Scopes(modifying variables in outer nonglobal scope)

```
def foo():
    a = 3
    def bar():
        'bar creates local copy of a the
moment it assigns to a'
        a = 5

    bar()
    print a #3

foo()
```

```
def foo():
   'foo creates an empty class and adds all
the variables to it to be modified by bar'
   class DFoo:pass
   DFoo.a = 3

   def bar():
      'bar can see DFoo in outer scope and
modify its members as side effect'
      DFoo.a = 5
   bar()
   print DFoo.a #5

foo()
```

Scopes(modifying variables in outer nonglobal scope)

Python 3 allows you to change the value of the closed-over variable without creating a new variable as a side effect. nonlocal tells the compiler that, despite the assignment happening in an inner block, the variable being modified is the one created in the outer block.

```
def foo():
    a=1
    def bar():
        "bar modifies a from outer scope"
        nonlocal a=2 #fixed in Python 3
```

Object Oriented Programming

An object oriented program is based on classes and a collection of interacting objects.

The four major principles of object orientation are:

- ✔Encapsulation
- ✔Data Abstraction
- ✔Polymorphism
- ✔Inheritance

OOP-Terminology

- Class
- Instance/Object
- Instantiation
- Class variable
- Instance variable
- Method
- Data member
- Operator/Function overloading
- Inheritence
- Polymorphism
- Encapsulation or Data abstraction

OOP-Class

class *name*:

"*documentation*"

Statements

class *name*(*base1*, *base2*, ...): pass

- Most, *statements* are method definitions:

def *name*(**self,** *arg1*, *arg2*, ...):

- May also be *class variable* assignments

OOP-methods

- Class methods have only one specific difference from ordinary functions--they must have an extra first name that has to be added to the beginning of the parameter list

- You do not give a value for this parameter(self) when you call the method, Python will provide it.

- This particular variable refers to the object itself, and by convention, it is given the name self.

OOP-Constructor

- Constructor __init__: A special method that is automatically invoked right after a new object is created

```
def __init__(self):
    print "A new object has been created!"
```

- Usually write one in each class
- Usually sets up the initial attribute values of new object in constructor

    self – first parameter in every instance method

    self receives reference to new object

```
def __init__(self, name):
    self.name = name #setting attribute name on new object self
```

OOP-class variables

```
class Employee: #class with no base
   'Common base class for all employees'
   EmpCount = 0 #class variable keeps track of number of employee instances

   ...
```

- Assignment statement in class but outside method creates class attribute
- Assignment statement executed only once, when Python first sees class definition
- Class attribute exists even before single object created
- Can use class attribute without any objects of class in existence
- Accessing a class attribute -> Employee.EmpCount

OOP-Example

```
class Employee: #class with no base
   'Common base class for all employees'
   EmpCount = 0 #class variable

   def __init__(self, name, salary): #constructor
      "constructor for employee class"
      self.name = name #instance variable
      self.salary = salary #instance variable
      Employee.empCount += 1

   def __del__(self): #destructor
      self.__class__.empCount -= 1
      print  self.__class__.__name__, "destroyed", "empCount = ",
self.__class__.empCount

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

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

OOP-Example

```
e1 = Employee("Foo", 1000)  #Instantiation:call to constructor
e2 = Employee("Bar", 2000)  #Instantiation:call to constructor
e1.displayEmployee()        #Name :  Foo , Salary:  1000
e2.displayEmployee()        #Name :  Bar , Salary:  2000
e1.displayCount()           #Total Employee 2
e2.displayCount()           #Total Employee 2
#special attributes on class
print "Employee.__doc__:", Employee.__doc__  #Employee.__doc__: Common base class for all employee
print "Employee.__name__:", Employee.__name__      #Employee.__name__: Employee
print "Employee.__module__:", Employee.__module__  #Employee.__module__: __main__
print "Employee.__bases__:", Employee.__bases__    #Employee.__bases__: ()
print "Employee.__dict__:", Employee.__dict__
```

```
# Employee.__dict__: {'__module__': '__main__', 'displayCount': <function displayCount at 0x7fe26f8c3758>,
'empCount': 2, 'displayEmployee': <function displayEmployee at 0x7fe26f8c37d0>, '__doc__': 'Common base class
for all employees', '__init__': <function __init__ at 0x7fe26f8c36e0>}
```

