Python Programming - VIII

By Nimesh Kumar Dagur

Python Object Oriented Programming

- Python has been an object-oriented language since it existed.
- Because of this, creating and using classes and objects are downright easy.

Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
- The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

- **Class variable:** A variable that is shared by all instances of a class.
- Class variables are defined within a class but outside any of the class's methods.
- Class variables are not used as frequently as instance variables are.

- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.

- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.

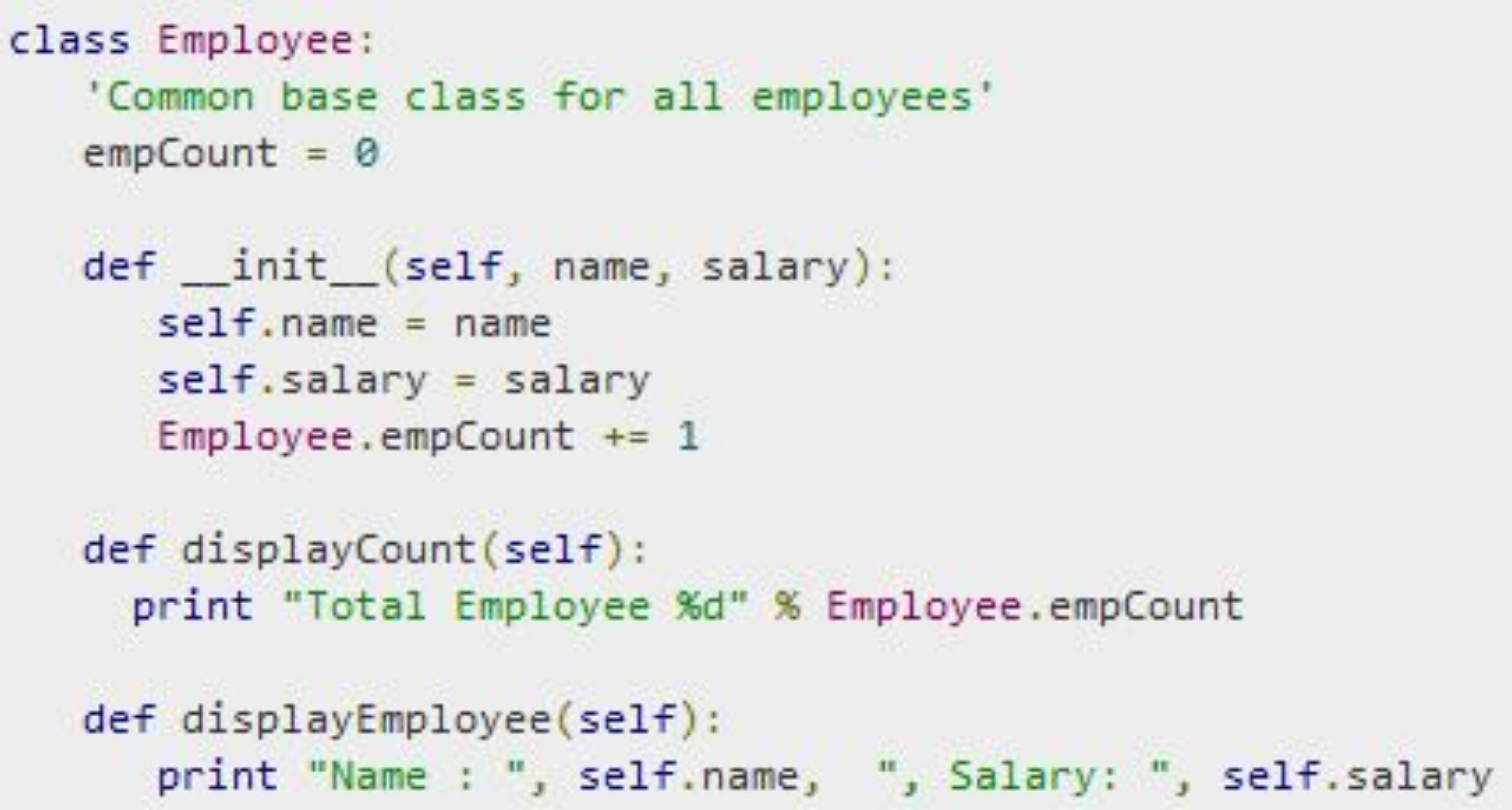
Overview of OOP Terminology

- **Method :** A special kind of function that is defined in a class definition.
- **Object:** A unique instance of a data structure that's defined by its class.
- An object comprises both data members (class variables and instance variables) and methods.
- **Instance:** An individual object of a certain class.
- An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation:** The creation of an instance of a class.

Overview of OOP Terminology

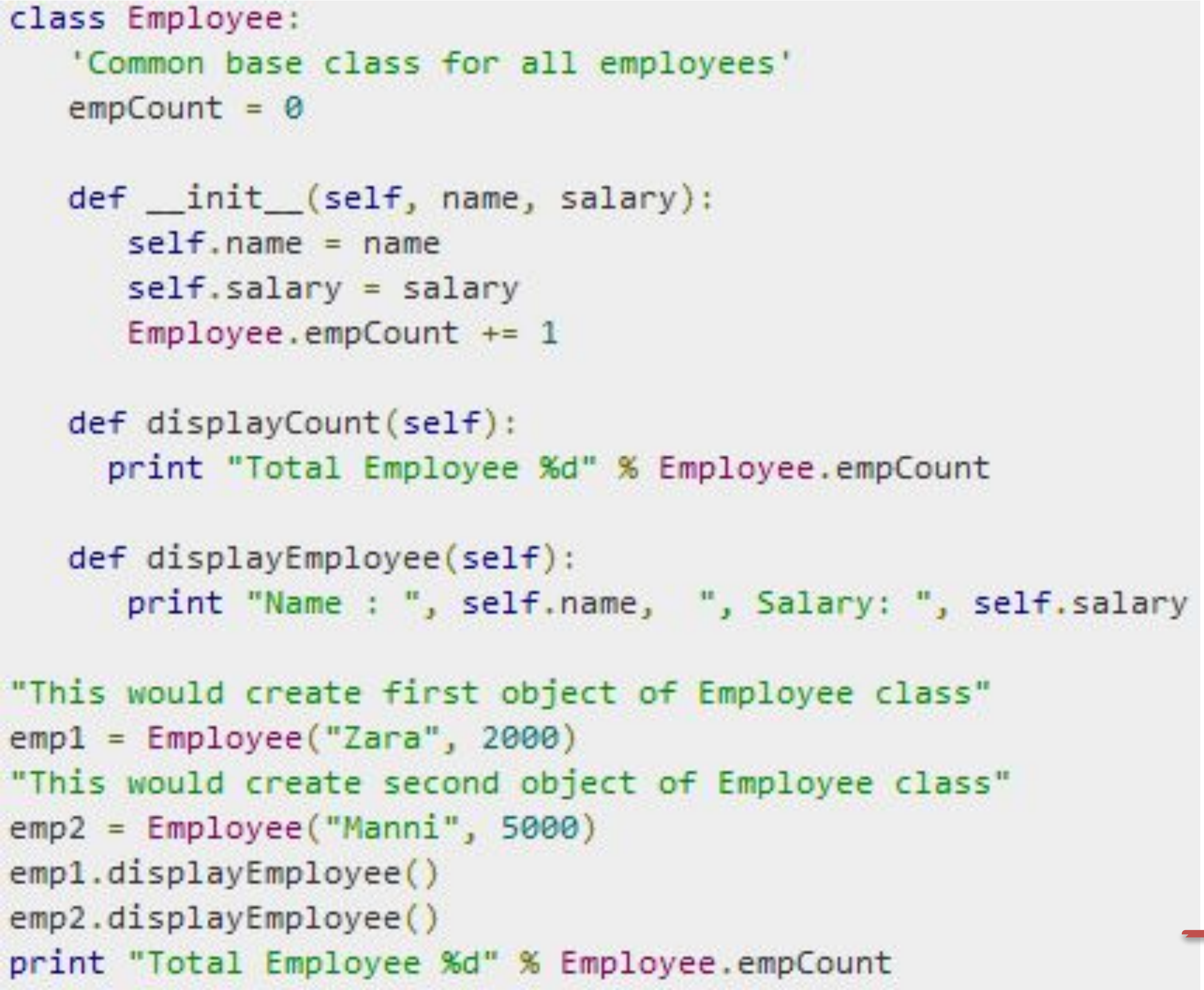
- **Function overloading:** The assignment of more than one behavior to a particular function.
- The operation performed varies by the types of objects or arguments involved.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Operator overloading:** The assignment of more than one function to a particular operator.

Example

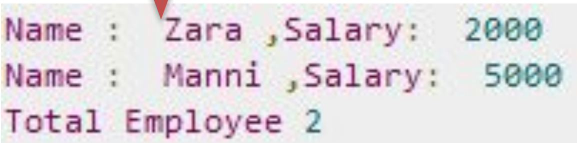


```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable ***empCount*** is a class variable whose value is shared among all instances of a this class.
- The first method ***__init__()*** is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*.
- Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

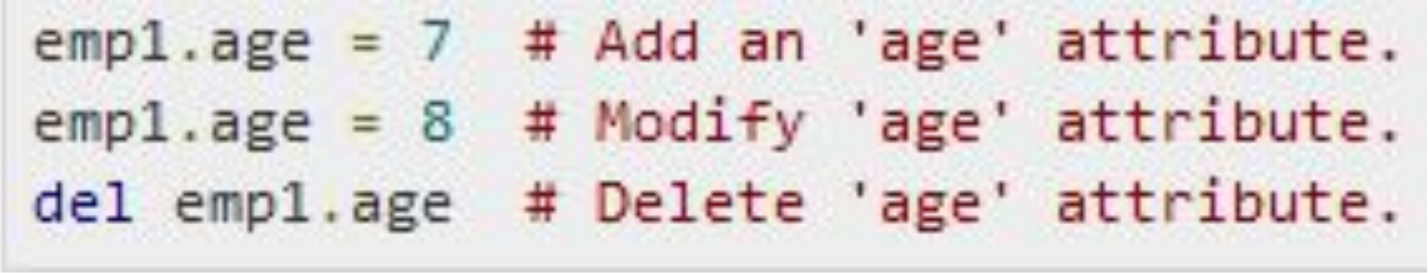


```python
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```



```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

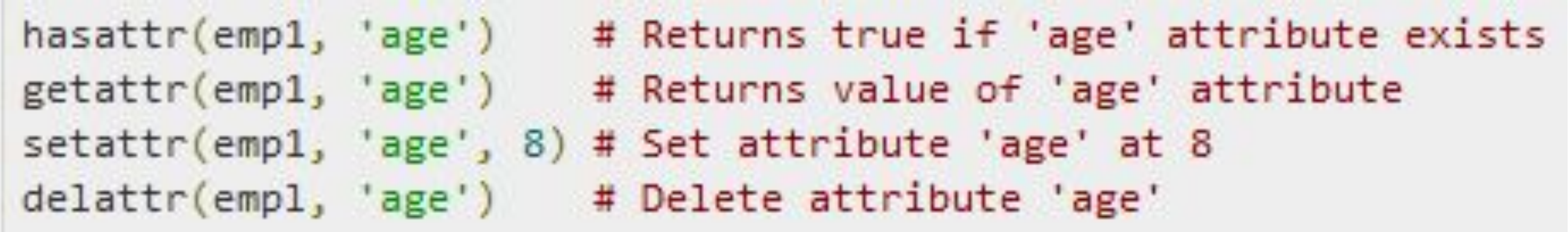
- You can add, remove, or modify attributes of classes and objects at any time.



