28/04/2021

Oops concepts:

class:
A class is a blueprint of an object

object:
An object is also called an instance of a class and the process of creating this object is called instantiation

self:
Whenever an object calls its method, the object itself passed as the first argument
The first argument of the function in the class must be object itself

__init__
The special function gets called whenever a new object of the class is instantiated

__str__ and __repr__
str() and repr() both are used to get a string representation of an object.

Encapsulation

This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as prefix i.e single " _ " or double " __".

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price
c = Computer()
c.sell()
# change the price
c.__maxprice = 1000
c.sell()
# using setter function
c.setMaxPrice(1000)
c.sell()
```

Output:

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

```
class Parrot:
    def fly(self):
        print("Parrot can fly")
    def swim(self):
        print("Parrot can't swim")
class Penguin:
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")
# common interface
def flying_test(bird):
    bird.fly()
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

Output:

```
Parrot can fly
Penguin can't fly
```

Operator overloading:

```
class student:
  def __init__(self, m1, m2):
    self.m1 = m1
    self.m2 = m2
  def __add__(self, other):
    m1 = self.m1+other.m1
    m2 = self.m2+other.m2
    s3 = student(m1,m2)
    return s3
  def __gt__(self, other):
    r1 = self.m1+self.m2
    r2 = other.m1+other.m2
    if r1>r2:
      return True
    else:
      return False
  def __str__(self):
    return '{} {}'.format(self.m1, self.m2)
s1 =student(58,59)
s2 =student(60,65)

s3 = s1+s2 #student.__add__(s1,s2)
print(s3.m2)
if s1>s2:
  print('s1 wins')
else:
  print('s2 wins')
a=9
print(a.__str__())
print(s1)
```

Output:

```
124
s2 wins
9
58 59
```

Note: Method overloading is not supported in python

Method overriding:

```
class A:
  def show(self):
     print("In A show")
class B(A):
  def show(self):
     print("in B show")

a1 = B()
a1.show()
```

Output:

```
in  B show
```

Inner Class:

```
class Student:
   def __init__(self, name, rollno):
       self.name = name
       self.rollno = rollno
       self.lap = self.Laptop()
   def show(self):
       print(self.name, self.rollno)
       self.lap.show()
   class Laptop:
       def __init__(self):
           self.brand = 'HP'
           self.cpu = 'i5'
           self.ram = 8
       def show(self):
           print(self.brand, self.cpu, self.ram)
s1 = Student('Navin', 22)
s2 = Student('Jenny', 3)
s1.show()
```

Output:

```
Navin 22
HP i5 8
```

Python by default doesn’t support abstract classes
To achieve abstract class we need to abc module (Abstract base classes)
You can have multiple abstract methods

super()

At a high level super() gives you access to methods in a superclass from the subclass that inherits from it. A common use case is building classes that extend the functionality of previously built classes.

Instance vs. Static vs. Class Methods in Python: The Important Differences

```
class MyClass:
  def method(self):
      return 'instance method called', self

  @classmethod
  def classmethod(cls):
      return 'class method called', cls

  @staticmethod
  def staticmethod():
      return 'static method called'
```

Python methods can often be confusing once you get into object orientated programming (OOP). This guide covers the three main types of methods in Python.

There are three types of methods in Python:

1. instance methods,
2. static methods, and
3. class methods.

Before looking at the differences, it's important to understand a design pattern known as the decorator pattern, or simply called a decorator.

Decorators sound complex, but there's nothing to fear. Decorators are simply functions. You can write them yourself, or use those included in libraries, or the Python standard library.

Like any function, decorators perform a task. The difference here is that decorators apply logic or change the behavior of other functions. They are an excellent way to reuse code, and can help to separate logic into individual concerns.

The decorator pattern is Python's preferred way of defining static or class methods. Here's what one looks like in Python:

```
class DecoratorExample:
  """ Example Class """
  def __init__(self):
    """ Example Setup """
    print('Hello, World!')

  @staticmethod
  def example_function():
    """ This method is decorated! """
    print('I\'m a decorated function!')

de = DecoratorExample()
de.example_function()
```

Output:

```
Hello, World!
I'm a decorated function!
```

Decorators have to immediately precede a function or class declaration.
They start with the @ sign, and unlike normal methods, you don't have to put parentheses on the end unless you are passing in arguments.
It's possible to combine multiple decorators, write your own, and apply them to classes as well, but you won't need to do any of that for these examples.

