Classes

--created by David Song

Contents

Classes in Python do not exhibit the traditional behavior in classic OO languages, they are more like namespaces. They hold related functions and variables.

Define a class

```
class Person(object):
    def __init__(self):
        self.name = ''
        self.age = 0

    def __str__(self):
        return '{} {}'.format(self.name, self.age)
```

Objects have a *__dict__* property that will display the attributes of an object as a dictionary.

```
person = Person()
person.name='Bob'
person.age=37
print person.__dict__  outputs {'age': 37, 'name': 'Bob'}
```

A class also has *__dict__* property,

```
print Person.__dict__ outputs the class's properties:

{'__module__': '__main__', '__str__': <function __str__ at 0x02621C30>,
'__dict__': <attribute '__dict__' of 'Person' objects>, '__weakref__':
<attribute '__weakref__' of 'Person' objects>, '__doc__': None, '__init__':
<function __init__ at 0x02621C70>}
```

The self parameter is in fact the first parameter to any instance method and represent the object calling the method. In Python, any call to a method in this form *instance.method(args)* will be translated into *Class.method(instance, args)*.

Note that, in the above class definition, Person inherits from object. This means Person follows the new style class definition. If it is defined without `(object)`, it is considered as classic or old style class definition, some new features will not be available for this class.

Variable scope

```
msg = 'hello 1'              # global variable
class Person(object):
    msg = 'hello 2'          # class variable

    def __init__(self):
        print ('in init:{}'.format(msg))     # print hello 1
        self.name = ''
        self.age = 0


    print ('in class:{}'.format(msg))        # print hello 2

person = Person()

print ('class variable:{}'.format(Person.msg))
```

When the module is loaded, module variable msg and the class is loaded, it prints out `'in class:hello 2'`. The statement `person = Person()` instantiates a class instance, calling `__init__` method, prints out `'in init:hello 1'`. Finally it prints out `'class variable:hello 2'`

Useful Magic Methods

The destructor **`__del__`** is called when del instance is executed, but is not commonly used. Instance will be automatically garbage collected when the instance is out of execution context.

1. __str__, __repr__

```
class Person(object):
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    def __str__(self):
        return '{} {}'.format(self.name, self.age)


person = Person('Bob', 37)
print(person)                # calling __str__
print(repr(person))          # calling __repr__
```

The goal of __str__ is to be human readable, while _repr__ is intended to represent the object, especially for debugging purpose.

2. __setitem__, __getitem__

These two methods provide a way to make the object as a dictionary. Need to implement __getitem__, __setitem__ method if you want to use notation like below:

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return '{} {}'.format(self.name, self.age)

    def __getitem__(self, key):
        print('__getitem__ gets called with {}'.format(key))
        return self.__dict__[key]

    def __setitem__(self, key, value):
        print('__setitem__ gets called with key:{} value:{}'.format(key, value))
        self.__dict__[key] = value

print person ['age']
person ['name'] = 'Robert'

person['height']=175
print person['name']
print person['height']
```

3. __setattr__, __getattr__

Python provides a means for custom lookup instance's attributes (or variables).

Below code is just an illustration of how to customize the get/set attr method, it may be confusing to compare with those get/set item methods.

```
class Person(object):
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    def __str__(self):
        return '{} {}'.format(self.name, self.age)

    def __getitem__(self, key):
        print('__getitem__ gets called with {}'.format(key))
        return self.__dict__[key]

    def __setitem__(self, key, value):
        print('__setitem__ gets called with key:{} value:{}'.format(key, value))
        self.__dict__[key] = value

    def __getattr__(self, item):
        print('__getattr__ gets called with {}'.format(item))
        if item in self.__dict__:
            return self.__dict__[item]
        else:
            return None

    def __setattr__(self, key, item):
        print('__setitem__ gets called with key:{} value:{}'.format(key, item))
        self.__dict__[key] = item
```

Note that, these two methods are accessed via the dot . notation but not the [ ] notation. For example, `p1.height` returns none, while `p1['height']` raises an exception if the __setitem__ method not provided in the class. The __setattr__ will be called when assignment involved, for example, p1.age=18; it is more like an assignment interceptor.

The differences between these get/set for item and attr are:

- get/set items -- are used only in indexed attributes like arrays, dictionaries, lists etc. without implementing those method, you cannot use [] notation.

- get/set attr – encapsulated solution; override it when necessary.
  Without implementation those method, you still can use the . notion like below:

```
person.name='John'
print person.name       # print out John
```

The Properties Decorator

Python defies a means for exposing methods as if they were simple properties, they are similar to Java code getter/setter.

