Introduction to Python 2 for statisticians

Will Landau

Iowa State University

November 18, 2013

Outline

Outline

Preliminaries

- Python is a high-level multipurpose interpreted language.
- It's clumsier than R at statistical number crunching, but better than R at string manipulation.
- Open the interpreter by typing `python` into the command line.

```
> python
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)]
      on darwin
Type "help", "copyright", "credits" or "license" for more
     information.
>>>
```

Preliminaries

- You can create variables and do arithmetic:

```
>>> a = 1
>>> b = 2
>>> c = "goober"
>>> c
'goober'
>>> a+b
3
>>>
```

- You can make a python script like `hello_world.py`:

```
s = "Hello World"
print(s)
```

- And run it in the command line:

```
> python hello_world.py
Hello World
```

Preliminaries

- Use the hash sign for single-line comments.

```
>>> # print("Hello world!")
>>>
```

- Triple-quoted strings span multiple lines and serve as multi-line comments.

```
# a.py
"""
This program
does nothing.
"""
```

```
> python a.py
>
```

Preliminaries

- Formatting strings

```
>>> s = "Today's date is {month}/{day}/{year}".format(month = 10, day =
      22, \
...         year = 2012)
>>>
... print(s)
Today's date is 10/22/2012
>>>
```

- Every string has builtin methods such as `format()`, which can be accessed with the dot operator.
- There are multiple ways to format output.

```
>>> a = 3
>>> b = 4.8878
>>> s = format("sample %d: mass= %0.3fg" % (a, b))
>>> print(s)
sample 3: mass= 4.888g
>>>
>>> print("sample %d: mass= %0.3fg" % (a, b))
sample 3: mass= 4.888g
>>>
```

User-defined functions

- I can define and use my own function like this:

```
>>> def f1(a):
...     if a == 0:
...         print("hi")
...         return(0)
...     elif a < 0:
...         print("stop")
...         return(1)
...     else:
...         return(5)
...
>>> f1(0)
hi
0
>>> f1(1)
5
>>> f1(-1)
stop
1
>>>
```

Indentation

- In python, indention is used to denote nested blocks of code (like { and } in C). Indentation must be consistent.
- The following script, `a.py`, has an indentation error.

```
# a.py
def f1(a):
  if a == 0:
    print("hi")
    return(0)
  elif a < 0:
    print("stop")
    return(1)
  else:
  return(5)
```

- When I try to run it,

```
> python a.py
  File "a.py", line 10
    return(5)
          ^
IndentationError: expected an indented block
```

Line continuation

- With the exception of multi-line quotes, you have to use the line continuation character, '\', when you want to wrap text in your code:

```
>>> 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1
29
>>> 1+1+1+1+1+1+1+
  File "<stdin>", line 1
    1+1+1+1+1+1+1+
                 ^
SyntaxError: invalid syntax
>>>
>>> 1+1+1+1+1+1+1+ \
... 1+1+1+1+1+1+1+1
15
>>>
```

Logic and control flow

```
>>> 1 and 2
2
>>> 1 == 1
True
>>> 1 == 0
False
>>> 1 == 1 and 2 == 0
False
>>> 1 > 1 or 2 <= 5
True
>>> not True
False
>>> True and not False
True
>>> if True:
...     print("yes")
... else:
...     print("no")
...
yes
>>>
```

Logic and control flow

```
>>> a = 1
>>> if a == 2:
...     print("two")
... elif a < -1000:
...     print("a is small")
... elif a > 100 and not a % 2:
...     print("a is big and even")
... else:
...     print("a is none of the above.")
...
a is none of the above.
>>>
```

Stings: where Python is strong

- You can use single, double, or triple quotes to denote string literals.

```
>>> a = "Hello World"
>>> b = 'Python is groovy'
>>> c = """Computer says 'No'"""
>>>
```

- Triple quotes can extend over multiple lines, but single and double quotes cannot.

```
>>> c = """Computer says 'no'
... because another computer
... says yes"""
>>> a = "hello
  File "<stdin>", line 1
    a = "hello
              ^
SyntaxError: EOL while scanning string literal
>>>
```

Strings: where Python is strong

- Strings are stored as sequences of characters.

```
>>> a = "Hello World"
>>> a[0]
'H'
>>> a[:5]
'Hello'
>>> a[6:]
'World'
>>> a[3:8]
'lo Wo'
>>>
```

- You can convert numeric types into strings and vice versa.

```
>>> z = "90"
>>> z
'90'
>>> int(z)
90
>>> float(z)
90.0
>>> str(90.25)
'90.25'
>>>
```

