SEES: OOP and more testing

FOSSEE

February 12, 2015

Outline

1. Object Oriented Programming
2. `unittest` framework

Outline

1. Object Oriented Programming

2. `unittest` framework

Objectives

At the end of this section, you will be able to -

- Understand the differences between Object Oriented Programming and Procedural Programming
- Appreciate the need for Object Oriented Programming
- Read and understand Object Oriented Programs
- Write simple Object Oriented Programs

Classes: the big picture

- Lets you create new data types
- Class is a template for an object belonging to that class
- Note: in Python a class is also an object
- Instantiating a class creates an instance (an object)
- An instance encapsulates the state (data) and behavior (methods)
- Allows you to define an inheritance hierarchy
  - “A Honda car is a car.”
  - “A car is an automobile.”
  - “A Python is a reptile.”
- Programmers need to think OO

Classes: what’s the big deal?

- Lets you create objects that mimic a real problem being simulated
- Makes problem solving more natural and elegant
- Easier to create code
- Allows for code-reuse
- Polymorphism

Class definition and instantiation

- Class definitions when executed create class objects
- Instantiating the class object creates an instance of the class

```
class Foo(object):
    pass
# class object created.

# Create an instance of Foo.
f = Foo()
# Can assign an attribute to the instance
f.a = 100
print f.a
100
```

Classes . . .

- All attributes are accessed via the `object.attribute` syntax
- Both class and instance attributes are supported
- *Methods* represent the behavior of an object: crudely think of them as functions “belonging” to the object
- All methods in Python are “virtual”
- Inheritance through subclassing
- Multiple inheritance is supported
- No special public and private attributes: only good conventions
  - `object.public()`: public
  - `object._private()` & `object.__priv()`: non-public

Classes: examples

```
class MyClass(object):
    """Example class (this is the class docstring
    i = 12345 # A class attribute
    def f(self):
        """This is the method docstring"""
        return 'hello world'

>>> a = MyClass() # creates an instance
>>> a.f()
'hello world'
>>> # a.f() is equivalent to MyClass.f(a)
... # This also explains why f has a 'self' argum
... MyClass.f(a)
'hello world'
```

Classes (continued)

- `self` is conventionally the first argument for a method
- In previous example, `a.f` is a method object
- When `a.f` is called, it is passed the instance `a` as the first argument
- If a method called `__init__` exists, it is called when the object is created
- If a method called `__del__` exists, it is called before the object is garbage collected
- Instance attributes are set by simply “setting” them in `self`
- Other special methods (by convention) like `__add__` let you define numeric types:

  `https://docs.python.org/2.7/reference/datamodel.html`

Classes: examples

```
class Bag(MyClass): # Shows how to derive classes
    def __init__(self): # called on object creati
        self.data = [] # an instance attribute
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
>>> a = Bag()
>>> a.f() # Inherited method
'hello world'
>>> a.add(1); a.addtwice(2)
>>> a.data
[1, 2, 2]
```

Derived classes

- Call the parent's `__init__` if needed
- If you don't need a new constructor, no need to define it in subclass
- Can also use the `super` built-in function

```
class AnotherBag(Bag):
    def __init__(self):
        # Must call parent's __init__ explicitly
        Bag.__init__(self)
        # Alternatively use this:
        super(AnotherBag, self).__init__()
        # Now setup any more data.
        self.more_data = []
```

Classes: polymorphism

```
class Drawable(object):
    def draw(self):
        # Just a specification.
        pass

class Square(Drawable):
    def draw(self):
        # draw a square.
class Circle(Drawable):
    def draw(self):
        # draw a circle.
```

Classes: polymorphism

```
class Drawable(object):
    def draw(self):
        # Just a specification.
        pass

class Square(Drawable):
    def draw(self):
        # draw a square.
class Circle(Drawable):
    def draw(self):
        # draw a circle.
```

Classes: polymorphism

```
class Artist(Drawable):
    def draw(self):
        for obj in self.drawables:
            obj.draw()
```

Example: Managing Talks

- A list of talks at a conference
- We want to manage the details of the talks

```
talk = {'Speaker': 'Guido van Rossum',
        'Title': 'The History of Python'
        'Tags': 'python,history,C,advanced'}

def get_first_name(talk):
    return talk['Speaker'].split()[0]

def get_tags(talk):
    return talk['Tags'].split(',')
```

- Not convenient to handle large number of talks

Objects and Methods

- Objects group data with the procedures/functions
- A single entity called `object`
- Everything in Python is an object
- Strings, Lists, Functions and even Modules

```
s = "Hello World"
s.lower()

l = [1, 2, 3, 4, 5]
l.append(6)
```

Objects . . .