```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions

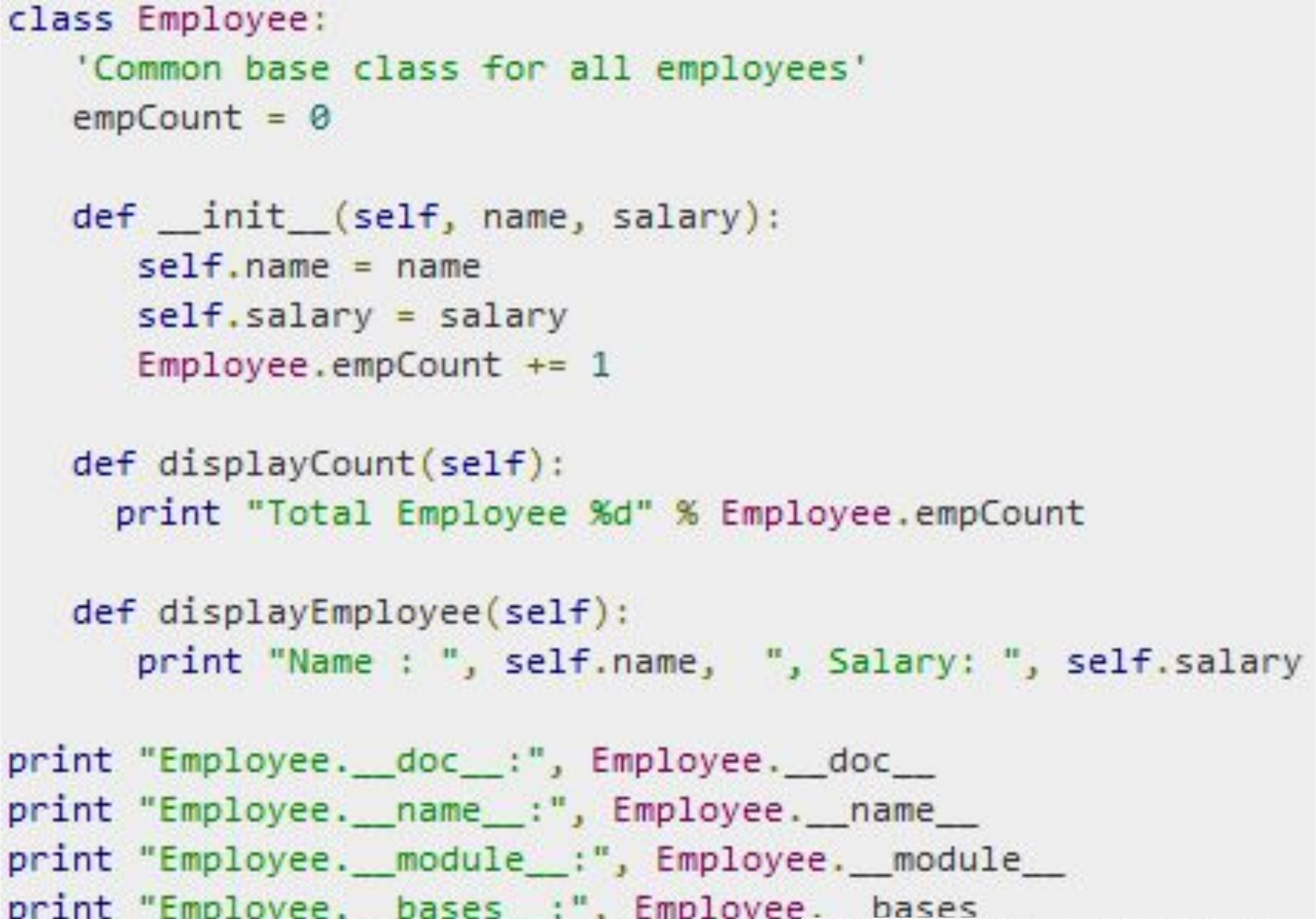
- The **getattr(obj, name[, default])** : to access the attribute of object.
- The **hasattr(obj,name)** : to check if an attribute exists or not.
- The **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** : to delete an attribute.



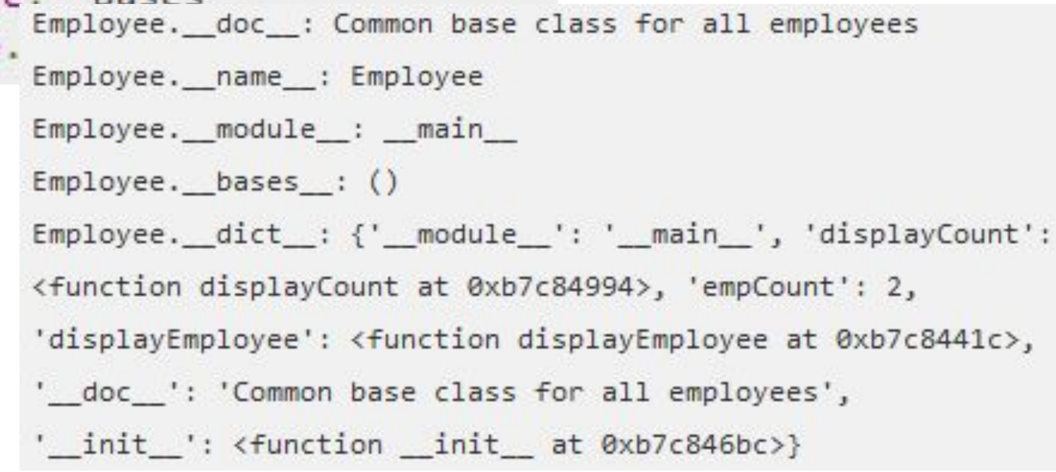
```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

Built-In Class Attributes

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −
- **__dict__:** Dictionary containing the class's namespace.
- **__doc__:** Class documentation string or none, if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__:** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.



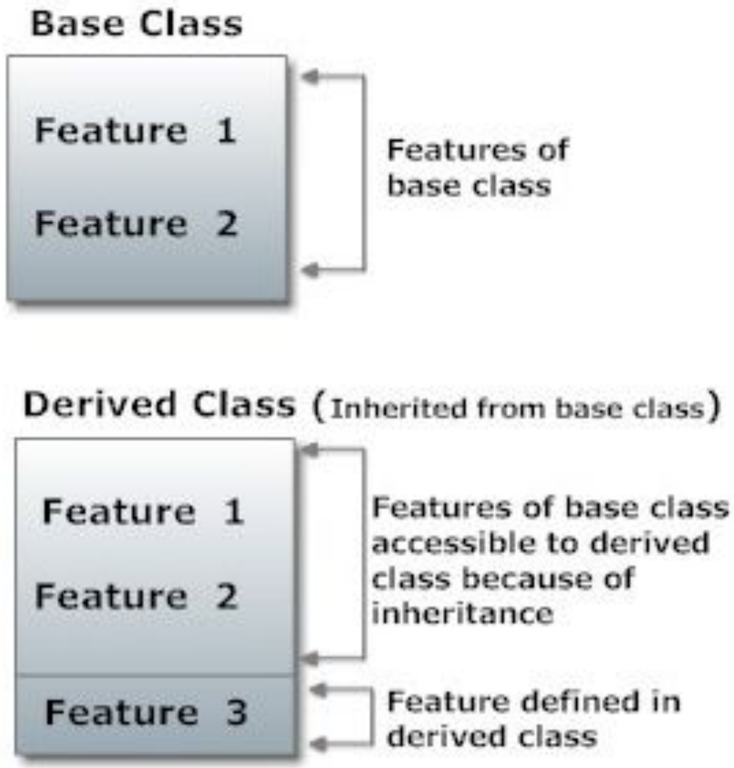
```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases
print "Employee.__dict__:", Employee.
```

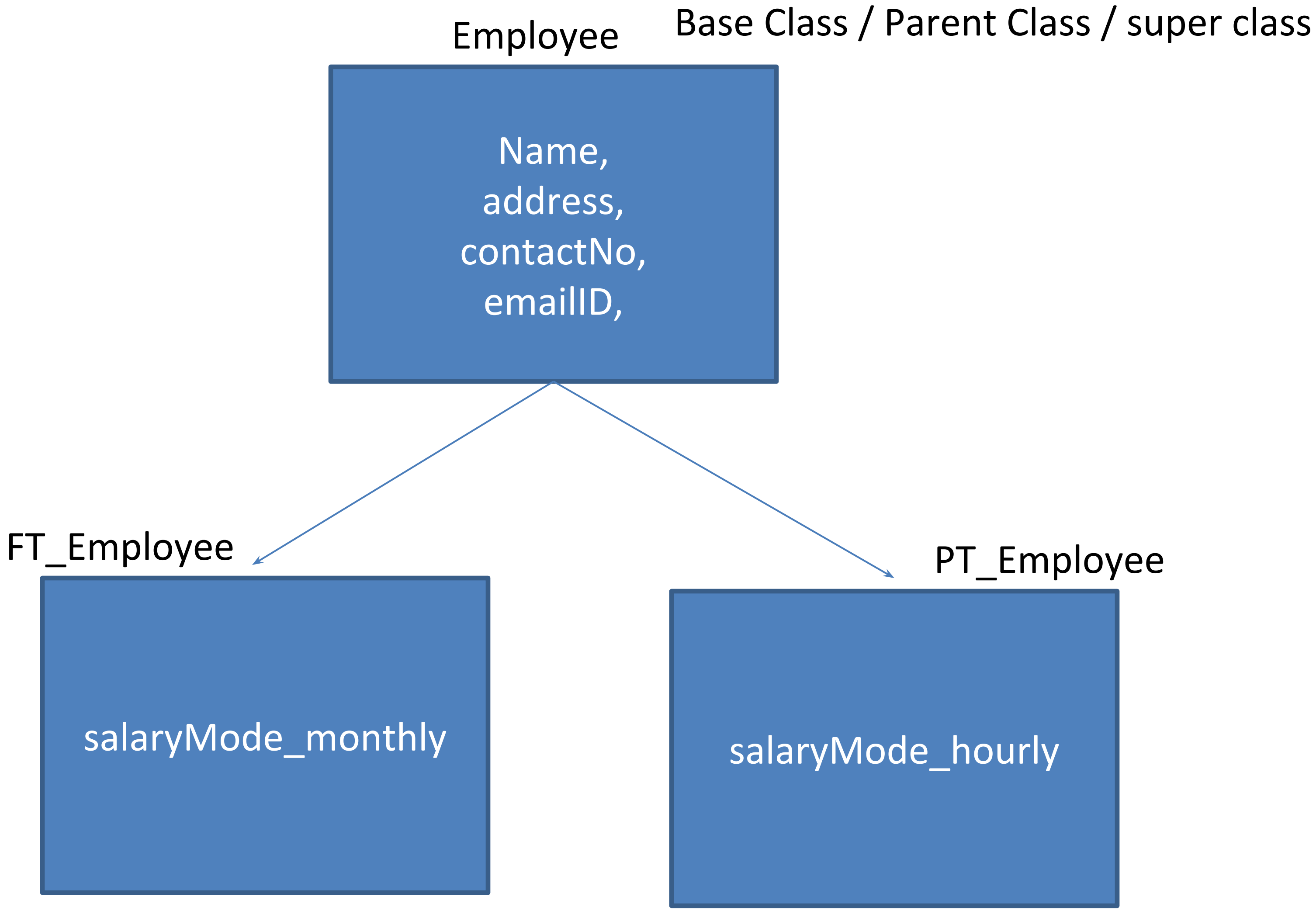


```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Python Inheritance

- Inheritance is a powerful feature in object oriented programming.
- It refers to defining a new class with little or no modification to an existing class.
- The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.
- Derived class inherits features from the base class, adding new features to it.
- This results into re-usability of code.


