Scientific Research Computing with Python

Files for Today:

`http://bit.ly/python-seminar-lecture0` = Lecture PDF

```
git clone git@github.com:profjsb/python-seminar.git
```

Band: *Decorate. Decorate* Song: *Surname of Copenhagen*

Welcome to the ***Scientific Research Computing with Python*** Seminar

AY 250: Monday 2-5pm (Wheeler 221)

Instructor: Josh Bloom

GSIs: Isaac Shivvers

Instructor+GSI email: ucbpythonclass+seminar@gmail.com

Course email: python@bspace.berkeley.edu

Award #0941742

Motivation:

short version

get you using Python to do cutting-edge research

long version

1) get you using Python to do cutting-edge research,
2) helping you realize that Python is a viable framework to do just about any 21st century problem well (and costs zero). "Super Glue"
3) fold you into the Python community so it benefits from having you part of it

How we plan to do this:

- "formal" lectures on specialized topics each week by leading experts & local practitioners (Monday)

- "breakout work sessions" interspersed within the lectures

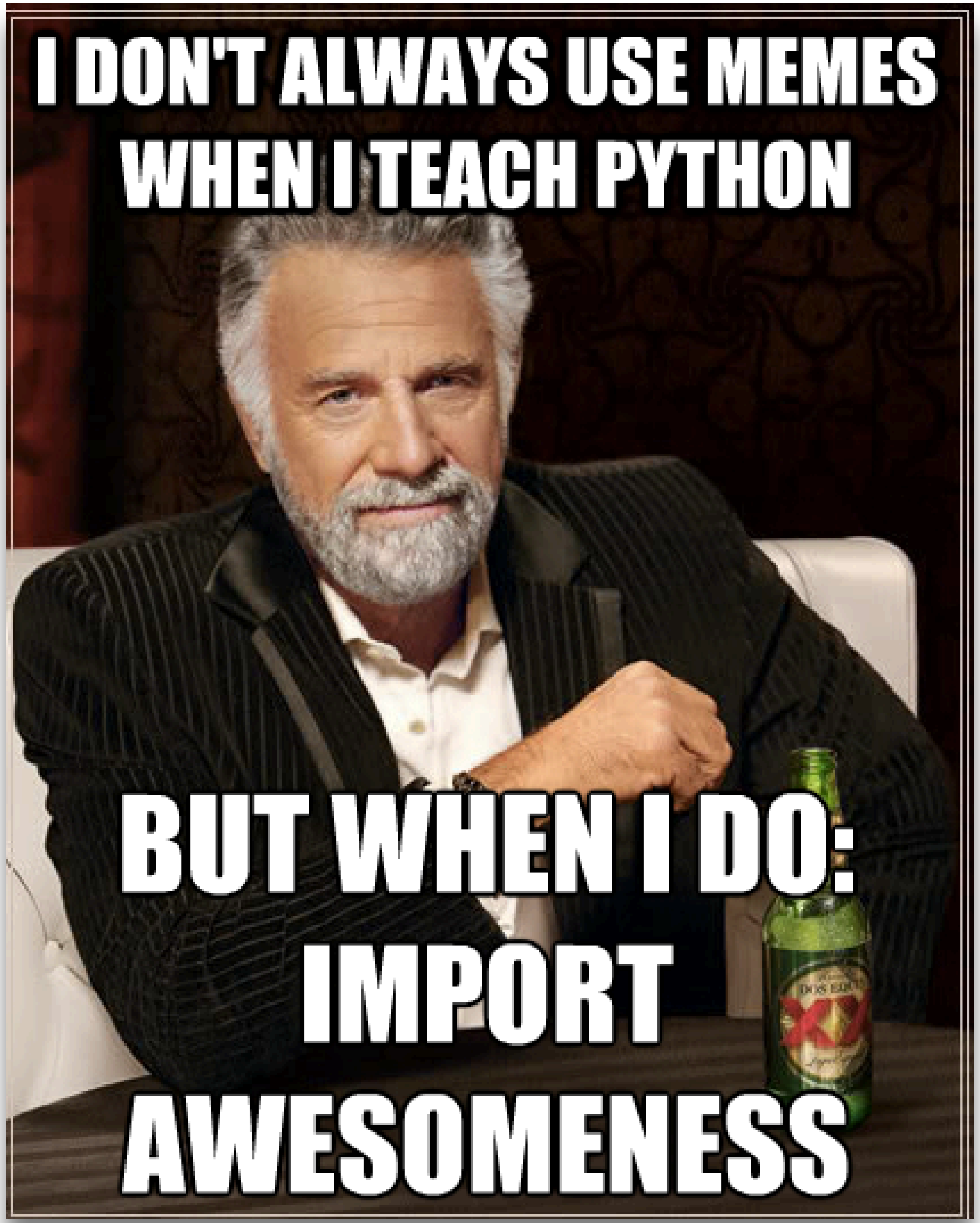
- homework assignments based on week's lecture