Instance Methods in Python

Instance methods are the most common type of methods in Python classes.
These are so called because they can access unique data of their instance.
If you have two objects each created from a car class, then they each may have different properties.
They may have different colors, engine sizes, seats, and so on.

Instance methods must have self as a parameter, but you don't need to pass this in every time.
Self is another Python special term. Inside any instance method, you can use self to access any data or methods that may reside in your class.
You won't be able to access them without going through self.

Finally, as instance methods are the most common, there's no decorator needed. Any method you create will automatically be created as an instance method, unless you tell Python otherwise.

Here’s an example:

```
class DecoratorExample:
  """ Example Class """
  def __init__(self):
    """ Example Setup """
    print('Hello, World!')
    self.name = 'Decorator_Example'

  def example_function(self):
    """ This method is an instance method! """
    print('I\'m an instance method!')
    print('My name is ' + self.name)

de = DecoratorExample()
de.example_function()
```

Output:

```
Hello, World!
I'm an instance method!
My name is Decorator_Example
```

The name variable is accessed through self. Notice that when example_function is called, you don’t have to pass self in—Python does this for you.

Static Methods in Python:

Static methods are methods that are related to a class in some way, but don’t need to access any class-specific data. You don’t have to use self, and you don’t even need to instantiate an instance, you can simply call your method:

```
class DecoratorExample:
  """ Example Class """
  def __init__(self):
    """ Example Setup """
    print('Hello, World!')

  @staticmethod
  def example_function():
    """ This method is a static method! """
    print('I\'m a static method!')

de = DecoratorExample.example_function()
```

Output:

```
I'm a static method!
```

The @staticmethod decorator was used to tell Python that this method is a static method.

Static methods are great for utility functions, which perform a task in isolation. They don't need to (and cannot) access class data. They should be completely self-contained, and only work with data passed in as arguments. You may use a static method to add two numbers together, or print a given string.

Class Methods in Python:

Class methods are the third and final OOP method type to know. Class methods know about their class. They can't access specific instance data, but they can call other static methods.

Class methods don't need self as an argument, but they do need a parameter called cls. This stands for class, and like self, gets automatically passed in by Python.

Class methods are created using the @classmethod decorator.

```
class DecoratorExample:
  """ Example Class """
  def __init__(self):
    """ Example Setup """
    print('Hello, World!')

  @classmethod
  def example_function(cls):
    """ This method is a class method! """
    print('I\'m a class method!')
    cls.some_other_function()

  @staticmethod
    def some_other_function():
    print('Hello!')

de = DecoratorExample()
de.example_function()
```

Output:

```
Hello, World!
I'm a class method!
Hello!
```

Class methods are possibly the more confusing method types of the three, but they do have their uses. Class methods can manipulate the class itself, which is useful when you're working on larger, more complex projects.

Example:

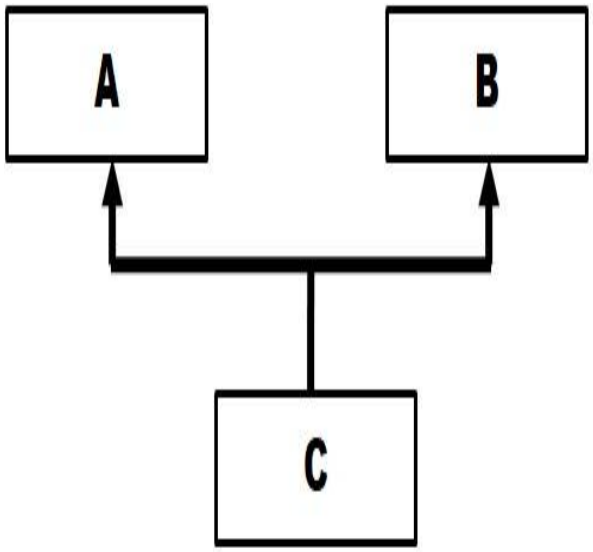
```
# Python program to demonstrate
# use of class method and static method.
from datetime import date
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
    # a class method to create a Person object by birth year.
  @classmethod
  def fromBirthYear(cls, name, year):
    return cls(name, date.today().year - year)
    # a static method to check if a Person is adult or not.
  @staticmethod
  def isAdult(age):
    return age > 18

person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)
print(person1.age)
print(person2.age)
# print the result
print(Person.isAdult(22))
```

Output:

```
21
21
True
```

MRO:

Case 1
This is a simple case where we have class C derived from both A and B. When method process() is called with an 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 the MRO of a class using the 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.

