

# Introduction to Python

## More OO – Iterators and Generators

Christopher Barker

UW Continuing Education / Isilon

August 08, 2012

# Table of Contents

- 1 Review/Questions
- 2 More OO
- 3 Properties
- 4 Special Attributes
- 5 Iterators / Generators

# Lightning Talks

NOTE: class canceled next week

Lightning Talks today:

Brett and Jeffery

# Review of Previous Class

- Really Quick OO overview
- Built an html generator, using:
  - A base class with a couple methods
  - Subclasses overriding class attributes
  - Subclasses overriding a method
  - Subclasses overriding the `__init__`

# Homework review

## Homework notes

Did anyone make classes for a real project?

How many of you finished the html building code from last class?

## 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...

## 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



# LAB

## Last class: 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

## Last class: 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

## Last class 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

# Lightning Talk

Lightning Talk:

Brett

## Accessing Attributes

One of the strengths of Python is lack of clutter

Simple attributes:

```
In [5]: class C(object):  
        def __init__(self):  
            self.x = 5
```

```
In [6]: c = C()
```

```
In [7]: c.x
```

```
Out[7]: 5
```

```
In [8]: c.x = 8
```

# Getter and Setters?

What if you need to add behavior later?

- do some calculation
- check data validity
- keep things in sync

## Getter and Setters?

```
class C(object):  
    def get_x(self):  
        return self.x  
    def set_x(self, x):  
        self.x = x  
  
>>> c = C()  
>>> c.get_x()  
>>> 5  
>>> c.set_x(8)  
>>> c.get_x()  
>>> 8
```

Ugly and verbose – Java?

<http://dirtsimple.org/2004/12/python-is-not-java.html>

# properties

When (and if) you need them:

```
class C(object):  
    def getx(self):  
        return self._x  
    def setx(self, value):  
        self._x = value  
    def delx(self):  
        del self._x  
    x = property(getx, setx, delx, "docstring")
```

Interface is still like simple attribute access  
(properties\_sample.py )



# properties

When (and if) you need them:

```
class C(object):  
    def getx(self):  
        return self._x  
    def setx(self, value):  
        self._x = value  
    def delx(self):  
        del self._x  
    x = property(getx, setx, delx, "docstring")
```

Interface is still like simple attribute access  
(properties\_sample.py )

## staticmethod

A method that doesn't get self!

```
class C(object):  
    def add(a, b):  
        return a + b  
    add = staticmethod(add)  
  
>>> C.add(3,4)  
7  
>>> c = C()  
>>> c.add(2, 2)  
4
```

When you don't need self – can be used from either an instance or the class itself

see: `static_method.py`

## classmethod

Method gets the class object, rather than an instance the first argument

```
class C(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def a_class_method(klass, y):  
        print "in a_class_method", klass  
        return klass( y, y**2 )  
    a_class_method = classmethod(a_class_method)
```

When you need the class object rather than an instance – plays well with subclassing

see: `class_method.py`

## dict.fromkeys()

classmethod often used for alternate constructors:

```
>>> d = dict([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot convert dictionary update
sequence element #0 to a sequence
>>> d = dict.fromkeys([1,2,3])
>>> d
{1: None, 2: None, 3: None}
```

## dict.fromkeys()

```
class Dict: ...
    def fromkeys(klass, iterable, value=None):
        "Emulate dict_fromkeys() in dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
    fromkeys = classmethod(fromkeys)
```

See also `datetime.datetime.now()`, etc....

For a low-level look:

<http://docs.python.org/howto/descriptor.html>

## super

getting the superclass:

```
class SafeVehicle(Vehicle):  
    """  
    Safe Vehicle subclass of Vehicle base class...  
    """  
    def __init__(self, position=0, velocity=0, icon='S'):  
        Vehicle.__init__(self, position, velocity, icon)
```

not DRY

also, what if we had a bunch of references to superclass?

## super

getting the superclass:

```
class SafeVehicle(Vehicle):  
    """  
    Safe Vehicle subclass of Vehicle base class  
    """  
    def __init__(self, position=0, velocity=0, icon='S'):  
        super(SafeVehicle, self).__init__(position, velocity)
```

“super() considered super!” by Raymond Hettinger

[http://rhettinger.wordpress.com/2011/05/26/  
super-considered-super/](http://rhettinger.wordpress.com/2011/05/26/super-considered-super/)

maybe use super() for your html subclassing...