- final project

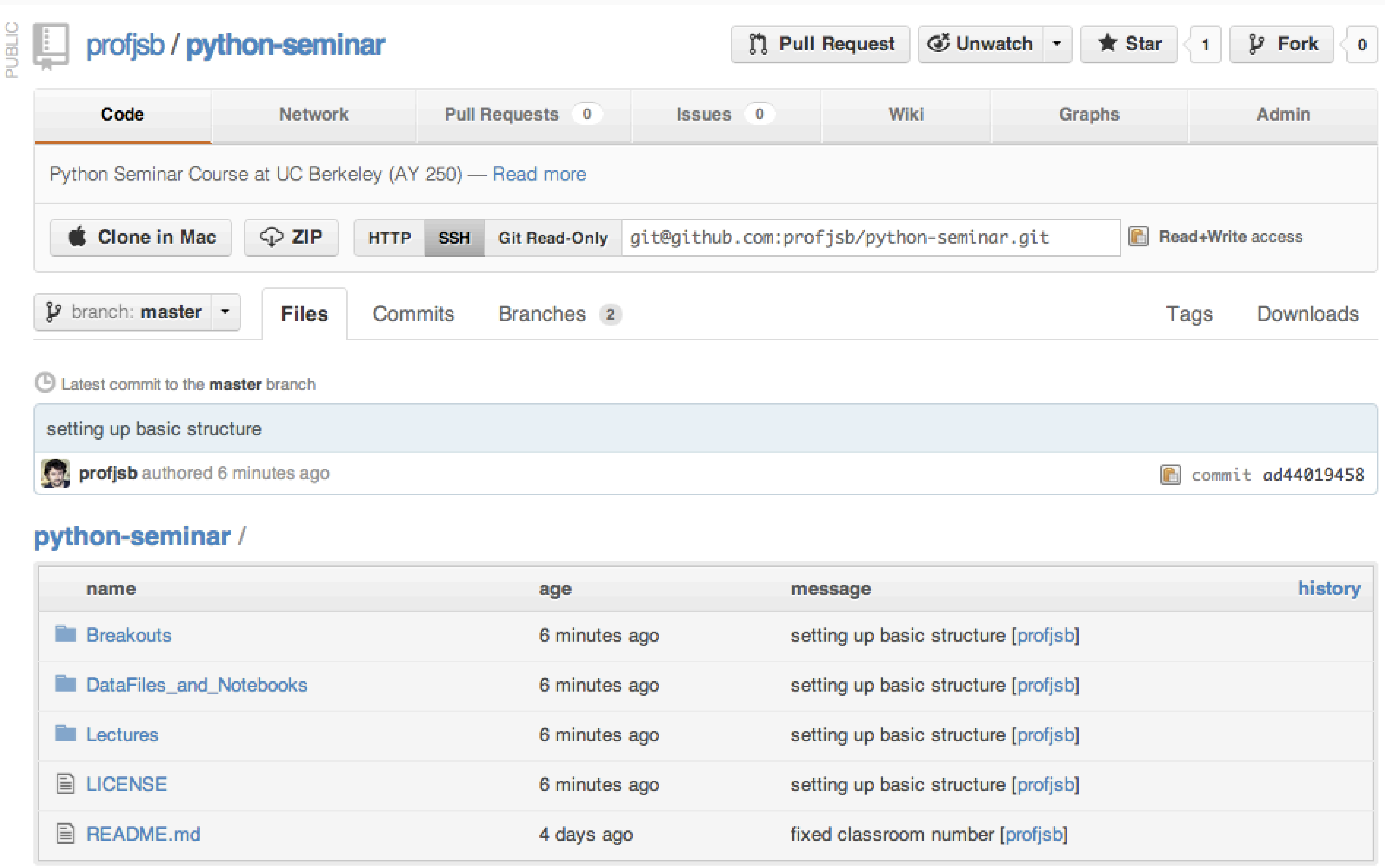
MONTY PYTHON

For awkward, forced laughter.

