Python:
Introduction and Basics

David Green
UMD NASA/GSFC

Objectives

- Introduce you to the Python language
- Get you writing Python code.
- Convince you its utility in your research life
- Instill good coding and curation practices
- We try not to proselytize, but sometimes it's too hard to resist

Organization

- 5 lectures
    - Basics, Data structures, Functions and Strings, Advanced misc, Science packages (numpy, pylab, scipy, ect…)
- Breakout coding sessions after each lecture
- Homework (small code projects)
- Blood, sweat, tears → a more productive you!

What is

python™

?

What is Python?

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

http://www.python.org/doc/essays/blurb/

What is Python?

interpreted	no need for a compiling stage
object-oriented	programming paradigm that uses objects (complex data structures with methods)
high level	abstraction from the way machine interprets & executes
dynamic semantics	can change meaning on-the-fly
built in	core language (not external)
data structures	ways of storing/manipulating data
script/glue	programs that control other programs
typing	the sort of variable (int, string)
syntax	grammar which defines the language
library	reusable collection of code
binary	a file that you can run/execute

Development History

- Open-sourced development from the start (BSD licensed now)
  http://www.opensource.org/licenses/bsd-license.php
- Relies on large community input (bugs, patches) and 3rd party add-on software
- Version 2.0 (2000), 2.6 (2008), 2.7 (2010).
  We're using **2.7.3** in this class
- Version 3.X (2008) is not backward compatible with 1.X & 2.X. But 2.7 code is "easily" migrated to 3.X

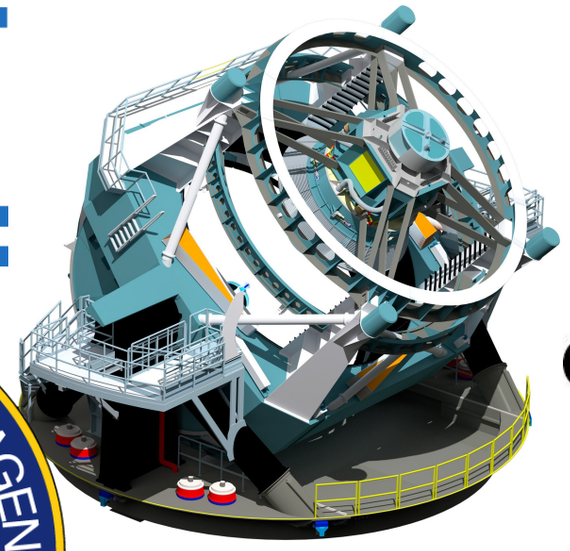
Why Python?

- **C, C++, FORTRAN**
  - Pros: great performance, backbone of legacy scientific computing codes
  - Cons: syntax not optimized for causal programming, no interactive facilities, difficult visualization, text processing, etc.
- **Mathematica, Maple, MATLAB, IDL**
  - Pros: Interactive, great visuals, extensive libraries
  - Cons: Costly, proprietary, unpleasant for large-scale programs and non-mathematical task

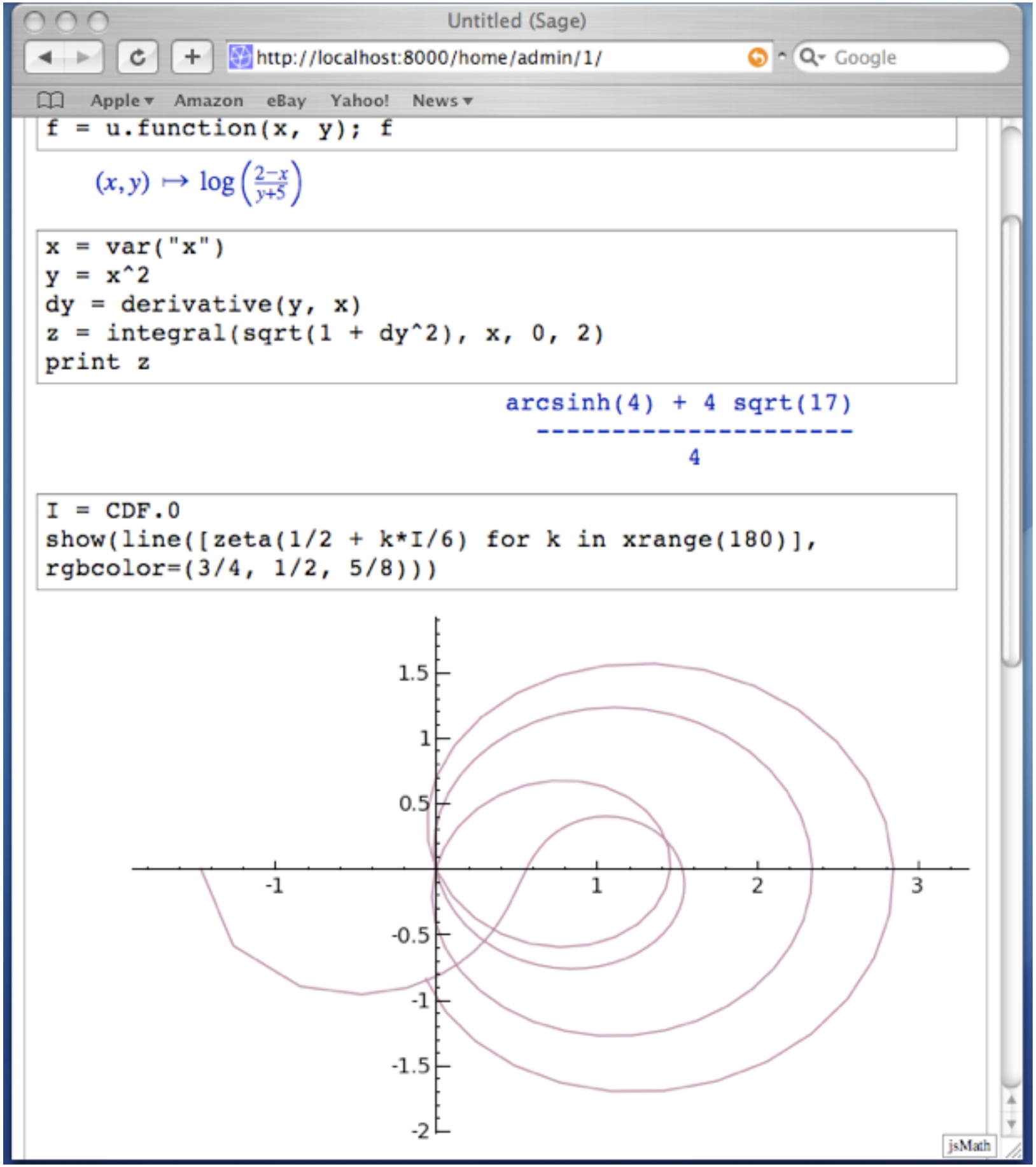
Why Python?

- Free, high portable (Linux, OSX, Windows, ect…)
- Interactive interpreter provided
- Extremely readable syntax (“executable pseudo-code”)
- Simple: non-professional programs can use it effectively
    - Great documentation
    - Total abstraction of memory management
- Rich build-in types: lists, sets, dictionaries (hash tables), strings, ect…
- Standard libraries for IDL/MATLAB-like arrays (NumPy)
- Easy to wrap existing C, C++, and FORTRAN codes

Who uses Python?



