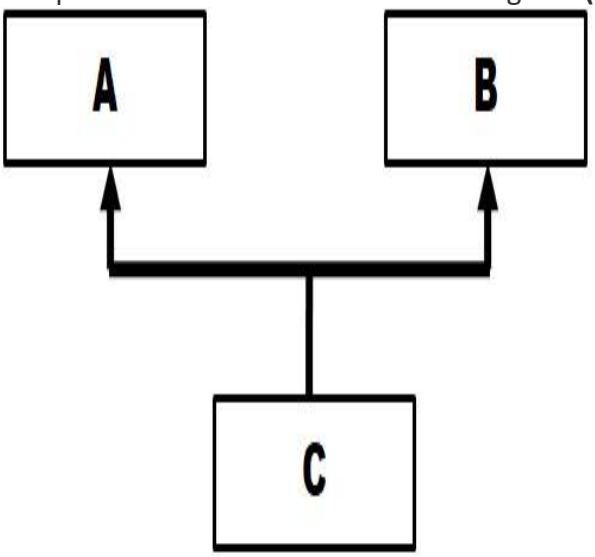
MRO (METHOD RESOLUTION ORDER)

Method Resolution Order (MRO) is the order in which Python looks for a method in a hierarchy of classes. Especially it plays vital role in the context of multiple inheritance as single method may be found in multiple super classes.
To understand the concept of MRO and its need, lets examine a few cases.
We are using Python 3.6.4 in this blog.
Here is my video on **Method Resolution Order(MRO)** in YouTube.

Case 1

This is a simple case where we have class C derived from both A and B. When method process() is called with object of class C then process() method in class A is called.
Python constructs the order in which it will look for a method in the hierarchy of classes. It uses this order, known as MRO, to determine which method it actually calls.
It is possible to see MRO of a class using **mro()** method of the class.




```
class A:
    def process(self):
        print('A process()')


class B:
    pass


class C(A, B):
    pass
```

```
obj = C()
obj.process()
print(C.mro())   # print MRO for class C
```

The above diagram illustrates hierarchy of classes.
When run, the above program displays the following output:

```
A process()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

From MRO of class C, we get to know that Python looks for a method first in class C. Then it goes to A and then to B. So, first it goes to super class given first in the list then second super class, from left to right order. Then finally Object class, which is a super class for all classes.

Case 2

Now, let's make it a little more complicated by adding process() method to class B also.

```
class A:
    def process(self):
        print('A process()')



class B:
    def process(self):
        print('B process()')

class C(A, B):
    pass

obj = C()
obj.process()
```

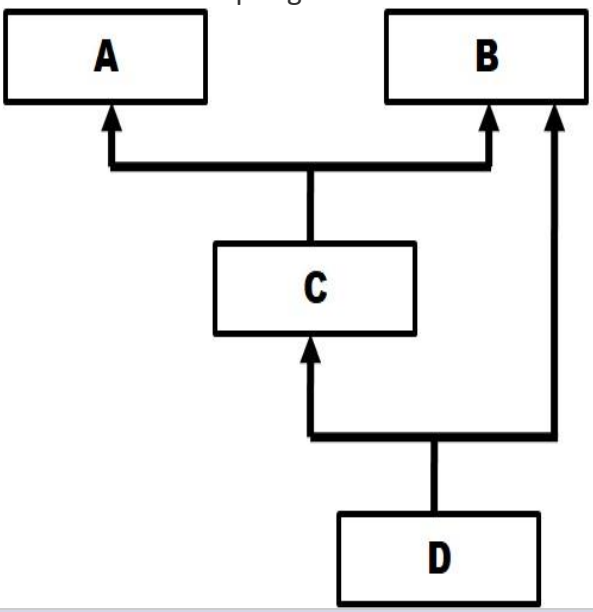
When you run the above code, it prints the following:

```
A process()
```

Python calls process() method in class A. According to MRO, it searches A first and then B. So if method is found in A then it calls that method.
However, if we remove process() method from class A then process() method in class B will be called as it is the next class to be searched according to MRO.
The ambiguity that arises from multiple inheritance is handled by Python using MRO.

Case 3

In this case, we create D from C and B. Classes C and B have process() method and as expected MRO chooses method from C. Remember it goes from left to right. So it searches C first and all its super classes of C and then B and all its super classes. We can observe that in MRO of the output given below.