Strings: where Python is strong

- And you can concatenate strings.

```
>>> "123" + "abc"
'123abc'
>>> "123" + str(123.45)
'123123.45'
>>> a = 1
>>> b = "2"
>>> str(a) + b
'12'
>>>
```

Strings: where Python is strong

- There are several useful methods for strings.

```
>>> s = "Hello world!"
>>> len(s)
12
>>>
>>> s = "5, 4, 2, 9, 8, 7, 28"
>>> s.count(",")
6
>>> s.find("9, ")
9
>>> s[9:12]
'9, '
>>> "abc123".isalpha()
False
>>> "abc123".isalnum()
True
>>> s.split(",")
['5', ' 4', ' 2', ' 9', ' 8', ' 7', ' 28']
>>> ", ".join(["ready", "set", "go"])
'ready, set, go'
>>> "ready\n set\n go".splitlines()
['ready', ' set', ' go']
>>> "ready set go".splitlines()
['ready set go']
>>>
```

Lists

- In python, a list is an ordered sequence of objects, each of which can have any type.

```
>>> s = [1, 2, "Five!", ["Three, sir!", "Three!"]]
>>> len(s)
4
>>>
>>> s[0:1]
[1]
>>> s[2]
'Five!'
>>> s[2][1]
'i'
>>> s[3]
['Three, sir!', 'Three!']
>>> s[3][0]
'Three, sir!'
>>> s[3][1]
'Three!'
>>> s[3][1][1]
'h'
>>> s.append("new element")
>>> s
[1, 2, 'Five!', ['Three, sir!', 'Three!'], 'new element']
```

Lists

- I can append and remove list elements.

```
208 >>> l = ["a", "b", "c"]
209 >>> l.append("d")
210 >>> l.append("c")
211 >>> l
212 ['a', 'b', 'c', 'd', 'c']
213 >>> l.remove("a")
214 >>> l
215 ['b', 'c', 'd', 'c']
216 >>> l.remove("c")
217 >>> l
218 ['b', 'd', 'c']
219 >>> l.remove("c")
220 >>> l
221 ['b', 'd']
222 >>>
```

Tuples

```
223 >>> a = ()
224 >>> b = (3,)
225 >>> c = (3,4,"thousand")
226 >>> len(c)
227 3
228 >>>
229 >>> number1, number2, word = c
230 >>> number1
231 3
232 >>> number2
233 4
234 >>> word
235 'thousand'
236 >>> keys =["name", "status", "ID"]
237 >>> values = ["Joe", "approved", 23425]
238 >>> z = zip(keys, values)
239 >>> z
240 [('name', 'Joe'), ('status', 'approved'), ('ID', 23425)]
```

Dictionaries

```
241 >>> stock = {
242  ... "name" : "GOOG",
243  ... "shares" : 100,
244  ... "price" : 490.10 }
245 >>> stock
246 {'price': 490.1, 'name': 'GOOG', 'shares': 100}
247 >>> stock["name"]
248 'GOOG'
249 >>> stock["date"] = "today"
250 >>> stock
251 {'date': 'today', 'price': 490.1, 'name': 'GOOG', 'shares': 100}
252 >>> keys = ["name", "status", "ID"]
253 >>> values = ["Joe", "approved", 23425]
254 >>> zip(keys, values)
255 [('name', 'Joe'), ('status', 'approved'), ('ID', 23425)]
256 >>> d = dict(zip(keys, values))
257 >>> d
258 {'status': 'approved', 'name': 'Joe', 'ID': 23425}
259 >>>
```

Iteration and looping

- There are many ways to iterate.

```
260 # a.py
261 a = "Hello World"
262 # Print out the individual characters in a
263 for c in a:
264     print c
```

```
265 > python a.py
266 H
267 e
268 l
269 l
270 o
271
272 W
273 o
274 r
275 l
276 d
```

Iteration and looping

```
277 # a.py
278 b = ["Dave","Mark","Ann","Phil"]
279 # Print out the members of a list
280 for name in b:
281     print name
```

```
282 > python a.py
283 Dave
284 Mark
285 Ann
286 Phil
```

```
287 # a.py
288 c = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
289 # Print out all of the members of a dictionary
290 for key in c:
291     print key, c[key]
```

```
292 > python a.py
293 GOOG 490.1
294 AAPL 123.15
295 IBM 91.5
```

