Week 6: Advanced OO – Numpy

Christopher Barker

UW Continuing Education

April 30, 2013

Table of Contents

Some updates:

Some folks have asked to learn about Desktop GUIs

And the class is smaller than it may have been

So: week 9 will be Desktop GUIS with wxPython

And we'll do all the student presentations on week 10: May 28th

BlueBox VMs

Does anyone still need their BlueBox VMs?

(Conor, you're all set)

class creation

What happens when a class instance is created?

```
class Class(object):
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2
        .....
```

- A new instance is created
- `__init__` is called
- The code in `__init__` is run to initialize the instance

class creation

What if you need to do something before creation?
Enter: `__new__`

```
class Class(object):
    def __new__(cls, arg1, arg2):
        some_code_here
        return cls()
        .....
```

- `__new__` is called: it returns a new instance
- The code in `__new__` is run to pre-initialize
- `__init__` is called
- The code in `__init__` is run to initialize the instance

class creation

`__new__` is a static method – but it must be called with a class object as the first argument. And it should return a class instance:

```
class Class(superclass):
    def __new__(cls, arg1, arg2):
        some_code_here
        return superclass.__new__(cls)
        .....
```

- `__new__` is called: it returns a new instance
- The code in `__new__` is run to pre-initialize
- `__init__` is called
- The code in `__init__` is run to initialize the instance

class creation

When would you need to use it:

- subclassing an immutable type:
  – It's too late to change it once you get to `__init__`
- When `__init__` not called:
  - unpickling
  - copying

  You may need to put some code in `__new__` to make sure things go right

More detail here: `http://www.python.org/download/releases/2.2/descrintro/#__new__`

LAB

Demo: `code/__new__/new_example.py`

Write a subclass of int that will always be an even number: round the input to the closest even number

`code/__new__/even_int.py`

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 classes `__init__` are optional – case dependent)

multiple inheritance

Method Resolution Order – left to right

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

(This can get complicated...more on that later...)

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: `wxPython FloatCanvas`

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, velocit
```

but super is about more than just DRY...

Remember the method resolution order?

What does super() do?

`super` returns a
"proxy object that delegates method calls"

It's not returning the object itself – but you can call methods on it

It runs through the method resolution order (MRO) to find the method you call.

Key point: the MRO is determined *at run time*

`http://docs.python.org/2/library/functions.html#super`

What does super() do?

Not the same as calling one superclass method...
`super()` will call all the sibling superclass methods

```
class D(C, B, A):
    def __init__(self):
        super(D, self).__init__()
```

same as

```
class D(C, B, A):
    def __init__(self):
        C.__init__()
        B.__init__()
        A.__init__()
```

you may not want that...

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....)

super issues...

Both actually say similar things:

- The method being called by super() needs to exist
- Every occurrence of the method needs to use super():
  Use it consistently, and document that you use it, as it is part of the external interface for your class, like it or not.
- The caller and callee need to have a matching argument signature:
  - Never call super with anything but the exact arguments you received, unless you really know what you're doing.
  - When you use it on methods whose acceptable arguments can be altered on a subclass via addition of more optional arguments, always accept *args, **kwargs, and call super like `super(MyClass, self).method(args_declared, *args, **kwar`

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 "

numpy

numpy

Not just for lots of numbers!

(but it's great for that!)

`http://www.numpy.org/`

what is numpy?

An N-Dimensional array object

A whole pile of tools for operations on/with that object.

Why numpy?

Classic answer: Lots of numbers

- Faster
- Less memory
- More data types

Even if you don't have lot of numbers:

- N-d array slicing
- Vector operations
- Flexible data types

why numpy?

Wrapper for a block of memory:

- Interfacing with C libs
- PyOpenGL
- GDAL
- NetCDF4
- Shapely

Image processing:

- PIL
- WxImage
- ndimage

What is an nd array?

- N-dimensional (up to 32!)
- Homogeneous array:
  - Every element is the same type
    (but that type can be a pyObject)
  - Int, float, char – more exotic types
- "rank" number of dimensions
- Strided data:
  - Describes how to index into block of memory
  - PEP 3118 – Revising the buffer protocol

demos: `memory.py` and `structure.py`

Built-in Data Types

- Signed and unsigned Integers
  8, 16, 32, 64 bits
- Floating Point
  32, 64, 96, 128 bits (not all platforms)
- Complex
  64, 128, 192, 256 bits
- String and unicode
  Static length
- Bool
  8 bit
- Python Object
  Really a pointer

demo: `object.py`

Compund dtypes

- Can define any combination of other types
  Still Homogenous: Array of structs.
- Can name the fields
- Can be like a database table
- Useful for reading binary data

demo: `dtypes.py`

Array Constructors:

From scratch:

`ones()`, `zeros()`, `empty()`, `arange()`, `linspace()`, `logspace()`

(Default dtype: np.float64)

From sequences:

`array()`, `asarray()`

(Build from any sequence)

From binary data:

`fromstring()`, `frombuffer()`, `fromfile()`

Assorted linear algebra standards:

`eye()`, `diag()`, `etc.`

demo: `constructors.py`

Broadcasting:

Element-wise operations among two different rank arrays:

Simple case: scalar and array:

```
In [37]: a
Out[37]: array([1, 2, 3])

In [38]: a*3
Out[38]: array([3, 6, 9])
```

Great for functions of more than one variable on a grid

demo: `broadcasting.py`

Slicing – views:

a slice is a "view" on the array:
new object, but shares memory:

```
In [12]: a = np.array((1,2,3,4))
In [13]: b = a[:]
# for lists -- [:] means copy -- not for arrays!
In [15]: a is b
Out[15]: False
# it's new array, but...
In [16]: b[2] = 5
In [17]: a
Out[17]: array([1, 2, 5, 4])
# a and b share data
```

demo: `slice.py`

Working with compiled code:

Wrapper around a C pointer to a block of data

- Some code can't be vectorized
- Interface with existing libraries

Tools:

- C API: you don't want to do that!
- Cython: typed arrays
- Ctypes
- SWIG: numpy.i
- Boost: boost array
- f2py

Example of numpy+cython:
`http://wiki.cython.org/examples/mandelbrot`

numpy persistance:

`.tofile() / fromfile()`
– Just the raw bytes, no metadata

pickle

`savez()` – numpy zip format
Compact: binary dump plus metadata

netcdf

Hdf

- Pyhdf
- pytables

Other stuff:

- Masked arrays
- Memory-mapped files
- Set operations: unique, etc
- Random numbers
- Polynomials
- FFT
- Sorting and searching
- Linear Algebra
- Statistics

(And all of scipy!)

numpy docs:

`www.numpy.org`
– Numpy reference Downloads, etc

`www.scipy.org`
– lots of docs

Scipy cookbook
`http://www.scipy.org/Cookbook`

“The Numpy Book”
`http://www.tramy.us/numpybook.pdf`

Next Week:

Threading / Multiprocessing

– Jeff

And of course, your projects...