UNIT-IV

INHERITANCE

INHERITANCE IN PYTHON

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**

The benefits of inheritance are:

1. In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.
2. It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.

Syntax:

```
class base-class:
     statement-1
     statement-2
          ...

class derived-class(base-class):
     statement-1
     statement-2
          ...
```

Example:

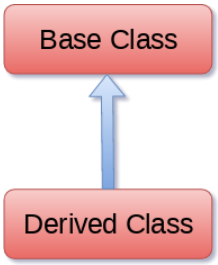
```
# Base or Super class
class Base:
    def baseMethod(self):
        print("This is a Base class method")

# Inherited or Sub class
class Derived(Base):
    def derivedMthod(self):
        print("This is a Derived class method")

obj=Base()
obj.baseMethod()
#obj.derivedMethod()     # Error

obj1=Derived()
obj1.baseMethod()
obj1.derivedMthod()
```

Output:

```
This is a Base class method
This is a Base class method
This is a Derived class method
```

TYPES OF INHERITANCE

1. **Single inheritance**: When a child class inherits from only one parent class, it is called as single inheritance.



Syntax:

```
class base-class:
      statement-1
      statement-2
           ...

class derived-class(base-class):
     statement-1
     statement-2
```

Example : Above Example

Example :

```
# Base/Super class
class Father:
    Surname="Banala"
    def fun1(self):
        print("This is the base class method")
        print("Surname:",self.Surname)

# Derived/Inherited/Sub class
class Son(Father):
    Name=""
    def fun2(self):
        print("This is the derived class method")
        print("Name:",self.Name)

obj1=Son()
obj1.Name="Eswar"
obj1.fun1()
obj1.fun2()
```

Output:

```
This is the base class method
Surname: Banala
This is the derived class method
Name: Eswar
```

Example:

```
class Person:
    Name=""
    Location=""
    def __init__(self,n,l):      #Base Class Constructor
        self.Name=n
        self.Location=l
    def getData(self):
        print("The details of the person are:")
        print("Name=",self.Name)
        print("Location=",self.Location)

class Student(Person):
    CGPA=9.4
    def getResult(self):
        print("The CGPA of the Student is ",self.CGPA)
        print("Thank You..")

class Employee(Person):
    Sal=1000
    def getSal(self):
        print("The Salary of an employee is ",self.Sal)
        print("Thank You..")

obj1=Student("Ram","Hyd")
obj1.getData()
obj1.getResult()

obj2=Employee("Eswar","Bangalore")
obj2.getData()
obj2.getSal()
```

Output:

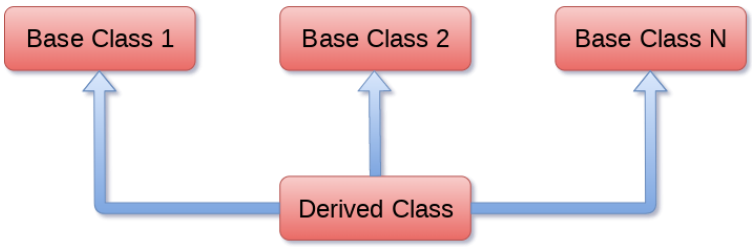
```
The details of the person are:
Name= Ram
Location= Hyd
The CGPA of the Student is  9.4
Thank You..
The details of the person are:
Name= Eswar
Location= Bangalore
The Salary of an employee is  1000
Thank You..
```

2. **Multiple inheritance:** When a child class inherits from multiple parent classes, it is called as multiple inheritance.

or

When a derived class inherits features from more than one base class, it is called multiple inheritance.

The derived class has all the features of both the base classes and in addition to them can have additional new features.