Iteration and looping

```
# a.py
for n in [0, 1,2,3,4,5,6,7,8,9]:
    print("2 to the %d power is %d" % (n, 2**n))
```

```
> python a.py
2 to the 0 power is 1
2 to the 1 power is 2
2 to the 2 power is 4
2 to the 3 power is 8
2 to the 4 power is 16
2 to the 5 power is 32
2 to the 6 power is 64
2 to the 7 power is 128
2 to the 8 power is 256
2 to the 9 power is 512
```

```
# a.py
for n in range(9):
    print("2 to the %d power is %d" % (n, 2**n))
```

```
> python a.py
2 to the 0 power is 1
2 to the 1 power is 2
2 to the 2 power is 4
2 to the 3 power is 8
2 to the 4 power is 16
2 to the 5 power is 32
2 to the 6 power is 64
2 to the 7 power is 128
2 to the 8 power is 256
2 to the 9 power is 512
```

range() and xrange()

- For lengthy iterations, don't use `range()` because it fully populates a list and takes up a lot of memory.
- Instead, use `xrange()`, which gives you your iteration indices on a need-to-know basis.

```
# a.py
x = 0
for n in xrange(999999999):
  x = x + 1
print(x)
```

Generators

- `xrange` is a special case of a larger class of functions called generators.

```
329 >>> def countdown(n):
330 ...     print "Counting down!"
331 ...     while n > 0:
332 ...         yield n # Generate a value (n)
333 ...         n -= 1
334 ...
335 >>> c = countdown(5)
336 >>> c.next()
337 Counting down!
338 5
339 >>> c.next()
340 4
341 >>> c.next()
342 3
343 >>>
```

List comprehensions

```
>>> nums = [1, 2, 3, 4, 5]
>>> squares = [n * n for n in nums]
>>> squares
[1, 4, 9, 16, 25]
>>> a = [-3,5,2,-10,7,8]
>>> b = 'abc'
>>> [2*s for s in a]
[-6, 10, 4, -20, 14, 16]
>>> [s for s in a if s >= 0]
[5, 2, 7, 8]
>>> [(x,y) for x in a
... for y in b if x > 0 ]
[(5, 'a'), (5, 'b'), (5, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (7, 'a'),
(7, 'b'), (7, 'c'), (8, 'a'), (8, 'b'), (8, 'c')]
>>> [(1,2), (3,4), (5,6)]
[(1, 2), (3, 4), (5, 6)]
```

- General syntax:

```
[expression for item_1 in iterable_1 if condition_1
            for item_2 in iterable_2 if condition_2
            ...
            for item_N in iterable_N if condition_N ]
```

Lambda functions, `filter()`, `map()`, and `reduce()`

- **Lambda function**: a compact way of writing a function. You can think of a lambda function as a "function literal".
- `filter(fun, list)`: returns a list of all the elements e in list for which fun(e) is true.
- `map(fun, list)`: applies fun to each element of list and returns the result in a new list.
- `reduce(fun, list)`: equivalent to the following (length of list is n).

```
364 value = fun(list[0], list[1])
365 value = fun(value, list[2])
366 value = fun(value, list[3])
367 ...
368 value = fun(value, list[n])
```

Lambda functions, `filter()`, `map()`, and `reduce()`

```
369 >>> f = lambda x: x > 3 and x % 2 != 0
370 >>> f(4)
371 False
372 >>> f(5)
373 True
374 >>> f(6)
375 False
376 >>>
377 >>> filter(lambda x: x > 3, [0, 1, 2, 3, 4, 5])
378 [4, 5]
379 >>>
380 >>>
381 >>> l = range(3)
382 >>> map(str, l)
383 ['0', '1', '2']
384 >>>
385 >>> map(lambda x: x*x, l)
386 [0, 1, 4]
387 >>>
388 >>> reduce(lambda x, y: x+y, range(1, 11)) # sum the numbers 1 to 10
389 55
390 >>>
```

File I/O

- If I run:

```
# a.py
import random

f = open("data.txt", "w")
f.write("x y\n")
for i in xrange(10):
  f.write("%0.3f %0.3f\n" % (random.random(), random.random()))
```

- The file, `data.txt`, is generated:

```
x y
0.506 0.570
0.887 0.792
0.921 0.641
0.894 0.664
0.494 1.000
0.745 0.734
0.274 0.127
0.075 0.381
0.449 0.995
0.355 0.807
```

