Introduction to Python
More OO – Inheritance and Duck Typing
Special methods

Christopher Barker

UW Continuing Education

November 12, 2013

Table of Contents

Lightning Talks

Lightning talks today:

Linh Tran

Maitri Kashyap

Sridharan Rajagopalan

Richard Smith

Review of Previous Class

- lambda
- Intro to OO
- Start of HTML generation code

Homework review

Questions?

Overview of my html-generating classes so far...

Demo of class vs. instance attributes

Lightning Talks

Lightning Talks:

Linh Tran

Maitri Kashyap

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.

More subclassing

You can also call the superclass' other methods:

```
class Circle(object):
...
    def get_area(self, diameter):
        return math.pi * (diameter/2.0)**2

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 – so should you subclass list?

Ask yourself:

– Is your 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 a list can be used.

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/`

`http://python-history.blogspot.com/2010/06/method-resolution-order.html`

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

We're going to do the rest: steps 4 - 8

(Still using `week-06/code/htmlrender`)

Step 4:

- Extend the Element class to accept a set of attributes as keywords to the constructor, i.e.:

  ```
  Element("some text content",
          id="TheList",
          style="line-height:200\%")
  ```

  (remember `**kwargs` ?)
- The render method will need to be extended to render the attributes properly.

You can now render some <p> tags (and others) with attributes

LAB

Step 5:

- Create a `SelfClosingTag` subclass of `Element`, to render tags like:
  `<hr />` and `<br />` (horizontal rule and line break).
- You will need to override the render method to render just the one tag and attributes.
- create a couple subclasses of SelfClosingTag for `<hr>` and `<br />` (Line break) or ??? if you like

You can now render an html page with a proper `<head>` (`<meta />` and `<title>` elements)

LAB

Step 6:

- Create an `A` class for an anchor (link) element. Its constructor should look like: `A(self, link, content)` – where link is the link, and content is what you see. It can be called like so: `A("http://google.com", "link")`
- You should be able to subclass from `Element`, and only override the `__init__`
  - Calling the `Element` `__init__` from the `A` `__init__`

You can now add a link to your web page.

LAB

Step 7:

- Create `Ul` class for an unordered list (really simple subclass of Element)
- Create `Li` class for an element in a list (also really simple)
- add a list to your web page.
- Create a Header class – this one should take an integer argument for the header level. i.e `<h1>, <h2>, <h3>`, called like:
- `H(2, "The text of the header")` for an `<h2>` header
- It can subclass from `OneLineTag` – overriding the `__init__`, then calling the superclass `__init__`

LAB

Step 8:

- Update the Html element class to render the "`<!DOCTYPE html>`" tag at the head of the page, before the `html` element.
- You can do this by subclassing `Element`, overriding `render()`, but then calling `Element.render()` from `Html.render()`.
- Create a subclass of `SelfClosingTag` for `<meta charset="UTF-8" />` and add the meta element to the beginning of the head element to give your document an encoding.
- The doctype and encoding are HTML 5 and you can check this at: `validator.w3.org`.

You now have a pretty full-featured html renderer

Review of HTML renderer lab

You have built an html generator, using:

- A Base Class with a couple methods
- Subclasses overriding class attributes
- Subclasses overriding a method
- Subclasses overriding the `__init__`

These are the core OO approaches

If you don't have it working, or don't think you "get" it:
work on it for homework, and ask questions.

Lightning Talks

Lightning Talks:

Sridharan Rajagopalan

Richard Smith

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 – case dependent)

multiple inheritance

Attribute resolution – left to right

1. Is it an instance attribute ?
2. Is it a class attribute ?
3. Is it a superclass attribute ?
    1. is the it an attribute of the left-most superclass?
    2. is the it an attribute of the next superclass?
    3. ....
4. Is it a super-superclass attribute ?
5. ...also left to right...

`http://python-history.blogspot.com/2010/06/method-resolution-order.html`

Mix-ins

Why would you want to do this?

Hierarchies are not always simple:

- Animal
  - Mammal
    - GiveBirth()
  - Bird
    - LayEggs()

Where do you put a Platypus or an Armadillo?

Real World Example: `FloatCanvas`

New Style classes

You will see reference to “new style” classes

These derive from `object`

Introduced in python2.2 to better merge types and classes, and clean up a few things

Differences in method resolution order and properties

Mostly the same, often makes no difference

My advice: always subclass from `object`

super

`super()`: use it to call a superclass method, rather than exlicitly calling it:

instead of:

