Introduction to classes and object-oriented programming

Fernando Pineda 140.636 Created: Oct. 11, 2018 version 0.2

Recall that objects contain data and functions. We refer to data in objects as *attributes* and we refer to functions in objects as *methods*.

Up to this point we have only used the built-in Python object types, e.g. `string`, `list`, `integer`, etc. But what if we wanted to build arbitrary objects? We would want to define the attributes and methods that such objects should have, AND we would want to have some means of creating these objects.

Of course, since everything in Python is an object, it should not be too surprising that the way to do this is by declaring an object that builds our objects. The `class` statement is used for this purpose.

classes and instances

To start, let's create a trivial class with no attributes and no methods

```
class Trivial:
  pass # pass is just a placeholder statement, it does nothing

# Now we can make many Trivial objects

t1 = Trivial()
t2 = Trivial()

# Let's investigate these

print(t1)
print(t2)
```

The two objects `t1` and `t2` are referred to as *instances* of the class `Trivial`.

Inheritance, `__new__` and `__init__`

Our class was rather trivial, yet it has a lot of methods that we can see with the `dir()` function.

```
dir(g1)
```

Where did all the methods come from? In Python3 all objects inherit from the "object" class. In other words they inherit methods from the object class.

Two of these methods are particularly important. The `__new__` method is the method that actually creates an instance of the class. The `__new__` function is automatically called.

The `__init__` method is called after `__new__` and is used to initialize attributes when you create an instance of a class. Let's look at `__init__` first.

initializing attributes

Let's create a simple class that has a single attribute and a single method

```
class Simple:

  # set the greeting when the class object is created
  greeting = "hello world" # All instances share this attribute value

  # return the greeting
  def greet(self):
    return(self.greeting)

# create two instances (objects) of the Simple class
g1 = Simple()
g2 = Simple()

print(g1.greet())
print(g1.greet())
```

The `greeting` variable is a class variable. All instances of the `Simple` class have the same value.

It would be useful to be able to initialize different greetings in different instances. We use the `__init__` method to initialize an object's attributes.

```
class Simple:

  # set the greeting
  def __init__(self,x):
    self.greeting = x    # this is an instance variable

  def greet(self):
    return(self.greeting)

# create two instances (objects) of the Simple class
g1 = Simple("hello")
g2 = Simple("greetings of the day")

print(g1.greet())
print(g2.greet())
```

Now let's look at `_new__`

fine control of instance creation with `super()` and `__new__`

The `__new___` method is called automatically by Python when we instantiate an object for a new class. The `super` method is how we invoke methods directly from the parent class.

So this class definition of the Trivial class is exactly the same as our previous Trivial class

```
class Trivial():

        def __new__(self):
                return( super().__new__(self) )
```

Sometimes we want to do something special when we create an instance of a class. For example, suppose that instead of creating a difference object each time we instantiate a class, we instead want to return exactly the same object every time. In this case, we want to save the object that **new** creates and then return it every subsequence time we instantiate the class. We can't use the class's `__new__` because it doesn't exist until after the object is instantiated, so we use the parent classe's `__new__`. We access the parent's `__new__` with the `super()` function:

```
class Singleton(object):
        _instance = None

        def __new__(self):
                if(not self._instance):
                        self._instance = super().__new__(self)
                        return(self._instance)
                else :
                        return(self._instance)


s1 = Singleton()
s2 = Singleton()
```

The first time we instantiate a Singleton the `__new__` method checks if the class variable `_instance` has been set. It has not, then we create the instance object. If it has a value, then we return that value instead of creating new object. A class that only ever returns the same object every time it is invoked, is called a "singleton" class.

References

1. http://CoreyMS.com