Operator Overloading

1. **Mathematical Operators:**

For the mathematical operators (`+`, `-`, `*`, `/`, `//`, `%`, `**`), you can define the following methods in your class:

- `__add__(self, other)`
- `__sub__(self, other)`
- `__mul__(self, other)`
- `__truediv__(self, other)`
- `__floordiv__(self, other)`
- `__mod__(self, other)`
- `__pow__(self, other)`

2. **Comparison Operators:**

For the comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`), you can define:

- `__lt__(self, other)` (less than)
- `__gt__(self, other)` (greater than)
- `__le__(self, other)` (less than or equal)
- `__ge__(self, other)` (greater than or equal)
- `__eq__(self, other)` (equal)
- `__ne__(self, other)` (not equal)

In [2]:

```python
class CustomNumber:
    def __init__(self, value):
        self.value = value

    # Mathematical operators
    def __add__(self, other):
        return CustomNumber(self.value + other.value)

    def __sub__(self, other):
        return CustomNumber(self.value - other.value)

    def __mul__(self, other):
        return CustomNumber(self.value * other.value)

    def __truediv__(self, other):
        return CustomNumber(self.value / other.value)

    def __floordiv__(self, other):
        return CustomNumber(self.value // other.value)

    def __mod__(self, other):
        return CustomNumber(self.value % other.value)

    def __pow__(self, other):
        return CustomNumber(self.value ** other.value)

    # Comparison operators
    def __lt__(self, other):
        return self.value < other.value

    def __gt__(self, other):
        return self.value > other.value

    def __le__(self, other):
        return self.value <= other.value

    def __ge__(self, other):
        return self.value >= other.value

    def __eq__(self, other):
        return self.value == other.value

    def __ne__(self, other):
        return self.value != other.value

    def __repr__(self):
        return f"CustomNumber({self.value})"

# Example Usage
a = CustomNumber(10)
b = CustomNumber(5)

print(a + b)    # CustomNumber(15)
print(a - b)    # CustomNumber(5)
print(a * b)    # CustomNumber(50)
print(a / b)    # CustomNumber(2.0)
print(a // b)   # CustomNumber(2)
print(a % b)    # CustomNumber(0)
print(a ** b)   # CustomNumber(100000)

print(a < b)    # False
```

```
print(a > b)    # True
print(a <= b)   # False
print(a >= b)   # True
print(a == b)   # False
print(a != b)   # True

```

```
CustomNumber(15)
CustomNumber(5)
CustomNumber(50)
CustomNumber(2.0)
CustomNumber(2)
CustomNumber(0)
CustomNumber(100000)
False
True
False
True
```

In [1]:
```
class Fraction:

  # parameterized constructor
  def __init__(self,x,y):
    self.num = x
    self.den = y

  def __str__(self):
    return '{}/{}'.format(self.num,self.den)
```

In [2]:
```
fr1 = Fraction(3,4)
fr2 = Fraction(1,2)
```

In [3]:
```
print(fr1)
```

```
3/4
```

In [4]:
```
print(fr2)
```

```
1/2
```

In [5]:
```python
class Fraction:

  # parameterized constructor
  def __init__(self,x,y):
    self.num = x
    self.den = y

  def __str__(self):
    return '{}/{}'.format(self.num,self.den)

  def __add__(self,other):
    new_num = self.num*other.den + other.num*self.den
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __sub__(self,other):
    new_num = self.num*other.den - other.num*self.den
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __mul__(self,other):
    new_num = self.num*other.num
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __truediv__(self,other):
    new_num = self.num*other.den
    new_den = self.den*other.num

    return '{}/{}'.format(new_num,new_den)

  def convert_to_decimal(self):
    return self.num/self.den
```

In [6]:
```python
fr1 = Fraction(3,4)
fr2 = Fraction(1,2)
```

In [7]:
```python
fr1.convert_to_decimal()
# 3/4
```

Out[7]: 0.75

In [8]:
```python
print(fr1 + fr2)
print(fr1 - fr2)
print(fr1 * fr2)
print(fr1 / fr2)
```

```
10/8
2/8
3/8
6/4
```

Write OOP classes to handle the following scenarios:

A user can create and view 2D coordinates

A user can find out the distance between 2 coordinates

A user can find find the distance of a coordinate from origin

A user can check if a point lies on a given line

A user can find the distance between a given 2D point and a given line

In [9]:
```
class Point:

  def __init__(self,x,y):
    self.x_cod = x
    self.y_cod = y

  def __str__(self):
    return '<{},{}>'.format(self.x_cod,self.y_cod)

  def euclidean_distance(self,other):
    return ((self.x_cod - other.x_cod)**2 + (self.y_cod - other.y_cod)**2)**0.5

  def distance_from_origin(self):
    return (self.x_cod**2 + self.y_cod**2)**0.5
    # return self.euclidean_distance(Point(0,0))


class Line:

  def __init__(self,A,B,C):
    self.A = A
    self.B = B
    self.C = C

  def __str__(self):
    return '{}x + {}y + {} = 0'.format(self.A,self.B,self.C)

  def point_on_line(line,point):#here line=self
    if line.A*point.x_cod + line.B*point.y_cod + line.C == 0:
      return "lies on the line"
    else:
      return "does not lie on the line"

  def shortest_distance(line,point):
    return abs(line.A*point.x_cod + line.B*point.y_cod + line.C)/(line.A**2 + li
```

In [10]:
```
l1 = Line(1,1,-2)
p1 = Point(1,10)
p2 = Point(10,10)
print(l1)
print(p1)
print(l1.point_on_line(Point(1,1)))
print(p1.euclidean_distance(p2))
print(l1.shortest_distance(p1))
```

```
1x + 1y + -2 = 0
<1,10>
lies on the line
9.0
6.363961030678928
```

Polymorphism

- Polymorphism is a word that came from two greek words, poly means many and morphos means forms.
- If a variable, object or method perform different behavior according to situation, it is called polymorphism.

- Duck Typing
- Method Overriding
- Method Overloading
- Operator Overloading

Duck Typing

- In Python, we follow a principle - If 'it walks like a duck and talks like a duck, it must be a duck' which means python doesn't care about which class of object it is, if it is an object and required behavior is present for that object then it will work. The type of object is distinguished only at runtime. This is called as duck typing.
- Python doesn't care about which class of object it is, in order to call an existing method on an object. If the method is defined on the object, then it will be called!

walk - thapak thapak

walk – tabdak tabdak

In [20]:
```
x=2
type(x)
```

Out[20]:
```
int
```

In [28]:

```
# Duck Typing
class Duck:
    def walk(self):
        print("thapak thapak thapak thapak")

class Horse:
    def walk(self):
        print("tabdak tabdak tabdak tabdak")

class Cat:
    def talk(self):
        print("Meow Meow")
def myfunction(obj):
    obj.walk()

d = Duck()
myfunction(d)

h = Horse()
myfunction(h)

c = Cat()
myfunction(c)
```

```
thapak thapak thapak thapak
tabdak tabdak tabdak tabdak

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-28-2a393904cc22> in <module>
     21
     22 c = Cat()
---> 23 myfunction(c)

<ipython-input-27-8bae1ca0732c> in myfunction(obj)
     12         print("Meow Meow")
     13 def myfunction(obj):
---> 14     obj.walk()
     15
     16 d = Duck()

AttributeError: 'Cat' object has no attribute 'walk'
```

Strong Typing

- We can check whether the object passed to the method has the method being invoked or not.
- hasattr ( ) Function is used to check whether the object has a method or not.
- Syntax:- hasattr(object, attribute)
- Where attribute can be a method or variable. If it is found in the object then this method returns True else False.

In [29]:
```
class Duck:
    def walk(self):
        print("thapak thapak thapak thapak")

class Horse:
    def walk(self):
        print("tabdak tabdak tabdak tabdak")

class Cat:
    def talk(self):
        print("Meow Meow")

def myfunction(obj):
    if hasattr(obj, 'walk'):
        obj.walk()
    if hasattr(obj, 'talk'):
        obj.talk()

d = Duck()
myfunction(d)

h = Horse()
myfunction(h)

c = Cat()
myfunction(c)
```

```
thapak thapak thapak thapak
tabdak tabdak tabdak tabdak
Meow Meow
```

In [23]:
```
hasattr(s,"area")
```

Out[23]:
```
True
```

In [24]:
```
hasattr(s,"Shape")
```

Out[24]:
```
False
```

Method Overriding

- If we write method in the both classes, parent class and child class then the parent class's method is not available to the child class.
- In this case only child class's method is accessible which means child class's method is replacing parent class's method. Method overriding is used when programmer want to modify the existing behavior of a Method.

