Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on.

In Python, polymorphism refers to the way in which different object classes can share the same method name,

and those methods can be called from the same place even though a variety of different objects might be passed in.

The best way to explain this is by example: python has builtin polymorphism and it can be achieved by

1.Duck typing philosophy of python

2.Operator overloading

3.Method overloading

4.Method overriding

Polymorphism is the ability to send the same message (request to run a method) to

different objects, each of which appear to perform the same function. However, the

way in which the message is handled depends on the object’s class.

Polymorphism is a strange sounding word, derived from Greek, for a relatively

simple concept. It is essentially the ability to request that the same operation be

performed by a wide range of different types of things. How the request is processed depends on the thing that receives the request. The programmer need not

worry about how the request is handled, only that it is. This is illustrated below.

Add (c, d)

Add(2,3.5)

Add(float c, float d)

Add(int c, int d,float a)

//overloading

Add(10,10.5)

base class

sleep

\derive class

eat

class base:

method1()

pass

class derive(base):

methods()

pass

class derive1(base):

pass

base \*ptr= &dervie ,&devive1

base->methods()

base class pointer = &dervieclass

bptr->eat

bptr->sleep

a = 10

a = first()

#Duck typing philosophy of python , it checks if the object passed to method belongs to that class it will invoke that method

def night\_out(p):

p.eat()

p.drink()

p.sleep()

In this example, the parameter passed into the night\_out() function expects to be given something that will respond to the methods eat(), drink() and sleep(). Any object that meets this requirement can be used with the function. We can define multiple classes that meet this informal contract, for example we can define a class hierarchy that provides these methods, or completely separate classes that implement the methods. In the case of the class hierarchy the methods may or may not override those from the parent class. Effectively, this means that you can ask many different things to perform the same action. For example, you might ask a range of objects to provide a printable string describing themselves. In fact in Python this is exactly what happens. For exmaple, if you ask an instance of a Manager class, a compiler object or a database object to return such a string, you use the same method (**str**(), in Python). The name polymorphism is unfortunate and often leads to confusion. It makes the whole process sound rather grander than it actually is. Note this is one of the most significant and flexible features of Python; it does not tie a variable to a specific type; instead via Duck Typing as long as the object provided meets the implied contract, then we are good. The following classes all meet the contract implied by the night\_out() function:

**class** Person:

**def** eat(self):

print('Person - Eat')

**def** drink(self):

print('Person - Drink')

**def** sleep(self):

print('Person - Sleep')

**class** Employee(Person):

**def** eat(self):

print('Employee - Eat')

**def** drink(self):

print('Employee - Drink')

**def** sleep(self):

print('Employee - Sleep')

**class** SalesPerson(Employee):

**def** eat(self):

print('SalesPerson - Eat')

**def** drink(self):

print('SalesPerson - Drink')

**class** Dog:

**def** eat(self):

print('Dog - Eat')

**def** drink(self):

print('Dog - Drink')

**def** sleep(self):

print('Dog - Sleep')

This means that instances of all of these classes can be used with the night\_out() function. Note that the SalesPerson class meets the implied contract partly via inheritance (the sleep() method is inherited from Employee).

**def** night\_out(p):

p.eat()

p.drink()

p.sleep()

p.bark() **--**error

​

d **=** Dog()

​

night\_out(d)

​

Sd **=** SalesPerson()

night\_out(Sd)

Dog - Eat

Dog - Drink

Dog - Sleep

SalesPerson - Eat

SalesPerson - Drink

Employee - Sleep

*#Method Overriding*

*#Where the method of same name present in base class when inherited can be over ridden in the derived class*

​

**class** Animal:

**def** \_\_init\_\_(self, name**=**''):

self.name **=** name

​

**def** talk(self):

**pass**

**def** getHeadCount(self):

**return** 1

​

**class** Cat(Animal):

**def** talk(self):

print ("Meow!")

