Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

We achieve polymorphism by overloading, overriding and duck typing.

Overloading:

- It means same name can have different meaning.
- Python doesn't support it directly like other languages (Java,C++) instead python has provided a different way to achieve it.
- We can achieve overloading by default arguments and variable length arguments.
- We can overload a function, method, constructor and operator.

Function overloading:

It means same function name with different meaning.

```
#Without overloading
def add(a,b): #old def
        print(a+b)

def add(a,b,c): #latest def
        print(a+b+c)

add(10,20) #Error
add(30,40,50)
```

There will be error in first function call as interpreter always consider latest definition and according to latest definition add function consists of 3 parameters but while calling 2 were given.

```
#Overloading by default arguments:
def add(a = None,b = None,c = None):
    if a!= None and b!= None and c != None:
        print(a+b+c)
    elif a!= None and b!= None:
        print(a+b)
    elif a!= None:
        print(a)
```

```
    else:
        print(0)

add()
add(10)
add(10,20)
add(10,20,30)
add(10,20,30,40) #Error
```

OUTPUT:

```
0
10
30
60
TypeError: add() takes from 0 to 3 positional arguments but 4 were given
```

In default arguments, there is one drawback that we can give at most that many arguments which we have defined as default in function definition. So last function call will generate error. We can overcome this drawback in variable length arguments.

#Overloading by variable length arguments:

```
def add(*args):
    sum = 0
    for i in args:
        sum += i
    print(sum)

add()
add(10)
add(10,20)
add(10,20,30)
add(10,20,30,40)
```

OUTPUT:

```
0
10
30
60
100
```

In above program, we can provide any no. of arguments to add function as we have written *args in function definition. The arguments will be collected in tuple format.

Method overloading:

#Default arguments:

```
class A:
    def add(self,a = None,b = None,c = None):
        if a!= None and b!= None and c != None:
            print(a+b+c)
        elif a!= None and b!= None:
            print(a+b)
        elif a!= None:
            print(a)
        else:
            print(0)

a = A()
a.add()
a.add(10)
a.add(10,20)
a.add(10,20,30)
a.add(10,20,30,40)
```

OUTPUT:

```
0
10
30
60
TypeError: add() takes from 1 to 4 positional arguments but 5 were given
```

#Variable length arguments:

```
class A:
    def add(self,*args):
        sum = 0
        for i in args:
            sum += i
        print(sum)
a = A()
a.add()
a.add(10)
```

a.add(10,20)
a.add(10,20,30)
a.add(10,20,30,40)

OUTPUT:
0
10
30
60
100

Constructor overloading:

#Default arguments

```
class A:
    def __init__(self,a = None,b = None,c = None):
        if a!= None and b!= None and c != None:
            print(a+b+c)
        elif a!= None and b!= None:
            print(a+b)
        elif a!= None:
            print(a)
        else:
            print(0)

a = A()
a1 = A(10)
a2 = A(10,20)
a3 = A(10,20,30)
a4 = A(10,20,30,40)
```

OUTPUT:
0
10
30
60
TypeError: __init__() takes from 1 to 4 positional arguments but 5 were given

#Variable length arguments:

```
class A:
    def __init__(self,*args):
        sum = 0
        for i in args:
```

```
        sum += i
    print(sum)


a = A()
a1 = A(10)
a2 = A(10,20)
a3 = A(10,20,30)
a4 = A(10,20,30,40)
```

OUTPUT:

```
0
10
30
60
100
```

Operator overloading:

Operator overloading means changing the default behaviour of an operator 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 simplest occurrences of polymorphism in Python.

For e.g,

```
#add 2 integers
print(10+20) #30

#add 2 strings
print('Hello'+'World') #HelloWorld

#merge 2 lists
print([1,2,3] + [4,5,6]) #[1,2,3,4,5,6]
```

Overloading operator for custom objects:

```
class Book:
    def __init__(self,pages):
        self.pages = pages
```

```
b1 = Book(400)
b2 = Book(300)

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.

```
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

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

Operator Name	Symbol	Magic method
Addition	+	__add__(self, other)
Subtraction	-	__sub__(self, other)
Multiplication	*	__mul__(self, other)
Division	/	__truediv__(self, other)
Floor Division	//	__floordiv__(self,other)
Modulus	%	__mod__(self, other)
Power	**	__pow__(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)

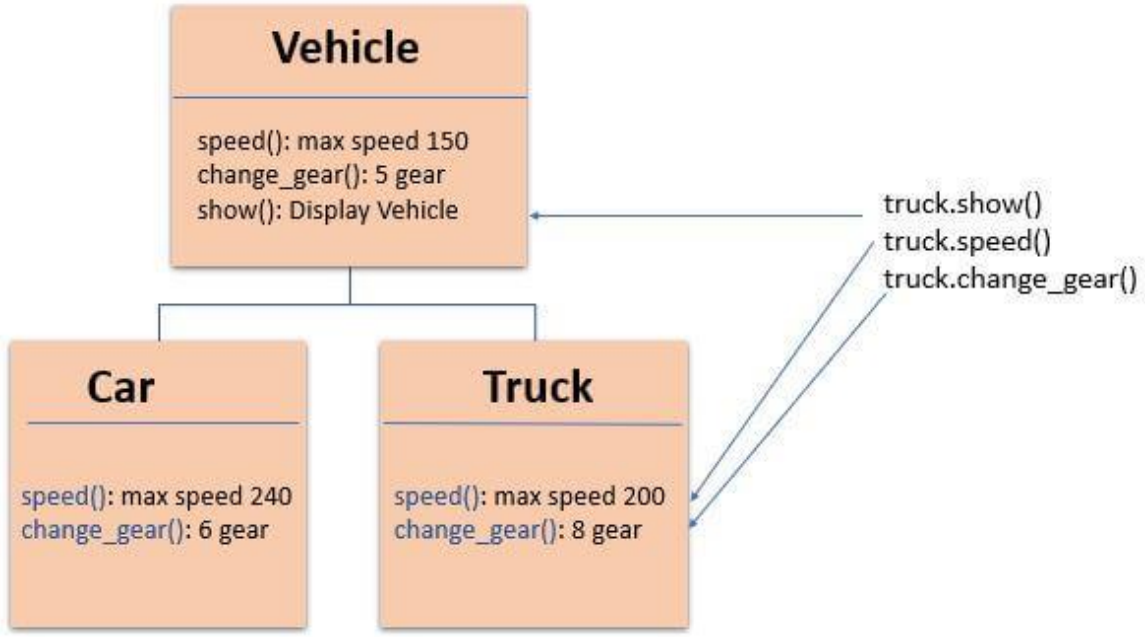
Overriding:

Polymorphism is mainly used with inheritance. In 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 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.



Method overridden in Car and Truck class

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 name in each class but with a different implementation. Using this code can be extended and easily maintained over time.

```
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')

class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')

    def change_gear(self):
        print('Car change 7 gear')

car = Car('Car x1', 'Red', 20000)
car.show()
car.max_speed()
car.change_gear()

vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
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
```

Duck Typing:

It means defining a common interface to call methods of different classes if the methods have same name.

We can create polymorphism with a function that can take any object 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.

```
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
```