```
class A:
    def process(self):
        print('A process()')


class B:
    def process(self):
        print('B process()')


class C(A, B):
    def process(self):
        print('C process()')
```

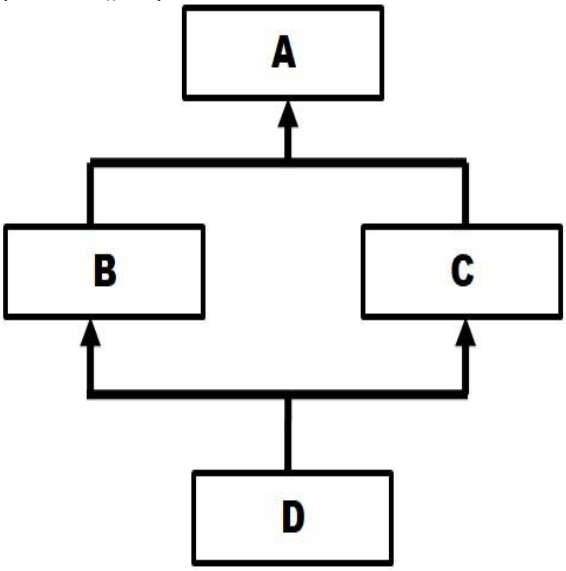
```
class D(C,B):
    pass



obj = D()
obj.process()

print(D.mro())
```

Running the above program will produce the following output:

```
C process()
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <
class '__main__.B'>, <class 'object'>]
```

Case 4

Now, lets change the hierarchy. We create B and C from A and then D from B and C. Method process() is present in both A and C.




```
class A:
    def process(self):
        print('A process()')
```

```
class B(A):
    pass



class C(A):
    def process(self):
        print('C process()')



class D(B,C):
    pass



obj = D()
obj.process()
```

Output of the above program is:

```
C process()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <
class '__main__.A'>, <class 'object'>]
```

When we call process() with an object of class D, it should start with first Super class – B (and its super classes) and then second super class – C (and its super classes). If that is the case then we will call process() method from class A as B doesn't have it and A is super class for B.

However, that is contradictory to rule of inheritance, as most specific version must be taken first and then least specific (generic) version. So, calling process() from A, which is super class of C, is not correct as C is a direct super class of D. That means C is more specific than A. So method must come from C and not from A.

This is where Python applies a simple rule that says (known as good head question) **when in MRO we have a super class before subclass then it must be removed from that position in MRO.**

So the original MRO will be:

```
D -> B -> A -> C -> A
```

If you include object class also in MRO then it will be:

```
D -> B-> A -> object -> C -> A -> object
```

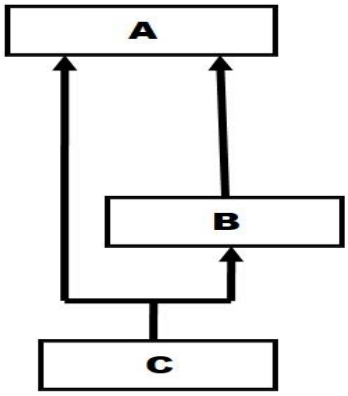
But as A is super class of C, it cannot be before C in MRO. So, Python removes A from that position, which results in new MRO as follows:

```
D -> B -> C -> A -> object
```

The output of the above program proves that.

Case 5

There are cases when Python cannot construct MRO owing to complexity of hierarchy. In such cases it will throw an error as demonstrated by the following code.




```
class A:
    def process(self):
        print('A process()')


class B(A):
    def process(self):
        print('B process()')


class C(A, B):
    pass


obj = C()
obj.process()
```

When you run the above code, the following error is shown:

```
TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

The problem comes from the fact that class A is a super class for both C and B. If you construct MRO then it should be like this:

```
C -> A -> B -> A
```

Then according to the rule (good head) A should NOT be ahead of B as A is super class of B. So new MRO must be like this:

```
C -> B -> A
```

But A is also direct super class of C. So, if a method is in both A and B classes then which version should class C call? According to new MRO, the version in B is called first ahead of A and that is not according to inheritance rules (specific to generic) resulting in Python to throw error.
Understanding MRO is very important for any Python programmer. I strongly recommend trying more cases until you completely understand how Python constructs MRO. Do not confuse yourself by taking old way of constructing MRO used in earlier versions of Python. It is better to consider only Python 3.