In [32]:
```
class Add:
    def result(self, a, b):
        print('Addition:', a+b)

class Multi(Add):
    def result(self, a, b):
        print('Multiplication:', a*b)

m = Multi()
m.result(10, 20)

m = Add()
m.result(10, 20)
```

```
Multiplication: 200
Addition: 30
```

In [33]:
```
# Method Overriding
class Add:
    def result(self, a, b):
        print('Addition:', a+b)

class Multi(Add):
    def result(self, a, b):
        super().result(10, 20) # Calling Parent Class's Method
        print('Multiplication:', a*b)

m = Multi()
m.result(10, 20)
```

```
Addition: 30
Multiplication: 200
```


Method Overloading

- When more than one method with the same name is defined in the same class, it is known as method overloading. In python, If a method is written such that it can perform more than one task, it is called method overloading.

In [1]:
```
class Shape:

  def area(self,a):
    return 3.14*a*a
  def area(self,a,b):
    return a*b

s = Shape()

print(s.area(2))
print(s.area(3,4))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-f3401eb34f3c> in <module>
      8 s = Shape()
      9
---> 10 print(s.area(2))
     11 print(s.area(3,4))

TypeError: area() missing 1 required positional argument: 'b'
```

In [2]:
```
class Shape:

  def area(self,a,b=0):
    if b == 0:
      return 3.14*a*a
    else:
      return a*b

s = Shape()

print(s.area(2))
print(s.area(3,4))
```

```
12.56
12
```

Operator Overloading

- If any operator performs additional actions other than what it is meant for, it is called operator overloading.
- For Operator Overloading, we will be using only Mathematical (add, sub, mul, truediv, floordiv, mod, pow), Comparison (lt, gt, le, ge, eq, ne)

Operator	Operator Name	Magic Method	Expression	Internal Calls
<	Less than	__lt__(SELF, OTHER)	C1<C2	C1.__lt__(C2)
>	Greater than	__gt__(SELF, OTHER)	C1>C2	C1.__gt__(C2)
<=	Less than or equal to	__le__(SELF, OTHER)	C1<=C2	C1.__le__(C2)
>=	Greater than or equal to	__ge__(SELF, OTHER)	C1>=C2	C1.__ge__(C2)
==	Equal to	__eq__(SELF, OTHER)	C1==C2	C1.__eq__(C2)
!=	Not equal to	__ne__(SELF, OTHER)	C1!=C2	C1.__ne__(C2)

Operator	Operator Name	Magic Method	Expression	Internal Calls
+	Addition	__add__(self, other)	C1-C2	C1.__sub__(C2)
-	Subtraction	__sub__(self, other)	C1+C2	C1.__add__(C2)

In [5]:
```
1+2
```

Out[5]: 3

In [7]:
```
"hi"+"hello"
```

Out[7]: 'hihello'

In [8]:
```
[1,2]+[3,4]
```

Out[8]: [1, 2, 3, 4]

In [34]:
```
# Operator Overloading
class A:
    def __init__(self, x):
        self.x = x
    def __add__(self, other):
        return self.x + other.x
class B:
    def __init__(self, x):
        self.x = x
a = A(100)
b = B(200)
print(a+b)
```

```
300
```

In [9]:
```
class Fraction:

  # parameterized constructor
  def __init__(self,x,y):
    self.num = x
    self.den = y

  def __str__(self):
    return '{}/{}'.format(self.num,self.den)

  def __add__(self,other):
    new_num = self.num*other.den + other.num*self.den
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __sub__(self,other):
    new_num = self.num*other.den - other.num*self.den
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __mul__(self,other):
    new_num = self.num*other.num
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __truediv__(self,other):
    new_num = self.num*other.den
    new_den = self.den*other.num

    return '{}/{}'.format(new_num,new_den)

  def convert_to_decimal(self):
    return self.num/self.den
```