## 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:

Example run code in `test_circle.py`

```
>> c = Circle(radius=3)
>> c.diameter
6
>> c.diameter = 8
>> c.radius
4
```

Use properties so you can keep the radius and diameter in sync  
Write an `__add__` method so you can add two circles  
(and have `__str__` and `__repr__` methods)

# Lightning Talk

## Lightning Talk: Jeffery



# Iterators

Iterators are one of the main reasons Python code is so readable:

```
for x in just_about_anything:  
    do_stuff(x)
```

you can loop through anything that satisfies the iterator protocol

<http://docs.python.org/library/stdtypes.html#iterator-types>

# Iterator Protocol

An iterator must have the following methods:

```
iterator.__iter__()
```

Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.

```
iterator.next()
```

Return the next item from the container. If there are no further items, raise the StopIteration exception.

## Example Iterator

```
class IterateMe_1(object):  
    def __init__(self, stop=5):  
        self.current = 0  
        self.stop = stop  
    def __iter__(self):  
        return self  
    def next(self):  
        if self.current < self.stop:  
            self.current += 1  
            return self.current  
        else:  
            raise StopIteration
```

This is a simple version of `xrange()`

# itertools

`itertools` is a collection of utilities that make it easy to build an iterator that iterates over sequences in various common ways

<http://docs.python.org/library/itertools.html>

# LAB

- Extend (iterator\_1.py) to be more like xrange() – add three input parameters:  
    iterator\_2(start, stop, step=1)
- See what happens if you break out in the middle of the loop:

```
it = IterateMe_2(2, 20, 2)
for i in it:
    if i > 10: break
    print i
```

And then pick up again:

```
for i in it:
    print i
```

- Does xrange() behave the same?  
– make yours match xrange().

# generators

Generators give you the iterator immediately: no access to the underlying data ... if it even exists

## **Conceptually:**

iterators are about various ways to loop over data,  
generators generate the data on the fly

## **Practically:**

You can use either either way (and a generator is one type of iterator)

Generators do some of the book-keeping for you.

# yield

yield is a way to make a quickie generator with a function:

```
def a_generator_function(params):  
    some_stuff  
    yield(something)
```

Generator functions "yield" a value, rather than returning it

State is preserved in between yields

# yield

A function with `yield` in it is a “factory” for a generator

Each time you call it, you get a new generator:

```
gen_a = a_generator()  
gen_b = a_generator()
```

Each instance keeps its own state.

Really just a shorthand for an iterator class that does the book keeping for you.



# yield

An example: like xrange()

```
def y_xrange(start, stop, step=1):  
    i = start  
    while i < stop:  
        yield i  
        i += step
```

Real World Example: FloatCanvas

# yield

Note:

```
In [164]: gen = y_xrange(2,6)
```

```
In [165]: type(gen)
```

```
Out[165]: generator
```

```
In [166]: dir(gen)
```

```
Out[166]:
```

```
...  
'__iter__',  
...  
'next',
```

So the generator **is** an iterator

# yield

A generator function can also be a method in a class

More about iterators and generators:

<http://www.learningpython.com/2009/02/23/iterators-iterables-and-generators-oh-my/>

`yield_example.py`

## generator comprehension

another way to make a generator:

```
>>> [x * 2 for x in [1, 2, 3]]  
[2, 4, 6]  
>>> (x * 2 for x in [1, 2, 3])  
<generator object <genexpr> at 0x10911bf50>  
>>> for n in (x * 2 for x in [1, 2, 3]):  
...     print n  
... 2 4 6
```

More interesting if `[1, 2, 3]` is also a generator

# LAB

generator lab:

`code/primer.py`

# Homework

NOTE: class canceled next week

Pickup from last week – For a portion of the system you're responsible for testing:

- Make a few classes to represent that portion. (No more than three or four classes)
- Try to make use of something from today:
  - Mix-ins
  - Properties
  - Magic methods
  - iterator or generator