classes_objects

February 7, 2020

0.1 namespace :

A namespace is a mapping from names to objects.

Examples of namespaces are: the set of built-in names (containing functions such as abs(), and built-in exception names);

0.2 The simplest form of class definition looks like this

Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below.

Object

```
<li>Class</li>
<li>Method</li>
<li>Inheritance</li>
```

Polymorphism

```
<li>Data Abstraction</li>
<li>Encapsulation</li>
    </ul>
<li>When a class definition is entered, a new namespace is created, and used as the local scop
<li>The object is an entity that has state and behavior. It may be any real-world object like
<li>The method is a function that is associated with an object.</li>
</ul>
```

[21]:
```
class Employee:
    id = 10;
    name = "John"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))

emp = Employee()
emp.display()
```

```
ID: 10
Name: John
```

[22]:
```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

[23]:
```
x = MyClass()
x.f()
```

[23]:
```
'hello world'
```

A constructor is a special kind of method that Python calls when it instantiates an object using the definitions found in your class.

```
<li> Python relies on the constructor to perform tasks such as initializing (assigning values t
```

The name of a constructor is always the same, **init**().

```
<li>The constructor can accept arguments when necessary to create the object. </li>
<li>When you create a class without a constructor, Python automatically creates a default const
<li>Every class must have a constructor, even if it simply relies on the default constructor.</
```

[24]:
```
class MyClass:
    Greeting =''
    def __init__(self, Name="there"):
        self.Greeting = Name + "!"
    def SayHello(self):
        print("Hello {0}".format(self.Greeting))
```

[29]:
```
hi=MyClass("Uttar")

hi.SayHello()
```

```
Hello Uttar!
```

In this case, there are two versions of **init**(). The first doesn't require any special input because it uses the default value for the Name of "there".The second requires a name as an input. It sets Greeting to the value of this name, plus an exclamation mark.

```
</ul>
```

[4]:
```
MyInstance = MyClass()
```

[5]:
```
MyInstance.SayHello()
```

```
Hello there!
```

[7]:
```
MyInstance = MyClass("Amy")
```

[8]:
```
MyInstance.SayHello()
```

```
Hello Amy!
```

[ ]:
```

```

[13]:
```
class Robot(object):
    def __init__(self):
      self.__version = 22

    def getVersion(self):
        print(self.__version)

    def setVersion(self, version):
        self.__version = version

obj = Robot()
obj.getVersion()
obj.setVersion(23)
obj.getVersion()
```

```
22
23
```

[31]:
```
class MyFirstClass:
    pass
```

[33]:
```
a=MyFirstClass()
print(a)
b=MyFirstClass()
print(b)
```

```
<__main__.MyFirstClass object at 0x0000022C708C0D48>
<__main__.MyFirstClass object at 0x0000022C7090F408>
```

[35]:
```
class Point:
    pass
```

[36]:
```
p1 = Point()
p2 = Point()
```

[37]:
```
p1.x = 5
p1.y = 4
```

[38]:
```
p2.x = 3
p2.y = 6
```

[39]:
```
print(p1.x, p1.y)
print(p2.x, p2.y)
```

```
5 4
3 6
```

[2]:
```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0
```

[42]:
```
p = Point()
p.reset()
print(p.x, p.y)
```

```
0 0
```

The self argument to a method is simply a reference to the object that the method is being invoked on. We can access attributes and methods of that object as if it were any another object.

[3]:
```
p = Point()
Point.reset(p)
print(p.x, p.y)
```

```
0 0
```

What happens if we forget to include the self argument in our class definition

[4]:
```
class Point:
    def reset():
        pass
```

[5]:
```
p=Point()
```

[6]:
```
p.reset()
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-6-22b6da648484> in <module>
   ----> 1 p.reset()


       TypeError: reset() takes 0 positional arguments but 1 was given
```

[7]:
```
Point.reset(p)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-7-048a0739cff2> in <module>
----> 1 Point.reset(p)


TypeError: reset() takes 0 positional arguments but 1 was given
```

[13]:
```
import math
class Point:
    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)

    def calculate_distance(self, other_point):
        return math.sqrt(
                (self.x - other_point.x)**2 +
                (self.y - other_point.y)**2)
```

[14]:
```
# how to use it:
point1 = Point()
point2 = Point()
```

[15]:
```
point1.reset()
```

[16]:
```
point2.move(5,0)
```

[17]:
```
print(point2.calculate_distance(point1))
```