​

​

**class** Dog(Animal):

**def** talk(self):

print ("Woof!")

a **=** Animal()

a.talk()

​

cat **=** Cat("Missy")

cat.talk()

print(cat.getHeadCount())

​

dog **=** Dog("Rocky")

dog.talk()

print(dog.getHeadCount())

Meow!

1

Woof!

1

*#Operator Overloading*

​

**class** BookX:

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

self.pages **=** pages

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

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

**class** BookY:

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

self.pages **=** pages

b1 **=** BookX(100)

b2 **=** BookY(200)

print('Total pages =',b1**+**b2)

*#Method Overloading*

*# Officially Python do not support method overloading and trying to achieve will result in errors as shown in below examples.*

**class** Cars(object):

**def** seats(self):

print("No parameters")

**def** seats(self, seat\_type, count):

print("seat\_type:", seat\_type)

print("count:", count)

**def** seats(self, seat\_type):

print("seat\_type:", seat\_type)

car **=** Cars()

**try**:

car.seats("leather")

**except** Exception **as** e:

print("Error:", e)

Error: seats() takes 2 positional arguments but 3 were given

In the above example, we had two functions **with** same name **and** different numbers of arguments,

Python only use the last defined function **and** previous one **is** discarded.

**class** Cars(object):

**def** seats(self, seat\_type):

print("seat\_type:", seat\_type)

**class** RaceCar(Cars):

**def** seats(self, seat\_type): *#through inheritence*

print("seat\_type:", seat\_type) *#---ignored*

**def** seats(self, seat\_type, count):

print("seat\_type:", seat\_type)

print("count:", count)

car **=** RaceCar()

**try**:

car.seats("leather",1)

**except** Exception **as** e:

print("Error:", e)

seat\_type: leather

count: 1

In the above example, we had two functions **with** same name **and** different numbers of arguments one **in** RaceCar **and**

another one **in** its parent **class** Cars, **and** when we call the function seats **from** an object of RaceCar it rightly calls

seats **from** RaceCar instead of **from** Cars.Now **if** you have a scenario like that **and** then one of the

solution which we can do **is** use the following process.

**class** Cars(object):

**def** seats(self, seat\_type):

print("seat\_type:", seat\_type)

**class** RaceCar(Cars): *#indirect way of achieving the methof overloading*

**def** seats(self, seat\_type, count**=None**):

**if** count **is** **None**:

super().seats(seat\_type)

**else**:

print("seat\_type:", seat\_type)

print("count:", count)

**def** seats(self,**\***args):

**for** i **in** args:

print(i) *#indirect way of achieving the methof overloading*

car **=** RaceCar()

car.seats(10,20,30)

**try**:

*# This will call parent seats*

car.seats(10, "leather")

*# This will call child seats*

print("")

car.seats(10)

**except** Exception **as** e:

print("Error:", e)

**Operator Overloading**

Operator overloading allows user defined classes to appear to have a natural way of using operators such as +, −, <, > or == as well as logical operators such as & (and) and | (or).

To implement operators such as '+' in a user defined class it is necessary to implement specific methods that are then mapped to the arithmetic or logical operators used by users of the class. These methods are considered special in that they start with and end with a double underscore ('\_\_'). Such methods are considered private and usually restricted for Python oriented implementations

​

*# 1+2 = 3*

​

*# "str"+"str" = strstr*

​

​

​

*# obj1 = A()*

*# obj2 = B()*

​

​

​

*# obj1+obj2 #error*

​

Numerical Operators

There are nine different numerical operators that can be implemented by special

methods; these operators are listed in the following table

​

​

Operator                   Expression     Method

Addition                   q1 + q2       \_\_add\_\_(self, q2)

Subtraction               q1 – q2       \_\_sub\_\_(self, q2)

Multiplication             q1 \* q2       \_\_mul\_\_(self, q2)

Power                     q1 \*\* q2     \_\_pow\_\_(self, q2)

Division                   q1 / q2       \_\_truediv\_\_(self, q2)

Floor Division             q1 // q2     \_\_floordiv\_\_(self, q2)

Modulo (Remainder)         q1 % q2       \_\_mod\_\_(self, q2)

Bitwise Left Shift         q1 << q2     \_\_lshift\_\_(self, q2)

Bitwise Right Shift       q1 >> q2     \_\_rshift\_\_(self, q2)

​

​

Comparison Operators

Numerical types (such as integers and real numbers) also support comparison

operators such as equals, not equals, greater than, less than as well as greater than or

equal to and less than or equal to.

Python allows these comparison operators to be defined for user defined types/

classes as well.

​

Operator                   Expression             Method

Less than                   q1 < q2               \_\_lt\_\_(q1, q2)

Less than or equal to       q1 <= q2             \_\_le\_\_(q1, q2)

Equal to                   q1 == q2             \_\_eq\_\_(q1, q2)

Not Equal to               q1 != q2             \_\_ne\_\_(q1, q2)

Greater than               q1 > q2               \_\_gt\_\_(q1, q2)

Greater than or equal to   q1 >= q2             \_\_ge\_\_(q1, q2)

Logical Operators

The final category of operators that can be defined on a class are Logical operators.

These are operators that can be used with and/or type tests; they typically return the

values True or False.

As with the numerical operators and the comparison operators; the logical

operators are implemented by a set of special methods

​

​

Operator Expression Method

AND           q1 & q2         \_\_and\_\_(q1, q2)

OR           q1 | q2         \_\_or\_\_(q1, q2)

XOR           q1 ^ q2         \_\_xor\_\_(q1, q2)

NOT           ~q1             \_\_invert\_\_()

*# Examples*

​

**class** Quantity:

**def** \_\_init\_\_(self, value**=**0):

self.value **=** value

*#Numeric Operators*

**def** \_\_add\_\_(self, other): *#overloading for + symbol*

new\_value **=** self.value **+** other.value

**return** Quantity(new\_value)

**def** \_\_sub\_\_(self, other): *#overloading for - symbol*

new\_value **=** self.value **-** other.value

**return** Quantity(new\_value)

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

new\_value **=** self.value **\*** other.value

**return** Quantity(new\_value)

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

new\_value **=** self.value **\*\*** other.value

**return** Quantity(new\_value)

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

new\_value **=** self.value **/** other.value

**return** Quantity(new\_value)

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

new\_value **=** self.value **//** other.value

**return** Quantity(new\_value)

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

new\_value **=** self.value **%** other.value

**return** Quantity(new\_value)

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

**return** 'Quantity[' **+** str(self.value) **+** ']'

*#Comparison operators*

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

**return** self.value **==** other.value

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

**return** self.value **!=** other.value

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

**return** self.value **>=** other.value

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

**return** self.value **>** other.value

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

**return** self.value **<** other.value

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

**return** self.value **<=** other.value

​

*#numerical operator*

q1 **=** Quantity(5)

q2 **=** Quantity(10)

print(q1)

print('q1 =', q1, ', q2 =', q2)

q3 **=** q1 **+** q2 *#q1.\_\_add\_\_(q2)*

print('q3 =', q3) *#q3.\_\_str\_\_()*

print('q2 - q1 =', q2 **-** q1) *#q2.\_\_sub\_\_(q1)*

print('q1 \* q2 =', q1 **\*** q2)

print('q1 / q2 =', q1 **/** q2)

​

​

*#Comparison operators*

print('q1 < q2: ', q1 **<** q2)

print('q3 > q2: ', q3 **>** q2)

print('q3 == q1: ', q3 **==** q1) *#Objects created will be having different id's hence while comparing it will be false*

​

​

​

Quantity[5]

q1 = Quantity[5] , q2 = Quantity[10]

q3 = Quantity[15]

q2 - q1 = Quantity[5]

q1 \* q2 = Quantity[50]

q1 / q2 = Quantity[0.5]

q1 < q2: True

q3 > q2: True

q3 == q1: False

​