```
#special attributes on instances
print "e1.__doc__:", e1.__doc__                          #e1.__doc__: Common base class for all employees
print "e1.__class__.__name__:", e1.__class__.__name__    #e1.__class__.__name__: Employee
print "e1.__module__:", e1.__module__                    #e1__module__: __main__
print "e1.__class__.__bases__:", e1.__class__.__bases__  #e1.__class__.__bases__: ()
print "e1.__dict__:", e1.__dict__                        #e1.__dict__: {'salary': 1000, 'name': 'Foo'}
```

Instance Variable Rules

- On use via instance (self.x), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (self.x = ... or self.x +=... etc.):
  - always makes an instance variable
  - can not be used to modify class variables
- Class variables "default" for instance variables
- But...!
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own

Module

If code is being imported __name__ is set to module name denotated with filename. This makes sure that module test code is not run when module is imported. e.g. if you call *python module_test.py* which has only one line.

```
import OOP_module
```

If code is directly run using python interpreter then __name__ will be set to '__main__'. e.g if you call *python OOP_module.py*

```
if __name__ == '__main__':
    e1 = Employee("Foo", 1000)  #Instantiation:call to constructor
    e2 = Employee("Bar", 2000)   #Instantiation:call to constructor
    e1.displayEmployee()          #Name :  Foo , Salary:  1000
    e2.displayEmployee()          #Name :  Bar , Salary:  2000
    e1.displayCount()             #Total Employee 2
    e2.displayCount()             #Total Employee 2
    print "Employee.__doc__:", Employee.__doc__  #Employee.__doc__: Common base class for all employees
    print "Employee.__name__:", Employee.__name__      #Employee.__name__: Employee
    print "Employee.__module__:", Employee.__module__ #Employee.__module__: __main__
    print "Employee.__bases__:", Employee.__bases__     #Employee.__bases__: ()
    print "Employee.__dict__:", Employee.__dict__
```

Static methods

- Static methods often work with class attributes
- showCount() is static method
- Doesn't have self in parameter list because method will be invoked through class not object
- staticmethod()is a built-in Python function which takes method and returns static method

```
class Puppy(object):
    count = 0
    def __init__(self, breed, name):
        self.breed = breed
        self.name = name
        Puppy.count +=1

    def showCount():
        print "total count=",Puppy.count

    showCount =
staticmethod(showCount)
```

```
p1 = Puppy('beagle', 'Tabby')
Puppy.showCount() #total count= 1
p2= Puppy('bulldog', 'Hary')
Puppy.showCount() #total count= 2
p3= Puppy('Golden Retriever', 'Sam')
Puppy.showCount() #total count= 3

# assignment creates a new variable count in the
instances,class variable can not be modified
from instance
p1.count += 1
p2.count += 2
p3.count += 3
#3 4 5 6
print Puppy.count, p1.count, p2.count, p3.count
```

Inheritance(extending the parent)

```
class MyRectangle(Shape):
    'rectangle class'
    def __init__(self, pt1, pt2, color):
        'constructor for rectangle'
        self.pt1 = pt1
        self.pt2 = pt2
        Shape.__init__(self, 'rectangle', 'this is a class for creating and
drawing rectangles', color)
```

Inheritance (code reuse)

```
class Shape:
   'Common base class for all shapes'
   shapeCount = 0
   win =  GraphWin("Shapes display", 400, 400)

   def __init__(self, name, description, color):
      "constructor for shape class"
      self.name = name
      self.description = description
      self.color = color
      Shape.shapeCount += 1

   def displayShape(self):
      print "Name : ", self.name,  ", Description: ",
self.description

   def displayCount(self):
      print "Number of shapes on window :
",Shape.shapeCount

   def draw(self): pass
```

```
class MyRectangle(Shape):
    'rectangle class'
    def __init__(self, pt1, pt2, color):
        'constructor for rectangle'
        self.pt1 = pt1
        self.pt2 = pt2
        Shape.__init__(self, 'rectangle', 'this is
a class for creating and drawing rectangles',
color)

    def draw(self):
       o = Rectangle(self.pt1, self.pt2)
       o.setFill(self.color)
       o.draw(self.win)
       self.win.getMouse()
```

MyRectangle inherits class variables shapeCount and win and class methods displayShape and displayCount, overrides __init__ and draw

Inheritance(Polymorphism)

Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms.