Interactive Notebooks



127.0.0.1:8888/bdfa2fd0-5df7-4c66-a205-8c9aa2b1a109

IP[y]: Notebook

Untitled0 Save QuickHelp

Notebook

Actions: New Open Download ipynb Print

Cell

Actions: Delete

Format: Code Markdown

Output: Toggle ClearAll

Insert: Above Below

Move: Up Down

Run: Selected All

Autoindent:

Kernel

Actions: Interrupt Restart

Kill kernel upon exit:

Help

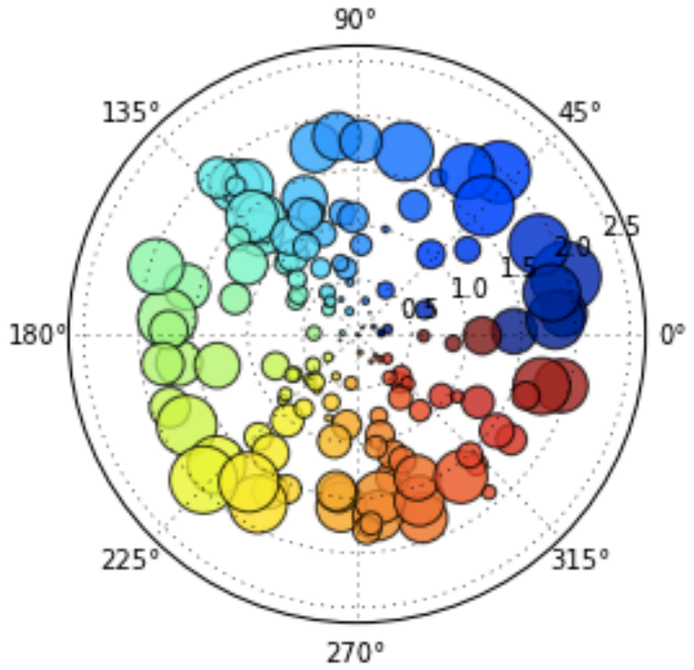
Links: Python IPython NumPy SciPy MPL SymPy

Shift-Enter : run selected cell
Ctrl-Enter : run selected cell in-place
Ctrl-m h : show keyboard shortcuts

Configuration

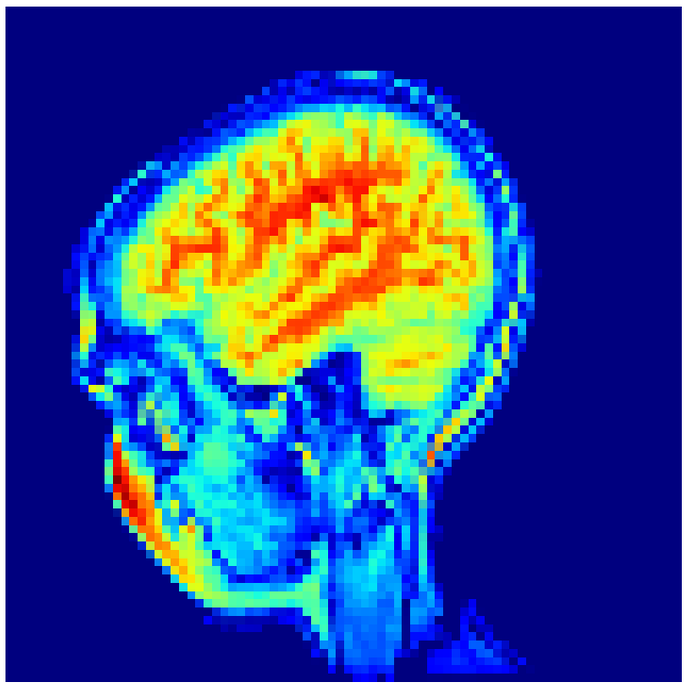
Tooltip on tab:

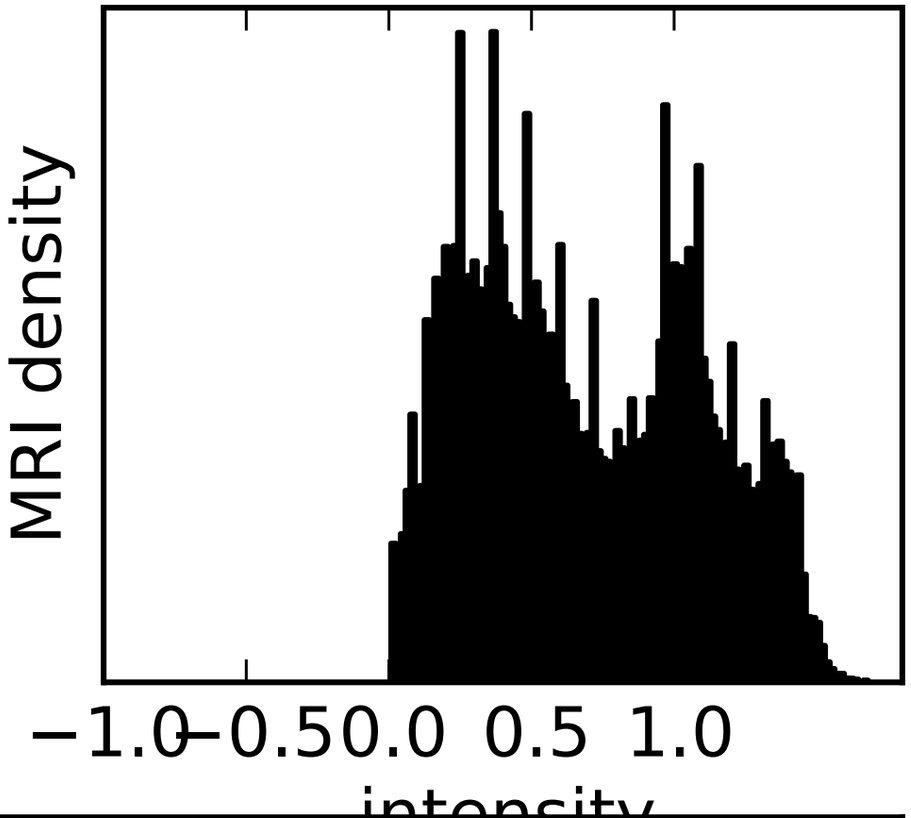
In [1]:
```
num_bootcampers=142
# note: rand not Ron
r = 2*rand(num_bootcampers)
theta = 2*pi*rand(num_bootcampers)
area = 200*r**2*rand(num_bootcampers)
colors = theta
ax = subplot(111, polar=True)
c = scatter(theta, r, c=colors, s=area)
c.set_alpha(0.75)
```

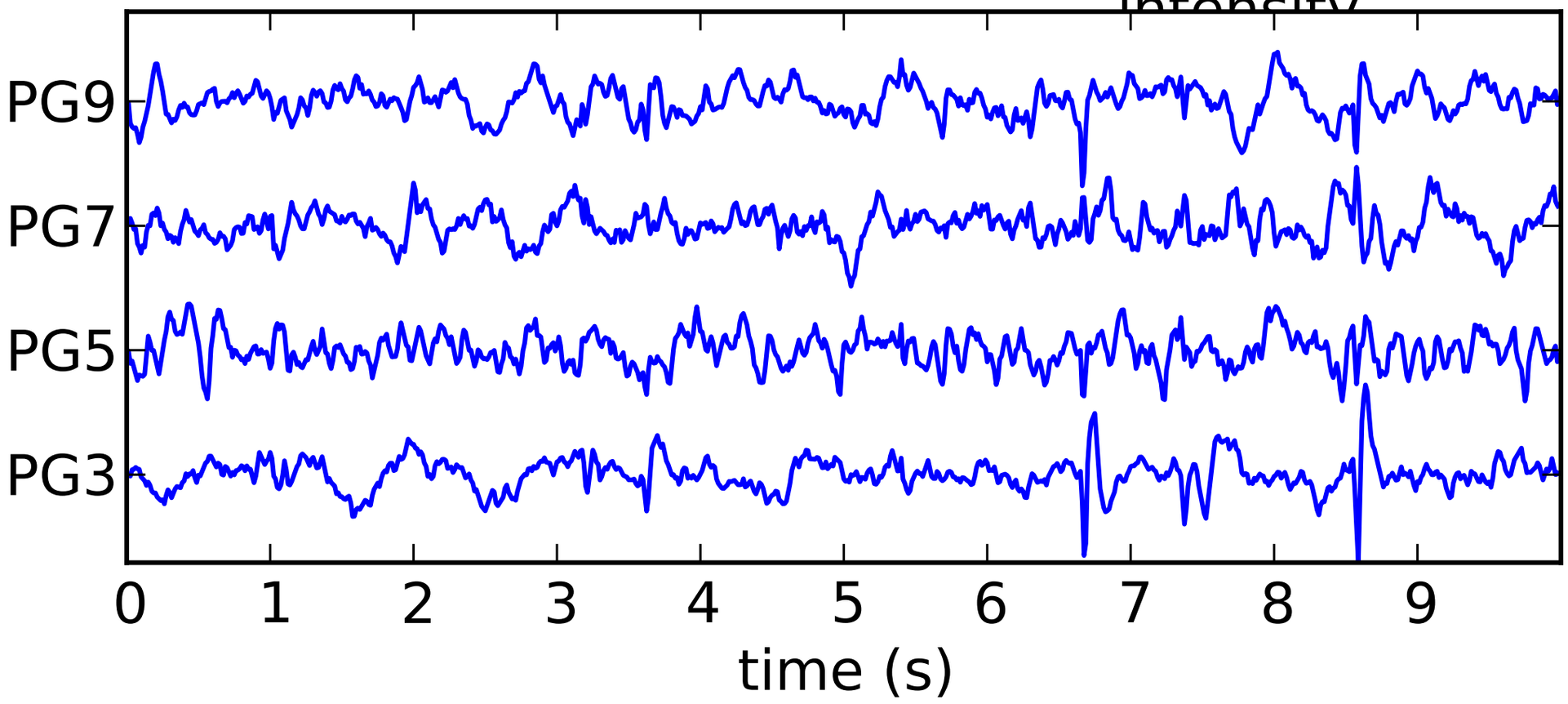


In [ ]:

iPython notebook

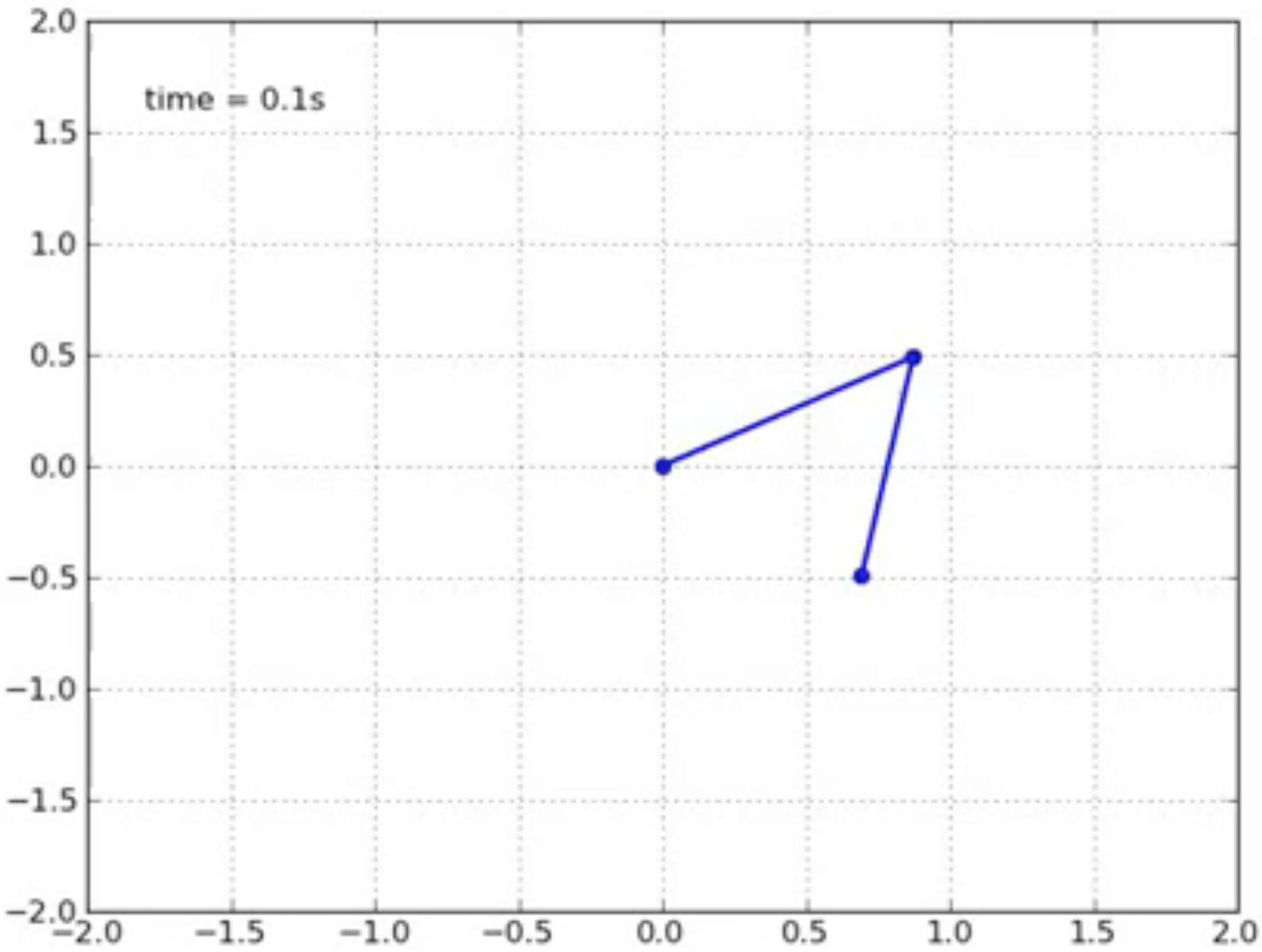



MRI density
−1.0
−0.5
0.0
0.5
1.0
intensity


PG9
PG7
PG5
PG3
0
1
2
3
4
5
6
7
8
9
time (s)