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Features of base class accessible to derived class because of inheritance
Feature 3
Feature defined in derived class



Employee

Base Class / Parent Class / super class

Name,
address,
contactNo,
emailID,

FT_Employee

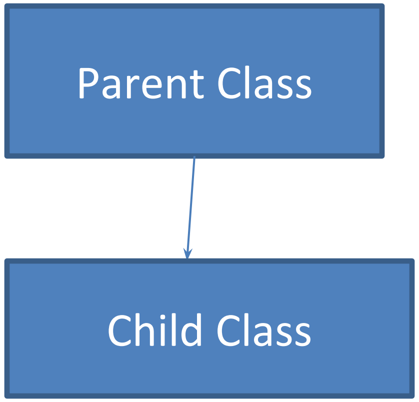
salaryMode_monthly

PT_Employee

salaryMode_hourly

Derived class/Child Class /sub class

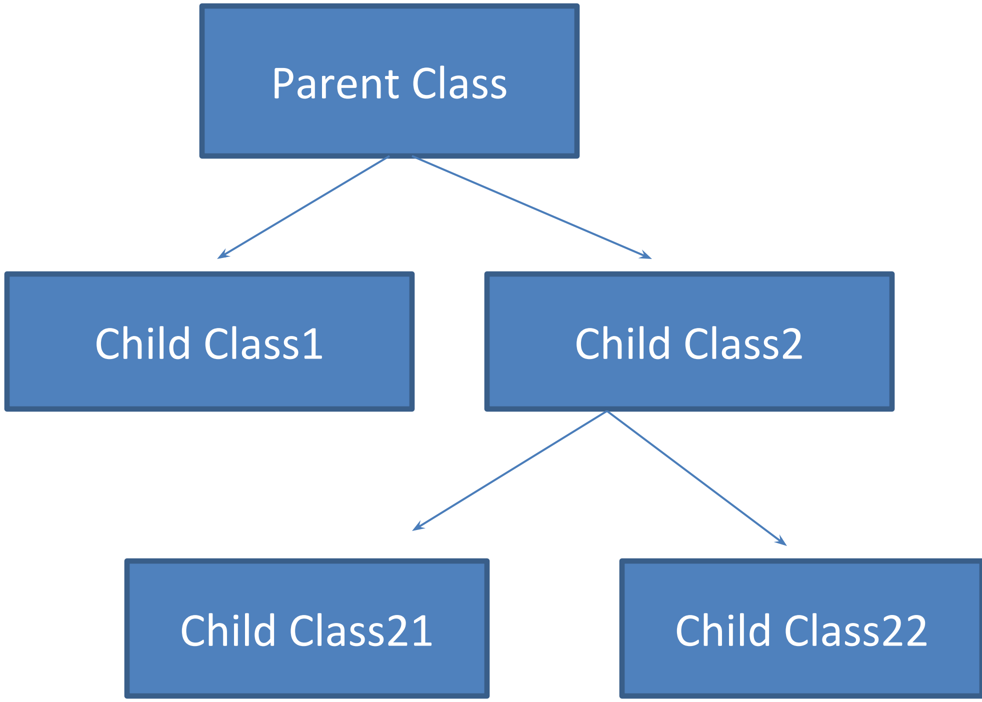
Reusability



Parent Class

Child Class

Single inheritance



Parent Class

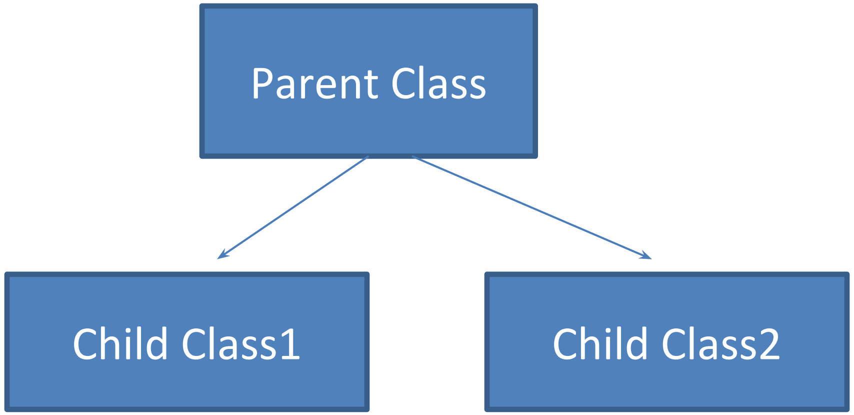
Child Class1

Child Class2

Child Class21

Child Class22

Multilevel Inheritance

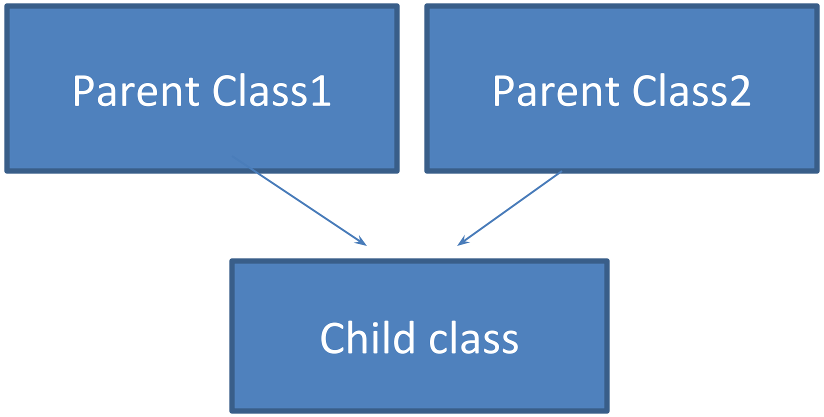


Parent Class

Child Class1

Child Class2

Hierarchical Inheritance

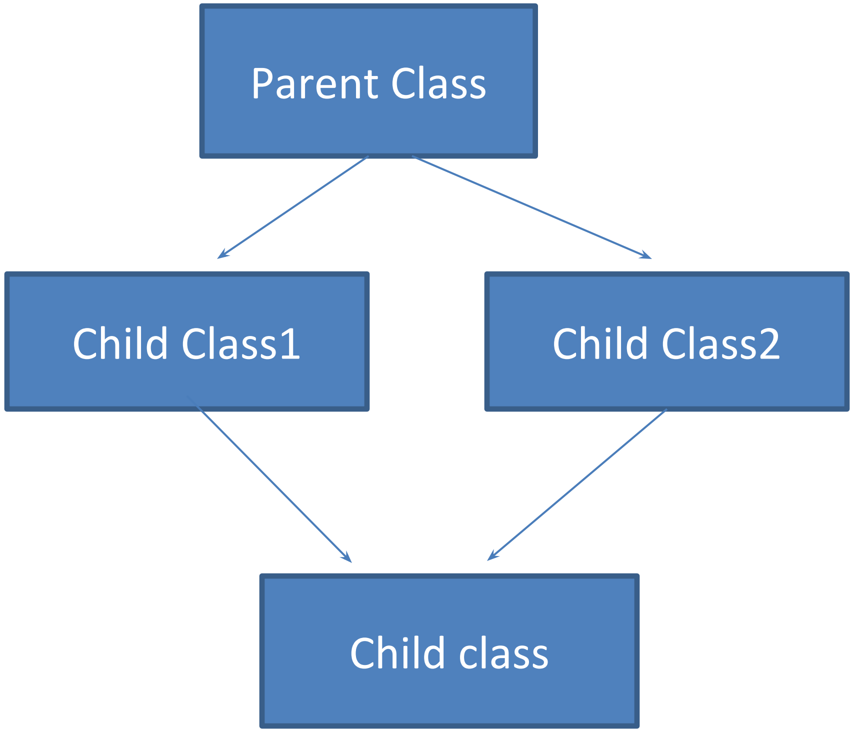


Parent Class1

Parent Class2

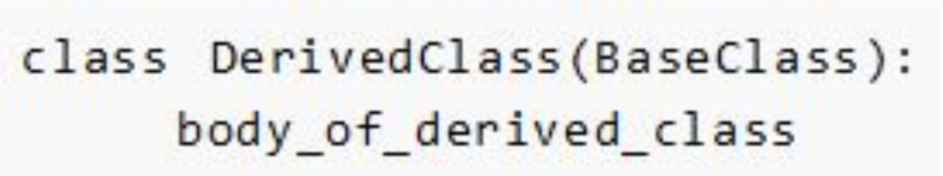
Child class

Multiple Inheritance



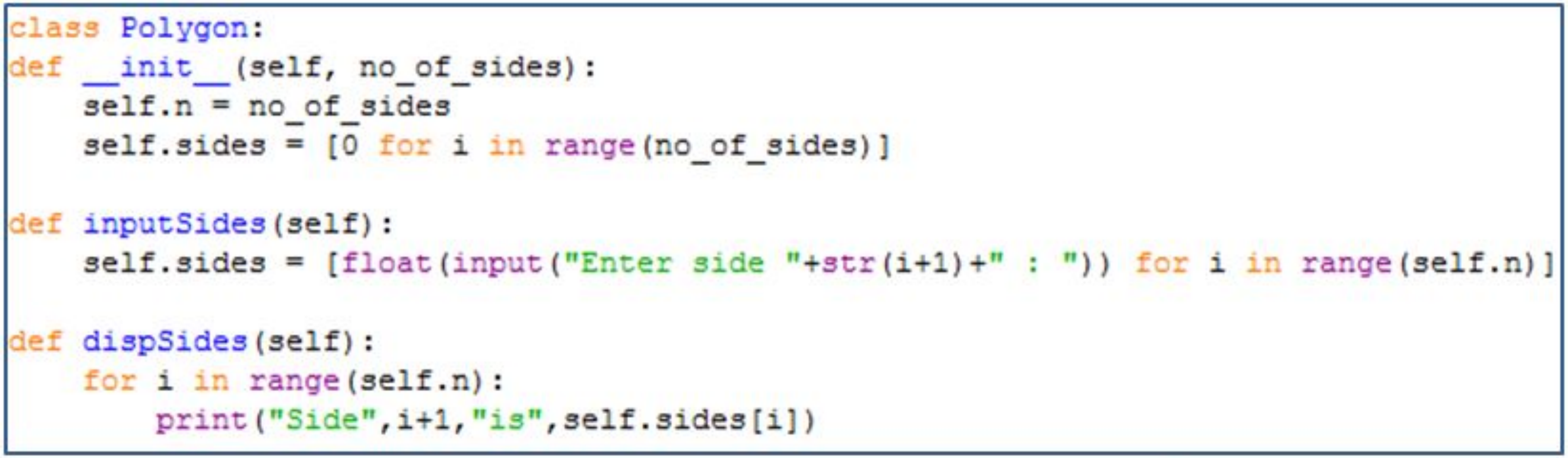
Hybrid Inheritance

Python Inheritance Syntax

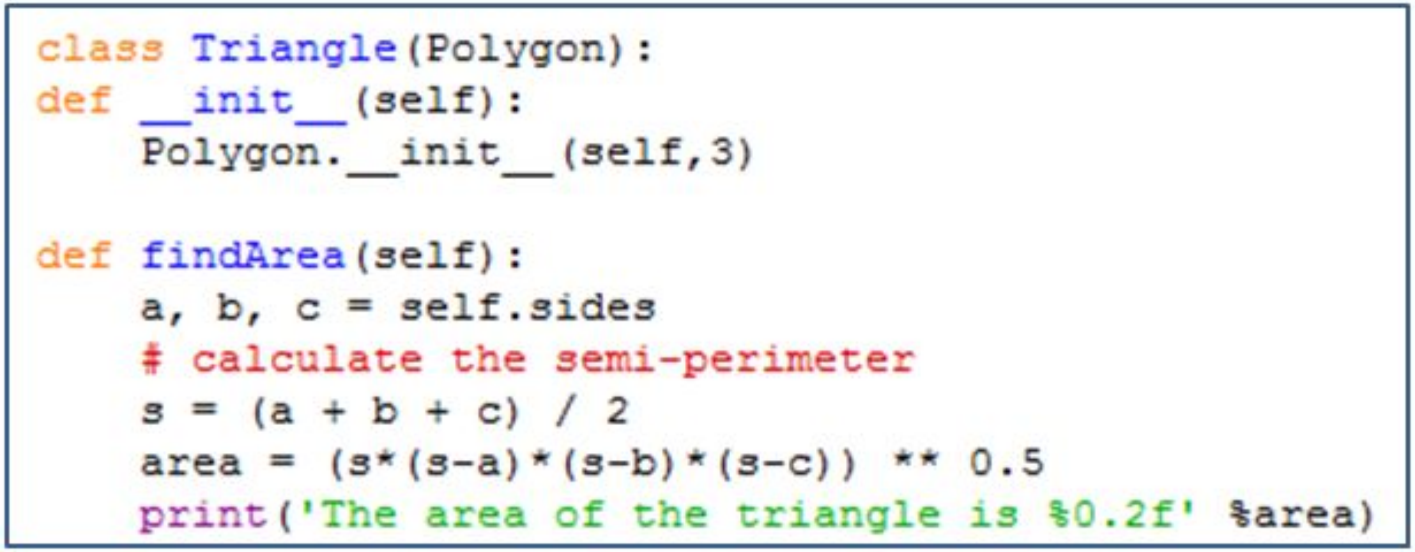


```
class DerivedClass(BaseClass):
    body_of_derived_class
```

