Programming in Python

Week 7

Inside Python

Syllabus:
`http://briandorsey.github.com/uwpython_onsite/`

Brian Dorsey
brian@dorseys.org
http://briandorsey.info

Introductions, comments (...)

class design assignment

design is hard

that's OK

a note about today:

pretty deep stuff

mostly want to make sure you've seen it, and know where to look it up later

today:

customizing classes
iterators and generators
decorators

Unit A
(25)

customizing classes

a bit more about classes

what if you need to add behavior later?

getters?

setters?

property()

what if you need to add behavior later?

```
class C(object):
    def __init__(self):
        self.x = None
```

we'll see a cleaner way to write this later today

```
class C(object):
    def __init__(self):
        self._x = None

    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.")
```

http://docs.python.org/library/functions.html#property

staticmethod()

```
class C(object):
    def add(a, b):
        return a + b
    add = staticmethod(add)
```

<-- no self!

```
>>> c = C()
>>> c.add(2, 2)
4
```

`classmethod()`

normal methods always get instance as first parameter

classmethods always get the class as the first parameter

fromkeys()

```
>>> 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}
```

```
# even true from instances!

>>> d = {}.fromkeys([1,2,3])
>>> d
{1: None, 2: None, 3: None}
```

```
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)
```

from http://docs.python.org/howto/descriptor.html

super()

```
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?
```

```
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, icon)
```

super() considered super!

by

Raymond Hettinger

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

special attributes

Magic Method	When it gets invoked (example)	Explanation
`__new__(cls [,...])`	`instance = MyClass(arg1, arg2)`	`__new__` is called on instance creation
`__init__(self [,...])`	`instance = MyClass(arg1, arg2)`	`__init__` is called on instance creation
`__cmp__(self, other)`	`self == other, self > other, etc.`	Called for any comparison
`__pos__(self)`	`+self`	Unary plus sign
`__neg__(self)`	`-self`	Unary minus sign
`__invert__(self)`	`~self`	Bitwise inversion
`__index__(self)`	`x[self]`	Conversion when object is used as index
`__nonzero__(self)`	`bool(self)`	Boolean value of the object
`__getattr__(self, name)`	`self.name # name doesn't exist`	Accessing nonexistent attribute
`__setattr__(self, name, val)`	`self.name = val`	Assigning to an attribute
`__delattr__(self, name)`	`del self.name`	Deleting an attribute
`__getattribute(self, name)`	`self.name`	Accessing any attribute
`__getitem__(self, key)`	`self[key]`	Accessing an item using an index
`__setitem__(self, key, val)`	`self[key] = val`	Assigning to an item using an index
`__delitem__(self, key)`	`del self[key]`	Deleting an item using an index
`__iter__(self)`	`for x in self`	Iteration
`__contains__(self, value)`	`value in self, value not in self`	Membership tests using `in`
`__concat__(self, value)`	`self + other`	Concatenation of two sequences
`__call__(self [,...])`	`self(args)`	"Calling" an instance
`__enter__(self)`	`with self as x:`	`with` statement context managers
`__exit__(self, exc, val, trace)`	`with self as x:`	`with` statement context managers
`__getstate__(self)`	`pickle.dump(pkl_file, self)`	Pickling
`__setstate__(self)`	`data = pickle.load(pkl_file)`	Pickling

double underscores are in charge

"__init__(self)"

how do you say them?

dunder

under-under

dir(2)

dir(list)

demo

you almost always want to define this

```
__repr__(self)
```

not going to say too much more... when you want your code to support certain operations, lookup the magic methods which implement them

Lightning Talks!

(15)

Unit B
(25)

iterators and generators

iterators and generators

iterators are one of the main reasons Python code reads well

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

demo

for loops exposed!

iterators are pretty cool, we can loop over almost anything now

… but what if they don't fit in memory?

... or can't be repeated?

… or are infinitely long?

generators

generators give you the iterator immediately

no access to the underlying data

... if it even exists

demo generator

maybe more?

three ways to implement them

just like Python objects do

implement iterator protocol directly in a custom class

```
class IterateMe(object):
    limit = 3
    current = 0

    def __iter__(self):
        return self

    def next(self):
        self.current += 1
        if self.current > self.limit:
            raise StopIteration
        else:
            return self.current
```

generator.py

```
>>> for i in IterateMe():
...     print i
...
1
2
3
>>>
```

generator comprehension

```
>>> [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
```

yield

demo

generators.py

examples

stdlib examples

files

```
for line in open('secrets.txt'):
print len(line)
```

itertools

```
itertools.cycle()
itertools.chain()
 itertools.tee()

 itertools.izip()
 itertools.imap()
 itertools.ifilter()
 itertools.islice()

itertools.combinations()
itertools.permutations()
```

Lab B (20)

```
git pull

svn update
```

Lab B

- Implement an object whose attributes all return itself. Name it Null.
  `n = Null(); n == n.a == n.b`
- Add support for any method, called with any arguments and keyword arguments. Should return itself.
  n == n.meth(1) == n.meth(a='abc') == n.a
- bonus: print all access as it happens
- double bonus: log all access and provide a method to return it

Null object pattern

http://code.activestate.com/recipes/68205-null-object-design-pattern/

Dummy()

http://code.activestate.com/recipes/576447/

mock object library for testing

action → assertion style

in stdlib starting in 3.3

http://www.voidspace.org.uk/python/mock/

Unit C
(25)
decorators

decorators are wrappers

they let you add code before and after every execution of a function

```
@logged
def add(a, b):
"""add() adds things"""
return a + b
```

demo

motivation

see the PEP for more details

http://www.python.org/dev/peps/pep-0318/

@ decorator operator is an abbreviation

```
@f
def g:
    pass
```

```
def g:
    pass
g = f(g)
```

demo

Memoize

see also:

@functools.lru_cache
(added in 3.2)

http://code.activestate.com/recipes/498245-lru-and-lfu-cache-decorators/

examples

stdlib examples

property

```
class C(object):
    def __init__(self):
        self._x = None

    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.")
```

http://docs.python.org/library/functions.html#property

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

http://docs.python.org/library/functions.html#property

staticmethod()

```
class C(object):
    def add(a, b):
        return a + b
    add = staticmethod(add)
```

```
>>> c = C()
>>> c.add(2, 2)
4
```

```
class C(object):
    @staticmethod
    def add(a, b):
        return a + b
```

```
>>> c = C()
>>> c.add(2, 2)
4
```

cherrypy

```
import cherrypy
class HelloWorld(object):
    def index(self):
        return "Hello World!"
    index.exposed = True

cherrypy.quickstart(HelloWorld())
```

```
import cherrypy
class HelloWorld(object):
    @cherrypy.expose
    def index(self):
        return "Hello World!"

cherrypy.quickstart(HelloWorld())
```

Lab C

(20)

```
git pull

svn update
```

Lab C

- All of these programs should avoid reading the entire file into memory.
- Write a program which reads a file and prints the length of each line and the line.
  What happens to line endings in this mode?
  How does that compare to `file.readlines()`?
- Read a file and print each line with the line number in front of the line.
- bonus: use itertools when doing the previous task.

wrapup

Assignment

Find and install a Python package and report your experience.

Report these items:

1. The name of the package
2. Location where you found the package
3. The format of the downloaded package, if any (.msi, .dmg, .tar.gz, ... or 'none')
4. The installation program you used (distutils, pip, easy_install, apt, macports ...)
5. Whether the installation was successful

If the installation was not successful, you don't have to fix it if you don't want to -- just report that it didn't work. If you do fix it, report that.