```
@property
def balance(self):
    return self.amount

@balance.setter
```

```
    def balance(self, value):
        self.amount = value

run:
      a.balance = 2000
```

The benefits to use getter/setter instead of directly update instances variables are you can add checking before updating, and perhaps there is a chain actions to update several variables or objects.

Overloading

There is no true way to overload functions or methods, it is possible to use default parameters and type checks to allow multiple initialization techniques.

Private keyword

There is no public or private keyword in Python. Python relies on the developer and good coding practices. Normally a single underscore _ would be assumed to be a private attribute.

Using double underscore __ invokes name mangling feature within Python. A property with __ cannot be access directly with the object name, such as p1.__age, while it can be accessed via p1._Person__age.

Class variables

Class variables are declared at the class level and are visible to all instances and the class itself.

A variable created on the fly is only added to the object scope, and not visible to the class and other objects. For example, p1.height will create a height variable to p1 itself, and cannot be seen from the class and other Person objects. This can be checked like below:

```
p1.__dict__
p1.__class__.__dict__
```

Static and class method

```
class A(object):
    def __init__(self,a):
        self.a=a

    @staticmethod
    def static_method(x): #a static method does not take the self instance
        print 'static method, arg=', x

    @classmethod
    def class_method(cls):
```

```
        print 'class method,', cls.__class__

a=A(10)

A.static_method(1)
a.static_method(1)

A.class_method()
a.class_method()
```

Note: a static method does not take the self instance as an input parameter, a class method takes the class as the first input parameter. They can be called from either an instance or the class.

A first look seems there is no differences on how to use those two methods, but class methods are intended to be used with class not instance, and it is useful in some circumstances, such as in class inheritance.

Inheritance

```
class C(A, B):
    def __init__(self, a,b,c):
        A.__init__(self,a)
        B.__init__(self,b)
        self.c = c

    def __str__(self):
        return A.__str__(self) +" "+ B.__str__(self)+""+self.c
```

As we can see, refer to a parent method or variable, just use the parent class name and the method, passes in *self* and parameters. The `super()` method can be used to refer to a parent class, with this class name and the self instance; but not sure it is useful in multiple inheritance. Using the parent class name to refer to a parent is straightforward and sufficient.

```
      super(C, self).__init__() #how to use in multiple inheritance?
```

Search order in Inheritance

When dealing with evaluating a value with both inheritance and instances involved, such as a variable is defined as a class variable and also an instance variable in several places in the hierarchy. Python looks to the instances first (going up the chain) and then to the classes next (again going up the chain).

Classes searching order in inheritance can be see by using: `print C.__mro__`

Lack of interfaces

Use introspection features such as *hasattr(), callable(), isinstance(inst, class),type()*

Operator Overloading

There are some magic methods can be overridden

Method	Description
__init__	Constructor
__del__	Desctructor
__str__	toString()
__add__	+ operator
__iter__	Iterator
__len__	Lenth of object
__cmp__	Compare
__eq__	Equal
__lt__	Less than

Abstract class

Use the *abc* module to provide two important items: a decorator and a metaclass. The metaclass prevents instantiation of the abstract class and the decorator is used to define which methods will be abstract.

```
import abc
class AbstractBase(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self, item):
        self.item = item
        self.do_action()

    @abc.abstractmethod
    def do_action(self):
        # subclass will define this work
        return
```

Q: How to define a class to be final?

Metaclass

Metaclasses are structures that define how classes can be constructed.

```
new_class= type('name',(base1, base2, ...), attributes)
```

```
class Mammal(object):
    def __init__(self):
        self.num_legs = 4


def dog_speak(self): print "Arf! arf!"

Dog = type('Dog', (Mammal,), {'breed':None, 'speak': dog_speak})

d = Dog()

print d.breed, d.num_legs
d.speak()
```

You can define your own metaclass:

```
class MyMetaClass(type):
    def __new__(meta, name, bases, attributes):
        bases = (Mammal,)
        return type.__new__(meta, name, bases, attributes)


class Dog(object):
    __metaclass__ = MyMetaClass


d = Dog()
print d.num_legs
print type(d)
print type(Dog)
```