Bring me
a shrubbery!!
ICANHASCHEEZBURGER.COM


I DON'T ALWAYS USE MEMES WHEN I TEACH PYTHON
BUT WHEN I DO: IMPORT AWESOMENESS

`github` is the main portal for us...



PUBLIC profjsb / python-seminar

Pull Request | Unwatch | Star 1 | Fork 0

Code | Network | Pull Requests 0 | Issues 0 | Wiki | Graphs | Admin

Python Seminar Course at UC Berkeley (AY 250) — Read more

Clone in Mac | ZIP | HTTP | SSH | Git Read-Only | `git@github.com:profjsb/python-seminar.git` | Read+Write access

branch: master | Files | Commits | Branches 2 | Tags | Downloads

Latest commit to the **master** branch

setting up basic structure

profjsb authored 6 minutes ago — commit ad44019458

python-seminar /

name	age	message	history
Breakouts	6 minutes ago	setting up basic structure [profjsb]	
DataFiles_and_Notebooks	6 minutes ago	setting up basic structure [profjsb]	
Lectures	6 minutes ago	setting up basic structure [profjsb]	
LICENSE	6 minutes ago	setting up basic structure [profjsb]	
README.md	4 days ago	fixed classroom number [profjsb]	

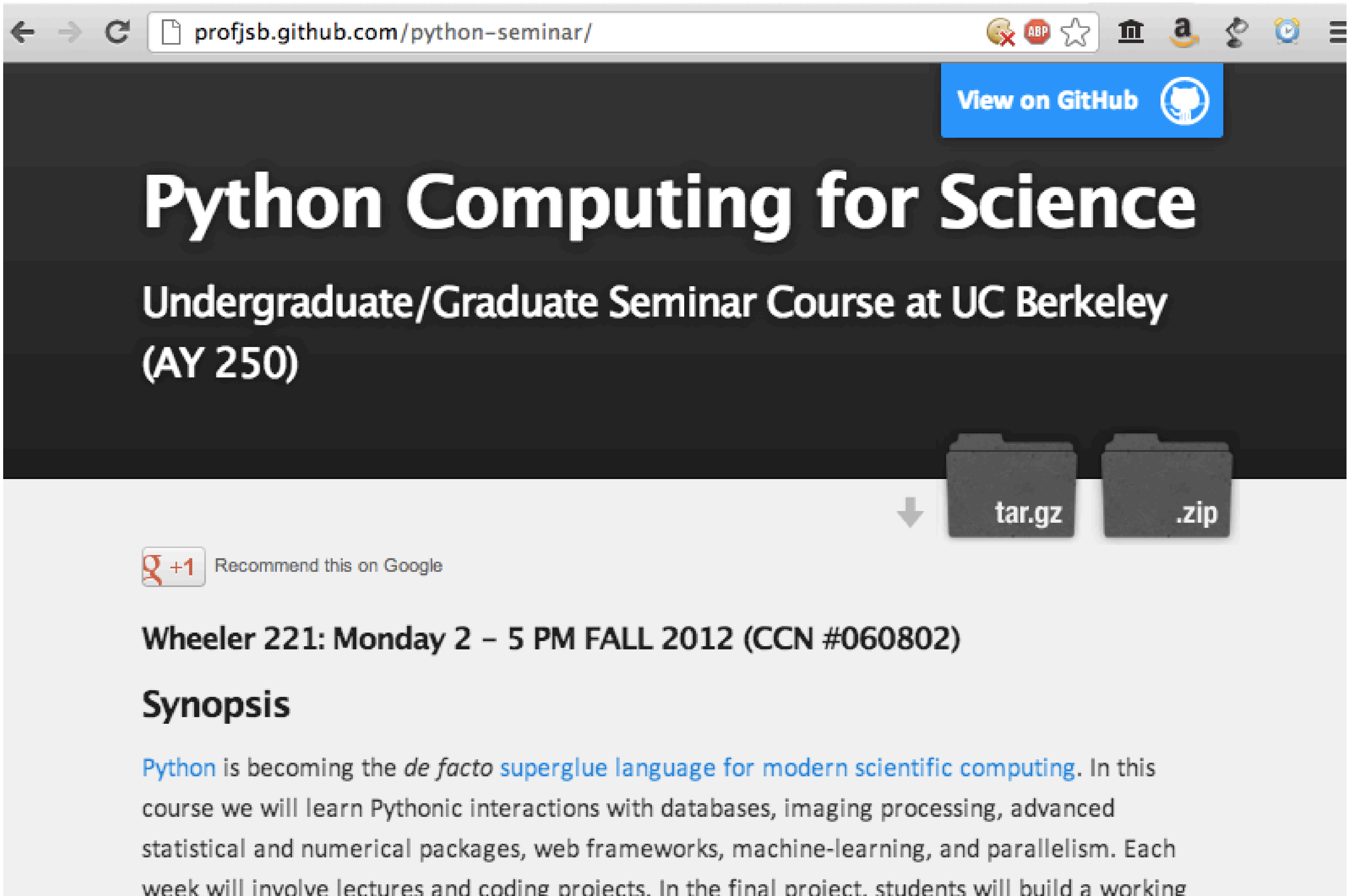
```
git clone git@github.com:profjsb/python-seminar.git
```