Example of Inheritance in Python



```
class Polygon:
def __init__(self, no_of_sides):
    self.n = no_of_sides
    self.sides = [0 for i in range(no_of_sides)]

def inputSides(self):
    self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

def dispSides(self):
    for i in range(self.n):
        print("Side",i+1,"is",self.sides[i])
```



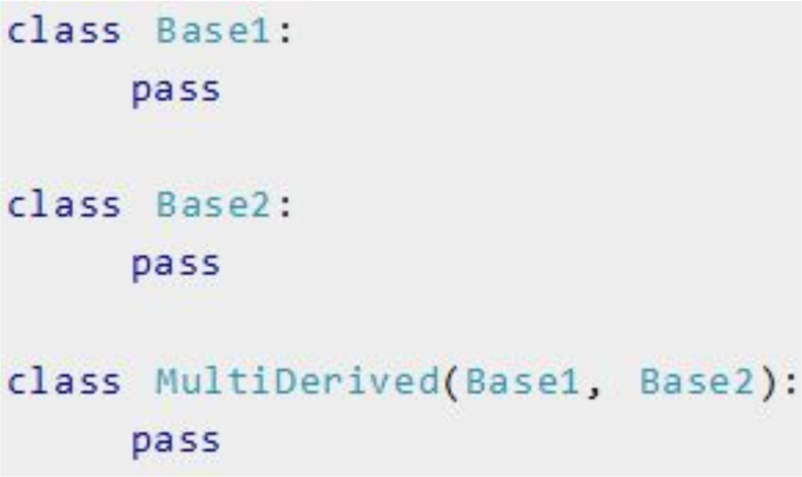
```
class Triangle(Polygon):
def __init__(self):
    Polygon.__init__(self,3)

def findArea(self):
    a, b, c = self.sides
    # calculate the semi-perimeter
    s = (a + b + c) / 2
    area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
    print('The area of the triangle is %0.2f' %area)
```

Example of Inheritance in Python

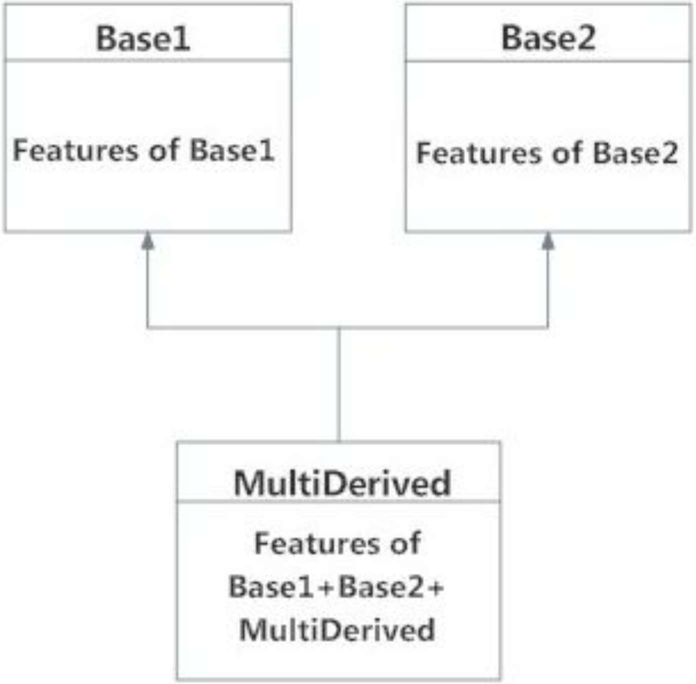
- Here, we did not define methods like inputSides() or dispSides() for class Triangle, we were able to use them.
- If an attribute is not found in the class, search continues to the base class.
- This repeats recursively, if the base class is itself derived from other classes.

Python Multiple Inheritance

- Multiple inheritance is possible in Python unlike other programming languages.
- A class can be derived from more than one base classes.
- The syntax for multiple inheritance is similar to single inheritance
- **Example:** The class MultiDerived inherits from both Base1 and Base2

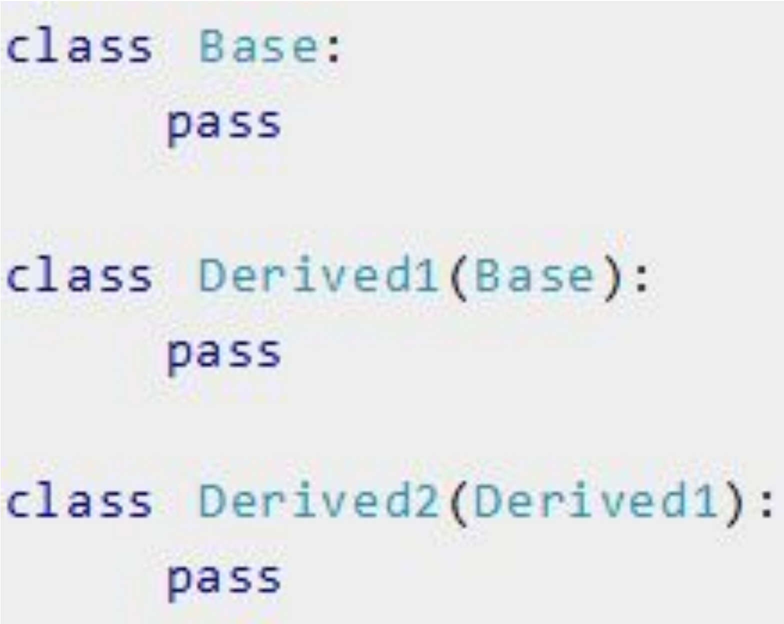


```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

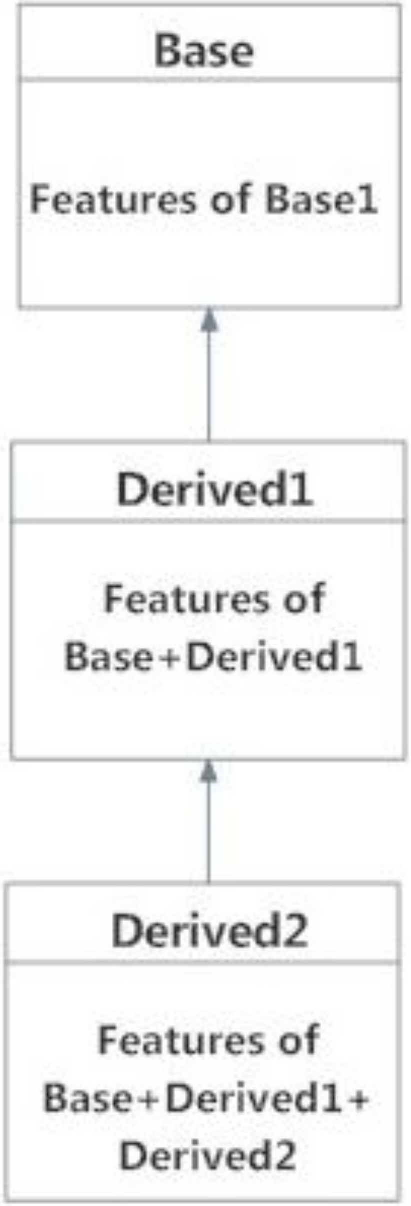


Multilevel Inheritance in Python

- we can inherit form a derived class.
- This is called multilevel inheritance.
- Multilevel inheritance can be of any depth in Python.
- **Example:**

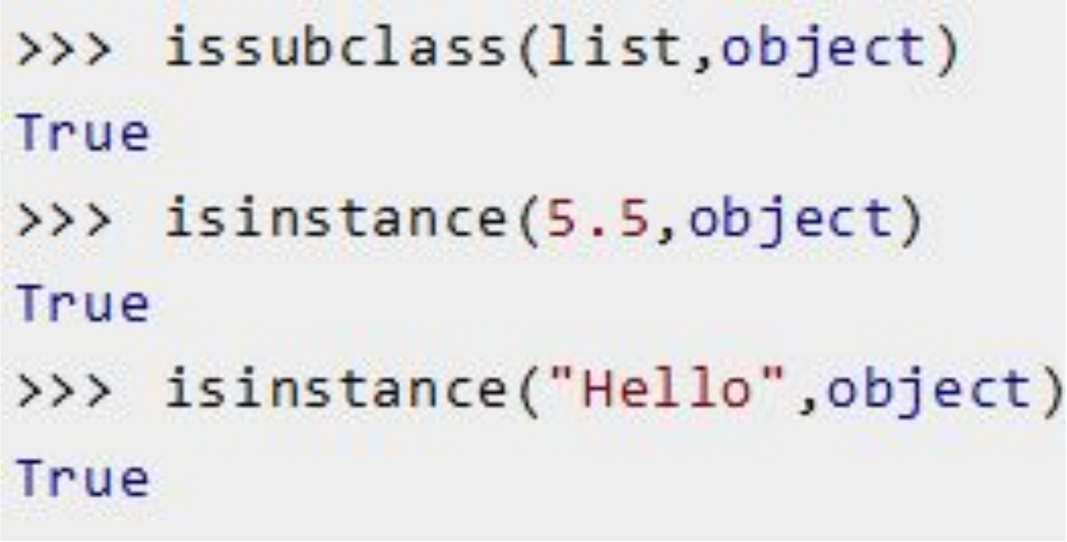


```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```



Method Resolution Order in Python

- Every class in Python is derived from the class object.
- It is the most base type in Python.
- So technically, all other class, either built-in or user-defines, are derived classes and all objects are instances of object class



```
>>> issubclass(list,object)
True
>>> isinstance(5.5,object)
True
>>> isinstance("Hello",object)
True
```

