Introduction to Python
More OO – Decorators and Generators

Christopher Barker

UW Continuing Education / Isilon

August 08, 2012

Table of Contents

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

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)

Attribute resolution – left to right

(Why would you want to do this?)

mix-ins

Hierarchies are not always simple

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

Where do you put a Platypus or an Armadillo?

Real World Example: `FloatCanvas`

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?

Getters 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?

property

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, "I'm the 'x' property.")
```

Interface is still like simple attribute access

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 = 5
    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 way st o loop over data, generators, generate the data on the fly

Practically, they behave similarly

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:

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


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

Each instance keeps its own state.

yield

An example: like `xrange()`

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

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 be a method in a class:

`A cool example here...`

`www.learningpython.com/2009/02/23/iterators-iterables--and-`