File I/O

- I can read `data.txt` with:

```
409 >>> f = open("data.txt")
410 >>> header = f.readline()
411 >>> data = f.readlines()
412 >>>
413 >>> header
414 'x y\n'
415 >>> data
416 ['0.506 0.570\n', '0.887 0.792\n', '0.921 0.641\n', '0.894 0.664\n', '
        0.494 1.000\n',
417  '0.745 0.734\n', '0.274 0.127\n', '0.075 0.381\n', '0.449 0.995\n', '
        0.355 0.807\n']
418 >>>
419 >>> header = header.replace("\n","")
420 >>> header
421 'x y'
422 >>>
423 >>> d = [d.replace("\n","") for d in data]
424 >>> d
425 ['0.506 0.570', '0.887 0.792', '0.921 0.641', '0.894 0.664', '0.494
        1.000',
426  '0.745 0.734', '0.274 0.127', '0.075 0.381', '0.449 0.995', '0.355
        0.807']
427 >>>
```

File I/O

- And then I can process it into a nicer format.

```
428 >>> data = [d.split(" ") for d in data]
429 >>> data
430 [['0.506', '0.570'], ['0.887', '0.792'], ['0.921', '0.641'], ['0.894', '
        0.664'],
431 ['0.494', '1.000'], ['0.745', '0.734'], ['0.274', '0.127'], ['0.075', '
        0.381'],
432 ['0.449', '0.995'], ['0.355', '0.807']]
433 >>>
434 >>> data = [map(float, d) for d in data]
435 >>> data
436 [[0.506, 0.57], [0.887, 0.792], [0.921, 0.641], [0.894, 0.664], [0.494,
        1.0],
437 [0.745, 0.734], [0.274, 0.127], [0.075, 0.381], [0.449, 0.995], [0.355,
        0.807]]
```

Modules

- Modules are external packages of code. The are not usually builtin, but they can be imported.

```
>>> sqrt(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>>
>>> import math
>>> sqrt(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>>
>>> math.sqrt(10)
3.1622776601683795
>>>
```

Modules

- If you don't want to write `math.sqrt()` every single time you want to compute a square root, you can use a shortcut.

```
>>> import math as m
>>> m.sqrt(10)
 3.1622776601683795
>>>
```

- Or better yet,

```
>>> from math import *
>>> sqrt(10)
 3.1622776601683795
>>>
```

- Where is the math module?

```
>>> import math
>>> math.__file__
 '/usr/lib64/python2.6/lib-dynload/mathmodule.so'
>>>
```

Installing modules locally in impact1

- We don't have permission to install modules ourselves in `/usr/`. We have to either bother the STAT IT people or install our package locally.
- Suppose I want to install the SQLAlchemy module. To download it into my home directory, I work from linux command line on impact1.

```
1  > cd ~
2  > pwd
3  /home/landau
4  > ls
5  stuff
6  > wget http://prdownloads.sourceforge.net/sqlalchemy/SQLAlchemy-0.7.9.
       tar.gz?download
7  # output of wget ...
8  > ls
9  stuff SQLAlchemy-0.7.9.tar.gz
10 > tar -zxvf SQLAlchemy-0.7.9.tar.gz
11 # output of tar...
12 > cd SQLAlchemy-0.7.9
13 > python setup.py build
14 # output of python...
15 > python setup.py install --user
16 # output of python...
17 Installed /home/landau/.local/lib/python2.6/site-packages/SQLAlchemy
       -0.7.9-py2.6-linux-x86_64.egg
18 Processing dependencies for SQLAlchemy==0.7.9
19 Finished processing dependencies for SQLAlchemy==0.7.9
```

Installing modules locally in impact1

- IMPORTANT: take note that SQLAlchemy was installed in the directory, `/home/landau/.local/lib/python2.6/site-packages/`. I must "export" this path in my `.bashrc` file so that Python knows where my new module lives.
- I move to my home directory and open `.bashrc`:

```
20 [landau@impact1 SQLAlchemy-0.7.9]$ cd ~
21 [landau@impact1 ~]$ emacs .bashrc
```

- The file itself currently looks like this:

```
1 # .bashrc
2
3 # Source global definitions
4 if [ -f /etc/bashrc ]; then
5         . /etc/bashrc
6 fi
7
8 # User specific aliases and functions
```

Installing modules locally in impact1

- I add a couple lines to the end so that Python knows where to find my package:

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions

export PYTHONPATH=$HOME/.local/lib/python2.6/site-packages:$PYTHONPATH
export PATH=$HOME/.local/bin:$PATH
```

- Once I've made the changes and I log out and in so that the changes take effect, I'm ready to import SQLAlchemy.