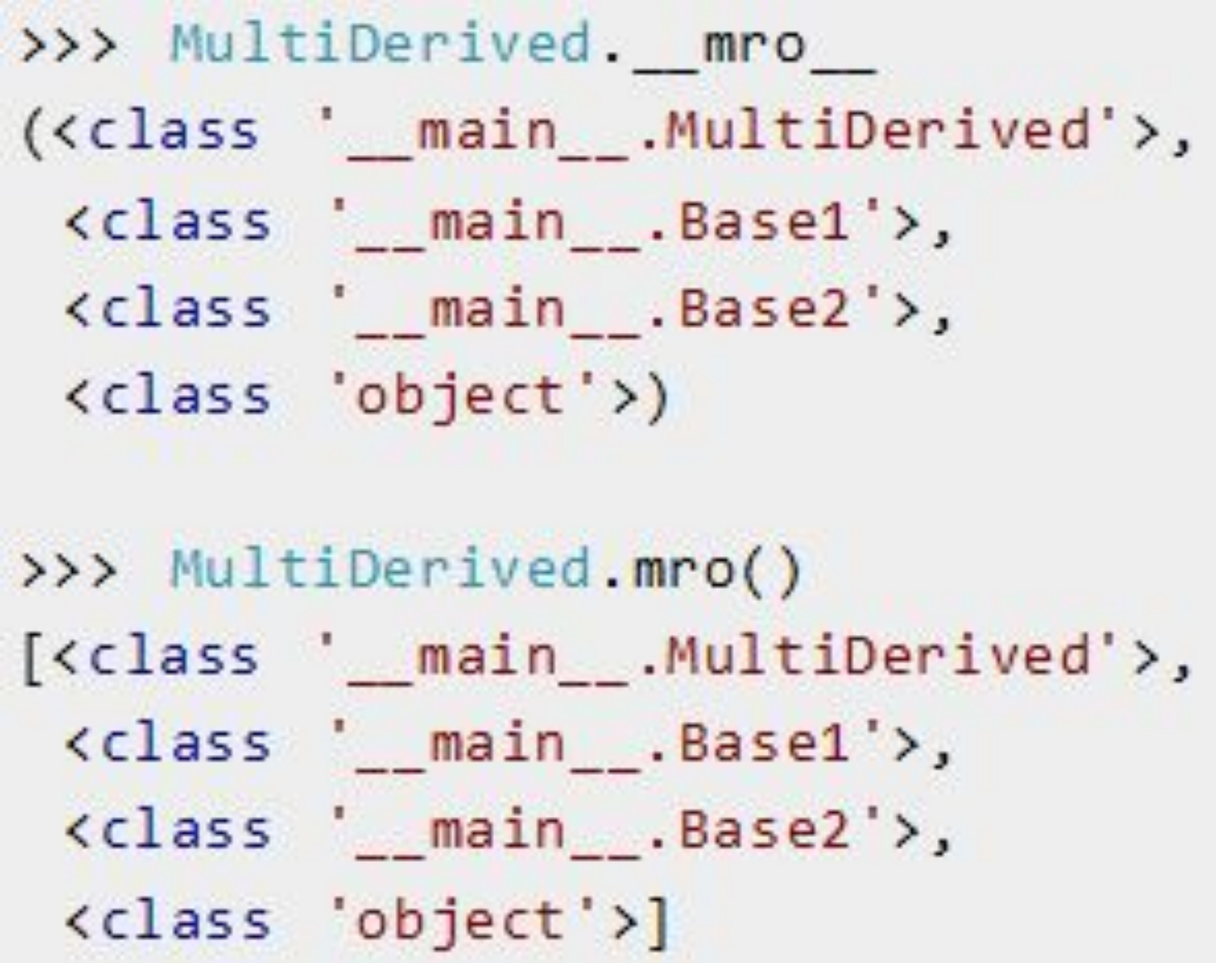
Method Resolution Order in Python

- In the multiple inheritance scenario, any specified attribute is searched first in the current class.
- If not found, the search continues into parent classes in depth-first (immediate Super class/Base Class) ,Then left-right fashion without searching same class twice.
- So, in the previous example of MultiDerived class the search order is [MultiDerived, Base1, Base2, object].
- This order is also called linearization of MultiDerived class and the set of rules used to find this order is called Method Resolution Order (MRO).

Method Resolution Order in Python

- MRO must prevent local precedence ordering and also provide monotonicity.
- It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.
- MRO of a class can be viewed as the __mro__ attribute or mro() method.
- The former returns a tuple while latter returns a list.

Method Resolution Order in Python

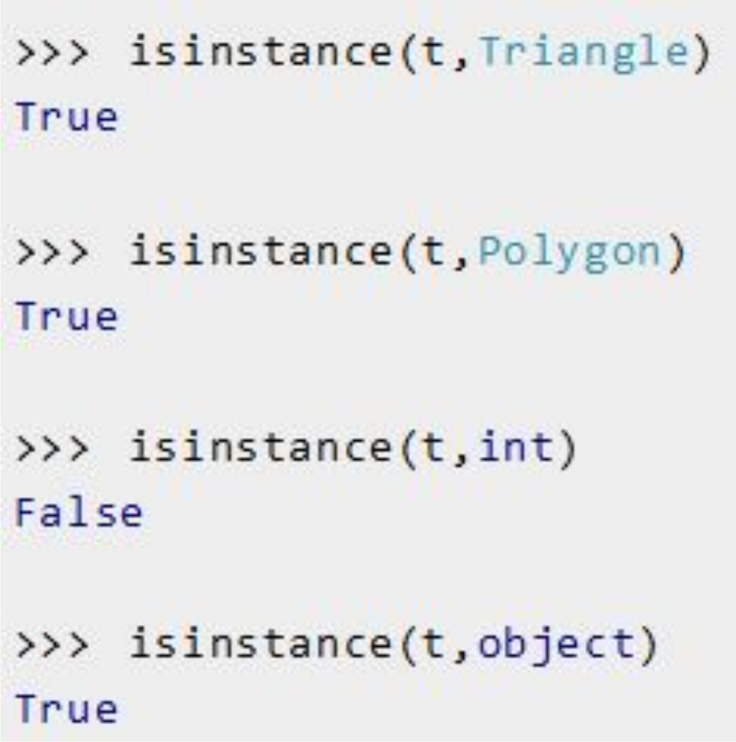


```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>]
```

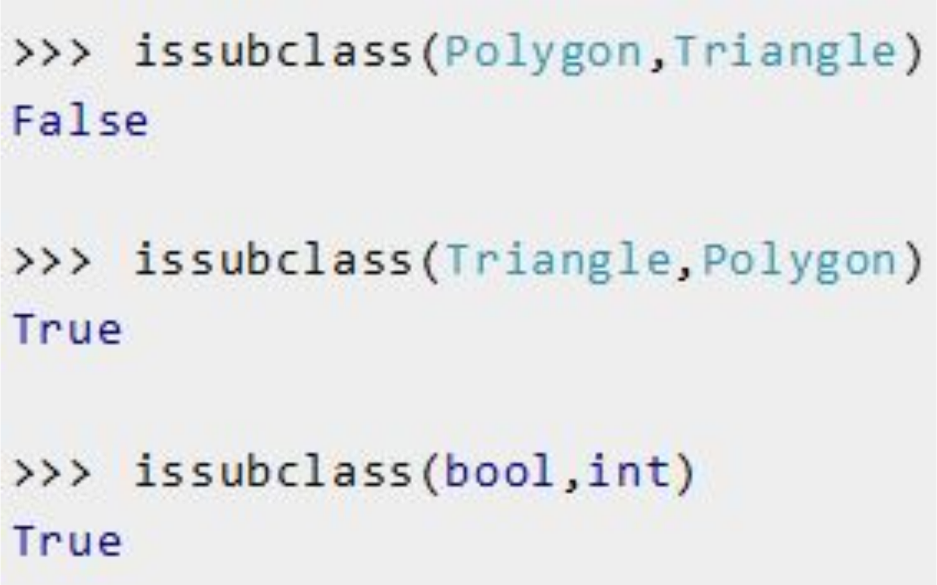
isinstance() and issubclass()

- Two built-in functions **isinstance()** and **issubclass()** are used to check inheritances.
- You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.
- The **issubclass(sub, sup):** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class):** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

isinstance() and issubclass()



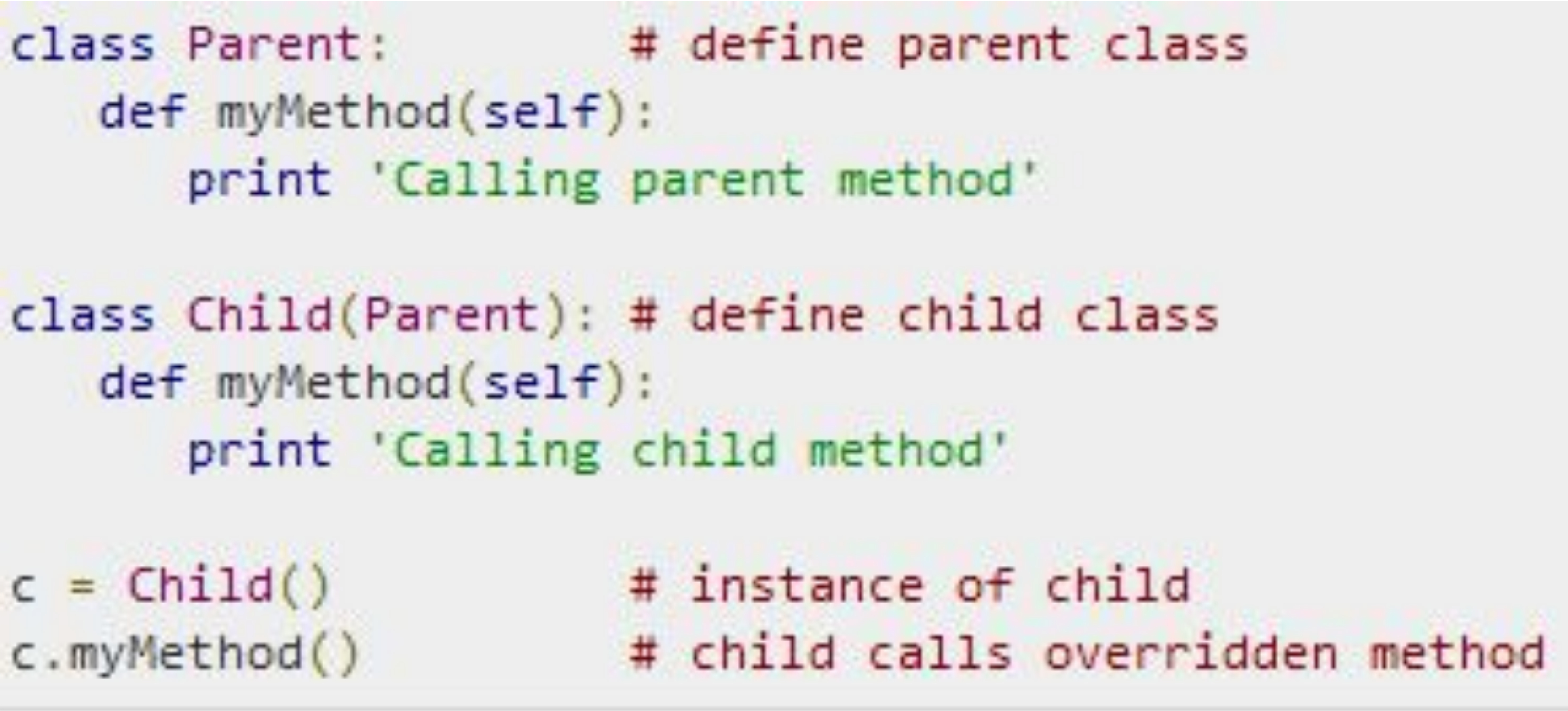
```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```



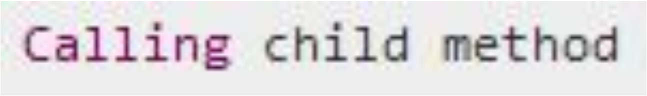
```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

Method Overriding in Python

- Overriding is a very important part of OOP since it is the feature that makes inheritance exploit its full power.
- Through method overriding a class may "copy" another class, avoiding duplicated code, and at the same time enhance or customize part of it.
- Method overriding is thus a strict part of the inheritance mechanism.
- In Python method overriding occurs simply defining in the child class a method with the same name of a method in the parent class.
- When you define a method in the object you make this latter able to satisfy that method call, so the implementations of its ancestors do not come in play.



