Introduction to Python: Object Oriented Programming

Christopher Barker

UW Continuing Education / Isilon

August 01, 2012

Table of Contents

Lightning Talks

Lightning Talks today:

Brett and Matt

Review of Previous Class

- Built an HTTP server
- Very basics of sockets
- Basics of HTTP protocol
- Got a server working (even proto-CGI!)

review of my `http_serve8.py`

Object Oriented Programming

More about Python implementation than OO design/strengths/weaknesses

One reason for this:
Folks can't even agree on what OO "really" means

The Quarks of Object-Oriented Development - Deborah J. Armstrong:
`http://agp.hx0.ru/oop/quarks.pdf`

Object Oriented Programming

Is Python a "True" Object-Oriented Language?

(Doesn't support full encapsulation, doesn't require objects, etc...)

Object Oriented Programming

I don't Care!

Good software design is about code-reuse, clean separation of concerns, refactorability, testability, etc...

OO can help with all that, but:

- it doesn't guarantee it
- it can get in the way

Object Oriented Programming

Python is a Dynamic Language

That clashes with "pure" OO

Think in terms of what makes sense for you project – not any one paradigm of software design.

Object Oriented Programming

OO for this class:

"Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use"

`http://en.wikipedia.org/wiki/Object-oriented_programming`

Object Oriented Programming

Even simpler:

Objects are data and the functions that act on them in one place.

In Python: just another namespace.

Object Oriented Programming

The OO buzzwords:

- data abstraction
- encapsulation
- messaging
- modularity
- polymorphism
- inheritance

Object Oriented Programming

You can do OO in C
(see the GTK+ project)

"OO languages" give you some handy tools to make it easier (and safer).

- polymorphism (duck typing gives you this anyway)
- inheritance

Object Oriented Programming

OO is the dominant model for the past couple decades

You will need to use it:

– It's a good idea for a lot of problems

– You'll need to work with OO packages

Object Oriented Programming

Some definitions

class A category of objects: particular data and behavior: A circle (same as a type in python)

instance A particular object of a class: a specific circle

object the general case of a instance – really any value (in Python anyway)

attribute something that belongs to an object (or class) – generally thought of as a variable, or single object, as apposed to a ...

method a function that belongs to a class

Python Classes

The `class` statement

`class` creates a new type object:

```
In [4]: class C(object):
    pass
   ...:
In [5]: type(C)
Out[5]: type
```

It is created when the statement is run – much like `def`

(note on "new style" classes)

Python Classes

Note about the book (TP):

Chapters 15 and 16 use a style that generally isn't recommended:

```
In [6]: class Point(object):
   ...:     pass
In [7]: p = Point()
In [8]: p.x = 4
In [9]: p.y = 2
```

Python is Dynamic – you can do this, but you generally want more structure, defaults, etc.

(it used to be a quick and dirty "struct" – but use a named tuple now)

Python Classes

About the simplest class:

```
>>> class Point:
...     x = 1
...     y = 2
>>> Point
<class __main__.Point at 0x2bf928>
>>> p
<__main__.Point instance at 0x2de918>
>>> Point.x
1
>>> p = Point()
>>> p.x
1
```

Python Classes

Basic Structure of a real class

```
class Point(object):
# everything defined in here is in the class namespace
    def __init__(self, x, y):
        self.x = x
        self.y = y

## create an instance of that class
p = Point(3,4)

## access the attributes
print "p.x is:", p.x
print "p.y is:", p.y
```

see: `simple_class` in code dir

Python Classes

The Initializer

The `__init__` special method is called when a new instance of a class is created.

You can use it to do any set-up you need

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

It gets the arguments passed to the class constructor

Python Classes

`self`

The instance of the class is passed as the first parameter for every method.

"`self`" is only a convention – but you DO want to use it.

```
class Point(object):
    def a_function(self, x, y):
...
```

Does this look familiar from C-style procedural programming?

Python Classes

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Anything assigned to a `self.` attribute is kept in the instance name space

That's where all the instance-specific data is.

Python Classes

```
class Point(object):
    size = 4
    color= "red"
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Anything assigned in the class scope is a class attribute – every instance of the class shares the same one.

Python Classes

```
class Point(object):
    size = 4
    color= "red"
...
    def get_color():
        return self.color

>>> p3.get_color()
 'red'
```

class attributes are accessed with `self` also..

Python Classes

Typical methods

```
class Circle(object):
    color = "red"
    def __init__(self, diameter):
        self.diameter = diameter

    def grow(self, factor=2):
        self.diameter = self.diameter * factor
```

methods take some parameters, manipulate the attributes in `self`

Python Classes

Gotcha!

```
...
    def grow(self, factor=2):
        self.diameter = self.diameter * factor
...