```
>>> import sqlalchemy
>>>
```

- Remember to use all lower case letters for modules in the `import` statement.

The sys module

- sys is a module of system-specific parameters and functions.

```
# a.py
import sys

for arg in sys.argv:
  print arg
```

```
> python a.py 1 2 3 4 5 3sir! 3!
a.py
1
2
3
4
5
3sir!
3!
```

Outline

The NumPy module

- Important module for arrays and matrices.

```
>>> from numpy import *
>>> a = arange(15)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
>>> a = a.reshape(3,5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.transpose()
>>> a.transpose()
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
>>> a.shape
(3, 5)
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>>
>>> zeros( (3,4) )
array([[0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.]])
>>>
```

The NumPy module

```
>>> ones( (2,3,4), dtype=int16 )                # dtype can also be
    specified
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)
>>>
>>> empty( (2,3) )
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
>>>
>>> b = array( [ [1.5,2,3], [4,5,6] ] )
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
>>> print(b)
[[ 1.5  2.   3. ]
 [ 4.   5.   6. ]]
>>>
>>> sum(b)
21.5
>>>
>>> a = array( [20,30,40,50] )
>>> b = arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
```

The NumPy module

```
>>> b**2
array([0, 1, 4, 9])
>>> 10*sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([True, True, False, False], dtype=bool)
```

- Elementwise product vs matrix product:

```
>>> A = array( [[1,1],
...             [0,1]] )
>>> B = array( [[2,0],
...             [3,4]] )
>>> A*B                         # elementwise product
array([[2, 0],
       [0, 4]])
>>> dot(A,B)                    # matrix product
array([[5, 4],
       [3, 4]])
```

The NumPy module

- Array indexing and slicing:

```
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a[0]
array([0, 1, 2, 3, 4])
>>> a[1]
array([5, 6, 7, 8, 9])
>>> a[0:2]
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>>
>>> a[0, 0]
0
>>> a[1, 2]
7
>>> a[0:2, 0:2]
array([[0, 1],
       [5, 6]])
>>>
>>> a[:,:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

The NumPy module

```
>>> a[:, 0]
array([ 0,  5, 10])
>>> a[:, 0:1]
array([[ 0],
       [ 5],
       [10]])
>>>
```

- Iterating over an array:

```
>>> for row in a:
...     print row
...
[0 1 2 3 4]
[5 6 7 8 9]
[10 11 12 13 14]
>>>
>>> for index in xrange(a.shape[1]):
...     print a[:, index]
...
[ 0  5 10]
[ 1  6 11]
[ 2  7 12]
[ 3  8 13]
[ 4  9 14]
>>>
>>> for elt in a.flat:
...     print elt,
...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

The NumPy module

- Array stacking:

```
142 >>> a = floor(10*random.random((2,2)))
143 >>> a
144 array([[ 1.,  1.],
145        [ 5.,  8.]])
146 >>> b = floor(10*random.random((2,2)))
147 >>> b
148 array([[ 3.,  3.],
149        [ 6.,  0.]])
150 >>> vstack((a,b))
151 array([[ 1.,  1.],
152        [ 5.,  8.],
153        [ 3.,  3.],
154        [ 6.,  0.]])
155 >>> hstack((a,b))
156 array([[ 1.,  1.,  3.,  3.],
157        [ 5.,  8.,  6.,  0.]])
```

The NumPy module

- Shallow copying:

```
158 >>> c = a.view()
159 >>> c == a
160 array([[ True,  True,  True,  True,  True],
161        [ True,  True,  True,  True,  True],
162        [ True,  True,  True,  True,  True]], dtype=bool)
163 >>> c is a
164 False
165 >>> a[0,0] = 1000
166 >>> a
167 array([[1000,    1,    2,    3,    4],
168        [   5,    6,    7,    8,    9],
169        [  10,   11,   12,   13,   14]])
170 >>> c
171 array([[1000,    1,    2,    3,    4],
172        [   5,    6,    7,    8,    9],
173        [  10,   11,   12,   13,   14]])
174 >>>
```

The NumPy module

- The default copy is the shallow copy:

```
>>> a
array([[1000,    1,    2,    3,    4],
       [   5,    6,    7,    8,    9],
       [  10,   11,   12,   13,   14]])
>>> b = a
>>> a[0,0] = 0
>>> b
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