Scientific Research Computing with Python

Files for Today:

`http://bit.ly/python-seminar-lecture0`	=	Lecture PDF
/DataFiles_and_Notebooks/00_AdvancedPythonConcepts	=	follow-along files (.tgz)

http://profjsb.github.com/python-seminar/


profjsb.github.com/python-seminar/
View on GitHub
tar.gz
.zip
+1 Recommend this on Google

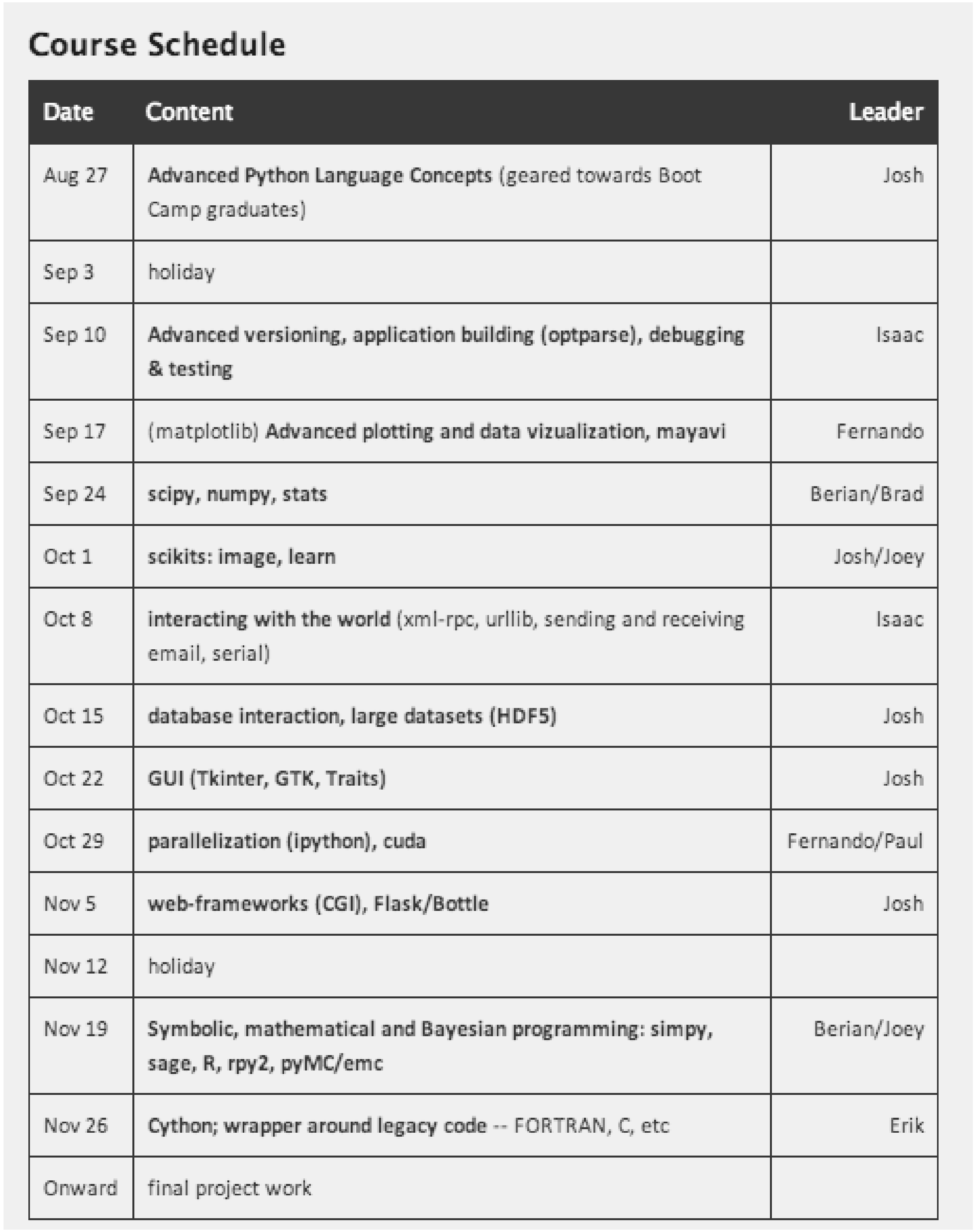
Python Computing for Science

Undergraduate/Graduate Seminar Course at UC Berkeley (AY 250)