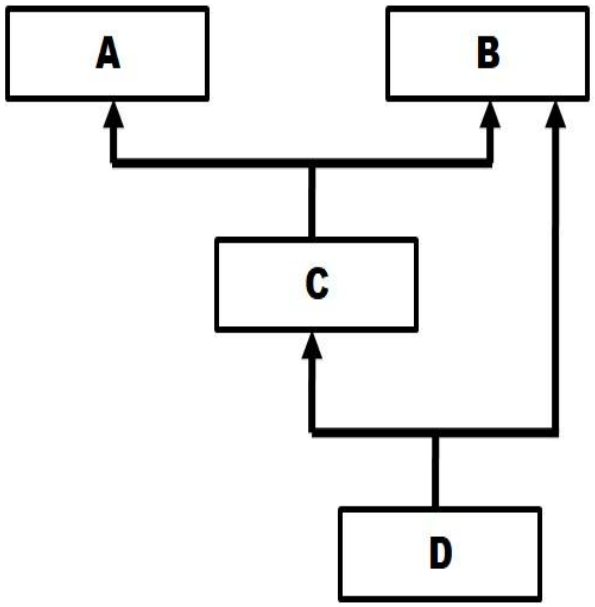
```
class A:
    def process(self):
        print('A process()')

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 the 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 a 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 the MRO of the output given below.



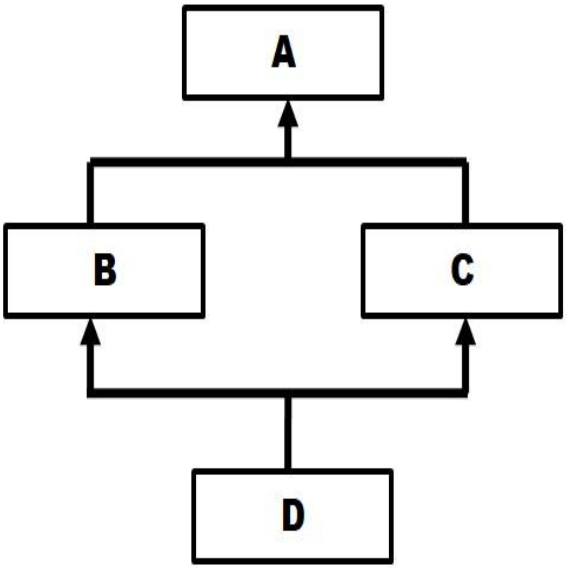

```
class A:
    def process(self):
        print('A process()')


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, let's 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

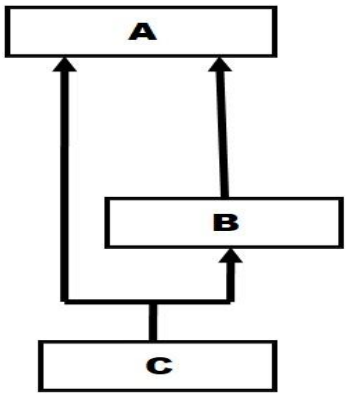
If you include object class also in MRO then it will be:
D -> B-> A -> object -> C -> A -> object

But as A is a 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 a direct superclass of C. So, if a method is in both A and B classes then which version should class C call? According to the 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 an 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 the old way of constructing MRO used in earlier versions of Python. It is better to consider only Python 3.

@property is a built-in decorator for the **property**() function in **Python**. It is used to give "special" functionality to certain methods to make them act as getters, setters, or deleters when we define **properties** in a class

Decorator vs inheritance

1. More flexibility than static inheritance.
2. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance.
3. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them.
4. In contrast, inheritance requires creating a new class for each additional responsibility (e.g., BorderedScrollableTextView, BorderedTextView).
5. This gives rise to many classes and increases the complexity of a system.
6. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities.