’t overridden in the Car class, so it is used from the Vehicle class.

**Overrride Built-in Functions**

In Python, we can change the default behavior of the built-in functions. For example, we can change or extend the built-in functions such as len(), abs(), or divmod() by redefining them in our class. Let’s see the example.

**Example**

In this example, we will redefine the function len()

**class** Shopping:

**def** \_\_init\_\_(self, basket, buyer):

self.basket = **list**(basket)

self.buyer = buyer

**def** \_\_len\_\_(self):

**print**('Redefine length')

count = **len**(self.basket)

# count total items in a different way

# pair of shoes and shir+pant

**return** count \* 2

shopping = Shopping(['Shoes', 'dress'], 'Jessa')

**print**(**len**(shopping))

**Output**

Redefine length

4

**Polymorphism In Class methods**

Polymorphism with class methods is useful when we group different objects having the same method. we can add them to a list or a tuple, and we don’t need to check the object type before calling their methods. Instead, Python will check object type attime and call the correct method. Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

Python allows different classes to have methods with the same name.

* Let’s design a different class in the same way by adding the same methods in two or more classes.
* Next, create an object of each class
* Next, add all objects in a [tuple](https://pynative.com/python-tuples/).
* In the end, iterate the tuple using a [for loop](https://pynative.com/python-for-loop/) and call methods of a object without checking its class.

**Example**

In the below example, fuel\_type() and max\_speed() are the instance methods created in both classes.

**class** Ferrari:

**def** fuel\_type(self):

**print**("Petrol")

**def** max\_speed(self):

**print**("Max speed 350")

**class** BMW:

**def** fuel\_type(self):

**print**("Diesel")

**def** max\_speed(self):

**print**("Max speed is 240")

ferrari = Ferrari()

bmw = BMW()

# iterate objects of same type

**for** car **in** (ferrari, bmw):

# call methods without checking class of object

car.fuel\_type()

car.max\_speed()

**Output**

Petrol

Max speed 350

Diesel

Max speed is 240

As you can see, we have created two classes Ferrari and BMW. They have the same instance method names fuel\_type() and max\_speed(). However, we have not linked both the classes nor have we used inheritance.

We packed two different objects into a [tuple](https://pynative.com/python-tuples/) and iterate through it using a car variable. It is possible due to polymorphism because we have added the same method in both classes Python first checks the object’s class type and executes the method present in its class.

**Polymorphism with Function and Objects**

We can create polymorphism with a function that can take any [object](https://pynative.com/python-classes-and-objects/) as a parameter and execute its method without checking its class type. Using this, we can call object actions using the same function instead of repeating method calls.

**Example**

**class** Ferrari:

**def** fuel\_type(self):

**print**("Petrol")

**def** max\_speed(self):

**print**("Max speed 350")

**class** BMW:

**def** fuel\_type(self):

**print**("Diesel")

**def** max\_speed(self):

**print**("Max speed is 240")

# normal function

**def** car\_details(obj):

obj.fuel\_type()

obj.max\_speed()

ferrari = Ferrari()

bmw = BMW()

car\_details(ferrari)

car\_details(bmw)

**Output**

Petrol

Max speed 350

Diesel

Max speed is 240

**Polymorphism In Built-in Methods**

The word polymorphism is taken from the Greek words poly (many) and morphism (forms). It means a **method can process objects differently depending on the class type or data type.**

The built-in function reversed(obj) returns the iterable by reversing the given object. For example, if you pass a string to it, it will reverse it. But if you pass a list of strings to it, it will return the iterable by reversing the order of elements (it will not reverse the individual string).

Let us see how a built-in method process objects having different data types.

**Example**:

students = ['Emma', 'Jessa', 'Kelly']

school = 'ABC School'

**print**('Reverse string')

**for** i **in** **reversed**('PYnative'):

**print**(i, end=' ')

**print**('\nReverse list')

**for** i **in** **reversed**(['Emma', 'Jessa', 'Kelly']):

**print**(i, end=' ')

**Output**:

Reverse string

e v i t a n Y P

Reverse list

Kelly Jessa Emma

**Method Overloading**

The process of calling the same method with different parameters is known as method overloading. Python does not support method overloading. Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method.

**Example**

**def** addition(a, b):

c = a + b

**print**(c)

**def** addition(a, b, c):

d = a + b + c

**print**(d)

# the below line shows an error

# addition(4, 5)

# This line will call the second product method

addition(3, 7, 5)

To overcome the above problem, we can use different ways to achieve the method overloading. In Python, to overload the class method, we need to write the method’s logic so that different code executes inside the function depending on the parameter passes.

For example, the built-in function [range()](https://pynative.com/python-range-function/) takes three parameters and produce different result depending upon the number of parameters passed to it.

**Example**:

**for** i **in** **range**(5): **print**(i, end=', ')

**print**()

**for** i **in** **range**(5, 10): **print**(i, end=', ')

**print**()

**for** i **in** **range**(2, 12, 2): **print**(i, end=', ')

**Output**:

0, 1, 2, 3, 4,

5, 6, 7, 8, 9,

2, 4, 6, 8, 10,

Let’s assume we have an area() [method](https://pynative.com/python-instance-methods/) to calculate the area of a square and rectangle. The method will calculate the area depending upon the number of parameters passed to it.

* If one parameter is passed, then the area of a square is calculated
* If two parameters are passed, then the area of a rectangle is calculated.

**Example**: User-defined polymorphic method

**class** Shape:

# function with two default parameters

**def** area(self, a, b=0):

**if** b > 0:

**print**('Area of Rectangle is:', a \* b)

**else**:

**print**('Area of Square is:', a \*\* 2)

square = Shape()

square.area(5)

rectangle = Shape()

rectangle.area(5, 3)

**Output**:

Area of Square is: 25

Area of Rectangle is: 15

**Operator Overloading in Python**

Operator overloading means changing the default behavior of an [operator](https://pynative.com/python-operators/) depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes.

For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings.

The operator + is used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

**Example**:

# add 2 numbers

**print**(100 + 200)

# concatenate two strings

**print**('Jess' + 'Roy')

# merger two list

**print**([10, 20, 30] + ['jessa', 'emma', 'kelly'])

**Output**:

300

JessRoy

[10, 20, 30, 'jessa', 'emma', 'kelly']

**Overloading + operator for custom objects**

Suppose we have two objects, and we want to add these two objects with a binary + operator. However, it will throw an error if we perform addition because the compiler doesn’t add two objects. See the following example for more details.

**Example**:

**class** Book:

**def** \_\_init\_\_(self, pages):

self.pages = pages

# creating two objects

b1 = Book(400)

b2 = Book(300)

# add two objects

**print**(b1 + b2)

**Output**

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'

We can overload + operator to work with custom objects also. Python provides some special or magic function that is automatically invoked when associated with that particular operator.

For example, when we use the + operator, the magic method \_\_add\_\_() is automatically invoked. Internally + operator is implemented by using \_\_add\_\_() method. We have to override this method in our class if you want to add two custom objects.

**Example**:

**class** Book:

**def** \_\_init\_\_(self, pages):

self.pages = pages

# Overloading + operator with magic method

**def** \_\_add\_\_(self, other):

**return** self.pages + other.pages

b1 = Book(400)

b2 = Book(300)

**print**("Total number of pages: ", b1 + b2)

**Output**

Total number of pages: 700

**Overloading the \* Operator**

The \* operator is used to perform the multiplication. Let’s see how to overload it to calculate the salary of an employee for a specific period. Internally \* operator is implemented by using the \_\_mul\_\_() method.

**Example**:

**class** Employee:

**def** \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

**def** \_\_mul\_\_(self, timesheet):

**print**('Worked for', timesheet.days, 'days')

# calculate salary

**return** self.salary \* timesheet.days

**class** TimeSheet:

**def** \_\_init\_\_(self, name, days):

self.name = name

self.days = days

emp = Employee("Jessa", 800)

timesheet = TimeSheet("Jessa", 50)

**print**("salary is: ", emp \* timesheet)

**Output**

Wroked for 50 days

salary is: 40000

**Magic Methods**

In Python, there are different magic methods available to perform overloading operations. The below table shows the magic methods names to overload the mathematical operator, assignment operator, and relational operators in Python.

| **Operator Name** | **Symbol** | **Magic method** |
| --- | --- | --- |
| Addition | + | \_\_add\_\_(self, other) |
| Subtraction | - | \_\_sub\_\_(self, other) |
| Multiplication | \* | \_\_mul**\_\_**(self, other) |
| Division | / | \_\_div\_\_(self, other) |
| Floor Division | // | \_\_floordiv\_\_(self,other) |
| Modulus | % | \_\_mod\_\_(self, other) |
| Power | \*\* | \_\_pow\_\_(self, other) |
| Increment | += | \_\_iadd\_\_(self, other) |
| Decrement | -= | \_\_isub\_\_(self, other) |
| Product | \*= | \_\_imul\_\_(self, other) |
| Division | /+ | \_\_idiv\_\_(self, other) |
| Modulus | %= | \_\_imod\_\_(self, other) |
| Power | \*\*= | \_\_ipow\_\_(self, other) |
| Less than | < | \_\_lt\_\_(self, other) |
| Greater than | > | \_\_gt\_\_(self, other) |
| Less than or equal to | <= | \_\_le\_\_(self, other) |
| Greater than or equal to | >= | \_\_ge\_\_(self, other) |
| Equal to | == | \_\_eq\_\_(self, other) |
| Not equal | != | \_\_ne\_\_(self, other) |