Introduction to Python
Decorators – Debugging – Packages and Packaging

Christopher Barker

UW Continuing Education / Isilon

August 22, 2012

Table of Contents

Review of Previous Class

Lightning talk today: Peter

- Some more OO
  - Multiple inheritance / mix-ins
  - Properties
  - `staticmethod` and `classmethod`
  - Special methods ("dunder")
- Iterators
- Generators

Homework review

Who added some classes to some "real" code?

- Multiple inheritance / mix-ins ?
- Property ?
- `staticmethod` or `classmethod` ?
- Special methods ?
- Iterator or Generators ?

Decorators

Decorators are wrappers around functions

They let you add code before an after the execution of a function

Creating a custom version of that function

Decorators

Syntax:

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

Demo and Motivation: `basicmath.py`

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

Decorators

@ decorator operator is an abbreviation:

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

same as

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

"Syntactic Sugar" – but really quite nice

Decorators

demo:

`memoize.py`

Decorator examples

Examples from the stdlib:

Does this structure:

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

look familiar from last class?

Decorator examples

`staticmethod()`

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

Decorator examples

`staticmethod()`

Decorator form:

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

(and `classmethod`)

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.")
```

becomes...

Decorator examples

```
class C(object):
    def __init__(self):
        self._x = None
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
    @x.deleter
    def x(self):
        del self._x
```

Puts the info close to where it is used

examples

CherryPy

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

examples

Pyramid

```
@template
def A_view_function(request)
    .....
@json
def A_view_function(request)
    ......
```

so you don't need to think about what your view is returning...

decorators...

For this class:

Mostly want to you to know how to use decorators that someone else has written

Have a basic idea what they do when you do use them

LAB

- Re-write the properties from last week's `Circle` class to use the decorator syntax (see a couple slides back for an example)
- Write a decorator that can be used to wrap any function that returns a string in a <p> element from the html builder from the previous couple classes (the `P Element` subclass).

Lightning Talk

Lightning Talk:

Peter

Debugging

Debugging

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program

We often spend more time debugging that we do writing the code in the first place

Debugging

Core Message:

Well structured code is less prone to bugs

Well structured code is easier to debug

Types of Bugs

- Syntax Errors
- Run Time Errors
- Logic Errors

(Usually show up in that order)

Syntax Errors

Common Causes

- Mismatched parenthesis, quotes, brackets, etc...
- Missing colons
- "=" vs "=="
- Indentation
- Using a keyword for a variable name

Hint: Make sure you are editing the same file you are running!

Runtime Errors

This may seem obvious, but...

Read the traceback carefully!

- What type of error?
  - ValueError
  - TypeError
  - NameError
  - Think about what that type of error means
- What module/function did it occur in?
- What line did it occur?
- Where was that function called from?

Logic Errors

No hints from the interpreter

- Make sure the code you think is executing is really executing
- Simplify your code
- Boil it down to the simplest version that shows the bug
  - Often you'll find it in the process
- Save (and print) intermediate results from long expressions
- Try out bits of code at the command line (or iPython)

Debugging Tools

Print statements

Interactive debuggers

Logging

Tests

Print Statements

Simple

Easy to understand

Quick (with no compile cycle)

Nice if something fails the 1000th time through a loop...

(I do most of my debugging with print statements)

Logging

“enterprise level print statements”

Standard library logging module

powerful, awesome, and a bit annoying

`http://docs.python.org/library/logging.html`
`http://docs.python.org/howto/logging.html#logging-basic-tutorial`

Logging Module

Using the standard module means you can share your logging with third party packages, etc.

- Customized levels
- String interpolation
- On the fly configuration
- etc, etc..

Logging Module

Output options:

- StreamHandler
- FileHandler
- BaseRotatingHandler
- RotatingFileHandler
- TimedRotatingFileHandler
- SocketHandler
- SMTPHandler
- SysLogHandler
- HTTPHandler
- NullHandler
- ...

Tests

Test Suites Find Bugs

And keep them from recurring

You can get closer to the bug by writing more tests

Tests

Test Suites are particularly helpful for Heisenbugs:

heisenbug: /hi-zen-buhg/, n.

A bug that disappears or alters its behavior when one attempts to probe or isolate it.

`http://www.catb.org/jargon/html/H/heisenbug.html`

More on testing next class

Interactive Debuggers

PDB

- in stdlib
- command line
- local
- in process

PDB

(I've never used any of these much – but ...)

Getting started with pdb (blog post):

`http://pythonconquerstheuniverse.wordpress.com/2009/09/10/debugging-in-python/`
(Nice simple intro)

Python Debugging Techniques

`http://aymanh.com/python-debugging-techniques`

Use pdb to debug Django (screencast):

`http://ericholscher.com/blog/2008/aug/31/using-pdb-python-debugger-django-debugging-series-/`

LAB

PDB lab

-

Modules and Packages

A module is a file with python code in it
A package is a directory with and `__init__.py` file in it
And usually other modules, packages, etc...

```
my_package
    __init__.py
    module_a.py
    module_b.py
```

```
import my_package
```

runs `my_package/__init__.py`

```
import my_package.module_a
```

runs `my_package/__init__.py` and `my_package.module_a.py`

Modules and Packages

```
import sys

for p in sys.path:
    print p
```

(demo)

Installing Python

Linux:

Usually part of the system – just use it

Windows:

Use the `python.org` version:

System Wide

Can install multiple versions if need be

Third party binaries for it.

Installing Python

OS-X:

Comes with the system, but:

- Apple has never upgraded within a release
- There are non-open source components
- Third party packages may or may not support it
- Apple does use it – so don't mess with it.
- I usually recommend the `python.org` version

(Also Macports, Fink, Home Brew...)

Installing Packages

Every Python installation has its own stdlib and `site-packages` folder

`site-packages` is the default place for third-party packages

Finding Packages

The Python Package Index:

PyPi

`http://pypi.python.org/pypi`

Installing Packages

From source (`setup.py install`)

With the system installer (apt-get, yum, etc...)

From binaries:

Windows: MSI installers

OS-X: dmg installers
(make sure to get compatible packages)

`easy_install` and `pip`

LAB

Wrap up

Homework