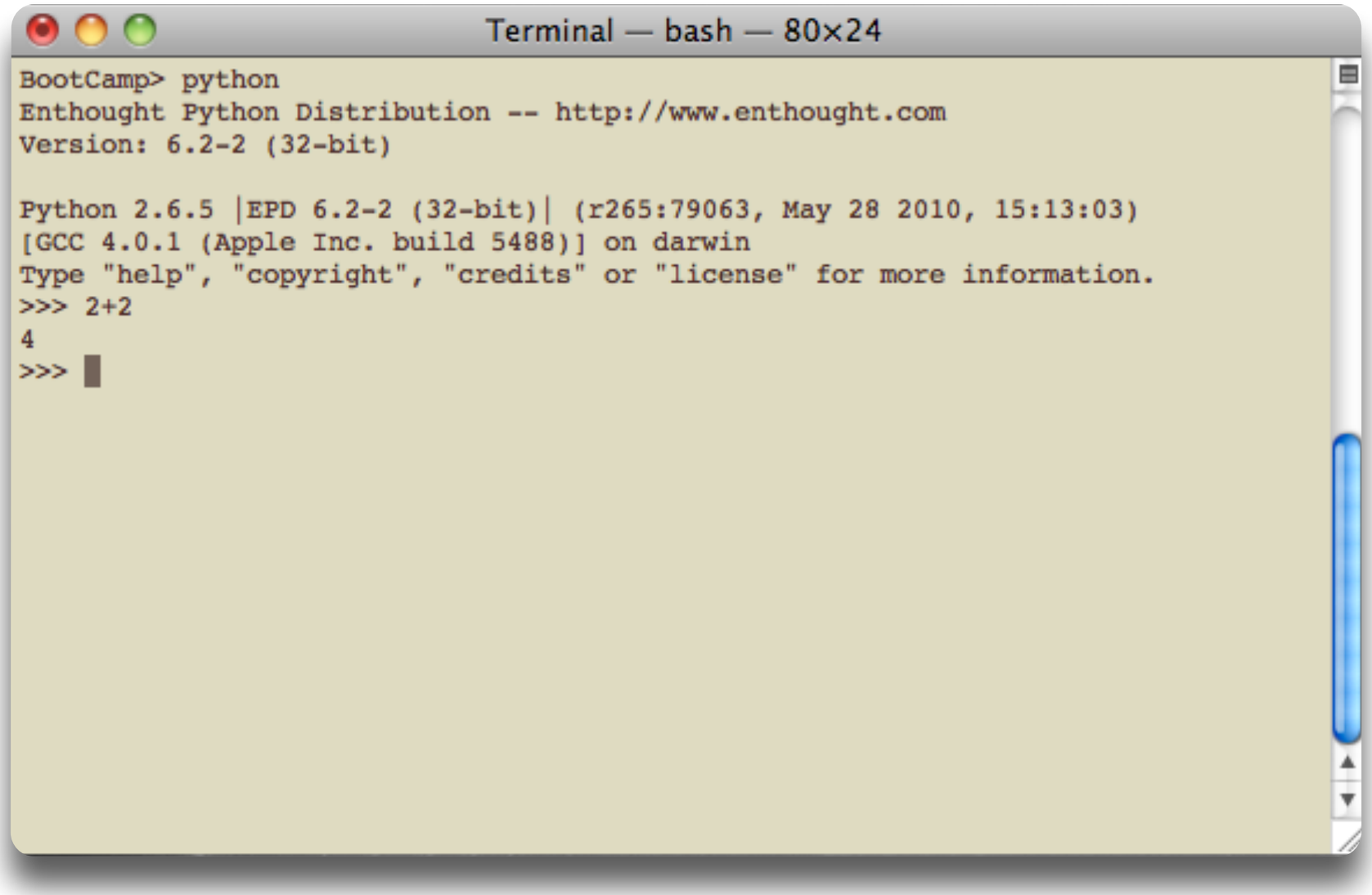
matplotlib

Animations



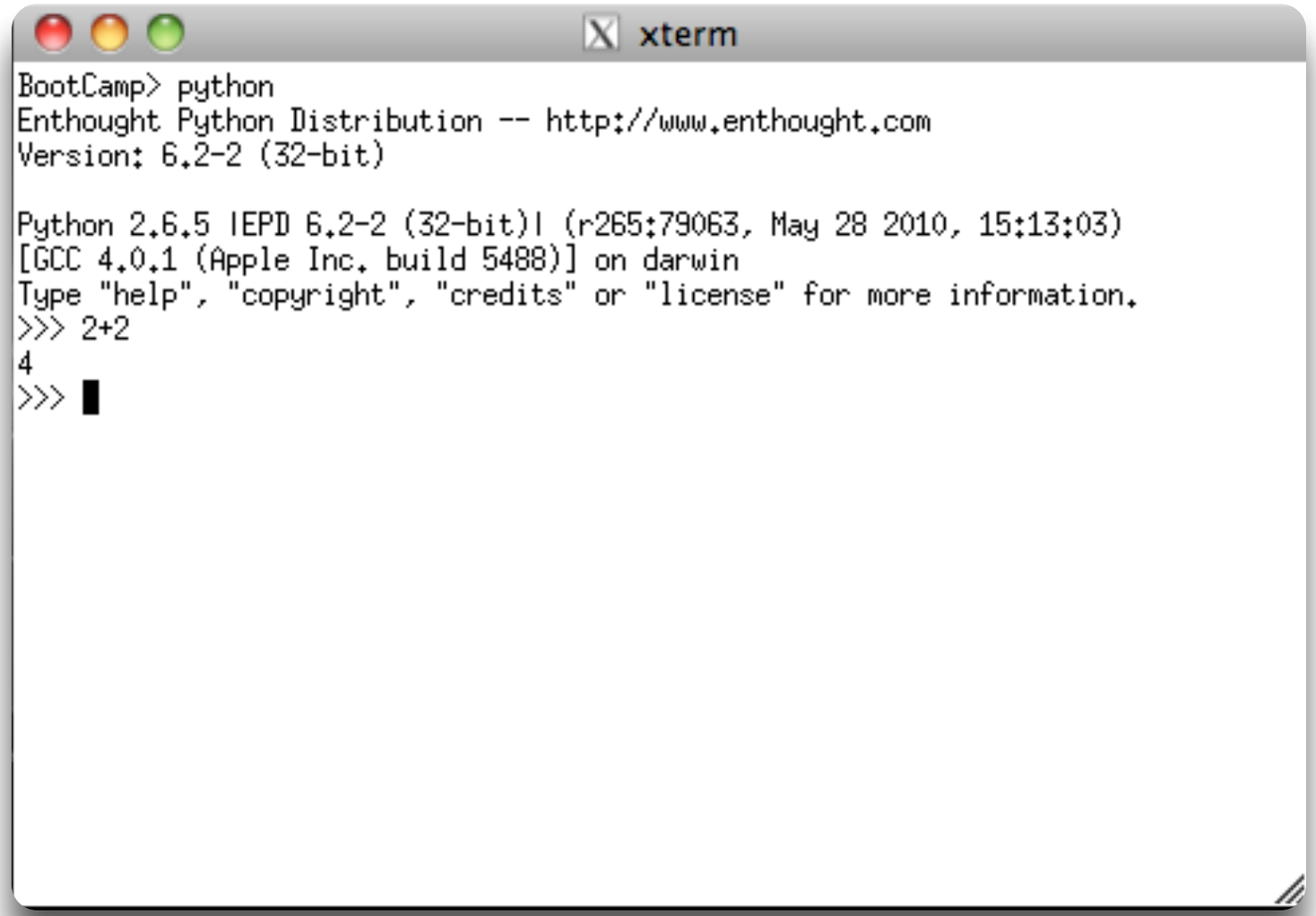
Ask Questions!!!

Firing up the interpreter



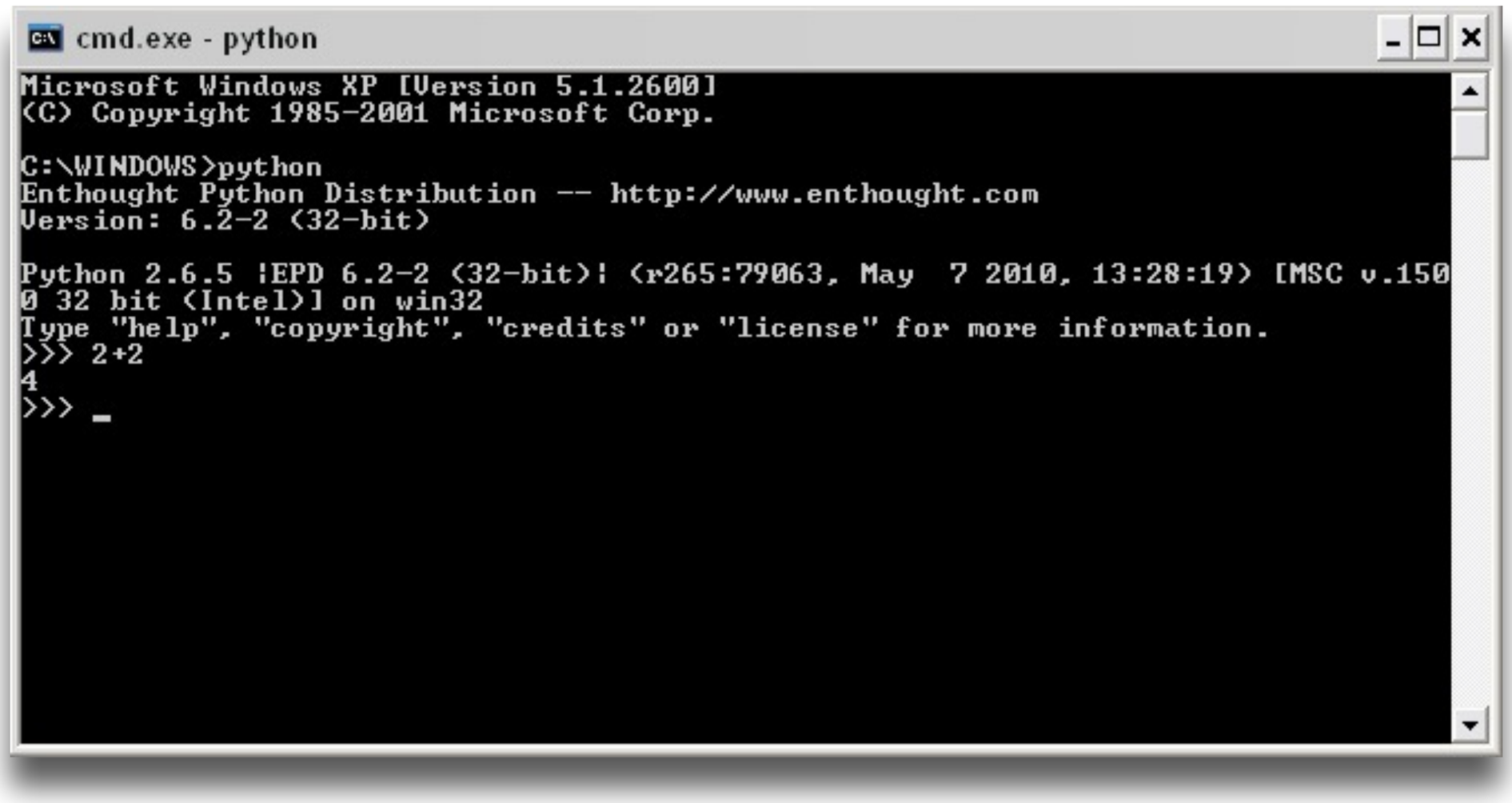
OS X (terminal)