```
# creating a list of shape objects to be rendered

lshapes = []
lshapes.append(MyCircle(Point(50,50), 50, 'blue'))
lshapes.append(MyLine(Point(100,100), Point(200,150),
'green'))
lshapes.append(MyOval(Point(150,150), Point(350,250), 'red'))
lshapes.append(MyRectangle(Point(10,200), Point(110,300),
'yellow'))

for shape in lshapes:
   shape.draw()              #overridden method
   shape.displayShape()  #inherited method
```

Polymorphism In Python

Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms.

```
#include <iostream>
using namespace std;
void f(int x, int y ) {
    cout << "values: " << x << ", " << x << endl;
}
void f(int x, double y ) {
    cout << "values: " << x << ", " << x << endl;
}
void f(double x, int y ) {
    cout << "values: " << x << ", " << x << endl;
}
void f(double x, double y ) {
    cout << "values: " << x << ", " << x << endl;
}
int main()
{
    f(42, 43);
    f(42, 43.7);
    f(42.3,43);
    f(42.0, 43.9);
}
```

Python is implicitly polymorphic

```
def f(x, y):
    print("values: ", x, y)

f(42, 43)
f(42, 43.7)
f(42.3, 43)
F(42.0, 43.9)
# and it supports other types ...
f([3,5,6],(3,5))
f("A String", ("A tuple", "with
Strings"))
f({2,3,9}, {"a":3.4,"b":7.8,
"c":9.04})
```

Encapsulation(data hiding)

In Python anything with two leading underscores is private

```
class Shape:
   'Common base class for all shapes'
   __shapeCount = 0  #1
   win =  GraphWin("Shapes display", 400, 400)

   def __init__(self, name, description, color):
      "constructor for shape class"
      self.name = name
      self.description = description
      self.color = color
      Shape.__shapeCount += 1 #2


   def displayCount(self):
      print "Number of shapes on window : ",Shape.__shapeCount #3
```

```
#private members can not be accessed directly
a= MyCircle(Point(50,50), 50, 'blue')

#AttributeError: MyCircle instance has no attribute '__shapeCount'
print "accesing shapeCount directly :", a.__shapeCount #4
```

Encapsulation(private members)

- No real support, but textual replacement (name mangling)

  `__var` is replaced by `_classname__var`
- prevents only accidental modification, not true protection

```
myobj= MyCircle(Point(50,50), 50, 'blue')

#accesing shapeCount indirectly : 5
print "accesing shapeCount indirectly :",myobj._Shape__shapeCount

#AttributeError: MyCircle instance has no attribute '__shapeCount'
print "accesing shapeCount directly :",myobj.__shapeCount
```

Encapsulation(data hiding)

- Data Protection:By defining a specific interface you can keep other modules from doing accidentally corrupt your data

- Implementation Independence:By limiting the functionality to supported interface, you leave yourself free to change the internal data without messing up your users

- Modularity and code maintainability: Makes code more modular and easy to maintain and develop, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions.

Destroying Objects: Reference Count

- When an object is created, its reference count is set to 1.

- Garbage Collector runs periodically and frees memory when object reference count reaches zero.

- An object reference count goes up
  - when it is created e.g a = 2
  - assigned a new name e.g b = a
  - placed in a container (list, tuple, dictionary) e.g. c = [1,2,b]

- An object reference count goes down
  - deleted with del e.g. del a
  - reference is reassigned e.g. b = 10, c[2] = 20
  - reference goes out of scope (local variables)

Destroying Objects: Reference Count

- When reference count goes down to 0, object is destroyed and its destructor __del__is called.
- Class can need a destructor __del__ to clean up nonmemory resources used by instance. Add this method to the class

```
def __del__(self):
    class_name = self.__class__.__name__
    #free up the resources
    . . .
    print class_name, "destroyed"
```

- When done with objects, call del on them which will reduce the reference count on object. But destructor __del__ will only be called when reference count goes down to 0.
- Use sys.getrefcount(obj) to check the reference count on obj

Destructor: resource allocation and deallocation