In [205]: C = Circle(5)
In [206]: C.grow(2,3)

TypeError: grow() takes at most 2 arguments (3 given)
```

Huh???? I only gave 2

LAB

We had such a good time last class – we'll do something similar

The goal is to build a set of classes that render an html page: `sample_html.html`

We'll start with a single class, then add some sub-classes to specialize the behavior

More details in `week-06/LAB_instuctions.txt`

LAB

Step 1:

- Create an "Element" class for rendering an html element (xml element).
- It should have class attributes for the tag name and the indentation
- the constructor signature should look like:
  `Element(content=None)` where content is a string
- It should have an "append" method that can add another string to the content
- It should have a `render(file_out, ind = "")` method that renders the tag and the strings in the content.
  `file_out` could be any file-like object.
  `ind` is a string with enough spaces to indent properly.

Lightning Talk

Lightning Talk:

Brett

Inheritance

In object-oriented programming (OOP), inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object.

...

objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, or super classes.

The resulting classes are known as derived classes or subclasses.

(`http://en.wikipedia.org/wiki/Inheritance_%28object-oriented_programming%29`)

Subclassing

A subclass "inherits" all the attributes (methods, etc) of the parent class.

You can then change ("override") some or all of the attributes to change the behavior.

The simplest subclass in Python:

```
class A_Subclass(The_SuperClass):
    pass
```

`A_subclass` now has exactly the same behavior as `The_SuperClass`

Overriding attributes

Overriding is as simple as creating a new attribute with the same name:

```
class Circle(object):
    color = "red"
...
class NewCircle(Circle):
    color = "blue"
>>> nc = NewCircle
>>> print nc.color
blue
```

all the `self` instances will have the new attribute

Overriding methods

Same thing, but with methods

```
class Circle(object):
...
    def grow(self, factor=2):
        """grows the circle's diameter by factor"""
        self.diameter = self.diameter * factor
...
class NewCircle(Circle):
...
    def grow(self, factor=2):
        """grows the area by factor..."""
        self.diameter = self.diameter * math.sqrt(2)
```

all the instances will have the new method

"Here's a program design suggestion: whenever you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you obey this rule, you will find that any function designed to work with an instance of a superclass, like a Deck, will also work with instances of subclasses like a Hand or PokerHand. If you violate this rule, your code will collapse like (sorry) a house of cards."

ThinkPython 18.10

LAB

Step 2:

- Create a couple subclasses of `Element`, for a `<body>` tag and `<p>` tag. Simply override the `tag` class attribute.
- Extend the `Element.render()` method so that it can render other elements inside the tag in addition to strings. Simple recursion should do it. i.e. it can call the `render()` method of the elements it contains.
- Deal with the content items that could be either simple strings or `Elements` with `render` methods...there are a few ways to handle that...

LAB

Step 3:

- Create a `<head>` element – simple subclass.
- Create a `OneLineTag` subclass of Element: It should override the render method, to render everything on one line – for the simple tags, like:
  `<title> PythonClass - Class 6 example </title>`
- Create a Title subclass of `OneLineTag` class for the title.
- You should now be able to render an html doc with a head element, with a `title` element in that, and a body element with some `<P>` elements and some text.

Lightning Talk

Lightning Talk:

Matt

Overriding __init__

__init__common method to override

You often need to call the super class `__init__` as well

```
class Circle(object):
    color = "red"
    def __init__(self, diameter):
        self.diameter = diameter
...
class CircleR(Circle):
    def __init__(self. radius):
        diameter = radius*2
        Circle.__init__(self, diameter)
```

exception to: "don't change the method signature" rule.

Overriding other methods

You can also call the superclasses other methods:

```
class Circle(object):
...
    def get_area(self, diameter):
        return math.pi * diameter / 2.0

class CircleR2(Circle):
...
    def get_area(self):
        return Circle.get_area(self, self.radius*2)
```

There is nothing special about `__init__` except that it gets called automatically.

When to Subclass

"Is a" relationship: Subclass/inheritance

"Has a" relationship: Composition

When to Subclass

"Is a" vs "Has a"

You may have a class that needs to accumulate an arbitrary number of objects.

A list can do that – do should you subclass list?

Ask yourself:

– IS you class a list (with some extra functionality)?
or
– Does you class HAVE a list?

You only want to subclass list if your class could be used anywhere is list is.

Attribute resolution order

When you access an attribute:
`An_Instance.something`

Python looks for it in this order:

1. Is it an instance attribute ?
2. Is it a class attribute ?
3. Is it a superclass attribute ?
4. Is it a super-superclass attribute ?
5. ...

It can get more complicated...
`http://www.python.org/getit/releases/2.3/mro/`

What are Python classes, really?

Putting aside the OO theory...

Python classes are:

- Namespaces
  - One for the class object
  - One for each instance
- Attribute resolution order
- Auto tacking-on of `self`

That's about it – really!

Type-Based dispatch

From Think Python:

```
if isinstance(other, A_Class):
    Do_something_with_other
else:
    Do_something_else
```

Usually better to use "duck typing" (polymorphism)
But when it's called for:

- `isinstance()`
- `issubclass()`

GvR: "Five Minute Multi- methods in Python":
`http://www.artima.com/weblogs/viewpost.jsp?thread=101605`

LAB

Keep going....

multiple inheritance

Multiple inheritance: Pulling from more than one class

```
class Combined(Super1, Super2, Super3):
    def __init__(self, something, something else):
        Super1.__init__(self, ......)
        Super2.__init__(self, ......)
        Super3.__init__(self, ......)
```

(calls to the super class `__init__` are optional – usage dependent)

Attribute resolution – right to left

Why would you want to do this?

mix-ins

Why you might want to do multiple inheritance

`FLoatCanvas` example