Firing up the interpreter



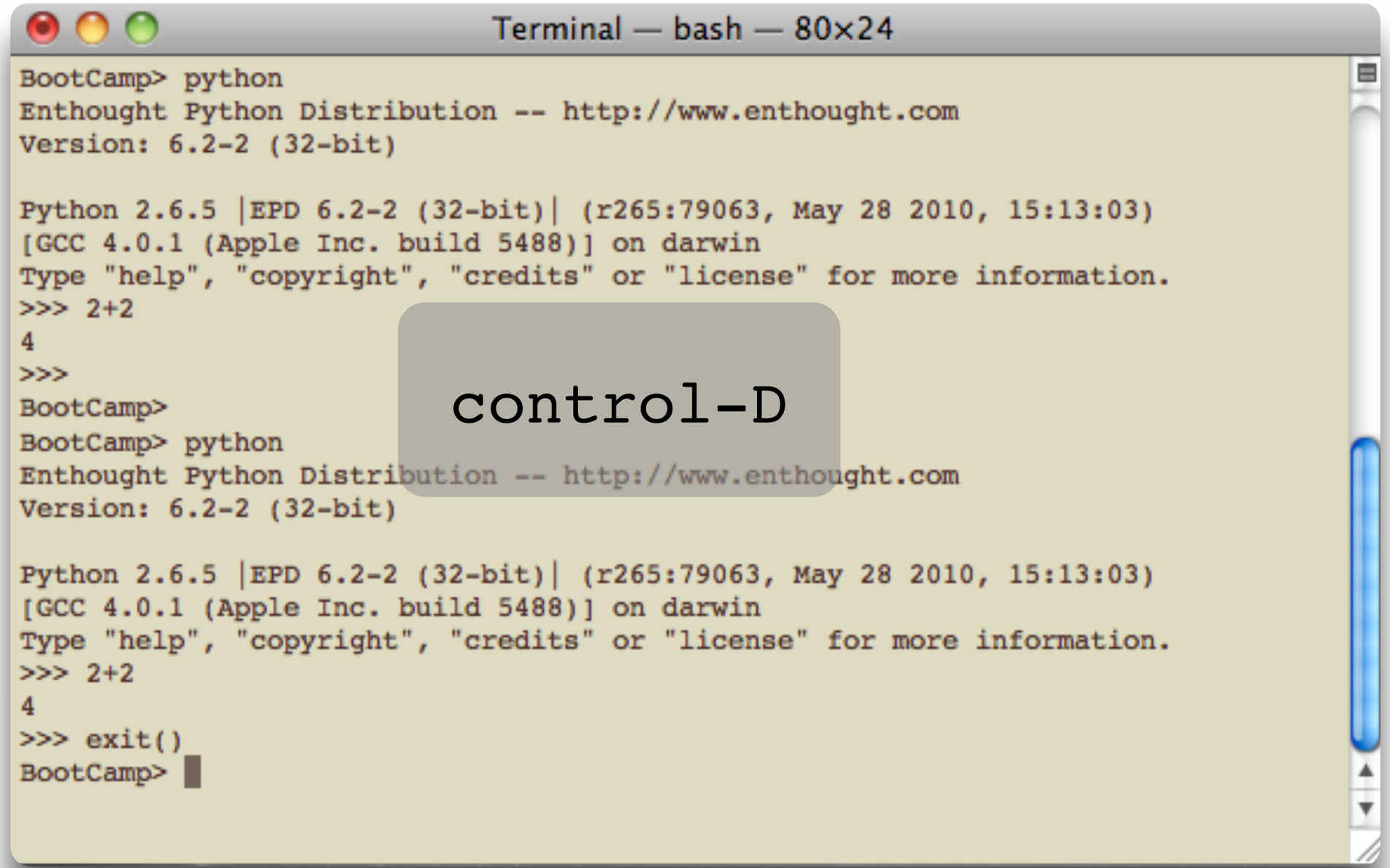
Linux/UNIX/Mac OS X (X11/Xterm)

Firing up the interpreter



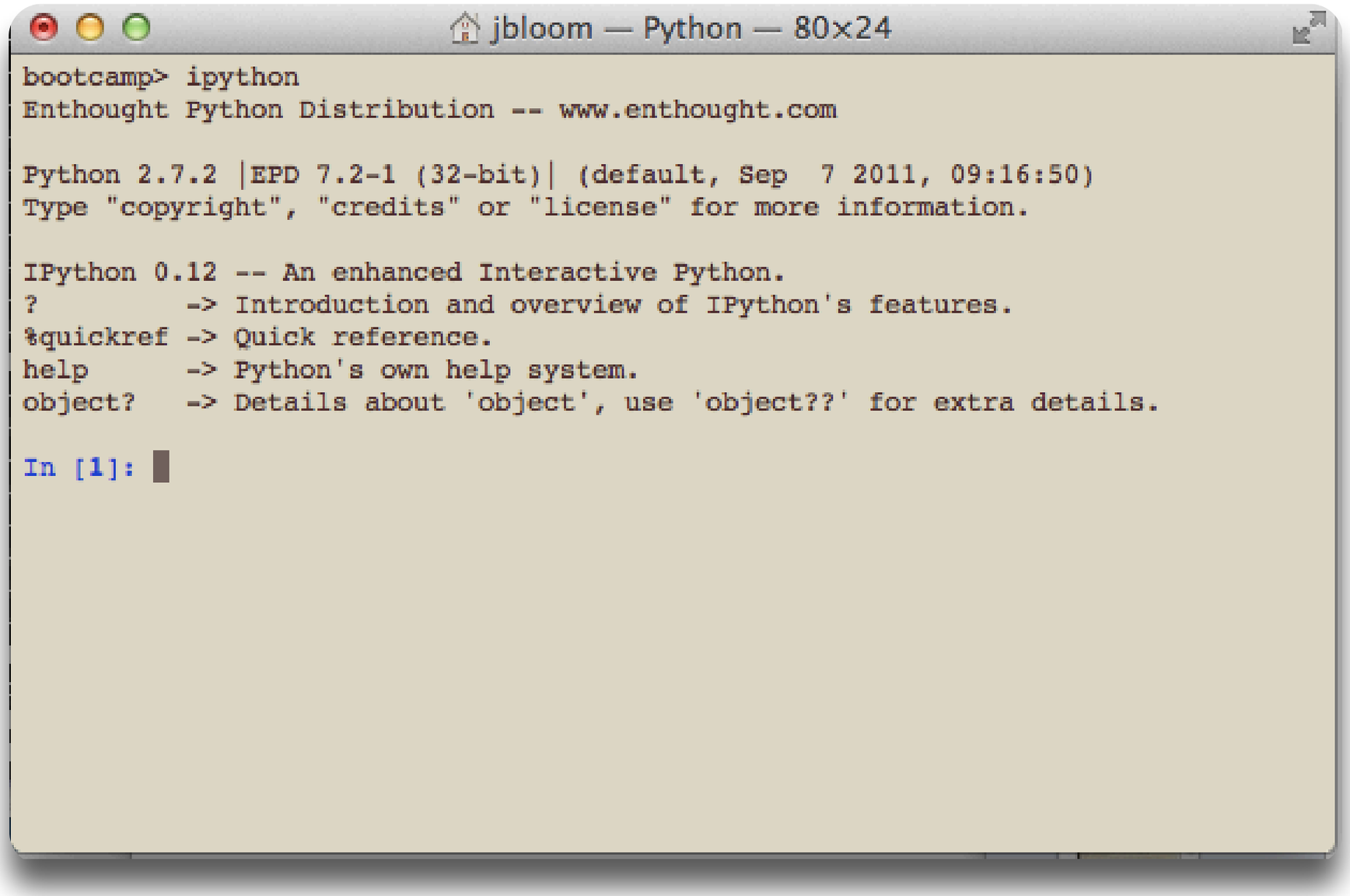
Windows

Firing up the interpreter



to exit: either control-D or exit()