Wheeler 221: Monday 2 – 5 PM FALL 2012 (CCN #060802)

Synopsis

Python is becoming the *de facto* superglue language for modern scientific computing. In this course we will learn Pythonic interactions with databases, imaging processing, advanced statistical and numerical packages, web frameworks, machine-learning, and parallelism. Each week will involve lectures and coding projects. In the final project, students will build a working



Course Schedule

Date	Content	Leader
Aug 27	**Advanced Python Language Concepts** (geared towards Boot Camp graduates)	Josh
Sep 3	holiday	
Sep 10	**Advanced versioning, application building (optparse), debugging & testing**	Isaac
Sep 17	(matplotlib) **Advanced plotting and data vizualization, mayavi**	Fernando
Sep 24	**scipy, numpy, stats**	Berian/Brad
Oct 1	**scikits: image, learn**	Josh/Joey
Oct 8	**interacting with the world** (xml-rpc, urllib, sending and receiving email, serial)	Isaac
Oct 15	**database interaction, large datasets (HDF5)**	Josh
Oct 22	**GUI (Tkinter, GTK, Traits)**	Josh
Oct 29	**parallelization (ipython), cuda**	Fernando/Paul
Nov 5	**web-frameworks (CGI), Flask/Bottle**	Josh
Nov 12	holiday	
Nov 19	**Symbolic, mathematical and Bayesian programming: simpy, sage, R, rpy2, pyMC/emc**	Berian/Joey
Nov 26	**Cython; wrapper around legacy code** -- FORTRAN, C, etc	Erik
Onward	final project work	

Workflow for a typical week

Weekend:

email from week's instructor w/ special installation instructions, reading/tutorials

Monday:

2:00	`git pull`
2:10 - 3pm:	Intro topics Lecture
3 - 3:30pm:	Breakout coding
3:30 - 4:30pm:	Detailed topics lecture
4:30-5:	Work on homework

Friday:

3-5 pm: Supervised help with homework
Evans 4th floor (time-series center)

Sunday:

Homework project due

Course Grade

10% participation in lectures/breakouts

60% Homeworks
there will be 11 assignments & we will drop your lowest scores

30% Final Project, due Dec11
(no final exam)

Final Project

a) Build a substantial framework for doing something in your own research, based on at least two topics from different weeks. Something you will use for a long time...

e.g., image analysis package, hardware control software, a webservice that does some crunching under the hood, provide a parallelization of some algorithm or code you use, etc.

- or -

b) Contribute code/functionality to a major open-source Python project (ipython,scipy, Cython,numpy, matplotlib, etc.)

Prerequisites:

- working knowledge (or more) of the core Python language

- and/or -

Python BootCamp graduate

- installation of Python (2.6.5 or 2.7.3), scientific 3rd party packages (EPD distro), & git

- laptop for use in class and for homeworks

- tolerance for our terrible computer humor

`http://www.pythonbootcamp.info/preparation/software`

Useful Advanced Concepts

- ***OrderedDict***
- building ***iterators*** (Classes & generating functions)
- with statements (***Context managers***)
- ***decorators***

OrderedDict

not a *core* type like sets, dict, list, tuple...
but still very useful

```
$ cd DataFiles_and_Notebooks/00_AdvancedPythonConcepts/
$ ipython notebook
```

You should be at least v0.13 for this. If not:

```
sudo enpkg ipython
```

Making Iterables

Python can loop over many different types

```
>>> for element in [1, 2, 3]:
    print element,
1 2 3
>>> for element in (1, 2, 3):
    print element
1 2 3
>>> for key in {'one':1, 'two':2}:
    print key,
one two
>>> for char in "123":
    print char,
1 2 3
>>> for a in {4,1,3,4,2}:
    print a,
1 2 3 4
>>> print {4,1,3,4,2,"a",0j}
set([0j, 'a', 2, 3, 4, 1])
```