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

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

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

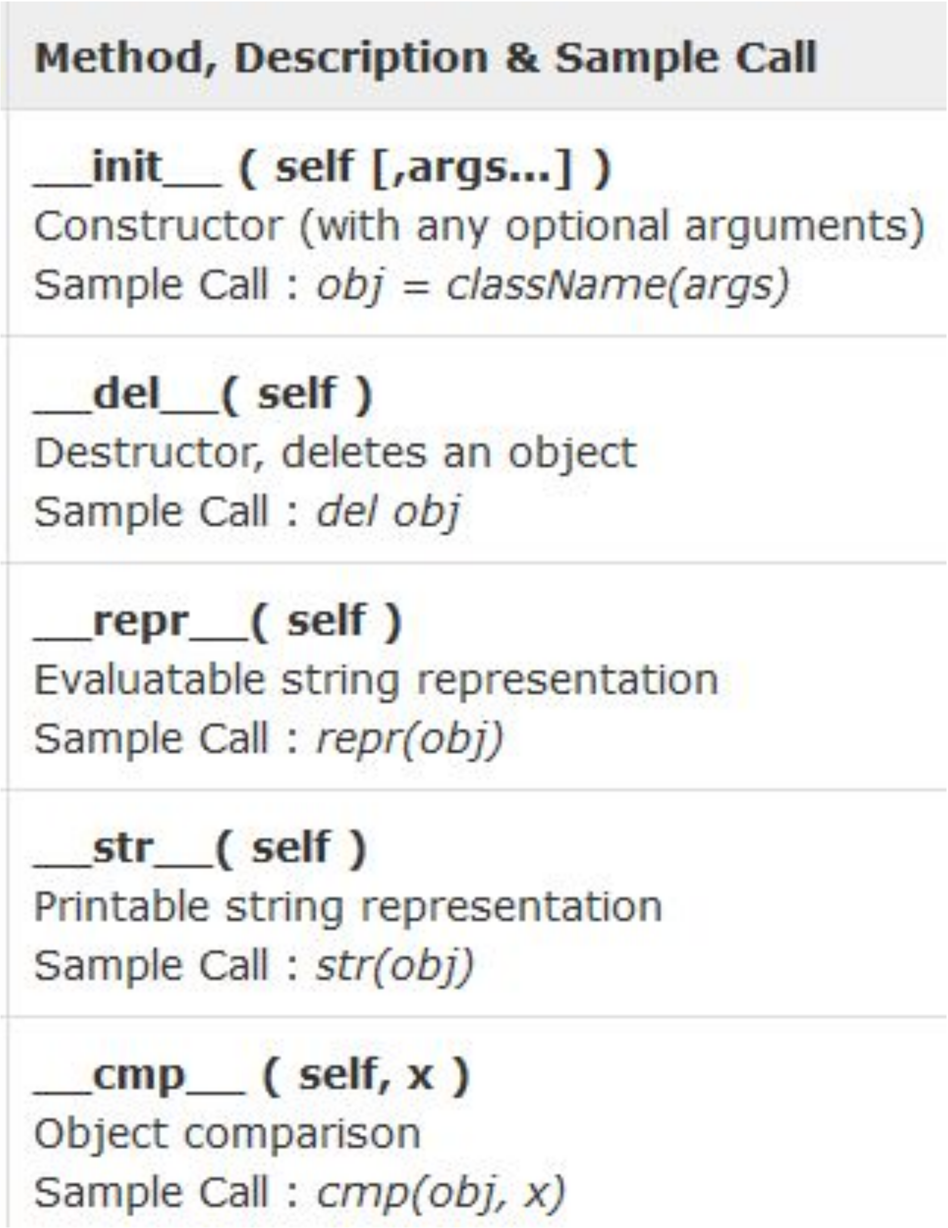


```
Calling child method
```

Special Functions in Python

- Class functions that begins with double underscore (__) are called special functions in Python.
- This is because, well, they are not ordinary.
- The __init__() function we defined above, is one of them.
- It gets called every time we create a new object of that class.

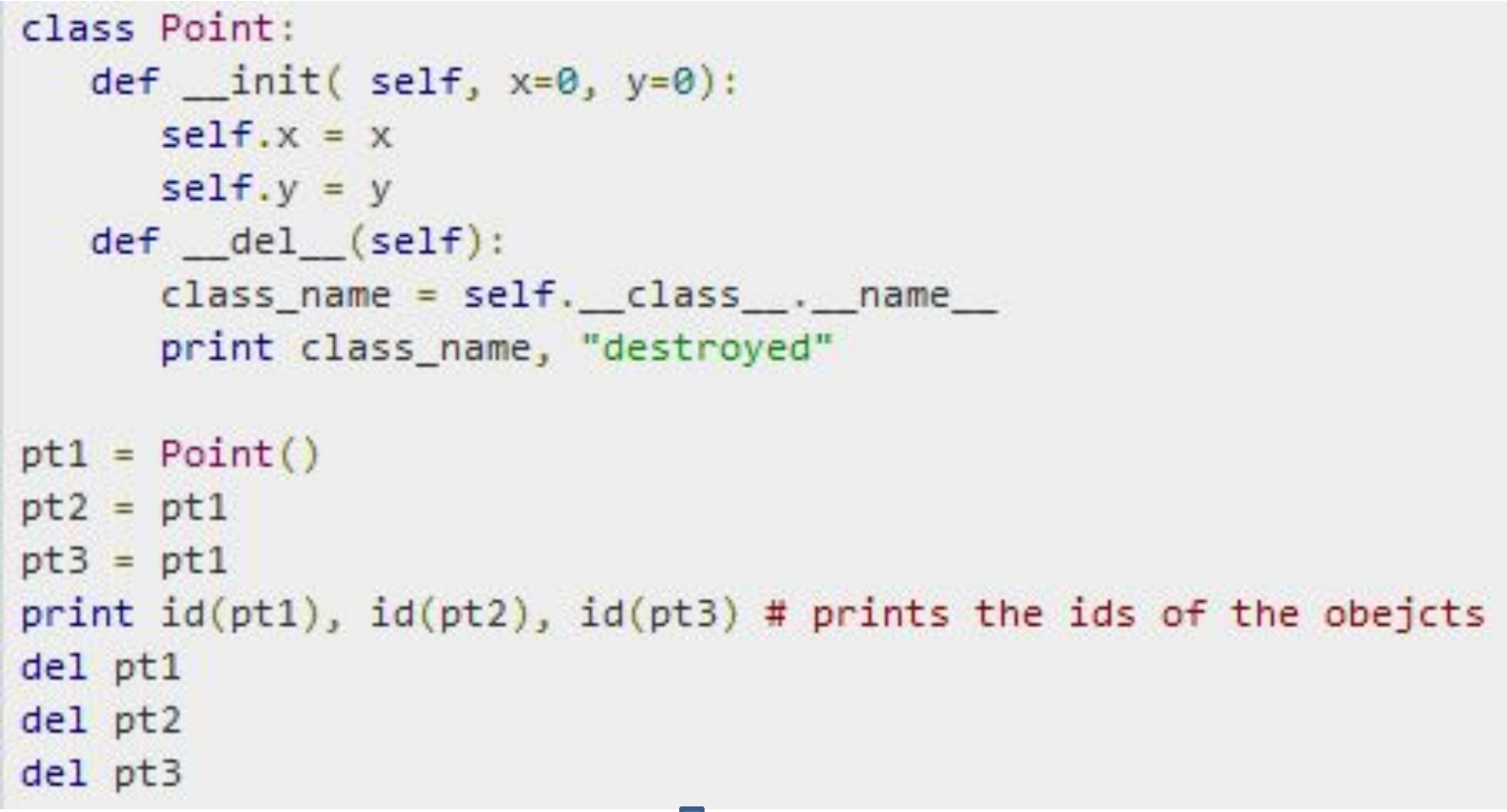
There are a ton of special functions in Python.



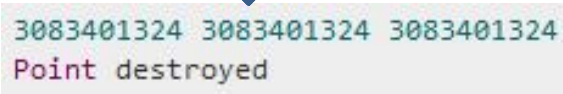
Method, Description & Sample Call
__init__ (self [,args...]) Constructor (with any optional arguments) Sample Call : *obj = className(args)*
__del__(self) Destructor, deletes an object Sample Call : *del obj*
__repr__(self) Evaluatable string representation Sample Call : *repr(obj)*
__str__(self) Printable string representation Sample Call : *str(obj)*
__cmp__ (self, x) Object comparison Sample Call : *cmp(obj, x)*

Special Functions in Python

- **Example:** This __del__() destructor prints the class name of an instance that is about to be destroyed



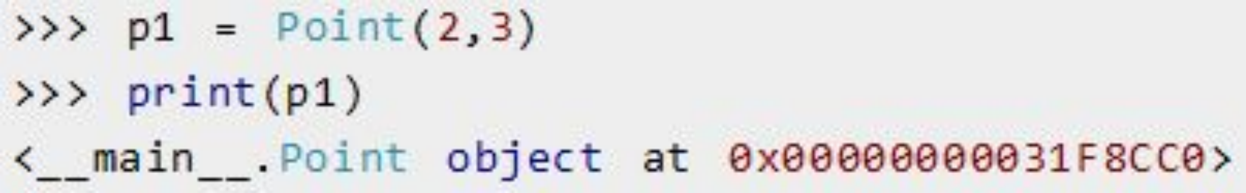
```
class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```



```
3083401324 3083401324 3083401324
Point destroyed
```

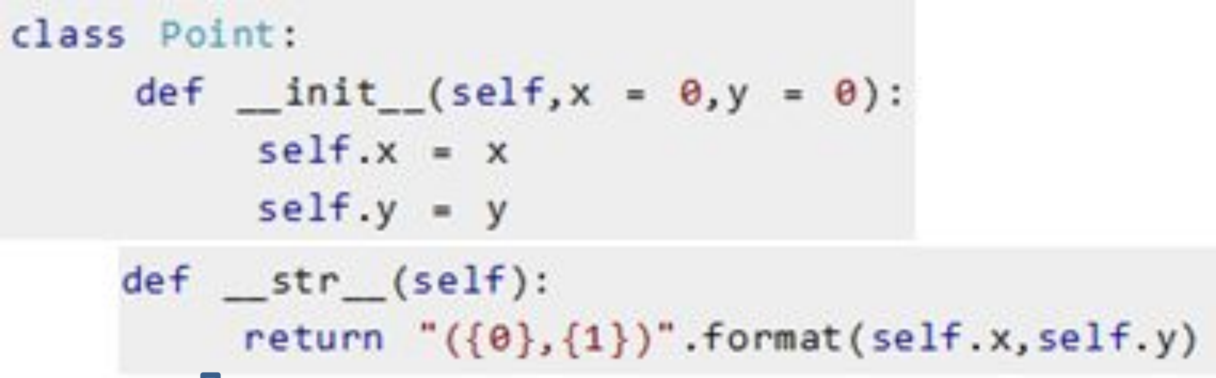
Special Functions in Python

- Using special functions, we can make our class compatible with built-in functions.

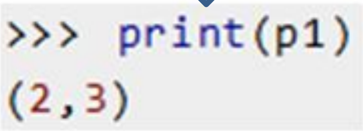


```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

- That did not print well. But if we define __str__() method in our class, we can control how it gets printed.