Firing up the interpreter

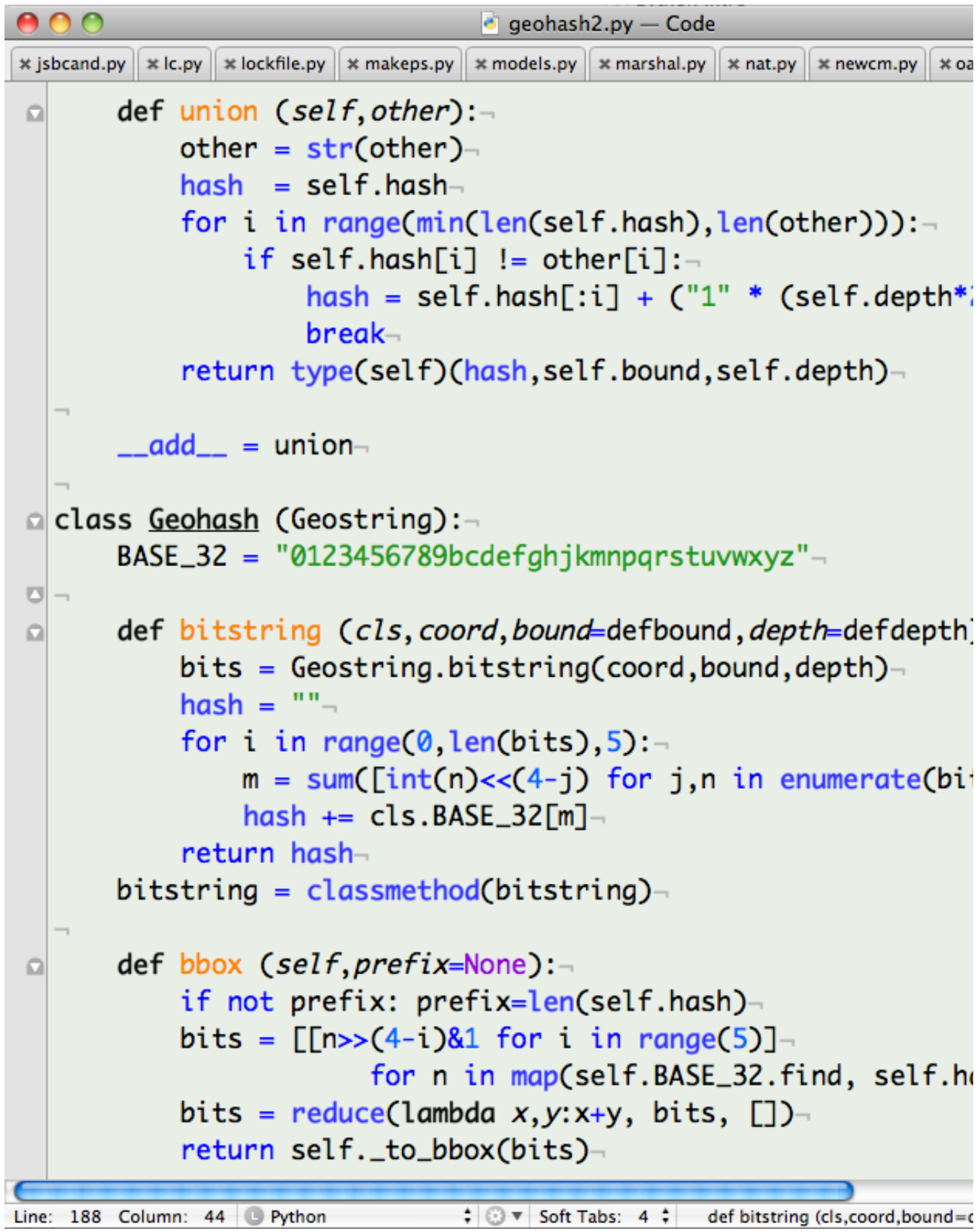


iPython

Editing Python Files

usually we name python files with a **.py** suffix

- New hotness
  - BBEdit, TextWrangler (Mac), NotePad++, SublimeText (windows), KWrite, Scribes, eggy (linux), Textmate
- Old standards
  - vim, emacs, nano,ect…



BASIC TRAINING

Outline

- Hello World!
- Calculator/basic math
- Strings
- Variables
- Basic control statements
  - Indentation!

Hello, World.

C++

file: hello.cpp

```
#include <iostream>
int main()
{
  std::cout << "Hello World!" << std::endl;
}
```

```
BootCamp> g++ -o hello hello.cpp
BootCamp> ./hello
Hello World!
BootCamp>
```

FORTRAN

file: hello.f

```
      PROGRAM HELLO
      WRITE (*,100)
      STOP
  100 FORMAT (' Hello World! ' /)
      END
```

```
BootCamp> g77 -o hello hello.f
BootCamp> ./hello
Hello World!
BootCamp>
```

Java

file: hello.java

```
class HelloWorld {
  static public void main( String args[] ) {
    System.out.println( "Hello World!" );
  }
}
```

```
BootCamp> javac hello.java
BootCamp> java HelloWorld
Hello World!
BootCamp>
```

Examples of complied languages

Scripted

file: hello.py

```
print "Hello World!"
```

```
BootCamp> python hello.py
Hello World!
BootCamp>
```

Interactive

```
BootCamp> python
>>> print "Hello World!"
Hello World!
>>>
```

2 Points

1. Python provides both an interactive way to develop code and a way to execute scripts

2. What you do interactively is basically the same things you (can) do in your scripts

Calculator

```
>>> print 2 + 2
4
>>> 2 + 2
4
>>> print 2.1 + 2
4.1
>>> 2.1 + 2 == 4.0999999999999996
True
```

- Python has int and floats (but not natively doubles)
- Python stores floats as their byte representation so it's limited by the same 16-bit issues as most other languages
- In doing calculations, unless you specify otherwise, Python will store the results in the smallest-byte representation

Indentation

- Indentation matters!!!
- When you mess up, python is gentle
- # starts a comment (until the end of the line)

```
>>> 2 + 2
4
>>>  2 + 2
  File "<stdin>", line 1
    2 + 2
    ^
IndentationError: unexpected indent
>>> 2 # this is a comment and is not printed
2
>>> # this is also a comment
>>>
```

handy error message!

Calculator

```
>>> (3.0*10.0 - 25.0)/5.0
1.0
>>> print 3.085e18*1e6  # this is a Megaparsec in units of cm!
3.085e+24
>>> t = 1.0   # declare a variable t (time)
>>> accel = 9.8  # acceleration in units of m/s^2
>>> # distance travelled in time t seconds is 1/2 a*t**2
>>> dist = 0.5*accel*t*t
>>> print dist # this is the distance in meters
4.9
>>> dist1 = accel*(t**2)/2
>>> print dist1
4.9
>>> dist2 = 0.5*accel*pow(t,2)
>>> print dist2
4.9
```

- **Variables** are assigned on the fly
- Multiplication, division, exponents work as you expect

Calculator

```
>>> 6 / 5 ; 9 / 5 #  integer division returns the floor
1
1
>>> 6 % 5 # mod operator
1
>>> 1 << 2 ## shift: move the number 1 by two bits to the left
           ##         that is make a new number 100 (base 2)
4
>>> 5 >> 1 ## shift: move the number 5 = 101 (base 2) one to
           ## to the right (10 = 2)
2
>>> x = 2 ; y = 3  ## assign two variables on the same line!
>>> x | y          ## bitwise OR
3
>>> x ^ y          ## exclusive OR (10 ^ 11 = 01)
1
>>> x & y          ## bitwise AND
2
>>> x = x ^ y ; print x
1
>>> x += 3 ; print x
4
>>> x /= 2.0
>>> print x
2.0
```

We'll see a lot more mathy operators and functions later

Calculator

Relationships

```
>>> # from before dist1  = 4.9 and dist = 4.9
>>> dist1 == dist
True
>>> dist < 10
True
>>> dist <= 4.9
True
>>> dist < (10 + 2j)
---------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/Users/damgreen/<ipython console> in <module>()

TypeError: no ordering relation is defined for complex numbers

>>> dist < -2.0
False
>>> dist != 3.1415
True
```

More on Variables and Types

None, numbers, and truth

```
>>> 0 == False
True
>>> not False
True
>>> 0.0 == False
True
>>> not (10.0 - 10.0)
True
>>> not -1
False
>>> not 3.1415
False
>>> x = None    # None is something special. Not true or false
>>> None == False
False
>>> None == True
False
>>> False or True
True
>>> False and True
False
```

More on Variables and Types

```
>>> print type(1)
<type 'int'>
>>> x = 2 ; type(x)
<type 'int'>
>>> type(2) == type(1)
True
>>> print type(True)
<type 'bool'>
>>> print type(type(1))
<type 'type'>
>>> print type(pow)
<type 'builtin_function_or_method'>
```

We can test whether something is a certain type with isinstance()

```
>>> isinstance(1,int)
True
>>> isinstance("spam",str)
True
>>> isinstance(1.212,int)
False
```

builtin-types: int, bool, str, float,complex, long,….