Making Iterables

Each of these types have built-in methods.
So do, even, `file` types:

```
>>> for l in open("password.file","r"):
    print l,
# here's some passwords I cracked
guido  Monty
cleese Python
```

Making Iterables

We can make classes that know how to iterate, becoming new iterables types. The key is to build to special methods: `.__iter__()` and `.next()`

`.__iter__()`	return an iterator object (usually just self)
`.next()`	return the next element in the iterator. raise a `StopIteration` exception if there is nothing left

http://docs.python.org/library/stdtypes.html#typeiter file: myits.py

```
class Reverse(object):
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
       self.data = data
       self.index = len(data)
    def __iter__(self):
       return self
    def next(self):
       if self.index == 0:
           raise StopIteration
       self.index = self.index - 1
       return self.data[self.index]
```

```
prompt> ipython
>>> run myits
>>> r = Reverse("god")
>>> for c in r: print c,
d o g
>>> r.next()
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
  File "myits.py", line 10, in next
    raise StopIteration
>>> r = Reverse("rats live on no evil star")
>>> for c in r: print c,
r a t s   l i v e   o n   n o   e v i l   s t a r
```

to the notebook...

Making Iterables

we can also make iterables using *generating functions*

`yield` inside of a function acts like a "temporary `return`" but saves the entire state of the local variables for further use

file: myits.py

```
def countdown(start,end=0,step=1.0):
     i = start
     while (i >= end) or end == None:
          yield i
          i -= step
```

when the function stops yielding, `StopIteration` is raised (implicitly)

```
>>> c = countdown(1)
>>> c
<generator object countdown at 0x13177b0>
>>> c.next()
1
>>> c.next()
0.0
>>> c.next()
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
StopIteration
```

http://stackoverflow.com/questions/231767/the-python-yield-keyword-explained

Fibonacci sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

file: myits.py

```
def fib(start=0,end=None):
    a = long(start)
    b = start + 1L
    while 1 and ((a < end) or (end is None)):
        yield a
        a, b = b, a + b
```

```
>>> a = fib()
>>> for i in range(10): print a.next(),
0 1 1 2 3 5 8 13 21 34
>>> for e in fib(start=-1,end=1000): print e,
-1 0 -1 -1 -2 -3 -5 -8 -13 -21 -34 -55 -89 -144 -233 -377 -610 -987
>>> for e in fib(start=-1,end=10000,maxnum=4): print e,
-1 0 -1 -1
>>> b = fib(start=1,end=10000,maxnum=2)
>>> b.next()
1L
>>> b.next()
2L
>>> b.next()
Traceback (most recent call last):
StopIteration
```

Breakout Work:

The infinite series:

`1 - 1/3 + 1/5 - 1/7 ...`

converges to $\pi/4$

a) write a generator function which progressively makes better and better approximations of π.

b) modify the generator to stop after it reaches within 0.1% of the true value of π. What value do you get?

c) "accelerate" convergence by writing a generator that takes your answer in a) as an argument and returns:

$$S_n = S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2\,S_n + S_{n+1}}$$

PROGRAMMING

```
In [1]: run week1-breakout1.py
3.1447274 ... 0.1% stops after: 319 iterations
Last breakout question:
******************************
fractional accuracy first 8 in accelerated series....:
7.9813e-03  2.6290e-03  1.1604e-03  6.0801e-04  3.5657e-04  2.2642e-04
1.5252e-04  1.0753e-04
------------------------------

fractional accuracy first 8 in un-accelerated series....:
2.7324e-01  1.5117e-01  1.0347e-01  7.8417e-02  6.3054e-02  5.2695e-02
4.5246e-02  3.9636e-02
In [2]:
```

solutions:

https://github.com/profjsb/python-seminar/blob/master/Breakouts/00_AdvancedPythonConcepts/answer.zip?raw=true

Context Managers

allow you to build classes that provide a context to what you do: everything inside of a **`with`** statement operates abides by the context you create. You decide how to build up the context and how to tear it down.

e.g., *holding a lockfile, running a database transaction*

```
>>> with open("password.file","r") as f:
    print f.readlines()
["# here's some passwords I cracked","guido  Monty","cleese Python"]
```

`f.close()` got called for us (and would have even under an exception)

http://www.python.org/dev/peps/pep-0343/

Context Managers

write __enter__() and __exit__() methods.
These get executed no matter what.