Syntax:

```
class Base-class1:
      statement-1
      statement-2
           ...

class Base-class2:
      statement-1
      statement-2
           ...

class derived-class(Base-class1, Baseclass2):
     statement-1
     statement-2
```

Example

```
class Base1:
    def base1Method(self):
        print("This is a base1Method of Baseclass-1")
class Base2:
    def base2Method(self):
        print("This is a base2Method of Baseclass-2")

class Derived(Base1,Base2):
     def derivedMethod(self):
        print("This is a derivedMethod of Derived class")

obj1=Derived()
obj1.base1Method()
obj1.base2Method()
obj1.derivedMethod()
```

Output:

```
This is a base1Method of Baseclass-1
This is a base2Method of Baseclass-2
This is a derivedMethod of Derived class
```

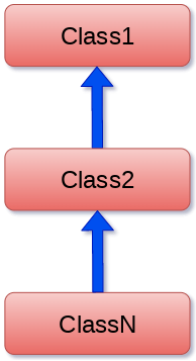
Example:

```
class Area:
    def getArea(length,breadth):
        a=length*breadth
        return(a)
class Perimeter:
    def getPerimeter(length,breadth):
        p=2*(length+breadth)
        return(p)
class Rectangle(Area,Perimeter):
    def __init__(self,l,b):
        self.length=l
        self.breadth=b
    def area(self):
        area= Area.getArea(self.length,self.breadth)
        print("Area of Rectangle= ",area)
    def perimeter(self):
        perimeter=Perimeter.getPerimeter(self.length,self.breadth)
        print("Perimeter of Rectangle= ",perimeter)

r1=Rectangle(7,4)
r1.area()
r1.perimeter()
```

Output:

```
Area of Rectangle=  28
Perimeter of Rectangle=  22
```

3. **Multi-Level Inheritance:** The technique of deriving a class from an already derived class is called multi - level inheritance.



Syntax:

```
class Base:
      statement-1
      statement-2
            ...
class Child(Base):
      statement-1
      statement-2
            ...
class GrandChild(Child):
      statement-1
      statement-2
            ...
```

Example:

```
class Base:
    def method1(self):
        print("method1");
class Child(Base):
    def method2(self):
        print("method2")
class GrandChild(Child):
    def method3(self):
        print("method3")


obj1=GrandChild()
obj1.method1()
obj1.method2()
obj1.method3()

obj2=Child()
obj2.method1()
obj2.method2()
#obj2.method3()       #Error

obj3=Base()
```

```
obj3.method1()
#obj3.method2()     #Error
#obj3.method3()     #Error
```

Output:

```
method1
method2
method3
method1
method2
method1
```

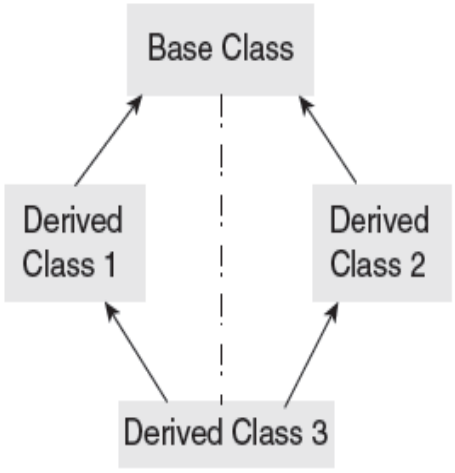
Example:

```
class Car:
    def vehicleType(self):
        print("VehicleType : car")
class Suziki(Car):
    def brand(self):
        print("Brand : Suziki")
    def speed(self):
        print("Max Speed : 120 KMPH")
class SwiftDezire(Suziki):
    def speed(self):
        print("Max Speed : 200 KMPH")

obj1=SwiftDezire()
obj1.vehicleType()
obj1.brand()
obj1.speed()

obj2=Suziki()
obj2.vehicleType()
obj2.brand()
obj2.speed()
```

Output:

```
VehicleType : car
Brand : Suziki
Max Speed : 200 KMPH
VehicleType : car
Brand : Suziki
Max Speed : 120 KMPH
```

4. **Multi-Path Inheritance:** Deriving a class from other derived classes that are in turn derived from the same base class is called multi-path inheritance.