Strings

- Strings are a sequence of characters
    - They can by indexed and sliced up if they were an array
    - you can glue strings together is + signs
- Strings are **immutable** (unlike in C), so you cannot change a string in place (not as bas as it sounds)
- Strings can be formatted and compared

Strings

```
>>> x = "spam" ; print type(x)
<type "str">
>>> print "hello!\n...my sire."
hello!
...my sire.
>>> "hello!\n...my sire."
'hello!\n...my sire.'
>>> "wah?!" == 'wah?!'
True
>>> print "'wah?!' said the student"
'wah?!' said the student
>>> print "\"wah?!\" said the student"
"wah?!" said the student
```

- backslaches (\) start special (escape) characters:
  - \n = newline (\r = return)
  - \t = tab
  - \a = bell
- String literals are defined with double quotes or quotes.
- The outermost quote type cannot be used inside the stings (unless it's escaped witha backlash)

Strings

```
>>> s = "spam" ; e = "eggs"
>>> print s + e
spameggs
>>> print s + " and " + e
spam and eggs
>>> print "green " + e + " and\n " + s
green eggs and
 spam
>>> print s*3 + e
spamspamspameggs
>>> print "*"*50
**************************************************
>>> print "spam" is "good" ; print "spam" is "spam"
False
True
>>> "spam" < "zoo"
True
>>> "s" < "spam"
True
```

- You can concatenate strings with + sign
- You can do multiple concatenations with the * sign
- Strings can be compared

Strings

```
>>> print 'I want' + 3 + ' eggs and no ' + s
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/Users/damgreen/<ipython console> in <module>()

TypeError: cannot concatenate 'str' and 'int' objects
>>> print 'I want ' + str(3) + ' eggs and no ' + s
I want 3 eggs and no spam
>>> pi = 3.14159
>>> print 'I want ' + str(pi) + ' eggs and no ' + s
I want 3.14159 eggs and no spam
>>> print str(True) + ":" + ' I want ' + str(pi) + ' eggs and no ' + s
True: I want 3.14159 eggs and no spam
```

You must concatenate only strings, coercing (“casting”) others variables to str

Strings

Getting input from the user: always a string response

```
>>> faren = raw_input("Enter the temperature (in Fahrenheit): ")
Enter the temperature (in Fahrenheit): 71
>>> cent = (5.0/9.0)*(faren - 32.0)
...
TypeError: unsupported operand type(s) for -: 'str' and 'float'
>>> faren = float(faren)
>>> cent = (5.0/9.0)*(faren - 32.0) ; print cent
21.6666666667
>>> faren = float(raw_input("Enter the temperature (in Fahrenheit): "))
Enter the temperature (in Fahrenheit): 71
>>> print (5.0/9.0)*(faren - 32.0)
21.6666666667
>>> faren = float(raw_input("Enter the temperature (in Fahrenheit): "))
Enter the temperature (in Fahrenheit): meh!
...
ValueError: invalid literal for float(): meh!
```

Strings

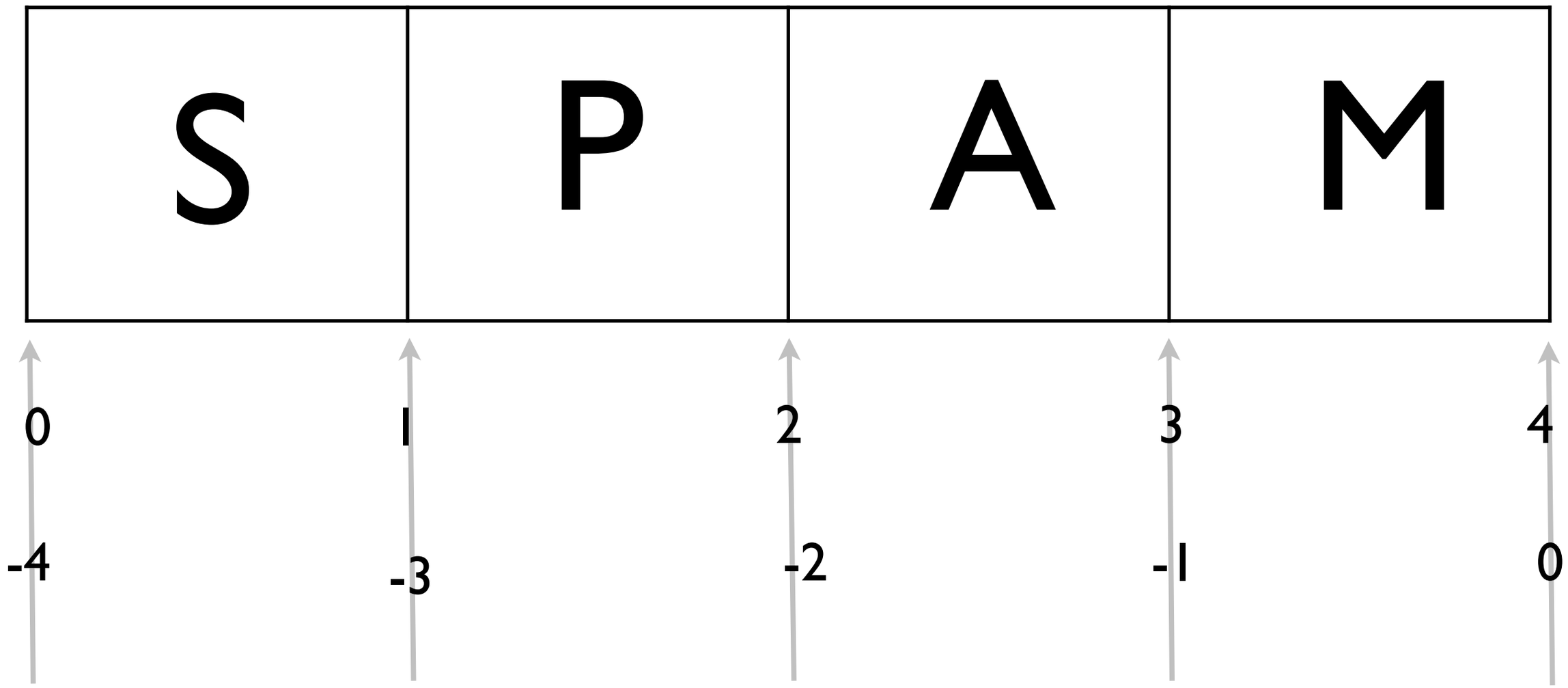
We can think of strings as arrays (although, unlike in C you never really need to deal with directly addressing characters locations in memory)

```
>>> s ="spam"
>>> len(s)
4
>>> len("eggs\n")
5
>>> len("")
0
>>> s[0]
's'
>>> s[-1]
'm'
```

- len() gives us the length of an array
- strings are zero indexed
- Can also count backwards