file: myctx.py

```
class MyDecor:
    def __enter__(self):
        print "Entered a wonderful technicolor world. Build it up"

    def __exit__(self,*args):    ## *args hold the exception args if needed
        print "...exiting this wonderful world. Tear it down."
```

```
>>> a = MyDecor()
>>> with MyDecor():
...     print " Do something!"
Entered a wonderful technicolor world...
 Do something!
...exiting this wonderful world.
```

***`__enter__`** and **`__exit__`** only get called when invoked with the `with` statement*

file: myctx.py

```python
class MyDecor1:

    def __init__(self,expression="None"):
        self.expression = expression
    def __enter__(self):
        print "Entered a wonderful technicolor world. Build it up"
        return eval(self.expression)
    def __exit__(self,*args):
        print "...exiting this wonderful world. Tear it down."
```

```
>>> with MyDecor1("2**3") as x:
...     print x
Entered a wonderful technicolor world. Build it up
8
...exiting this wonderful world. Tear it down.
>>> with MyDecor1("2") as x:
...     print x/0
Entered a wonderful technicolor world. Build it up
...exiting this wonderful world. Tear it down.
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
```

INTERIOR DECORATORS

Call 1-800-DEVALUE-UR-HOME

Decorators

special functions/classes that augment the functionality of other functions or classes (called in other languages *macros* or *annotations*)

denoted with an @ sign, immediately preceding decorator name, e.g. `@require_login` or `@testinput` file: myctx.py

```
def entryExit(f):
    def new_f():
        print "Entering", f.__name__
        f()
        print "Exited", f.__name__
    return new_f

@entryExit
def func1():
    print "inside func1()"

@entryExit
def func2():
    print "inside func2()"
```

```
>>> func1()
Entering func1
inside func1()
Exited func1
>>> func2()
Entering func2
inside func2()
Exited func2
```

file: myctx.py

```
def introspect(f):
    def wrapper(*arg,**kwarg):
        print "Function name = %s" % f.__name__
        print " docstring = %s" % f.__doc__
        if len(arg) > 0:
            print "   ... got passed args: %s " % str(arg)
        if len(kwarg.keys()) > 0:
            print "   ... got passed keywords: %s " % str(kwarg)
        return f(*arg,**kwarg)
    return wrapper
```

```
>>> myrange(1,10,2)
Function name = myrange
 docstring = None
   ... got passed args: (1, 10, 2)
[1, 3, 5, 7, 9]
```

```
def accepts(*types):
    """ Function decorator. Checks that inputs given to decorated function
      are of the expected type.

      Parameters:
      types -- The expected types of the inputs to the decorated function.
               Must specify type for each parameter.
    """
    def decorator(f):
        def newf(*args):
            assert len(args) == len(types)
            argtypes = tuple(map(type, args))
            if argtypes != types:
                a = "in %s "  % f.__name__
                a += "got %s but expected %s" % (argtypes,types)
                raise TypeError, a
            return f(*args)
        return newf
    return decorator
```

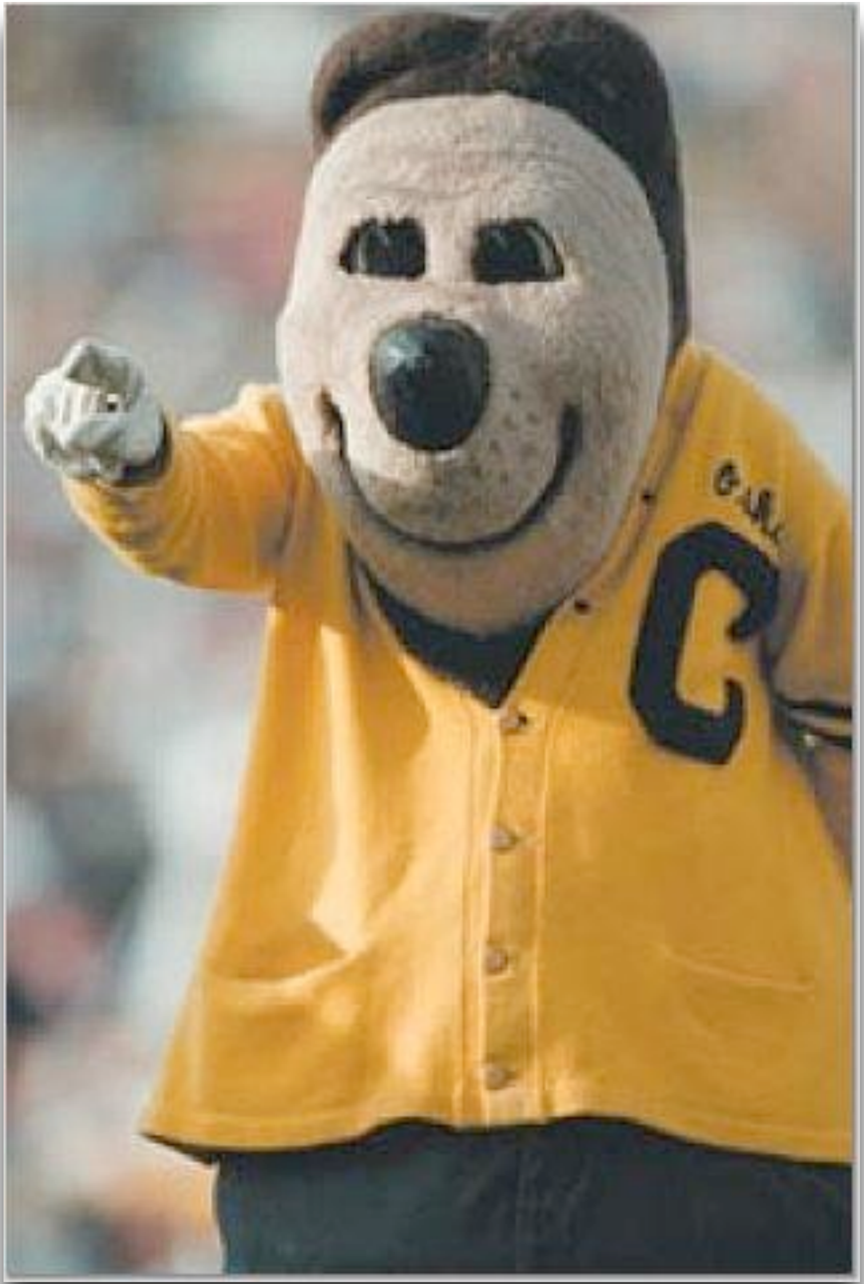
file: myctx.py

```
>>> @accepts(int,int,int)
... def myrange(start,stop,step): return range(start,stop,step)
>>> myrange(1,10,1)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> @accepts(long,int,int)
def myrange(start,stop,step): return range(start,stop,step)
>>> myrange(1,10,1)
TypeError: in myrange got (<type 'int'>, <type 'int'>, <type 'int'>) but
expected (<type 'long'>, <type 'int'>, <type 'int'>)
>>> myrange(1L,10,1)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A Little Teaser: the Google App Engine

```
def loginRequired(func):
  def wrapper(self, *args, **kw):
    user = users.get_current_user()
    if not user:
      self.redirect(users.create_login_url(self.request.uri))
    else:
      func(self, *args, **kw)
  return wrapper
```

```
class NewProject(webapp.RequestHandler):
   @loginRequired
    def get(self):
        user = users.GetCurrentUser()
        tmp = db.GqlQuery("SELECT * FROM SiteUser WHERE login_user  = :1", user).get()
        if not tmp:
            ## this dude is logged into Google, but he/she aint part of our project
            self.redirect("/newsiteuser")
        path = 'templates/newproject.html'
        self.response.out.write(template.render(path, {}))
        return
```


O.B.
C

Homework #0

due Sept 9, 2012 @ 5pm

Build a Bear population over 150 years, starting from bear cubs named Adam, Eve, and Mary.

<u>Rules</u>:

- Bears live for an average of 35 years (1 sigma = 5 years)

- Bears procreate starting at 5 years old until death (assume no gestation period). They produce no more than 1 cub every five years.

- No Bear can procreate with another Bear that has the same mother and father and must procreate with other Bears that are within 10 years of their own age and of the opposite sex.
- Male and Female cubs are equality likely: *P(male)* = 1 - *P(female)* = 0.5

- No new cubs can be named the same as Bears that are currently alive in the population

Homework #0

due Sept 9, 2012 @ 5pm

Questions:

a. On average, how many Bears are born in the first 100 years? How many Bears are alive at the end of 150 years?
b. What must be the minimum value of *P(male)* such that the population does not die out in 150 years?
c. Build and use a plotting routine to show the genealogy tree of a given Bear. Show all Bears at the same generation and earlier who are directly related.

Hints:

- use numpy.random to satisfy your random needs
- use a webservice to build a name generating function
- play around with *networkx* to help you build a genealogy tree

Enjoy!

Help online:
ucbpythonclass+seminar@gmail.com
or in person:
Friday 3-5pm Evans 4th floor (481)

See you in *two* Mondays from now!

remember to email us if you are "sitting in"...