- Deep copying:

```
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> b = a.copy()
>>> b[0,0] = 1000
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> b
array([[1000,    1,    2,    3,    4],
       [   5,    6,    7,    8,    9],
       [  10,   11,   12,   13,   14]])
```

The NumPy module

- Logical arrays:

```
199 >>> a = arange(12).reshape(3,4)
200 >>> b = a > 4
201 >>> b                    # b is a boolean with the same shape as a
202 array([[False, False, False, False],
203        [False, True, True, True],
204        [True, True, True, True]], dtype=bool)
205 >>> a[b]                                          # 1d array with the
        selected elements
206 array([ 5,  6,  7,  8,  9, 10, 11])
207 >>>
208 >>> a[b] = 0                                      # All elements of 'a'
        higher than 4 become 0
209 >>> a
210 array([[0, 1, 2, 3],
211        [4, 0, 0, 0],
212        [0, 0, 0, 0]])
```

The NumPy module

- Simple linear algebra:

```
>>> from numpy import *
>>> from numpy.linalg import *

>>> a = array([[1.0, 2.0], [3.0, 4.0]])
>>> print a
[[ 1.  2.]
 [ 3.  4.]]

>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])

>>> inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> u = eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> j = array([[0.0, -1.0], [1.0, 0.0]])

>>> dot (j, j) # matrix product
array([[-1.,  0.],
       [ 0., -1.]])
```

The NumPy module

```
>>> trace(u)  # trace
2.0

>>> y = array([[5.], [7.]])
>>> solve(a, y)
array([[-3.],
       [ 4.]])

>>> eig(j)
(array([ 0.+1.j,  0.-1.j]),
array([[ 0.70710678+0.j,  0.70710678+0.j],
       [ 0.00000000-0.70710678j,  0.00000000+0.70710678j]]))
Parameters:
    square matrix

Returns
    The eigenvalues, each repeated according to its multiplicity.

    The normalized (unit "length") eigenvectors, such that the
    column ``v[:,i]`` is the eigenvector corresponding to the
    eigenvalue ``w[i]`` .
```

The NumPy module

- Matrices:

```
>>> A = matrix('1.0 2.0; 3.0 4.0')
>>> A
[[ 1.  2.]
 [ 3.  4.]]
>>> type(A)  # file where class is defined
<class 'numpy.matrixlib.defmatrix.matrix'>

>>> A.T  # transpose
[[ 1.  3.]
 [ 2.  4.]]

>>> X = matrix('5.0 7.0')
>>> Y = X.T
>>> Y
[[5.]
 [7.]]

>>> print A*Y  # matrix multiplication
[[19.]
 [43.]]

>>> print A.I  # inverse
[[-2.   1. ]
 [ 1.5 -0.5]]

>>> solve(A, Y)  # solving linear equation
matrix([[-3.],
        [ 4.]])
```

The NumPy module

- Caution: indexing and slicing are different between matrices and arrays.

```
287 >>> A = arange(12).reshape(3,4)
288 >>> M =  mat(A.copy())
289 >>>
290 >>> print A[:,1]
291 [1  5  9]
292 >>> print M[:,1]
293 [[1]
294  [5]
295  [9]]
296 >>>
```

Outline

Other useful modules

- `SciPy`: a module for scientific computing

submodule	contents
`cluster`	clustering algorithms
`constants`	physical and mathematical constants
`fftpack`	fast Fourier transform
`integrate`	integration and ordinary differential equation solvers
`interpolate`	interpolation and smoothing splines
`io`	input and output
`linalg`	linear algebra
`ndimage`	N-dimensional image processing
`odr`	orthogonal distance regression
`optimize`	optimization and root-finding routines
`signal`	signal processing
`sparse`	sparse matrix routines
`spatial`	spatial data structures and algorithms
`special`	"special" functions
`stats`	statistical distributions and functions
`weave`	integration with C/C++

- `matplotlib`: graphics and plotting

Other useful modules

- `PyCUDA`: writing and executing GPU kernels from within Python.
  - Stay tuned ...

Outline

Resources

- Guides:
  1. David M. Beazley. *Python Essential Reference: Fourth Edition*. Addison-Wesley, 2009.
  2. Tentative NumPy Tutorial. http://www.scipy.org/Tentative_NumPy_Tutorial.
  3. SciPy Tutorial. http://docs.scipy.org/doc/scipy/reference/tutorial/general.html
  4. Matplotlib. http://matplotlib.org/.
- Code from today:
  - IntroPython.py

That's all for today.

- Series materials are available at http://will-landau.com/gpu.