```
class Shape(object):
    'Common base class for all shapes'
    shapeCount = 0  #1
    win =  None

    def __init__(self, name, description, color):
        "constructor for shape class"
        self.name = name
        self.description = description
        self.color = color
        print '------ shape->constructor'
        Shape.shapeCount += 1
        if (Shape.shapeCount == 1):
            Shape.win =  GraphWin("Shapes display", 500, 500)

    def __del__(self):
        "destructor for shape class"
        print "------ shape->Destructor:", str(self)
        Shape.shapeCount -= 1
        if (Shape.shapeCount == 0):
          print 'reference count has reached zero: closing window'
          Shape.win.close()
```

super: extending __init__ and __del__

A built-in function called “super” returns a proxy object to delegate method calls to a class – which can be either parent or sibling in nature.

```
class MyCircle(Shape):
    'circle class'
    def __init__(self, center, radius, color):
        self.center = center
        self.radius = radius
        super(MyCircle, self).__init__('circle', 'this is a class for creating
and drawing circles',color)
        print Shape.shapeCount,' circle->constructor:', str(self)

    def __del__(self):
        print  Shape.shapeCount,' circle->destructor'
        super(MyCircle, self).__del__()
```

Printing an Object

`__str__` is a special method that returns string representation of object

```
class MyCircle(Shape):
    'circle class'

    def __str__(self):
        ‘overloading str operator’
        return "Name:{0} center={1} radius={2}".format(self.name,
self.center, self.radius)

...
```

Operator Overloading:(str, +, -)

```
class MyCircle(Shape):
    'circle class'

    def __str__(self):
        ‘overloading str operator’
        return "Name:{0} center={1} radius={2}".format(self.name, self.center, self.radius)

    def __add__(self, obj):
        ‘overloading + operator’
        new_pt = Point(self.center.x + obj.center.x, self.center.y + obj.center.y)
        new_rad = self.radius + obj.radius
        new_obj = MyCircle(new_pt, new_rad, 'green')
        print str(new_obj)
        return new_obj

    def __sub__(self, obj):
        ‘overloading - operator’
        new_pt = Point(self.center.x - obj.center.x, self.center.y - obj.center.y)
        new_rad = self.radius - obj.radius
        if new_rad < 0:
            new_rad = 0 - new_rad

        new_obj = MyCircle(new_pt, new_rad, 'yellow')
        print str(new_obj)
        return new_obj
```

Operator Overloading:(+, -)

```
a = MyCircle(Point(50,50), 50, 'blue')
a.draw()

b = MyCircle(Point(150,150), 100, 'red')
b.draw()

#adding two circles
c = a + b
c.draw()

#subtracting two circles
d = b - a
d.draw()
```

Function Overloading: (methods with common name but with different signature)

- ***NOT** possible to have 2 functions with the same name but different signatures in python without overriding*

- can achieve almost the same effect using optional parameters

```
class MyCircle(Shape):
    'circle class'

    def __init__(self, center, radius, name='circle', description='this is a class for creating and drawing circles'):
        self.center = center
        self.radius = radius
        super(MyCircle, self).__init__(name, description)
        print Shape.shapeCount,' circle->constructor:'

    def draw(self, fillColor='brown', borderColor='gray', borderWidth = 5):
        o = Circle(self.center, self.radius)
        o.setFill(fillColor)
        o.setOutline(borderColor)
        o.setWidth(borderWidth)
        o.draw(self.win)
        super(MyCircle, self).win.getMouse()

    def __del__(self):
        print  Shape.shapeCount,' circle->destructor'
        super(MyCircle, self).__del__()
```

Function Overloading: (methods with common name but with different signature)

Use optional arguments in functions to achieve function overloading to some extent.