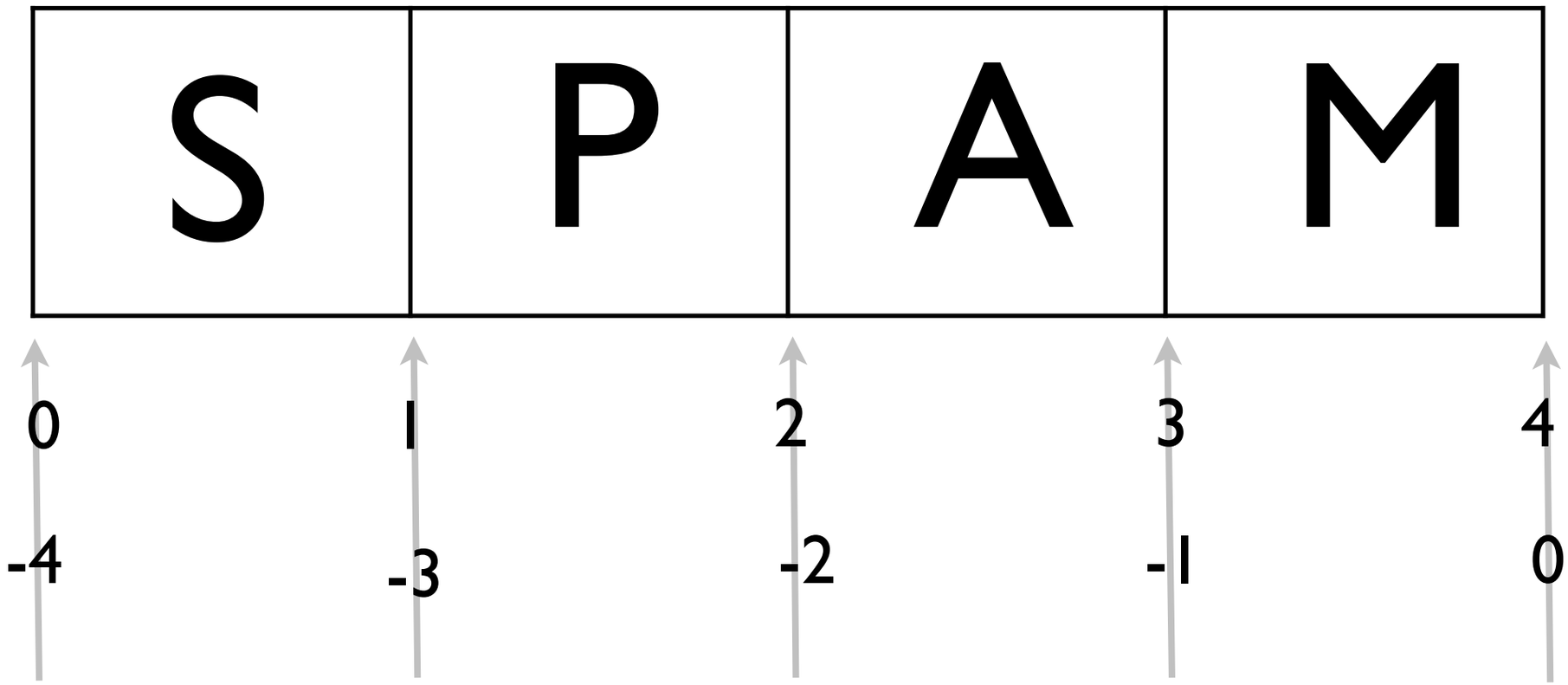
Strings

We can think of strings as arrays (although, unlike in C you never really need to deal with directly addressing characters locations in memory)



Useful for slicing: indices are between the characters

Strings




```
>>> s[0:1]  # get every character between 0 and 1
's'
>>> s[1:4]  # get every character between 1 and 4
'pam'
>>> s[-2:-1]
"a"
>>> ## slicing [m:n] will return abs(n-m) characters
>>> s[0:100] # if the index is beyond the len(str), you dont segfault!
'spam'
>>> s[1:]  # python runs the index to the end
'pam'
>>> s[:2]  # python runs the index to the beginning
'sp'
```

s = s[:n] + s[n:] for all n

Basic Control (Flow)

- Python has pretty much all what you use
  - if, elif, else, for, while, break, and continue
- Does not have:
  - case (explicitiy), goto
- Does have:
  - pass

Basic Control (Flow)

Flow is does within blocks (where indentation matters)

```
>>> x = 1
>>> if x > 0:
        print "yo"
    else:
        print "dude"

yo
```

```
>>> x = 1
>>> if x > 0:
...     print "yo"
... else:
...     print "dude"
...
yo
```

looks like this
(note the ...)

Note colons and indentations (tabbed or spaced)

```
>>> x = 1
>>> if x > 0:
      print "yo"
    else:
            print "dude"

yo
```

Indentation within the same block must be the same but not within different blocks (thought this is ugly)

Basic Control (Flow)

one-liners

```
>>> print "yo" if x > 0 else "dude"
"dude"
```

a small program...

```
>>> x = 1
>>> while True:
        print "yo" if x > 0 else "dude"
        x *= -1
yo
dude
yo
dude
...
yo
dude
yo
^Cdude
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

Control-C usually drops you back to the prompt

Basic Control (Flow)

Blocks cannot be empty

```
>>> x = "fried goldfish"
>>> if x == "spam for dinner":
       print "I will destroy the universe"
    else:
       # I'm fine with that. I'll do nothing

  File "<stdin>", line 5

    ^
IndentationError: expected an indented block
>>>
```

pass is a "do nothing" statement

```
>>> if x == "spam for dinner":
       print "I will destroy the universe"
    else:
       # I'm fine with that. I'll do nothing
     pass

>>>
```

```
# set some initial variables. Set the initial temperature low
faren = -1000

# we dont want this going on forever, let's make sure we cannot have too many attempts
max_attempts = 6
attempt = 0

while faren < 100:
     # let's get the user to tell us what temperature it is
     newfaren = float(raw_input("Enter the temperature (in Fahrenheit): "))
     if newfaren > faren:
             print "It's getting hotter"
     elif newfaren < faren:
             print "It's getting cooler"
     else:
         # nothing has changed, just continue in the loop
         continue
     faren = newfaren # now set the current temp to the new temp just entered
     attempt += 1 # bump up the attempt number
     if attempt >= max_attempts:
         # we have to bail out
         break

if attempt >= max_attempts:
     # we bailed out because of too many attempts
     print "Too many attempts at raising the temperature."
else:
     # we got here because it's hot
     print "it's hot here, man."
```

file: temp1.py

```
BootCamp> python temp1.py
Enter the temperature (in Fahrenheit): 1
It's getting hotter
Enter the temperature (in Fahrenheit): 2
It's getting hotter
Enter the temperature (in Fahrenheit): 3
It's getting hotter
Enter the temperature (in Fahrenheit): 4
It's getting hotter
Enter the temperature (in Fahrenheit): -1
It's getting cooler
Enter the temperature (in Fahrenheit): 10
It's getting hotter
Too many attempts at raising the temperature.
BootCamp>
```

```
BootCamp> python temp1.py
Enter the temperature (in Fahrenheit): 3
It's getting hotter
Enter the temperature (in Fahrenheit): -45
It's getting cooler
Enter the temperature (in Fahrenheit): 101
It's getting hotter
it's hot here, man.
BootCamp>
```

```
# set some initial variables. Set the initial temperature low
faren = -1000

# we dont want this going on forever, let's make sure we cannot have too many attempts
max_attempts = 6
attempt = 0

while faren < 100 and (attempt < max_attempts):
     # let's get the user to tell us what temperature it is
     newfaren = float(raw_input("Enter the temperature (in Fahrenheit): "))
     if newfaren > faren:
             print "It's getting hotter"
     elif newfaren < faren:
             print "It's getting cooler"
     else:
         # nothing has changed, just continue in the loop
         continue
     faren = newfaren # now set the current temp to the new temp just entered
     attempt += 1 # bump up the attempt number

if attempt >= max_attempts:
     # we bailed out because of too many attempts
     print "Too many attempts at raising the temperature."
else:
     # we got here because it's hot
     print "it's hot here, man."
```

file: temp2.py

Exercise for the Breakout

Write a program which allows the user to build up a sentence on word a time, stopping when they under a period (.), an exclamation (!), or a question mark (?)

Example interaction:

```
Please enter a word in the sentence (enter . ! or ? to end.): My
...currently: My
Please enter a word in the sentence (enter . ! or ? to end.): name
...currently: My name
Please enter a word in the sentence (enter . ! or ? to end.): is
...currently: My name is
Please enter a word in the sentence (enter . ! or ? to end.): Slim
...currently: My name is Slim
Please enter a word in the sentence (enter . ! or ? to end.): Shady
...currently: My name is Slim Shady
Please enter a word in the sentence (enter . ! or ? to end.): !
--->My name is Slim Shady!
```