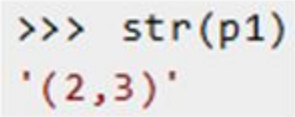


```
class Point:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```



```
>>> print(p1)
(2,3)
```

=



```
>>> str(p1)
'(2,3)'
```

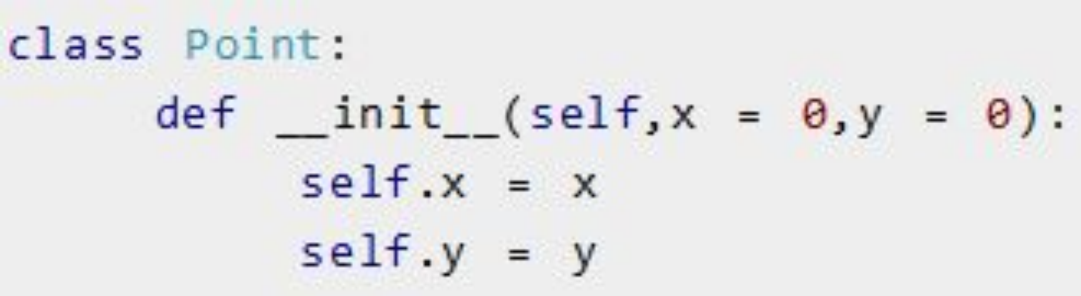
when you do str(p1) ,Python is internally doing p1.__str__(). Hence the name, special functions.

Python Operator Overloading

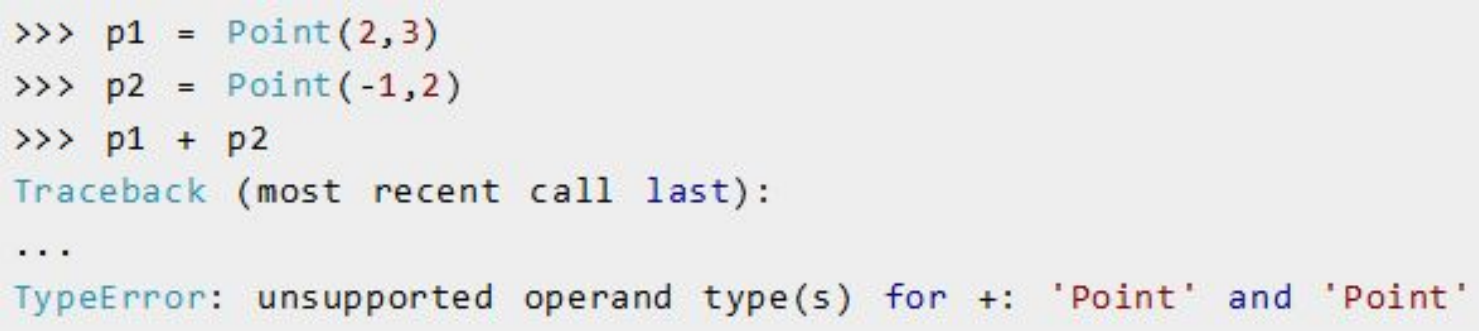
- Python operators work for built-in classes.
- But same operator behaves differently with different types.
- For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.
- This feature in Python, that allows same operator to have different meaning according to the context is called **operator overloading.**

Python Operator Overloading

- **Example: Let** us consider the following class, which tries to simulate a point in 2-D coordinate system.



```python
class Point:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
```



```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> p1 + p2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

***TypeError* was raised since Python didn't know how to add two *Point* objects together.**

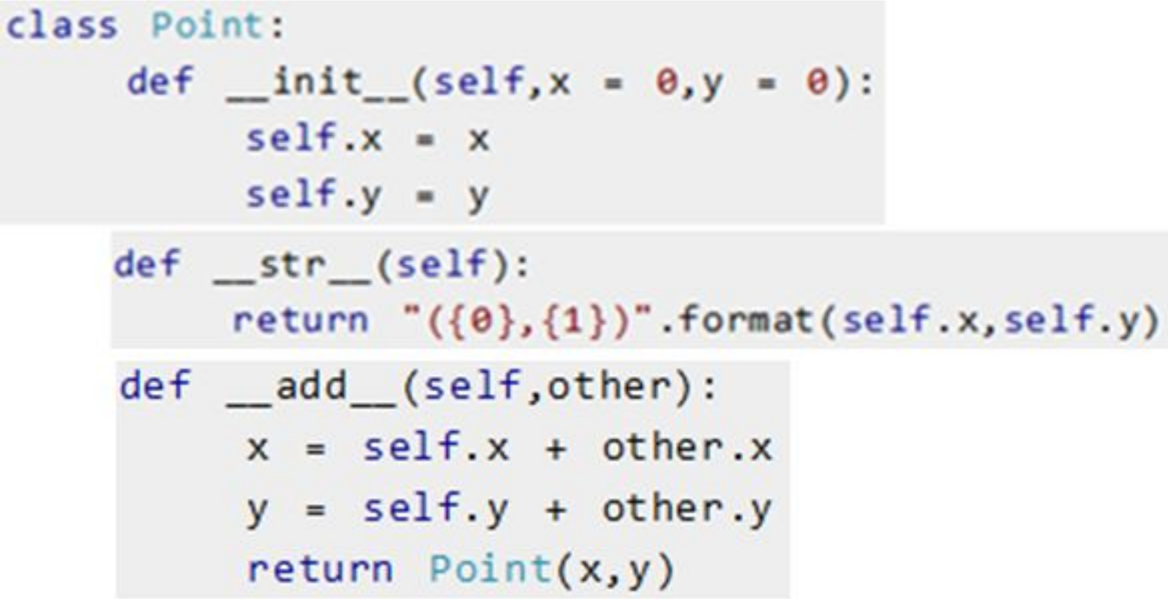
However, the good news is that we can teach this to Python through operator overloading.

Python Operator Overloading

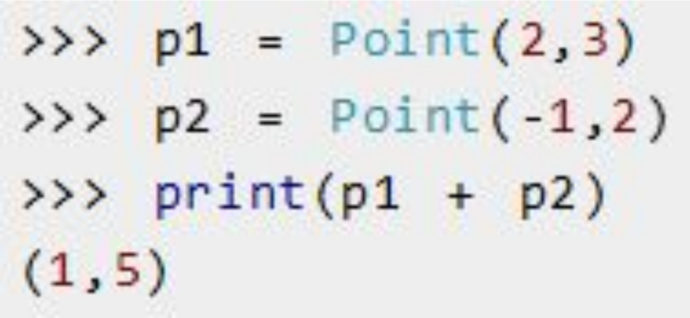
Overloading the + Operator:

- To overload the + sign, we will need to implement __add__() function in the class.
- With great power comes great responsibility.
- We can do whatever we like, inside this function. But it is sensible to return a Point object of the coordinate sum.

Python Operator Overloading



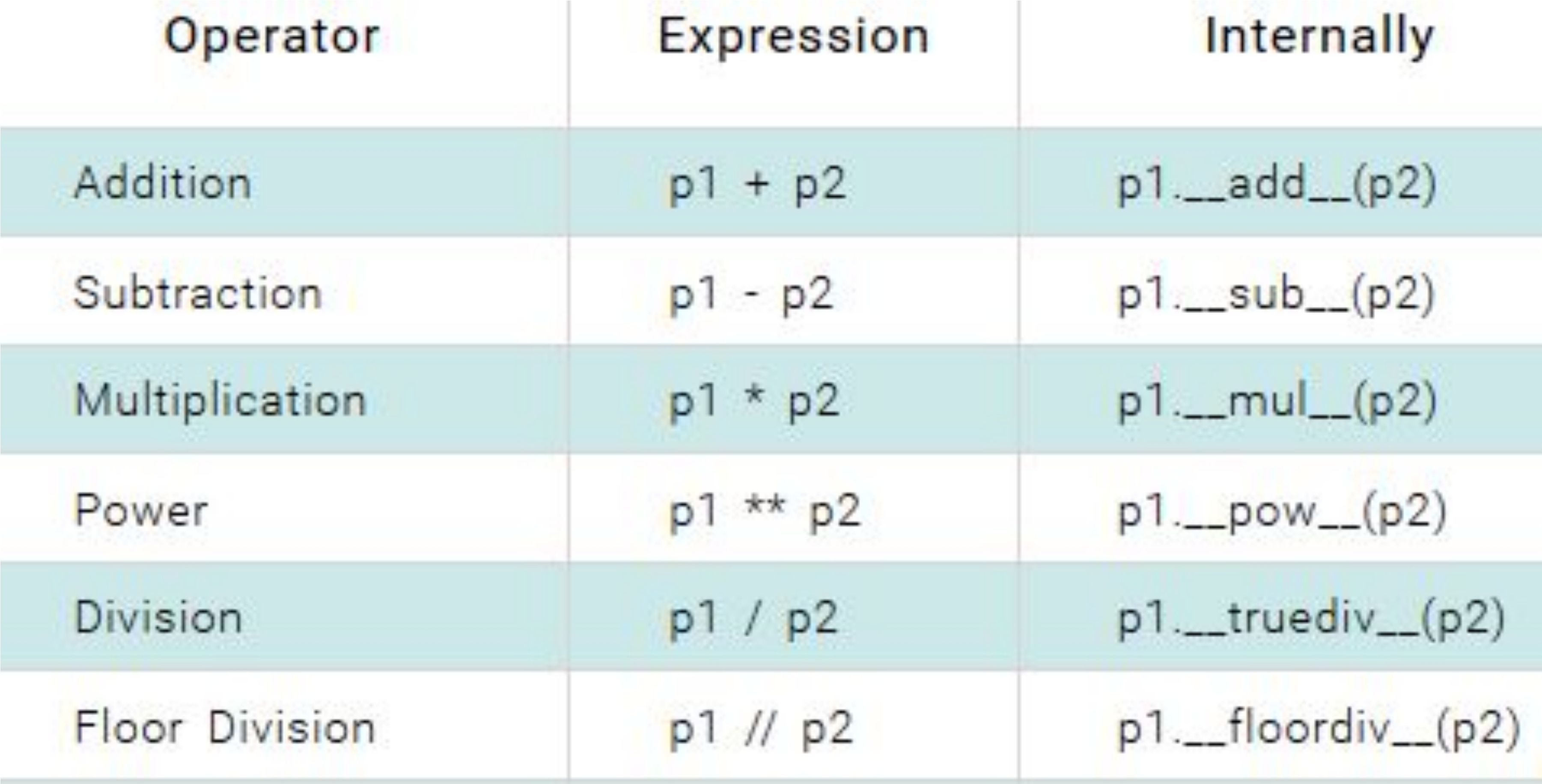
```
class Point:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
```



```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> print(p1 + p2)
(1,5)
```

What actually happens is that, when you do p1 + p2, Python will call p1.__add__(p2) which in turn is Point.__add__(p1,p2).