```
# MyCircle is called with 4 arguments
a = MyCircle(Point(50,50), 50, "circle-50-50-50", "circle created using optional arguments")

# draw is called with 3 arguments
a.draw('yellow', 'purple', 15)

# MyCircle is called with 3 arguments
b = MyCircle(Point(150,150), 50, 'circle-150-150-100')

# draw is called with 2 arguments
b.draw('green', 'pink')

# MyCircle is called with 2 arguments
c = MyCircle(Point(200,200), 50)

# draw is called with 1 arguments
c.draw('blue')

# MyCircle is called with 2 arguments
d = MyCircle(Point(250,250), 50)

# draw is called with 0 arguments
d.draw()
```

2 More Special Methods: Classes that look like arrays

```
class Puppy(object):
    def __init__(self, breed):
        self.breed = breed
        self.name = []
        self.color = []

    def __getitem__(self, name):
        if name in self.name:
            return self.color[self.name.index(name)]
        else:
            return None

    def __setitem__(self, name, color):
        self.name.append(name)
        self.color.append(color)
```

```
p = Puppy('beagle')
p['Tabby'] = 'cream'
p['Toeffee'] = 'black'
p['Sweetie'] = 'red'

#beagle :Tabby is cream
print p.breed, ':Tabby is', p['Tabby']
```

1 More Special Method: Classes that look like functions

```
class Puppy(object):
    def __init__(self, breed, name, color):
        self.breed = breed
        self.name = name
        self.color = color

    def __call__(self, mood):
        if mood == 0:
            print "Happy - Bhoo"
        elif mood == 1:
            print "Request - Iyoo"
        elif mood == 2:
            print "Pain - Kquee"
        else:
            print "Neutral - silent"
```

```
p = Puppy('beagle', 'tabby', 'brown')

p(0) #Happy - Bhoo

p(1) #Request - Iyoo

p(2) #Pain - Kquee

p(3) #Neutral - silent
```

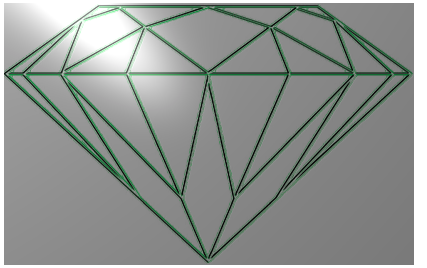
Empty Class

- You could use a dict but you like the look of dotted attributes in a class object. It is usually a question of aesthetics.
- Good for short term convenience only, not a good practice for long term

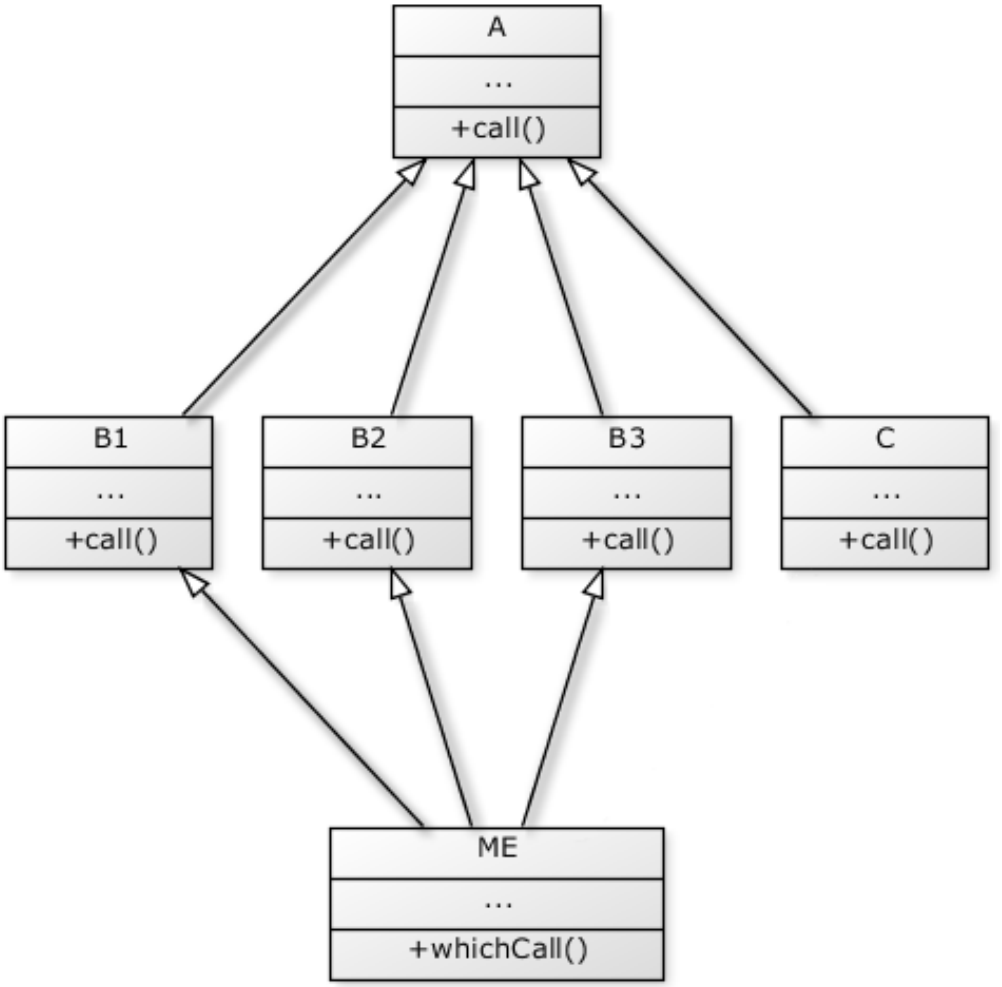
```
class Employee: pass