Example:

```
class A:
    def method1(self):
        print("Base Class A")
class B(A):
    def method2(self):
        print("Derived Class-1 of Class A")
class C(A):
    def method3(self):
        print("Derived Class-2 of Class A")
class D(B,C):
    def method4(self):
        print("Derived Class of Class B & Class C")

d=D()
d.method1()
d.method2()
d.method3()
d.method4()

b=B()
b.method1()
b.method2()

c=C()
c.method1()
c.method3()

a=A()
a.method1()
```

Output:

```
Base Class A
Derived Class-1 of Class A
Derived Class-2 of Class A
Derived Class of Class B & Class C
Base Class A
Derived Class-1 of Class A
Base Class A
Derived Class-2 of Class A
Base Class A
```

Example:

```
class Student:
    def __init__(self,n):
        self.name=n
    def getName(self):
        print("Name : ",self.name)

class Branch(Student):
    def getBranch(self, dept):
        print("Branch : ", dept)

class Score(Student):
    def getScore(self,avg):
        print("Academic Score : ", avg)

class Result(Branch,Score):
    def getResult(self):
        print('Result'.center(20,'*'))
        self.getName()
        self.getBranch('IT')
        self.getScore(65)

r=Result("Eeswar")
r.getResult()
```

Output:

```
*******Result*******
Name :  Eeswar
Branch :  IT
Academic Score :  65
```

OVERRIDING METHODS (POLYMORPHISM)

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

```
College =  VJIT  JBIT
Add(a,b)
Add(a,b,c) – overridden method
```

Note: Private methods will not be overridden.

Example:

```
class A:
    def display(self):
        print("This class is used for addition operation")
        print("Class A object can add two variables")
        print("Class B object can add three variables")
    def add(self,a,b):
        print("result of a+b =",a+b)

class B(A):
    def add(self,a,b,c):
        print("result of a+b+c =",a+b+c)

b=B()           # instance of child
b.display()
b.add(10,20,30) # child calls overridden method
a=A()
a.add(10,20)
```

Output:

```
This class is used for addition operation
Class A object can add two variables
Class B object can add three variables
result of a+b+c = 60
result of a+b = 30
```

Example2:

```
class Parent:
   def myMethod(self):
      print('Calling parent method')

class Child(Parent):
   def myMethod(self):
      print('Calling child method')

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

Example3:

```
class Bank:
    def getroi(self):
        print(“This is overriding method”);
class SBI(Bank):
    def getroi(self):
        print(“This is overriding  getroi() method”);
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi())
print("SBI Rate of interest:",b2.getroi())
print("ICICI Rate of interest:",b3.getroi())
```

Output:

Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8

COMPOSITION / CONTAINERSHIP / COMPLEX OBJECTS

- Complex objects are objects that are built from smaller or simpler objects.
- Process of building complex objects from small objects is called composition or containership.
- In object-oriented programming languages, object composition is used for objects that have a **has-a** relationship to each other.

Example1:

```
class A:
    def fun1(self):
        print("fun1 method")

class B:
    def fun3(self):
        a=A()
        a.fun1()
    def fun2(self):
        print("fun2 method")
# class B has-a class A

b=B()
b.fun2()
b.fun3()
```

Output:

```
fun2 method
fun1 method
```

Example2:

```
class A:
    def set(self,x):
        self.a=x
        print("assigned a value for an attribute 'a' of class A")
    def get(self):
        return self.a

class B:
    def __init__(self,y):
        self.var=A()     #Object of Class A
        self.var.set(y)  #method of class A is invoked using its object
    def show(self):
        print("value of attaribute 'a' = ", self.var.get())

b=B(20)
b.show()
```

Output:

```
assigned a value for an attribute 'a' of class A
value of attaribute 'a' =  20
```

Example3:

```
class College:
    def getBranch(self):
        branches=['ECE','CSE','IT']
        print(branches)

class University:
    def getCol(self):
        ColList=['JBIT', 'Narayanamma','Mallareddy']
        print(ColList)
    def __init__(self):
        obj1=College()
        obj1.getBranch()

obj2=University()
obj2.getCol()
```

Output:

```
['ECE', 'CSE', 'IT']
['JBIT', 'Narayanamma', 'Mallareddy']
```

Inheritance	Containership
• Enables a class to inherit data and functions from a base class by extending it.	• Enables a class to contain objects of different classes as its data member.
• The derived class may override the functionality of base class.	• The container class cannot alter or override the functionality of the contained class.
• The derived class may add data or functions to the base class.	• The container class cannot add anything to the contained class.
• Inheritance represents a "is-a" relationship.	• Containership represents a "has-a" relationship.
• Example: A Student is a Person.	• Example: class One has a class Two.

ABSTRACT CLASSES

A class which cannot be instantiated is known as an **Abstract class**. This means that, we cannot create an object of abstract class.

If a class contains any abstract method, then that class is called as an Abstract Class.
Abstract Method – a method without definition/implementation

- Abstract classes could only be inherited and then an object of the derived class was used to access the features of the base class (abstract class).
- Abstract class is an incomplete class; users are not allowed to create its object. To use such a class, we must derive it and override the features specified in that class.
- An abstract class just serves as a template for other classes by defining a list of methods that the classes must implement. In Python, we use the **NotImplementedError** to restrict the instantiation of a class. Any class that has the NotImplementedError inside method definitions cannot be instantiated.

Example:

```
class Base:     #Abstract Class
    def fun(self):
        raise NotImplementedError()
class Derived(Base):
    def fun(self):
        print("This is an Example of Abstract class")
        print("fun() can defined here")

obj=Derived()
obj.fun()
```

Output:

```
This is an Example of Abstract class
fun() can defined here
```

Note: An abstract class can contain constructors in Python. And a constructor of abstract class is invoked when an instance of a inherited class is created.

Example1:

```
class Base:     #Abstract Class
    def __init__(self):
        print("This is a constructor of abstract class.")
    def fun(self):
        raise NotImplementedError()
class Derived(Base):
    def fun(self):
        print("This is an Example of Abstract class")
        print("fun() can defined here")

print("Abstract class Demo")
obj=Derived()
obj.fun()
```

Output:

```
Abstract class Demo
This is a constructor of abstract class.
This is an Example of Abstract class
fun() can defined here
```

Example2:

```
class A:
    def addition(self,a,b):
        raise NotImplementedError()

    def subtraction(self,a,b):
        raise NotImplementedError()
# addition() & subtraction() methods are abstract methods
# therfore Class A is called an Abstract class

class B(A):
    def addition(self,a,b):
        sum1=a+b
        return(sum1)
    def subtraction(self,a,b):
        diff=a-b
        return diff

b=B()
sum1= b.addition(1,2)
print("sum=", sum1)
diff= b.subtraction(1,2)
print("difference=",diff)
```

Example3:

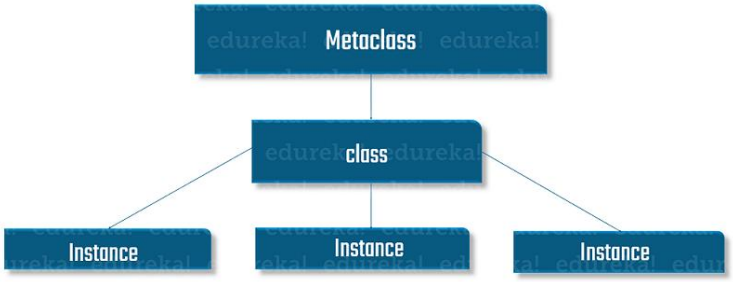
```
import math
class Shape:          #Abstract Class
    def calculateArea(self):
        raise NotImplementedError()

class Circle(Shape):
    def __init__(self,r):
        self.radius=r
    def calculateArea(self):
        area = math.pi * math.pow(self.radius,2)
        print("Area of circle = ",area)

class Traingle(Shape):
    def __init__(self,h,b):
        self.height=h
        self.base=b
    def calculateArea(self):
        area = (self.height * self.base)/2
        print("Area of Traingle = ",area)

class Rectangle(Shape):
    def __init__(self,l,w):
        self.lenghth=l
        self.width=w
    def calculateArea(self):
        area = self.lenghth * self.width
        print("Area of Rectangle = ",area)

c1=Circle(2)
t1=Traingle(5,3)
r1=Rectangle(2,4)

c1.calculateArea()
t1.calculateArea()
r1.calculateArea()
```

Output:

```
Area of circle =  12.566370614359172
Area of Traingle =  7.5
Area of Rectangle =  8
```

META CLASS

Python Metaclass is one of the advanced features associated with Object-Oriented Programming concepts of Python. It determines the behavior of a class and further helps in its modification.

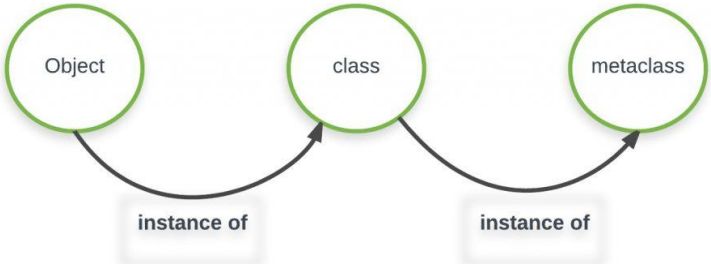


Every class created in Python has an underlying Metaclass. So when you're creating a class, you are indirectly using the Metaclass. It happens implicitly, you don't need to specify anything.

Metaclass when associated with meta programming determines the ability of a program to manipulate itself.

Note: In Python, a Class is also an object.

Note: A metaclass is the class of a class. While a class defines how an instance of the class behaves, a metaclass, on the other hand, defines how a class behaves. Every class that we create in Python, is an instance of a metaclass.



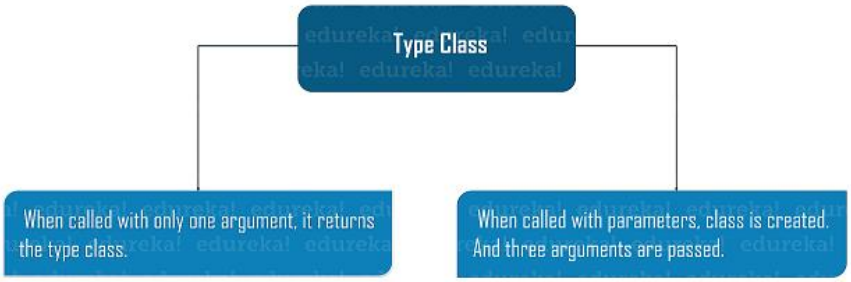
Creating a Metaclass

Whenever a class is created the default Metaclass which is type class gets called instead. Metaclass carries information like name, set of a base class, and attributes associated with the class. Hence, when a class is instantiated type class is called carrying these arguments. Metaclass can be created in two methods:

1. Type class
2. Custom Metaclass

1. Creating metaclass using type()

We can create meta classes using type() function directly. The function type() can be called in following ways,



i) When called with only one argument, it returns the type. We have seen it before in above examples like i=10; type(i), returns int.

ii) When called with three parameters, it creates a class. Following arguments are passed to it as

a) Class name

b) Tuple having base classes inherited by class

c) Class Dictionary: It serves as local namespace for the class, populated with class methods and variables

When we pass parameters through type class, it uses the following syntax.

Syntax:

```
type(__name__, __base__, attributes)
(or)
Class_Name=type('Class_Name',(base classes),{attributes/methods dict})
```

where,

- the name is a string and carries the class name
- the base is a tuple and helps create subclasses
- an attribute is a dictionary and assigns key-value pairs

Example:

```
Test=type('Test',(),{'x':5, 'func':func})
```

Example: Demonstrating attributes/methods using type()

```
-----------------------------------------------
class Test:
    x=5
    def func(self):
        print("This is func() definition")

print(Test)          #<class '__main__.Test'>
print(type(Test))    #<class 'type'>
-----------------------------------------------
```

(or, this is same as)

```
-----------------------------------------------
def func(self):
    print("This is func() definition")

Test=type('Test',(),{'x':5, 'func':func})
print(Test)          #<class '__main__.Test'>
print(type(Test))    #<class 'type'>
-----------------------------------------------
t=Test()
print(t.x)           #5
t.func()             #This is func() definition
```

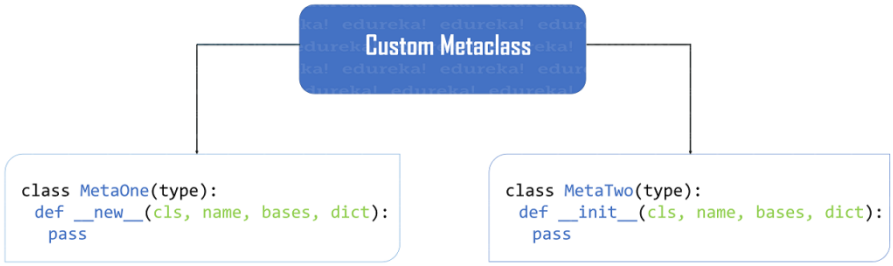
Example: Demonstrating Base-classes/inheritance using type()

```
class Base1:
    def show(self):
        print("Hello")


Test=type('Test',(Base1,),{'x':5})
#same as; class Test(Base1): x=5
t=Test()
print(t.x)      #5
t.show()        #Hello
```

2. Creating our custom Metaclass

Pass metaclass as a keyword while creating a class definition. Alternatively, we can achieve this by simply inheriting a class that has been instantiated through this Metaclass keyword.

There are two methods that we can use to create custom Metaclass.



1. **__new__():** is used when a user wants to define a dictionary of tuples before the class creation. It returns an instance of a class and is easy to override/manage the flow of objects.
2. **__init__()**: It's called after the object has already been created and simply initializes it.

Example1: __new__()

```
class Meta(type):
     def __new__(self,class_name,bases,attrs):
        print(attrs)
        #return(class_name,bases,attrs)

class Test(metaclass=Meta):
    x=5
    y=5

#observation: run the code with out creating an object, we can observe that the attributes are displayed
#metaclass can be specified by 'metaclass' keyword argument
```

Output:

```
{'__module__': '__main__', '__qualname__': 'Test', 'x': 5, 'y': 5}
```

Example2: __init__()

```
class Meta(type):
    def __init__(self, name, base, dct):
        self.attribute = 200
class Test(metaclass = Meta):
    pass

t=Test()
print(t.attribute)    #200
```

Note: Whenever a class is instantiated __new__ and __init__ methods are called. __new__ method will be called when an object is created and __init__ method will be called to initialize the object.

Operator Overloading in Python

Python allows programmers to redefine the meaning of operators when they operate on class objects. This feature is called operator overloading. Operator overloading allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can be also applied to user defined data types.

In Python few operators behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.

This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

Example:

```
>>>10+20
>>>30
>>>'Eswar'+'Banala'
>>>'EswarBanala'
```

Here + operator has two interpretations. When used with numbers it is interpreted as an addition operator whereas with strings it is interpreted as the concatenation operator. In other words, we can say that the + operator is overloaded for int class and str class.

Operator Overloading is achieved by defining a special method in the class definition. The names of these methods start and end with double underscores (__). The special method used to overload + operator is called __add__(). Both int class and str class implements __add__() method.

Example:

```
>>>x,y=10,20
>>>x+y
>>>30
>>>x.__add__(y)  # similar to x+y
>>>30
```

Example:

```
class A:
    def __init__(self,a):
        self.a=a
    # adding two objects
    def __add__(self,o):
        return self.a + o.a

    def __mul__(self,o):
        return self.a * o.a

a=A(10)
b=A(20)
print(a+b)            #30
print(a*b)            #200
```

Note: In the above program addition of objects concept was extended for + operator.

Operators and Their Corresponding Function Names

Operator	Special Method	Description
+	__add__(self, object)	Addition
-	__sub__(self, object)	Subtraction
*	__mul__(self, object)	Multiplication
**	__pow__(self, object)	Exponentiation
/	__truediv__(self, object)	Division
//	__floordiv__(self, object)	Integer Division

Operator	Function Name	Operator	Function Name
+	__add__	+=	__iadd__
-	__sub__	-=	__isub__
*	__mul__	*=	__imul__
/	__truediv__	/=	__idiv__
**	__pow__	**=	__ipow__
%	__mod__	%=	__imod__
>>	__rshift__	>>=	__irshift__
&	__and__	&=	__iand__
|	__or__	|=	__ior__
^	__xor__	^=	__ixor__
~	__invert__	~=	__iinvert__
<<	__lshift__	<<=	__ilshift__
>	__gt__	<=	__le__
<	__lt__	==	__eq__
>=	__ge__	!=	__ne__