6. OOP in Python – Inheritance and Polymorphism

Inheritance and polymorphism – this is a very important concept in Python. You must understand it better if you want to learn.

Inheritance

One of the major advantages of Object Oriented Programming is re-use. Inheritance is one of the mechanisms to achieve the same. Inheritance allows programmer to create a general or a base class first and then later extend it to more specialized class. It allows programmer to write better code.

Using inheritance you can use or inherit all the data fields and methods available in your base class. Later you can add you own methods and data fields, thus inheritance provides a way to organize code, rather than rewriting it from scratch.

In object-oriented terminology when class X extend class Y, then Y is called super/parent/base class and X is called subclass/child/derived class. One point to note here is that only data fields and method which are not private are accessible by child classes. Private data fields and methods are accessible only inside the class.

Syntax to create a derived class is:

```
class BaseClass:
      Body of base class
class DerivedClass(BaseClass):
      Body of derived class
```

Inheriting Attributes

Now look at the below example:

```
# Inheriting_Attributes

class Date(object):               # Inherits from the 'object' Class
    def get_date(self):
        return '2018-02-02'


class Time(Date):               # Inherits from the 'Date' Class
    def get_time(self):
        return '09:00:00'

dt = Date()
print("Get date from the Date class: ",dt.get_date())

tm = Time()
print("Get time from the Time class: ",tm.get_time())
print('Get date from time class by inheriting or calling Date class method: ',tm.get_date())          # Found this method in the 'Date' class
```

Output

```
Get date from the Date class:  2018-02-02
Get time from the Time class:  09:00:00
Get date from time class by inheriting or calling Date class method:  2018-02-02
```

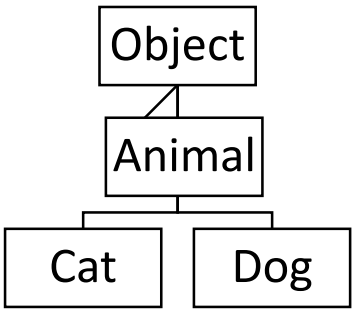
We first created a class called Date and pass the object as an argument, here-object is built-in class provided by Python. Later we created another class called time and called the Date class as an argument. Through this call we get access to all the data and attributes of Date class into the Time class. Because of that when we try to get the get_date method from the Time class object tm we created earlier possible.

Object.Attribute Lookup Hierarchy

- The instance
- The class
- Any class from which this class inherits

Inheritance Examples

Let's take a closure look into the inheritance example:



Let's create couple of classes to participate in examples:

- Animal: Class simulate an animal
- Cat: Subclass of Animal
- Dog: Subclass of Animal

In Python, constructor of class used to create an object (instance), and assign the value for the attributes.

Constructor of subclasses always called to a constructor of parent class to initialize value for the attributes in the parent class, then it start assign value for its attributes.

```
#Inheritance Example
class Animal(object):

    def __init__(self, name):
        self.name = name
    def eat(self, food):          # Common method(or property) of both subclass
        print ('%s is eating %s. ' %(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print('%s goes after the %s!' %(self.name, thing))

class Cat(Animal):

    def swatstring(self):
        print('%s shreds the string!' %(self.name))

d = Dog('Ranger')       # Created Dog object, d
c = Cat('MeOw')         # Created Cat object, c

d.fetch('ball')        # Rover goes after the paper!=
c.swatstring()          # Fluffy shreds the string!
d.eat('Dog Food')       # Rover is eating dog Food
c.eat('Cat Food')       #Fluffy is eating cat food.
d.swatstring()          #Attribute Error: 'Dog' object has no Attribute 'swatstring'
```

Output

```
Ranger goes after the ball!
MeOw shreds the string!
Ranger is eating Dog Food.
MeOw is eating Cat Food.
Traceback (most recent call last):
  File "animal.py", line 27, in <module>
    d.swatstring()          #Attribute Error: 'Dog' object has no Attribute 'swatstring'
AttributeError: 'Dog' object has no attribute 'swatstring'
```

In the above example, we see the command attributes or methods we put in the parent class so that all subclasses or child classes will inherits that property from the parent class.

If a subclass try to inherits methods or data from another subclass then it will through an error as we see when Dog class try to call swatstring() methods from that cat class, it throws an error(like AttributeError in our case).

Polymorphism ("MANY SHAPES")

Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This permits functions to use entities of different types at different times. So, it provides flexibility and loose coupling so that code can be extended and easily maintained over time.

This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.

Polymorphism can be carried out through inheritance, with subclasses making use of base class methods or overriding them.

Let understand the concept of polymorphism with our previous inheritance example and add one common method called show_affection in both subclasses:

From the example we can see, it refers to a design in which object of dissimilar type can be treated in the same manner or more specifically two or more classes with method of the same name or common interface because same method(show_affection in below example) is called with either type of objects.

```
#Polymorphism Example
class Animal(object):

    def __init__(self, name):
        self.name = name
    def eat(self, food):           # Common method(or property) of both subclass
        print ('{0} eats {1}'.format(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print('{0} goes after the {1}!'.format(self.name, thing))

    def show_affection(self):      # Same method is called in Dog class
        print('{0} wags tail'.format(self.name))

class Cat(Animal):

    def swatstring(self):
        print('{0} shreds the string!'.format(self.name))

    def show_affection(self):      # Same method is called in Cat class
        print('{0} purrs'.format(self.name))


for a in (Dog('Rover'), Cat('Fluffy'), Cat('Precious'), Dog('Scout')): a.show_affection()      # Same method is called with different attribute
```

Output

```
Rover wags tail
Fluffy purrs
Precious purrs
Scout wags tail
```

So, all animals show affections (show_affection), but they do differently. The "show_affection" behaviors is thus polymorphic in the sense that it acted differently depending on the animal. So, the abstract "animal" concept does not actually "show_affection", but specific animals(like dogs and cats) have a concrete implementation of the action "show_affection".

Python itself have classes that are polymorphic. Example, the len() function can be used with multiple objects and all return the correct output based on the input parameter.

```
>>> len('hello')   # passing an string to len function
5
>>> len([1, 2, 3]) # passing a list of 3 elements
3
>>> len(('x', 'y', 'z')) # passing a tuple of 3 elements
3
>>> len({'a':9, 'b':18}) # passing a dictionary of 2 pairs
2
>>> # illustrate it further,
>>> dir('hello')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '
__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__
', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace'
, 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartit
ion', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
>>> dir([1,2,3])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get
attribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__
len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__
', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop'
, 'remove', 'reverse', 'sort']
>>> dir({'a':9})
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__
', '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new_
_', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'c
opy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
>>>
```

Overriding

In Python, when a subclass contains a method that overrides a method of the superclass, you can also call the superclass method by calling
Super(Subclass, self).method instead of self.method.

Example

```
class Thought(object):

    def __init__(self):

        pass

    def message(self):

        print("Thought, always come and go")


class Advice(Thought):

    def __init__(self):

        super(Advice, self).__init__()

    def message(self):

        print('Warning: Risk is always involved when you are dealing with
market!')
```

Inheriting the Constructor

If we see from our previous inheritance example, __init__ was located in the parent class in the up 'cause the child class dog or cat didn't've __init__ method in it. Python used the inheritance attribute lookup to find __init__ in animal class. When we created the child class, first it will look the __init__ method in the dog class, then it didn't find it then looked into parent class Animal and found there and called that there. So as our class design became complex we may wish to initialize a instance firstly processing it through parent class constructor and then through child class constructor.

```
import random

class Animal(object):

    def __init__(self, name):
        self.name =name

class Dog(Animal):

    def __init__(self, name):
        super(Dog, self).__init__(name)
        self.breed = random.choice(['Doberman', 'German shepherd', 'Beagle'])

    def fetch(self, thing):
        print('%s goes after the %s!' %(self.name, thing))

d = Dog('dogname')

print(d.name)
print(d.breed)
```

Output

```
dogname
German shepherd
```

In above example- all animals have a name and all dogs a particular breed. We called parent class constructor with super. So dog has its own __init__ but the first thing that happen is we call super. Super is built in function and it is designed to relate a class to its super class or its parent class.

In this case we saying that get the super class of dog and pass the dog instance to whatever method we say here the constructor __init__. So in another words we are calling parent class Animal __init__ with the dog object. You may ask why we won't just say Animal __init__ with the dog instance, we could do this but if the name of animal class were to change, sometime in the future. What if we wanna rearrange the class hierarchy,

so the dog inherited from another class. Using super in this case allows us to keep things modular and easy to change and maintain.

So in this example we are able to combine general __init__ functionality with more specific functionality. This gives us opportunity to separate common functionality from the specific functionality which can eliminate code duplication and relate class to one another in a way that reflects the system overall design.

Conclusion

- __init__ is like any other method; it can be inherited
- If a class does not have a __init__ constructor, Python will check its parent class to see if it can find one.
- As soon as it finds one, Python calls it and stops looking
- We can use the super () function to call methods in the parent class.
- We may want to initialize in the parent as well as our own class.

Multiple Inheritance and the Lookup Tree

As its name indicates, multiple inheritance is Python is when a class inherits from multiple classes.
For example, a child inherits personality traits from both parents (Mother and Father).

Python Multiple Inheritance Syntax

To make a class inherits from multiple parents classes, we write the the names of these classes inside the parentheses to the derived class while defining it. We separate these names with comma.
Below is an example of that:

```
>>> class Mother:
      pass

>>> class Father:
     pass

>>> class Child(Mother, Father):
     pass

>>> issubclass(Child, Mother) and issubclass(Child, Father)
True
```

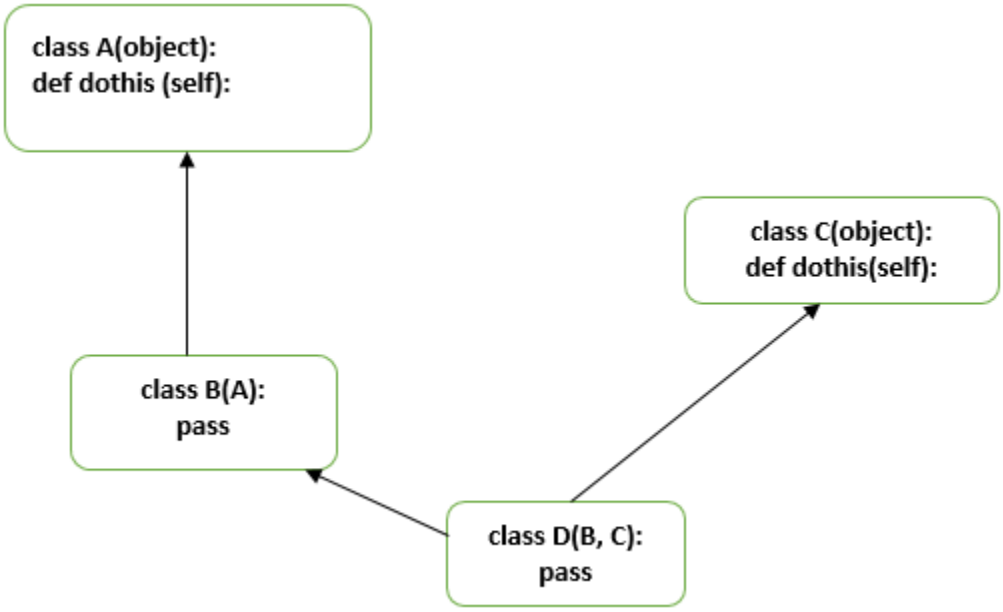
Multiple inheritance refers to the ability of inheriting from two or more than two class. The complexity arises as child inherits from parent and parents inherits from the grandparent class. Python climbs an inheriting tree looking for attributes that is being requested to be read from an object. It will check the in the instance, within class then

parent class and lastly from the grandparent class. Now the question arises in what order the classes will be searched - breath-first or depth-first. By default, Python goes with the depth-first.

That's is why in the below diagram the Python searches the dothis() method first in class A. So the method resolution order in the below example will be

Mro- D->B->A->C

Look at the below multiple inheritance diagram:



Let’s go through an example to understand the “mro” feature of an Python.

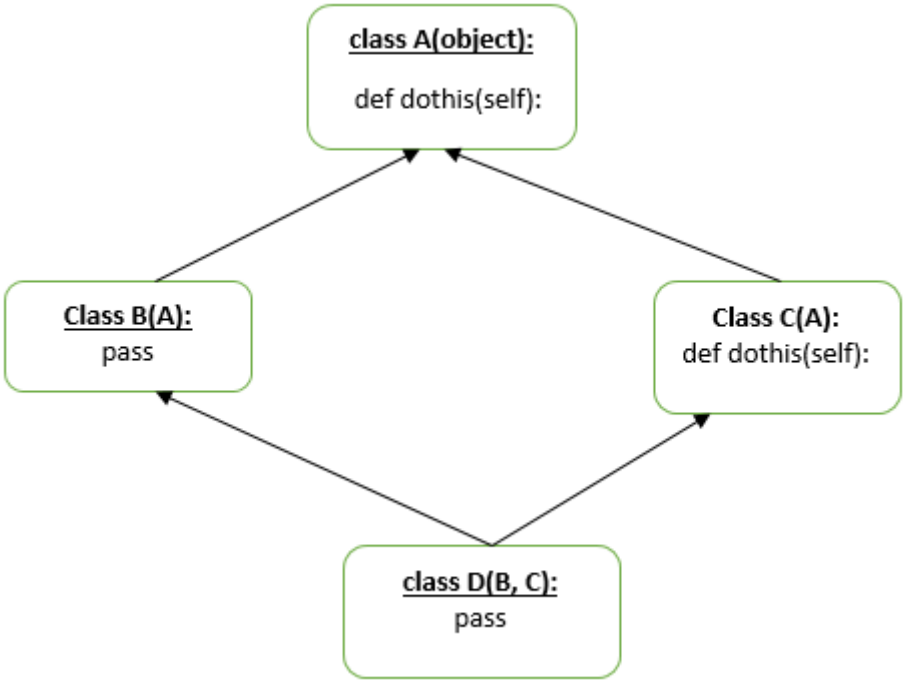
```
class A(object):
    def dothis(self):
        print('doing this in A')

class B(A):
    pass

class C(object):
    def dothis(self):
        print('do this in C')

class D(B,C):
    pass

d_instance = D()

d_instance.dothis()

print(D.mro())
```

Output

```
doing this in A
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>]
```

Example 3

Let's take another example of "diamond shape" multiple inheritance.



Above diagram will be considered ambiguous. From our previous example understanding "method resolution order" .i.e. mro will be D->B->A->C->A but it's not. On getting the second A from the C, Python will ignore the previous A. so the mro will be in this case will be D->B->C->A.

Let's create an example based on above diagram:

```
class A(object):
    def dothis(self):
        print('doing this in A')

class B(A):
    pass

class C(A):
    def dothis(self):
        print('do this in C')

class D(B,C):
    pass

d_instance = D()
d_instance.dothis()

print(D.mro())
```

Output

```
do this in C
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Simple rule to understand the above output is- if the same class appear in the method resolution order, the earlier appearances of this class will be remove from the method resolution order.

In conclusion:

- Any class can inherit from multiple classes
- Python normally uses a "depth-first" order when searching inheriting classes.
- But when two classes inherit from the same class, Python eliminates the first appearances of that class from the mro.

Decorators, Static and Class Methods

Functions(or methods) are created by def statement.

Though methods works in exactly the same way as a function except one point where method first argument is instance object.

We can classify methods based on how they behave, like

- **Simple method**: defined outside of a class. This function can access class attributes by feeding instance argument:

```
def outside_func(():
```

- **Instance method**:

```
def func(self,)
```

- **Class method**: if we need to use class attributes

```
                @classmethod
          def cfunc(cls,)
```

- **Static method**: do not have any info about the class

```
          @staticmethod
          def sfoo()
```

Till now we have seen the instance method, now is the time to get some insight into the other two methods,

Class Method

The @classmethod decorator, is a builtin function decorator that gets passed the class it was called on or the class of the instance it was called on as first argument. The result of that evaluation shadows your function definition.

Syntax

```
class C(object):
    @classmethod
    def fun(cls, arg1, arg2, ...):
        ....
fun: function that needs to be converted into a class method
returns: a class method for function
```

They have the access to this cls argument, it can't modify object instance state. That would require access to self.

- It is bound to the class and not the object of the class.
- Class methods can still modify class state that applies across all instances of the class.

Static Method

A static method takes neither a self nor a cls(class) parameter but it's free to accept an arbitrary number of other parameters.

Syntax

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...

returns: a static method for function funself.
```

- A static method can neither modify object state nor class state.
- They are restricted in what data they can access.

When to use what

- We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.