*ADDITIONAL TOPICS*

1. **Method Resolution Operator**

The Python Method Resolution Order defines the class search path used by Python to search for the right method to use in classes having multi-inheritance.

class A:

def who\_am\_i(self):

print("I am a A")

class B(A):

def who\_am\_i(self):

print("I am a B")

class C(A):

def who\_am\_i(self):

print("I am a C")

class D(B,C):

def who\_am\_i(self):

print("I am a D")

d1 = D()

d1.who\_am\_i()

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.

This algorithm is a inorder tree routing, and works this way, deep first, from left to right :

1. Look if method exists in instance class

2. If not, looks if it exists in its first parent, then in the parent of the parent and so on

3. If not, it looks if current class inherits from others classes up to the current instance others parents.

So in our example, algorithm search path is : D, B, A, C, A.

A class cannot appears twice in search path, so the final version is D, B, A, C:

* Looking in D
* If not found, looking in B
* If not found, looking un B first parent A
* If not found, going back in B others parents (none)
* If not found, looking in D others parents : C

Quiz.

class A1():

# def who\_am\_i(self):

# print("I am a A1")

pass

class A2():

def who\_am\_i(self):

print("I am a A2")

class A3():

def who\_am\_i(self):

print("I am a A3")

class B(A1, A2):

# def who\_am\_i(self):

# print("I am a B")

pass

class C(A3):

def who\_am\_i(self):

print("I am a C")

class D(B,C):

# def who\_am\_i(self):

# print("I am a D")

pass

d1 = D()

d1.who\_am\_i()

The search path when invoking d1.who\_am\_i() is : D, B, A1, A2, C, A3

1. **Singleton Design Pattern**

**The design pattern that allows you to create only one instance of a class throughout the lifetime of a program is called singleton.**

*A Singleton is a class that makes sure only one instance of it is ever created. Typically, such a class is used to manage resources that by their nature can exist only once*

*Singleton is the best candidate when the requirements are as follows:*

1. *Controlling concurrent access to a shared resource*
2. *If you need a global point of access for the resource from multiple or different parts of the system*
3. *When you need to have only one object*

***Use Cases :***

* *Managing a connection to a database*
* *File manager*
* *Retrieving and storing information on external configuration files*
* *Read-only singletons storing some global states (user language, time, time zone, application path, and so on)*

***Usage Example: Modules Intiliaziation***

* *Check whether a module is already imported. If yes, return it. If not, find a module, initialize it, and return it.*
* *Initializing a module means executing a code, including all module-level assignments.*
* *When you import the module for the first time, all of the initializations will be done; however, if you try to import the module for the second time,* [*Python*](https://www.packtpub.com/tech/Python) *will return the initialized module. Thus, the initialization will not be done, and you get a previously imported module with all of its data.*

*Here, if you try to import a global variable in a singleton module and change its value in the module1 module, module2 will get a changed variable.*

***Implementation***

***In classic singleton in Python, we check whether an instance is already created. If it is created, we return it; otherwise, we create a new instance, assign it to a class attribute, and return it. To create a dedicated singleton class***

class Singleton(object):  
  
 def \_\_new\_\_(cls):  
  
 if not hasattr(cls, 'instance'):  
  
 cls.instance = super(Singleton, cls).\_\_new\_\_(cls)  
  
 return cls.instance

Here, before creating the instance, we check for the special *\_\_new\_\_* method, which is called right before *\_\_init\_\_* if we had created an instance earlier. If not, we create a new instance; otherwise, we return the already created instance.

>>> singleton = Singleton()  
>>> another\_singleton = Singleton()  
>>> singleton is another\_singleton  
**True**  
>>> singleton.only\_one\_var = "I'm only one var"  
>>> another\_singleton.only\_one\_var  
**I'm only one var**