```
class A(B):
    def __init__(self, *args, **kwargs)
        B.__init__(self, *argw, **kwargs)
        ...
```

You can do:

```
class A(B):
    def __init__(self, *args, **kwargs)
        super(B, self).__init__(self, *argw, **kwargs)
        ...
```

There are some subtle differences with multiple inheritance

super

Two seminal articles about `super()`:

"Super Considered Harmful"

– James Knight

`https://fuhm.net/super-harmful/`

"super() considered super!"

– Raymond Hettinger

`http://rhettinger.wordpress.com/2011/05/26/super-considered-super/`

(Both worth reading....)

special methods

Python's Duck typing:

Defining special (or magic) methods in your classes is how you make your class act like standard classes

special methods

We've seen at least one:

`__init__`

it's all in the double underscores...

Pronounced "dunder" (or "under-under")

try: `dir(2)` or `dir(list)`

special methods

Emulating Numeric types

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

special methods

Emulating container types:

```
object.__len__(self)
object.__getitem__(self, key)
object.__setitem__(self, key, value)
object.__delitem__(self, key)
object.__iter__(self)
object.__reversed__(self)
object.__contains__(self, item)
object.__getslice__(self, i, j)
object.__setslice__(self, i, j, sequence)
object.__delslice__(self, i, j)
```

special methods

Example – to define addition:

```
def __add__(self, v):
    """
    redefine + as element-wise vector sum
    """
    assert len(self) == len(v)
    return vector([x1 + x2 for x1, x2 in zip(self, v)])
```

(from a nice complete example in `code/vector.py`)

special methods

You get the idea...

You only need to define the ones that are going to get used

But you probably want to define at least these:

`object.__str__`: Called by the str() built-in function and by the print statement to compute the informal string representation of an object.

`object.__repr__`: Called by the repr() built-in function and by string conversions (reverse quotes) to compute the official string representation of an object.

special methods

When you want your class to act like a "standard" class in some way:

Look up the magic methods you need and define them

`http://docs.python.org/reference/datamodel.html#special-method-names`

`http://www.rafekettler.com/magicmethods.html`

LAB

Write a "Circle" class:

A Circle has a radius and can compute its area:

```
In [2]: c = Circle(3)
In [3]: c.radius
Out[3]: 3
In [4]: c.get_area()
Out[4]: 28.274333882308138
In [5]: print c
Circle Object with radius: 3.000000
```

Write an `__add__` method so you can add two circles
Have `__str__` and `__repr__` methods
Extra credit: also compare them... (`c1 > c2`, etc)
`code/circle.py` and `code/test_circle.py`

Wrap Up

Thinking OO in Python:

Think about what makes sense for your code:

- Code re-use
- Clean APIs
- ...

Don't be a slave to what OO is *supposed* to look like.

Let OO work for you, not *create* work for you

Wrap Up

OO in Python:

The Art of Subclassing: Raymond Hettinger

`http://pyvideo.org/video/879/the-art-of-subclassing`

"classes are for code re-use – not creating taxonomies"

Stop Writing Classes: Jack Diederich

`http://pyvideo.org/video/880/stop-writing-classes`

"If your class has only two methods – and one of them is `__init__` – you don't need a class "

Homework

Finish the labs.

Watch the videos.

Readup more on OO design.

Your Project:

- By next week, send me a project proposal: can be short and sweet.
- Think about how you might use OO:
  - What classes naturally fall out of the problem?
  - NOTE: maybe none!