Operator Overloading Special Functions in Python

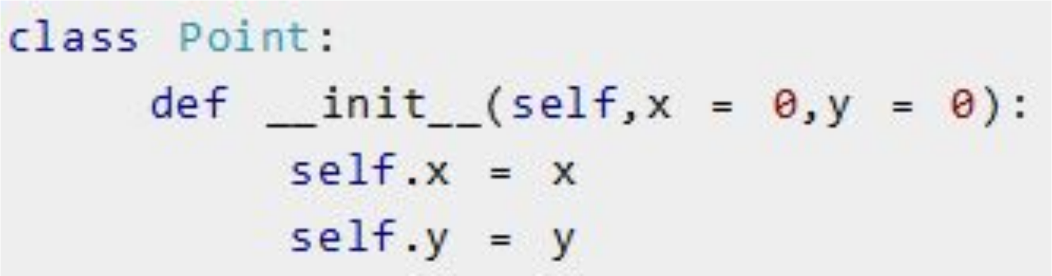


Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)

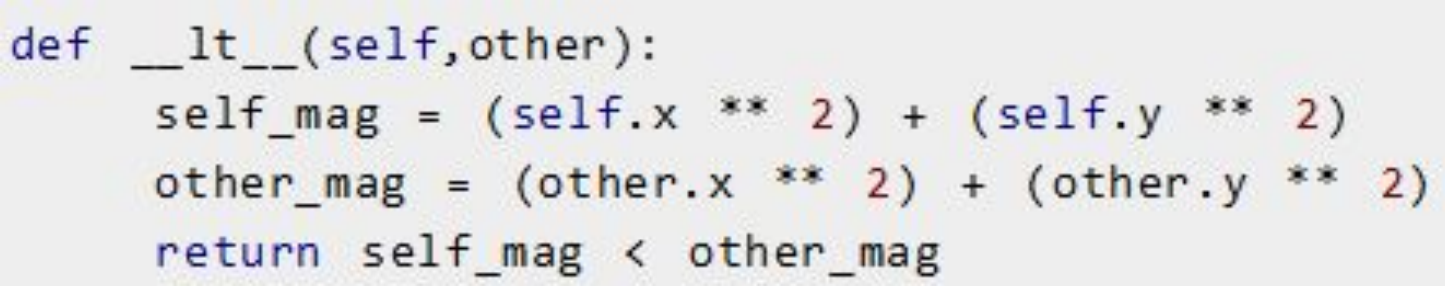
Overloading Comparison Operators in Python

- Python does not limit operator overloading to arithmetic operators only.
- We can overload comparison operators as well.
- Suppose, we wanted to implement the less than symbol < symbol in our Point class.
- Let us compare the magnitude of these points from the origin and return the result for this purpose.

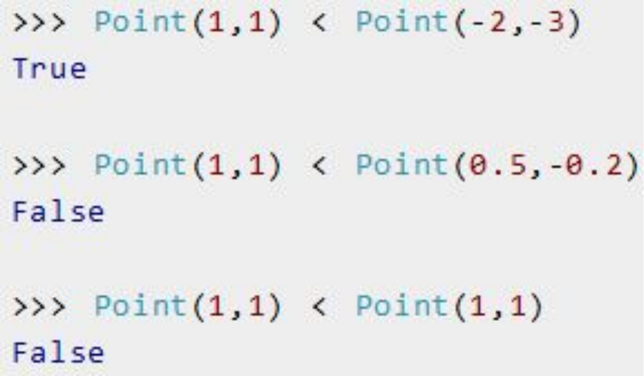
Overloading Comparison Operators in Python



```
class Point:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
```

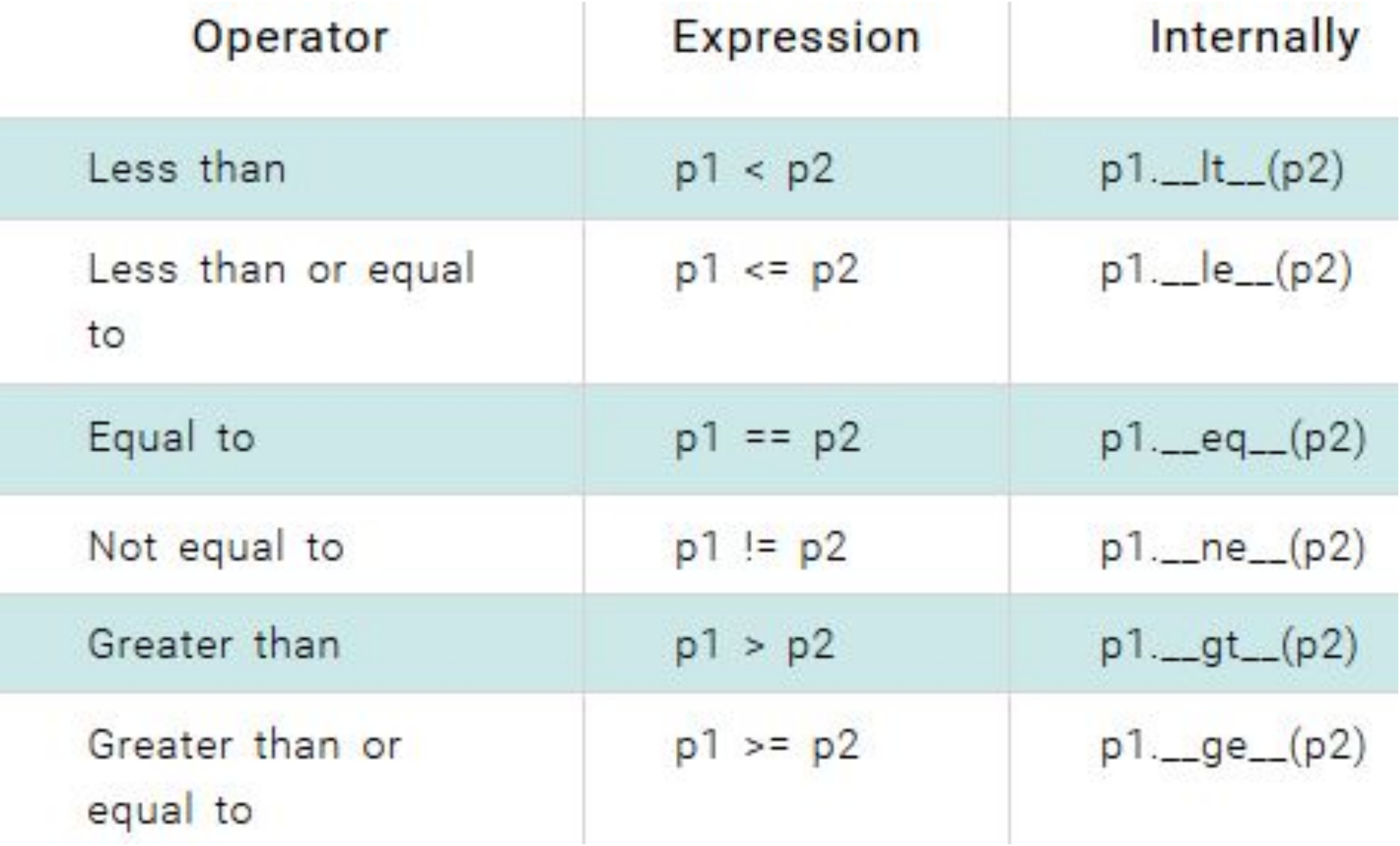


```
    def __lt__(self,other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag
```



```
>>> Point(1,1) < Point(-2,-3)
True

>>> Point(1,1) < Point(0.5,-0.2)
False

>>> Point(1,1) < Point(1,1)
False
```

Comparison Operator Overloading in Python



Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

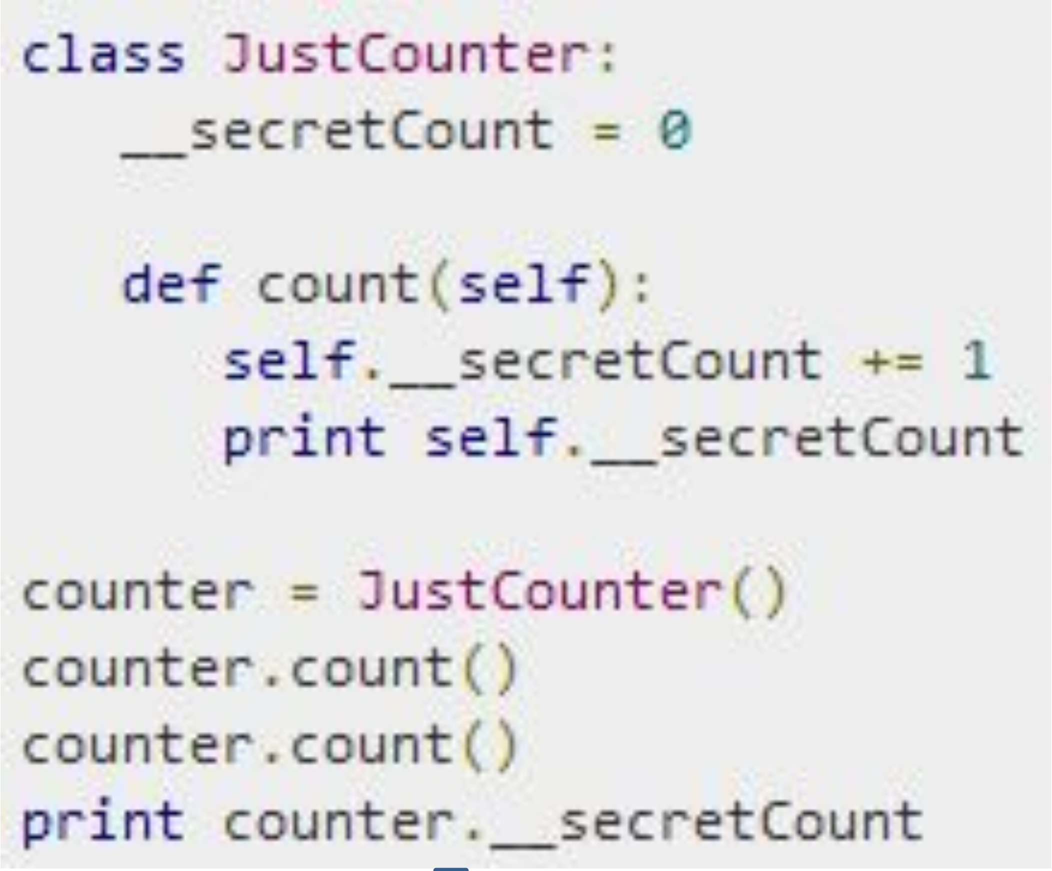
Data Hiding

- An object's attributes may or may not be visible outside the class definition.
- You need to name attributes with a **double underscore** prefix, and those attributes then are not be directly visible to outsiders.

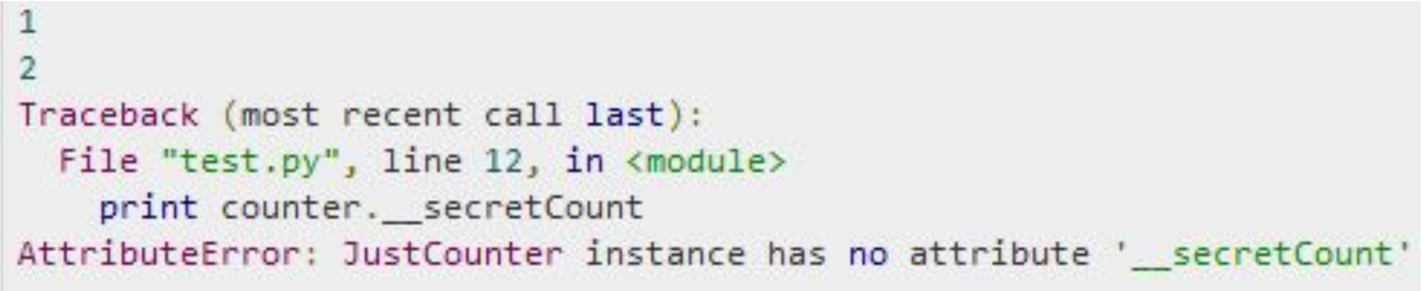
__x =>private variable x

_x => protected (accessible only within base class and it derived classes)

Data Hiding



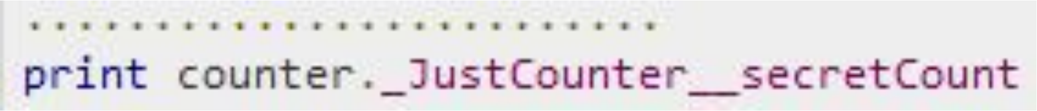
```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```



```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

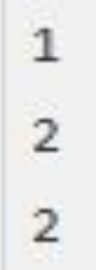
Data Hiding

- Python protects those members by internally changing the name to include the class name.
- You can access such attributes as ***object._className__attrName***.
- If you would replace your last line in previous example as following, then it works for you



```
...........................
print counter._JustCounter__secretCount
```

Output



```
1
2
2
```