- Objects provide a consistent interface

```
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
for line in urllib2.urlopen('http://site.com'
    print line
```

Classes

- A new string, comes along with methods
- A template or a blue-print, where these definitions lie
- This blue print for building objects is called a `class`
- `s` is an object of the `str` class
- An object is an “instance” of a class

```
s = "Hello World"
type(s)
```

Defining Classes

- A class equivalent of the talk dictionary
- Combines data and methods into a single entity

```python
class Talk:
    """A class for the Talks."""

    def __init__(self, speaker, title, tags):
        self.speaker = speaker
        self.title = title
        self.tags = tags

    def get_speaker_firstname(self):
        return self.speaker.split()[0]

    def get_tags(self):
        return self.tags.split(',')
```

class block

- Defined just like a function block
- `class` is a keyword
- `Talk` is the name of the class
- Classes also come with doc-strings
- All the statements of within the class are inside the block

```
class Talk:
    """A class for the Talks."""

    def __init__(self, speaker, title, tags):
        self.speaker = speaker
        self.title = title
        self.tags = tags
```

self

- Every method has an additional first argument, `self`
- `self` is a reference to the object itself, of which the method is a part of
- Variables of the class are referred to as `self.variablename`

```
def get_speaker_firstname(self):
    return self.speaker.split()[0]

def get_tags(self):
    return self.tags.split(',')
```

Instantiating a Class

- Creating objects or instances of a class is simple
- We call the class name, with arguments as required by it's `__init__` function.

```
bdfl = Talk('Guido van Rossum',
            'The History of Python',
            'python,history,C,advanced')
```

- We can now call the methods of the Class

```
bdfl.get_tags()
bdfl.get_speaker_firstname()
```

`__init__` method

- A special method
- Called every time an instance of the class is created

```
print bdfl.speaker
print bdfl.tags
print bdfl.title
```

Inheritance I

- Suppose, we wish to write a `Tutorial` class
- It's almost same as `Talk` except for minor differences
- We can "inherit" from `Talk`

```
class Tutorial(Talk):
    """A class for the tutorials."""

    def __init__(self, speaker, title, tags,
                            handson=True):
        Talk.__init__(self, speaker, title,
                            tags)
        self.handson = handson
```

Inheritance II

```
def is_handson(self):
    return self.handson
```

- Modified `__init__` method
- New `is_handson` method
- It also has, `get_tags` and `get_speaker_firstname`

```
numpy = Tutorial('Travis Oliphant',
                 'Numpy Basics',
                 'numpy,python,beginner')
numpy.is_handson()
numpy.get_speaker_firstname()
```

Summary

In this section we have learnt,

- the fundamental difference in paradigm, between Object Oriented Programming and Procedural Programming
- to write our own classes
- to write new classes that inherit from existing classes

Outline

1. Object Oriented Programming

2. unittest framework

unittest

- unittest framework can efficiently automate tests
- Easily initialize code and data for executing the specific tests
- Cleanly shut them down once the tests are executed
- Easily aggregate tests into collections and improved reporting

unittesting gcd.py

- Subclass the TestCase class in unittest
- Place all the test code as methods of this class
- Place the code in test_gcd.py

test_gcd.py I

```python
import gcd
import unittest

class TestGcdFunction(unittest.TestCase):
    def setUp(self):
        # Called before each test case.
        print "In setUp"

    def tearDown(self):
        print "In tearDown"

    def test_gcd(self):
        self.assertEqual(gcd.gcd(45, 5), 5)
        self.assertEqual(gcd.gcd(45, 5), 5)
```

test_gcd.py II

```
    def test_gcd_correctly_handles_floats(self):
        # Write appropriate tests here.
        pass

if __name__ == '__main__':
    unittest.main()
```

test_gcd.py

- setUp – called before every test_* method
- tearDown – called after every test
- setUp and tearDown – useful to perform common operations, make a temporary directory, delete it when done etc.
- test_gcd – actual test code
- assertEqual – compare actual result with expected one
- Also see: docs.python.org/2.7/library/unittest.html