```
5.0
```

0.3 Initializing the object

constructor, a special method that creates and initializes the object when it is created. Python is a little different; it has a constructor and an initializer.

The Python initialization method is the same as any other method, except it has a special name, **init**.

The leading and trailing double underscores mean this is a special method that the Python interpreter will treat as a special case.

[52]:
```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

[38]:
```
print(MyClass)
```

```
<class '__main__.MyClass'>
```

[22]:
```
MyClass.i
```

[22]:
```
12345
```

[27]:
```
MyClass.f
```

[27]:
```
<function __main__.MyClass.f(self)>
```

[28]:
```
MyClass.__doc__
```

[28]:
```
'A simple example class'
```

[30]:
```
MyClass.__dict__
```

[30]:
```
mappingproxy({'__module__': '__main__',
              '__doc__': 'A simple example class',
              'i': 12345,
              'f': <function __main__.MyClass.f(self)>,
              '__dict__': <attribute '__dict__' of 'MyClass' objects>,
              '__weakref__': <attribute '__weakref__' of 'MyClass' objects>})
```

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class.

[54]:
```
x = MyClass()
```

creates a new instance of the class and assigns this object to the local variable x.

[37]:
```
print(x)
```

```
<__main__.MyClass object at 0x000002A6FF809A88>
```

The instantiation operation ("calling" a class object) creates an empty object.
Many classes like to create objects with instances customized to a specific initial state.
Therefore a class may define a special method named **init**()

[39]:
```
def __init__(self):
    self.data = []
```

[40]:
```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
```

[42]:
```
x = Complex(3.0, -4.5)
```

[45]:
```
x.r,x.i
```

[45]:
```
(3.0, -4.5)
```

0.4 Instance Objects

The only operations understood by instance objects are attribute references.

There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to "instance variables"

Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

[46]:
```
x.counter=1
```

[47]:
```
while x.counter < 10:
    x.counter = x.counter * 2
```

[48]:
```
x.counter
```

[48]:
```
16
```

[49]:
```
del x.counter
```

[50]:
```
x.counter
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-50-5b599638899f> in <module>
----> 1 x.counter


AttributeError: 'Complex' object has no attribute 'counter'
```

The other kind of instance attribute reference is a method.

A method is a function that "belongs to" an object.

Valid method names of an instance object depend on its class

By definition, all attributes of a class that are function objects define corresponding methods of its instances.

0.5 Method Objects

[55]:
```
x.f()
```

[55]:
```
'hello world'
```

[57]:
```
xf=x.f
for i in range(10):
    print(xf()) #if u dont mention the braces it will print object refrence
```

```
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
```

0.6 Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class

[75]:
```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance
```

[76]:
```
d = Dog('Fido')
e = Dog('Buddy')
```

[77]:
```
d.kind='hi'
```

[80]:
```
f=Dog('Hello')
```

[81]:
```
f.kind
```

[81]: 'canine'

[82]:
```
e.kind
```

[82]: 'canine'

[83]:
```
d.kind
```

[83]: 'hi'

[84]:
```
d.name
```

[84]: 'Fido'

[85]:
```
e.name
```

[85]: 'Buddy'

shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries.

[86]:
```
class Dog:

    tricks = []             # mistaken use of a class variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)
```

[87]:
```
d = Dog('Fido')
e = Dog('Buddy')
```

[88]:
```
d.add_trick('roll over')
e.add_trick('play dead')
```

[90]:
```
d.tricks
```

[90]:
```
['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

[91]:
```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)
```

[92]:
```
d = Dog('Fido')
e = Dog('Buddy')
```

[93]:
```
d.add_trick('roll over')
e.add_trick('play dead')
```

[94]:
```
d.tricks
```

[94]:
```
['roll over']
```

[95]:
```
e.tricks
```

[95]:
```
['play dead']
```

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok.

[96]:
```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)
```

```
class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Now f, g and h are all attributes of class C that refer to function objects, and consequently they are all methods of instances of C — h being exactly equivalent to g. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the self argument:

[97]:
```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

0.7 Private Variables

```
<li><pre>However, there is a convention that is followed by most Python code: a name prefixed w
<li><pre>Any identifier of the form __spam (at least two leading underscores, at most one trail
```

This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

```
<li>Name mangling is helpful for letting subclasses override methods without breaking intraclas
```

[89]:
```
class Point:
    def __init__(self, x, y):
        self.move(x, y)

    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)
```

[19]:
```
point = Point(3, 5)
```

[20]:
```
print(point.x, point.y)
```

3 5