In [10]:
```
fr1 = Fraction(3,4)
fr2 = Fraction(1,2)
```

In [11]:
```
fr1.convert_to_decimal()
# 3/4
```

Out[11]:
```
0.75
```

In [12]:
```
print(fr1 + fr2)
print(fr1 - fr2)
print(fr1 * fr2)
print(fr1 / fr2)
```

```
10/8
2/8
3/8
6/4
```

In [14]:
```
1 print(fr1 % fr2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-46f44ec7b0f2> in <module>
----> 1 print(fr1 % fr2)

TypeError: unsupported operand type(s) for %: 'Fraction' and 'Fraction'
```


In [15]:
```
1 print(fr1 > fr2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-d4a4ac298f3e> in <module>
----> 1 print(fr1 > fr2)

TypeError: '>' not supported between instances of 'Fraction' and 'Fraction'
```

Abstraction

- A class derived from ABC class which belongs to abc module, is known as abstract class in Python.
- ABC Class is known as Meta Class which means a class that defines the behavior of other classes. So we can say, Meta Class ABC defines that the class which is derived from it becomes an abstract class.
- Abstract Class can have abstract method and concrete methods.
- Abstract Class needs to be extended and its method implemented.
- not create objects of an abstract class!

from abc import ABC, abstractmethod

Class Father(ABC):

Abstract Method

- A abstract method is a method whose action is redefined in the child classes as per the requirement of the object.
- We can declare a method as abstract method by using @abstractmethod decorator.

from abc import ABC, abstractmethod

Class Father(ABC):

```
    @abstractmethod

     def disp(self):

            pass
```

- A Concrete method is a method whose action is defined in the abstract class itself.

```
from abc import ABC, abstractmethod
Class Father(ABC):
    @abstractmethod
    def disp(self):
        pass
    def show(self):
```

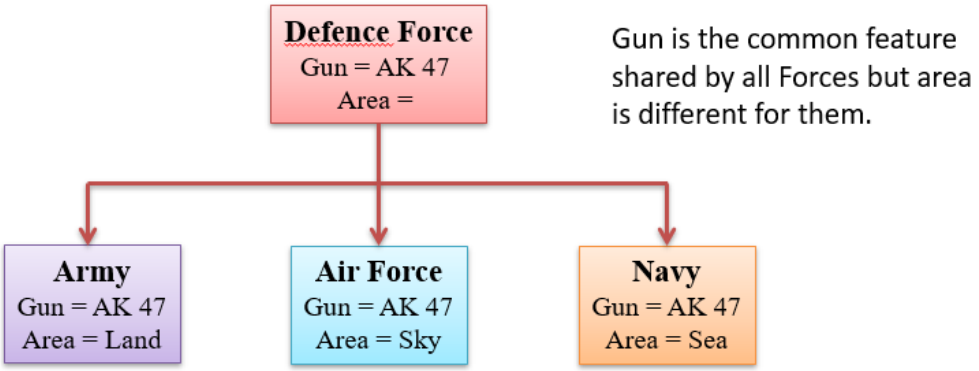
Abstract Method / Method Without Body

Rules

- can not create objects of an abstract class.
- It is not necessary to declare all methods abstract in a abstract class.
- Abstract Class can have abstract method and concrete methods.
- If there is any abstract method in a class, that class must be abstract.
- The abstract methods of an abstract class must be defined in its child class/subclass.
- If you are inheriting any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

When use Abstract Class

- We use abstract class when there are some common feature shared by all the objects as they are.



Why Abstraction is Important?

- In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency.

In [36]:

```
from abc import ABC, abstractmethod
class Father(ABC):

    @abstractmethod
    def disp(self): # Abstract Method
        pass

    def show(self): # Concrete Method
        print('Concrete Method')

#my = Father() # Not possible to create object of a abstract class

class Child(Father):
    def disp(self):
        print("Defining Abstract Method")


c = Child()
c.disp()
c.show()
```

```
Defining Abstract Method
Concrete Method
```

In [37]:

```
my = Father()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-25b17222ec92> in <module>
----> 1 my = Father()

TypeError: Can't instantiate abstract class Father with abstract methods disp
```

In [40]:

```
from abc import ABC, abstractmethod

class DefenceForce (ABC):
    def __init__(self):
        self.id = 101

    @abstractmethod
    def area(self):
        pass

    def gun(self):
        print("Gun = AK47")

class Army(DefenceForce):
    def area(self):
        print("Army Area = Land", self.id)

class AirForce(DefenceForce):
    def area(self):
        print("AirForce Area = Sky", self.id)

class Navy(DefenceForce):
    def area(self):
        print("Navy Area = Sea", self.id)

a = Army()
af = AirForce()
n = Navy()

a.gun()
a.area()
print()
af.gun()
af.area()
print()
n.gun()
n.area()
```

```
Gun = AK47
Army Area = Land 101

Gun = AK47
AirForce Area = Sky 101

Gun = AK47
Navy Area = Sea 101
```

In [16]:

```
from abc import ABC,abstractmethod
class BankApp(ABC):

  def database(self):
    print('connected to database')

  @abstractmethod
  def security(self):
    pass

  @abstractmethod
  def display(self):
    pass
```

In [17]:
```
class MobileApp(BankApp):

  def mobile_login(self):
    print('login into mobile')

  def security(self):
    print('mobile security')

  def display(self):
    print('display')
```

In [18]:
```
mob = MobileApp()
```

In [19]:
```
mob.security()
```

```
mobile security
```

In [35]:
```
v=BankApp()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-cfc3ce3b7340> in <module>
----> 1 v=BankApp()

TypeError: Can't instantiate abstract class BankApp with abstract methods displ
ay, security
```

In [41]:

```
from abc import ABC, abstractmethod
class Car(ABC):
    def mileage(self):
        pass

class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
     def mileage(self):
          print("The mileage is 24kmph ")

class Renault(Car):
    def mileage(self):
            print("The mileage is 27kmph ")

# Driver code
t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

```
The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph
```

In [43]:

```
from abc import ABC, abstractmethod
class Bank(ABC):
    def branch(self, RD):
        print("Fees submitted : ",RD)
        @staticmethod
        @abstractmethod
        def Bank(RD):
            pass
class private(Bank):
    @staticmethod
    def Bank(RD):
        print("Total RD Value here: ",RD)
class XXX(Bank):
    @staticmethod
    def Bank(RD):
        print("Total RD Value here:",RD)
private.Bank(500)
XXX.Bank(200)
```

```
Total RD Value here:  500
Total RD Value here: 200
```

In [44]:
```
from abc import ABC, abstractmethod
class ljclass(ABC):
    def print(self,a):
        print("The value is: ", a)
        @abstractmethod
        def course(self):
            print("This is educlass")
class learn(ljclass):
    def course(self):
        print("This is test class")
class demo_class(ljclass):
    def course(self):
       print("This is demo class")
t1 = learn()
t1.course()
t1.print(500)
ex = demo_class()
ex.course()
ex.print(850)
print("t1 is instance of educlass? ", isinstance(t1, ljclass))
print("ex is instance of educlass? ", isinstance(ex, ljclass))
```

```
This is test class
The value is:  500
This is demo class
The value is:  850
t1 is instance of educlass?  True
ex is instance of educlass?  True
```

In [1]:
```
class Father: # Parent Class
    def __init__(self, m):
        self.money = m
        print("Father Class Constructor")
    def show(self):
        print("Father Class Instance Method:", self.money)
class Son(Father): # Child Class
    def __init__(self, j, m):
        super().money       # Calling Parent Class Constructor
        self.job = j
        print("Son Class Constructor")
    def disp(self):
        print("Son Class Instance Method", self.job)
```

In [2]:
```
s=Son(4,5)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-66aaf414f825> in <module>
----> 1 s=Son(4,5)

<ipython-input-1-7342395e8a47> in __init__(self, j, m)
      7 class Son(Father): # Child Class
      8     def __init__(self, j, m):
----> 9         super().money           # Calling Parent Class Constructor
     10         self.job = j
     11         print("Son Class Constructor")

AttributeError: 'super' object has no attribute 'money'
```

In [ ]:
```
1
```