john = Employee()
john.name = 'John Doe'
john.dept = 'CS'
john.salary = 1000

print john.__dict_        #{'salary': 1000, 'dept': 'CS', 'name': 'John Doe'}
```

```
Employee=type('Employee',(),{})
```

Multiple Inheritence



problem: Diamond: class derived from two classes with a common base class. The ordering of base classes defines the order of search.




```
class ME(B1, B2, B3):
    def whichCall(self):
        print self.call()
```

If call is not implemented in B1, which one will be looked up next, A or B2?

Multiple Inheritence(diamond)

- Python attacks this tiny little problem with what's called the Method Resolution Order (MRO).

- When a class inherits from multiple parents, Python build a list of classes to search for when it needs to resolve which method has to be called when one in invoked by an instance.

Method Resolution Order

Assuming everything descends from object (you are on your own if it doesn't), Python computes a method resolution order (MRO) based on your class inheritance tree. The MRO satisfies 3 properties:

- *Children of a class come before their parents*
- *Left parents come before right parents*
- *A class only appears once in the MRO in preorder*

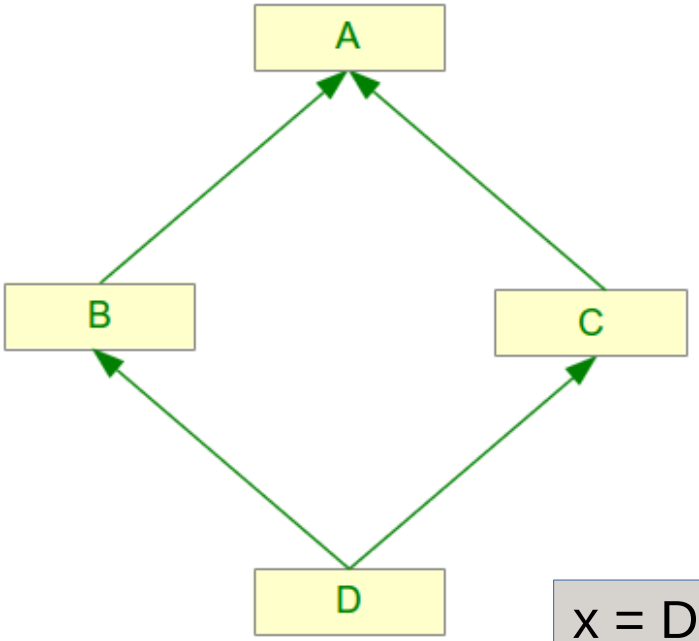
If no such ordering exists, Python errors.

https://stackoverflow.com/questions/3277367/how-does-pythons-super-work-with-multiple-inheritance

Multiple Inheritence(diamond)

In python, you can explicitly call a particular method on (one of) your parent class(es):

class_name.method(self, ...) #self has to be passed explicitly here




```
x = C()
x.m()
m of C called
m of A called

Xx = B()
x.m()
m of B called
m of A called
```

```
x = D()
x.m() #A is called twice
m of D called
m of B called
m of A called
m of C called
m of A called
```

```
class A:
    def m(self):
        print("m of A called")
class B(A):
    def m(self):
        print("m of B called")
        A.m(self)

class C(A):
    def m(self):
        print("m of C called")
        A.m(self)
class D(B,C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)
```

Multiple Inheritence(diamond)

You can also chain the calls using the super() , which will call the "first" parent in MRO order. The exact order is dynamic, but by default it will do left-to-right, depth-first, pre-order (children before parent) scans of your inheritance hierarchy.

```
class A(object):
    def m(self):
        print("m of A called")
class B(A):
    def m(self):
        print("m of B called")
        super(B,self).m()

class C(A):
    def m(self):
        print("m of C called")
        super(C,self).m()

class D(B,C):
    def m(self):
        print("m of D called")
        super(D,self).m()
```

```
#super pythonic way
x = D()
x.m()
m of D called
m of B called
m of C called
m of A called
```

Multiple Inheritence(diamond)

- Do not mix the usage of super and explicit calling.

- Use super when method in each base class has be called exactly once.

- The super function is often used when instances are initialized with the __init__ method

Method Resolution Order(super in leading position)

MRO rule: Left to right, depth first, remove duplicates except last one
In example below MRO is *child, Left, Right, Parent*

```
class Parent(object):
    def __init__(self):
        super(Parent, self).__init__()
        print "parent"
class Left(Parent):
    def __init__(self):
        super(Left, self).__init__()
        print "left"
class Right(Parent):
    def __init__(self):
        super(Right, self).__init__()
        print "right"
class Child(Left, Right):
    def __init__(self):
        super(Child, self).__init__()
        print "child"
```

Child() outputs:
parent
right
left
child

https://stackoverflow.com/questions/3277367/how-does-pythons-super-work-with-multiple-inheritance

Method Resolution Order(super in trailing position)

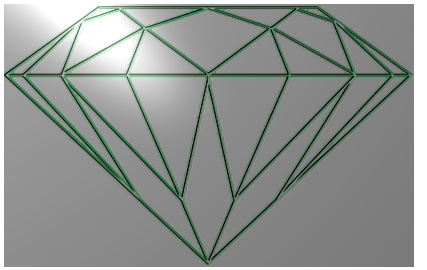
MRO rule: Left to right, depth first, remove duplicates except last one
In example below MRO is *child, Left, Right, Parent*

