INTRODUCTION TO PYTHON (VERSION 2) FOR STATISTICIANS

Will Landau, Prof. Jarad Niemi

OUTLINE

- Basic elements
  - Random preliminaries
  - User-defined functions
  - Code formatting: Indentation and line continuation
  - Logic and control flow
  - Data types: strings, lists, tuples, and dictionaries
  - Iteration, looping, and generators
  - Functional programming: list comprehensions, lambda functions, `filter()`, `map()`, and `reduce()`
  - File I/O
  - Modules
- The NumPy Module
- Other useful modules

RANDOM PRELIMINARIES

Like R, Python is an interpreted language. You can open the interpreter by typing `python` into the command line:

```
[landau@impact1 ~]$ python
Python 2.6.6 (r266:84292, May  1 2012, 13:52:17)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can create variables and do arithmetic operations essentially like in R:

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

You can also write a script and save it to a `.py` file, such as hello_world.py:

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

And then run it in the command line like this:

```
~/Desktop> python hello_world.py
Hello World
~/Desktop>
```

Denote single-line comments with the hash sign:

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

Triple-quoted strings serve as multi-line comments:

```
# a.py
"""
This program
```

```
does nothing.
"""
```

```
~/Desktop> python a.py
~/Desktop>
```

You can format strings and print them to the console like this:

```
>>> 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 built-in methods such as `format`, which can be accessed with the `.` operator.

You an also format your output like:

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

```
>>> print(s)
sample 3: mass= 4.888g
>>>
```

or:

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

USER-DEFINED FUNCTIONS

I can define 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). Thus, indentation has to be consistent.

If I have a script like this:

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

I get the following error if I try to run it:

```
~/Desktop> 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+1+1+1+1+1+1+1+1
29
>>> 1+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+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
>>>
```

```
>>> 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.
>>>
```

STRINGS

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 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'
>>>
```

And you can concatenate strings

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

There are several useful methods for strings. To demonstrate just a few:

```
>>> 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']
```

I can append and remove list elements:

```
>>> l = ["a", "b", "c"]
>>> l.append("d")
>>> l.append("c")
>>> l
['a', 'b', 'c', 'd', 'c']
>>> l.remove("a")
>>> l
['b', 'c', 'd', 'c']
>>> l.remove("c")
>>> l
['b', 'd', 'c']
>>> l.remove("c")
>>> l
['b', 'd']
>>>
```

TUPLES

```
>>> a = ()
>>> b = (3,)
>>> c = (3,4,"thousand")
>>> len(c)
3
>>>
>>> number1, number2, word = c
>>> number1
3
>>> number2
4
>>> word
'thousand'
>>> keys =["name", "status", "ID"]
>>> values = ["Joe", "approved", 23425]
>>> z = zip(keys, values)
>>> z
[('name', 'Joe'), ('status', 'approved'), ('ID', 23425)]
```

DICTIONARIES

```
>>> stock = {
... "name" : "GOOG",
... "shares" : 100,
... "price" : 490.10 }
>>> stock
{'price': 490.1, 'name': 'GOOG', 'shares': 100}
>>> stock["name"]
'GOOG'
>>> stock["date"] = "today"
>>> stock
{'date': 'today', 'price': 490.1, 'name': 'GOOG', 'shares': 100}
>>> keys = ["name", "status", "ID"]
>>> values = ["Joe", "approved", 23425]
>>> zip(keys, values)
[('name', 'Joe'), ('status', 'approved'), ('ID', 23425)]
>>> d = dict(zip(keys, values))
>>> d
{'status': 'approved', 'name': 'Joe', 'ID': 23425}
>>>
```

ITERATION AND LOOPING

There are many ways to iterate:

```
# a.py
a = "Hello World"
# Print out the individual characters in a
for c in a:
  print c
```

```
~/Desktop> python a.py
H
e
l
l
o

W
o
r
l
d
```

```
~/Desktop>
```

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

```
~/Desktop> python a.py
Dave
Mark
Ann
Phil
~/Desktop>
```

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

```
~/Desktop> python a.py
GOOG 490.1
AAPL 123.15
IBM 91.5
~/Desktop>
```

```
# 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))
```

```
~/Desktop> 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
```

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

```
~/Desktop> 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()`

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1,8)
[1, 2, 3, 4, 5, 6, 7]
>>> range(0, 14, 3)
[0, 3, 6, 9, 12]
>>> range(8, 1, -1)
[8, 7, 6, 5, 4, 3, 2]
>>>
```

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)
```

```
~/Desktop> python a.py
99999
~/Desktop>
```

GENERATORS

`range` is a special case of a larger class of functions called generators:

```
>>> def countdown(n):
...   print "Counting down!"
...   while n > 0:
...     yield n # Generate a value (n)
...     n -= 1
...
>>> c = countdown(5)
>>> c.next()
Counting down!
5
>>> c.next()
4
>>> c.next()
3
>>>
```

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 item1 in iterable1 if condition1
                    for item2 in iterable2 if condition2
                    ...
                    for itemN in iterableN if conditionN ]
```

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):

```
value <- fun(list[0], list[1])
value <- fun(value, list[2])
value <- fun(value, list[3])
...
value <- fun(value, list[n])
```

Examples:

```
>>> f = lambda x: x > 3 and x % 2 != 0
>>> f(4)
False
>>> f(5)
True
>>> f(6)
```

```
False
>>>
>>> filter(lambda x: x > 3, [0, 1, 2, 3, 4, 5])
[4, 5]
>>>
>>>
>>> l = range(3)
>>> map(str, l)
['0', '1', '2']
>>>
>>> map(lambda x: x*x, l)
[0, 1, 4]
>>>
>>> reduce(lambda x, y: x+y, range(1, 11)) # sum the numbers 1 through 10
55
>>>
```

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
```

I can read that file with:

```
>>> f = open("data.txt")
>>> header = f.readline()
>>> data = f.readlines()
>>>
>>> header
'x y\n'
>>> data
['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',
 '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']
>>>
>>> header = header.replace("\n","")
>>> header
'x y'
>>>
>>> d = [d.replace("\n","") for d in data]
>>> d
['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']
>>>
```

And then I can process it into a nicer format

```
>>> data = [d.split(" ") for d in data]
>>> data
[['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']]
>>>
>>> data = [map(float, d) for d in data]
>>> data
[[0.506, 0.57], [0.887, 0.792], [0.921, 0.641], [0.894, 0.664], [0.494, 1.0],
[0.745, 0.734], [0.274, 0.127], [0.075, 0.381], [0.449, 0.995], [0.355, 0.807]]
```

MODULES

Modules are Python libraries. You can use a library in your code with the `import` command.

```
>>> 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
>>>
```

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
>>>
```

`sys` MODULE

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

```
# a.py
import sys

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

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

NumPy MODULE

Important module for arrays and matrices. Here is some example code demonstrating basic array creation and operations.

```
>>> 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.]])
>>>
>>> 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])
>>> 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]])
```

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]])
>>>
>>> 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
>>>
```

Array stacking:

```
>>> a = floor(10*random.random((2,2)))
>>> a
array([[ 1.,  1.],
       [ 5.,  8.]])
>>> b = floor(10*random.random((2,2)))
>>> b
array([[ 3.,  3.],
       [ 6.,  0.]])
>>> vstack((a,b))
array([[ 1.,  1.],
       [ 5.,  8.],
       [ 3.,  3.],
       [ 6.,  0.]])
>>> hstack((a,b))
array([[ 1.,  1.,  3.,  3.],
       [ 5.,  8.,  6.,  0.]])
```

Shallow copying:

```
>>> c = a.view()
>>> c == a
array([[ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True]], dtype=bool)
>>> c is a
False
>>> a[0,0] = 1000
>>> a
array([[1000,    1,    2,    3,    4],
       [   5,    6,    7,    8,    9],
       [  10,   11,   12,   13,   14]])
>>> c
array([[1000,    1,    2,    3,    4],
       [   5,    6,    7,    8,    9],
       [  10,   11,   12,   13,   14]])
>>>
```

The default copy is a 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]])
>>>
```

Logical arrays:

```
>>> a = arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                          # b is a boolean with a's shape
array([[False, False, False, False],
       [False, True, True, True],
       [True, True, True, True]], dtype=bool)
>>> a[b]                                       # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
>>>
>>> a[b] = 0                                   # All elements of 'a' higher than 4 become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

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.]])

>>> 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]`` .
```

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.]])
```

Beware: indexing and slicing are slightly different for matrices.

```
>>> A = arange(12).reshape(3,4)
>>> M =  mat(A.copy())
>>>
>>> print A[:,1]
[1 5 9]
>>> print M[:,1]
[[1]
 [5]
 [9]]
>>>
```

OTHER USEFUL MODULES

`SciPy`:

Subpackage	Description
`cluster`	Clustering algorithms
`constants`	Physical and mathematical constants
`fftpack`	Fast Fourier Transform routines
`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 matrices and associated routines
`spatial`	Spatial data structures and algorithms
`special`	Special functions
`stats`	Statistical distributions and functions
`weave`	C/C++ integration

`matplotlib`: A popular graphics and plotting module.

OUTLINE

- Basic elements
  - Random preliminaries
  - User-defined functions
  - Code formatting: Indentation and line continuation
  - Logic and control flow
  - Data types: strings, lists, tuples, and dictionaries
  - Iteration, looping, and generators
  - Functional programming: list comprehensions, lambda functions, `filter()`, `map()`, and `reduce()`
  - File I/O
  - Modules
- The NumPy Module
- Other useful modules

GPU SERIES MATERIALS

These slides, a tentative syllabus for the whole series, and code are available at:

https://github.com/wlandau/gpu.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the materials.

REFERENCES

David M. Beaz.ley. Python Essential Reference: Fourth Edition.
Addison-Wesley, 2009.

Tentative NumPy Tutorial. http://www.scipy.org/Tentative_NumPy_Tutorial

SciPy Tutorial. http://docs.scipy.org/doc/scipy/reference/tutorial/general.html

Matplotlib. http://matplotlib.org/