```
class Parent(object):
    def __init__(self):
        print "parent"
        super(Parent, self).__init__()
class Left(Parent):
    def __init__(self):
        print "left"
        super(Left, self).__init__()
class Right(Parent):
    def __init__(self):
        print "right"
        super(Right, self).__init__()
class Child(Left, Right):
    def __init__(self):
        print "child"
        super(Child, self).__init__()
```

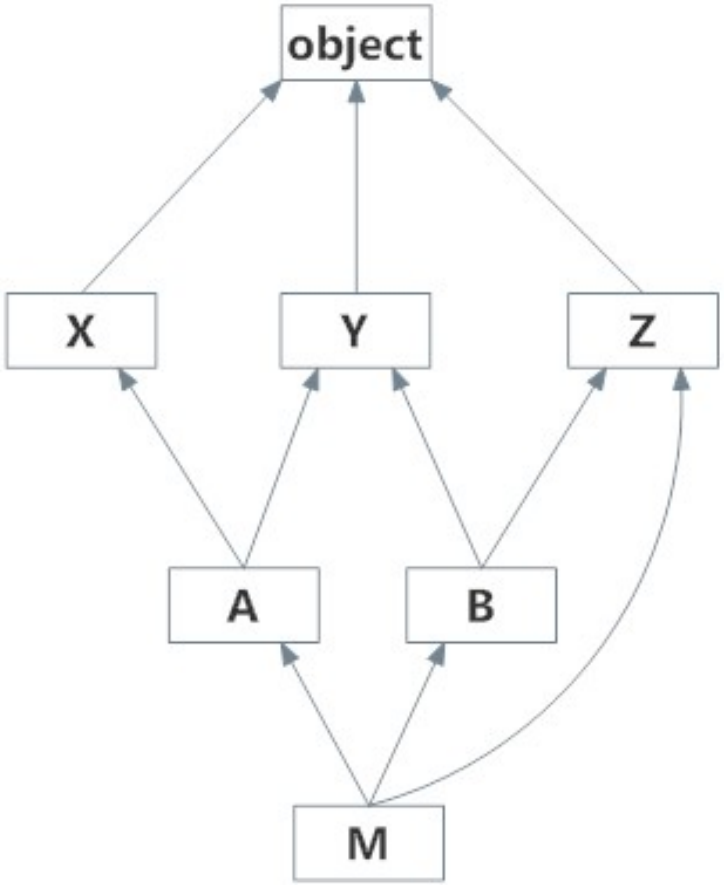
Child() outputs:
child
left
right
parent

https://stackoverflow.com/questions/3277367/how-does-pythons-super-work-with-multiple-inheritance

Multiple Inheritence

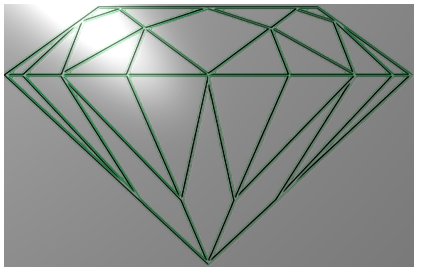


problem: Diamond: class derived from two classes with a common base class. The ordering of base classes defines the order of search.

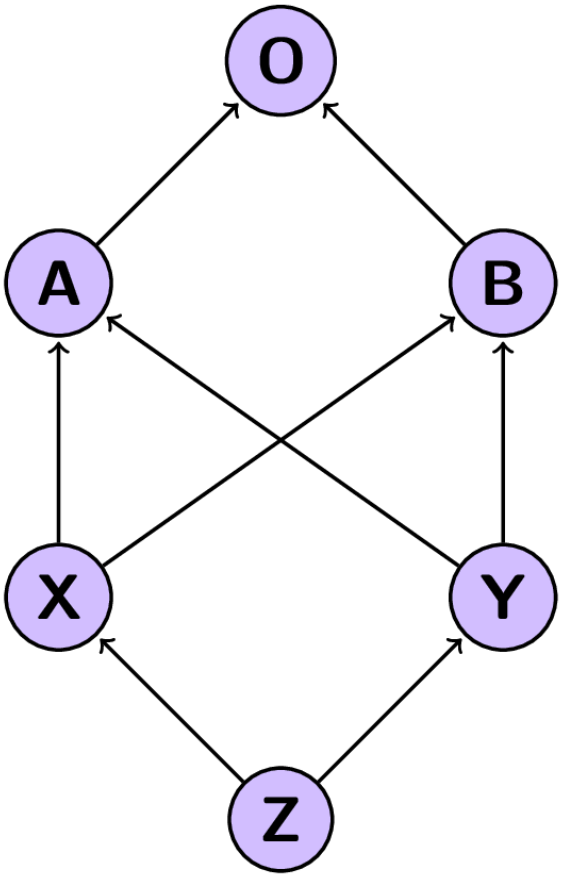



```
class X(O): pass
class Y(O): pass
class Z(O): pass
class A(X,Y): pass
class B(Y,Z): pass
class M(B,A,Z): pass
print "mro=", M.mro() #M B A X Y Z O
class M(A,B,Z): pass
print "mro=", M.mro() #M A X B Y Z O
```

Multiple Inheritence



problem: Diamond: class derived from two classes with a common base class. The ordering of base classes defines the order of search.




```
# cross reference
class A(O): pass
class B(O): pass
class Y(B,A): pass
class X(A,B): pass
class Z(X,Y):pass
print(Z.mro()) #Error
```

TypeError: Error when calling the metaclass bases
Cannot create a consistent method resolution order (MRO) for bases B,A

Summary: OOP

- Object-oriented Programming (OOP) is a methodology of programming where new types of objects are defined
- An object is a single software unit that combines attributes and methods
- An attribute is a “characteristic” of an object; it’s a variable associated with an object (“instance variable”)
- A method is a “behavior” of an object; it’s a function associated with an object
- A class defines the attributes and methods of a kind of object

Summary: (OOP - continued)

- Each instance method must have a special first parameter, called `self` by convention, which provides a way for a method to refer to object itself
- A constructor is a special method that is automatically invoked right after a new object is created
- A class attribute is a single attribute that's associated with a class itself
- A static method is a method that's associated with a class itself