AMath 483/583 — Lecture 25 — May 23, 2011

Today:

- Python concepts and objects
- Data types, lists, tuples
- Modules
- NumPy arrays

Next:

- More about NumPy and SciPy
- Python exception handling
- Python functions
- Iterators, booleans

Read: Class notes and references

Python

Python is an object oriented general-purpose language

Advantages:

- Can be used interactively from a Python shell
  (similar to Matlab)
- Can also write scripts to execute from Unix shell
- Little overhead to start programming
- Powerful modern language
- Many modules are available for specialized work
- Good graphics and visualization modules
- Easy to combine with other languages (e.g. Fortran)
- Open source and runs on all platforms

Python

Disadvantage: Can be slow to do certain things,
such as looping over arrays.

Code is interpreted rather than compiled

Need to use suitable modules (e.g. NumPy) for speed.

Can easily create custom modules from compiled code written in Fortran, C, etc.

Can also use extensions such as Cython that makes it easier to mix Python with C code that will be compiled.

Python is often used for high-level scripts that e.g., download data from the web, run a set of experiments, collate and plot results.

Interactive Python

Python can be used in a mode where commands are executed as typed....

Standard Python shell:

```
$ python
Python 2.5.2 (r252:60911, Jul 31 2008, 17:31:22)
>>>
```

Better shell: ipython

```
$ ipython
IPython 0.8.1 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works,
                                   ?? prints more.

In [1]:
```

Or use Sage shell or notebook (web interface):
www.sagemath.org

Executing Python scripts

Can collect commands in a file somename.py
(a script or program) Similar to m-files in Matlab.

Then you can execute the commands either from the Unix prompt:

```
$ python somename.py
```

or in a Python or IPython shell:

```
>>> execfile('somename.py')

In [13]: execfile('somename.py')
In [14]: run somename.py  # only works in IPython
```

Later we'll see you might instead want to `import somename` as a module.

Program structure

Python has no begin $\cdots$ end keywords or braces.

Blocks are determined entirely by indentation.

Forces you to write readable code!

```
def f(x):
    """Funny way to define abs(x)."""
    if x<0:
        y = -x
    else:
        z = 5.*x
        y = z/5.
    return y

for x in [-3., 0., 3.]:
    print "f at ", x, " is ", f(x)
```

Object-oriented language

Nearly everything in Python is an object of some class.

The class description tells what data the object holds and what operations (methods or functions) are defined to interact with the object.

Object-oriented language

Nearly everything in Python is an object of some class.

The class description tells what data the object holds and what operations (methods or functions) are defined to interact with the object.

Every “variable” is really just a pointer to some object. You can reset it to point to some other object at will.

So variables don’t have “type” (e.g. integer, float, string).
(But the objects they currently point to do.)

Object-oriented language

```
>>> x = 3.4
>>> print id(x), type(x)   # id() returns memory add
8645588 <type 'float'>

>>> x = 5
>>> print id(x), type(x)
8401752 <type 'int'>

>>> x = [4,5,6]
>>> print id(x), type(x)
1819752 <type 'list'>

>>> x = [7,8,9]
>>> print id(x), type(x)
1843808 <type 'list'>
```

Object-oriented language

```
>>> x = [7,8,9]
>>> print id(x), type(x)
1843808 <type 'list'>

>>> x.append(10)
>>> x
[7, 8, 9, 10]
>>> print id(x), type(x)
1843808 <type 'list'>
```

Note: Object of type 'list' has a method 'append' that changes the object.

A list is a mutable object.

Object-oriented language — gotcha

```
>>> x = [1,2,3]
>>> print id(x), x
1845768 [1, 2, 3]

>>> y = x
>>> print id(y), y
1845768 [1, 2, 3]

>>> y.append(27)
>>> y
[1, 2, 3, 27]

>>> x
[1, 2, 3, 27]
```

Note: x and y point to the same object!

Making a copy

```
>>> x = [1,2,3]
>>> print id(x), x
1845768 [1, 2, 3]

>>> y = list(x)    # creates new list object
>>> print id(y), y
1846488 [1, 2, 3]

>>> y.append(27)

>>> y
[1, 2, 3, 27]

>>> x
[1, 2, 3]
```

integers and floats are immutable

If `type(x) in [int,float]`, then setting `y = x` creates a new object `y` pointing to a new location.

```
>>> x = 3.4
>>> print id(x), x
8645588 3.4

>>> y = x
>>> print id(y), y
8645572 3.4

>>> y = y+1

>>> print id(y), y
8645572 4.4

>>> print id(x), x
8645588 3.4
```

Lists

The elements of a list can be any objects
(need not be same type):

```
>>> L = [3, 4.5, 'abc', [1,2]]
```

Indexing starts at 0:

```
>>> L[0]
3

>>> L[2]
'abc'

>>> L[3]
[1, 2]

>>> L[3][0]   # element 0 of L[3]
1
```

Lists

Lists have several built-in methods, e.g. append, insert, sort, pop, reverse, remove, etc.

```
>>> L = [3, 4.5, 'abc', [1,2]]

>>> L2 = L.pop(2)
>>> L2
'abc'

>>> L
[3, 4.5, [1, 2]]
```

Note: L still points to the same object, but it has changed.

In IPython: Type L. followed by Tab to see all attributes and methods.

Lists and tuples

```
>>> L = [3, 4.5, 'abc']
>>> L[0] = 'xy'
>>> L
['xy', 4.5, 'abc']
```

A tuple is like a list but is immutable:

```
>>> T = (3, 4.5, 'abc')
>>> T[0]
3
>>> T[0] = 'xy'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
           item assignment
```

Python modules

When you start Python it has a few basic built-in types and functions.

To do something fancier you will probably import modules.

Example: to determine what directory we are in or change directory, we need the module os (operating system):

```
>>> import os
>>> os.getcwd()
'/Users/rjl'

>>> os.chdir('uwamath583s11/codes/python')
>>> os.getcwd()
'/Users/rjl/uwamath583s11/codes/python'
```

Python modules

When type `import modname`, Python looks on its search path for the file modname.py.

You can add more directories using the Unix environment variable PYTHONPATH.

Or, in Python, using the sys module:

```
>>> import sys
>>> sys.path    # returns list of directories
['', '/usr/bin', ....]

>>> sys.path.append('newdirectory')
```

The empty string '' in the search path means it looks first in the current directory.

Python modules

Searches first in current directory, e.g.

```
>>> os.getcwd()
'/Users/rjl/uwamath583s11/codes/python'

>>> import myfcns

>>> myfcns.f1
<function f1 at 0x1bf4b0>

>>> myfcns.f1(2.)    # evaluate f1 at 2.
3.0
```

Python modules

Different ways to import:

```
>>> import os
>>> os.getcwd()
'/Users/rjl/uwamath583s11/codes/python'

>>> from os import getcwd
>>> getcwd()
'/Users/rjl/uwamath583s11/codes/python'

>>> from os import *
>>> getcwd()
'/Users/rjl/uwamath583s11/codes/python'

>>> import myfcns as MF
>>> MF.f1(2.)
3.0
```

Modules and functions are objects

```
>>> import myfcns
>>> f = myfcns.f1
>>> f(2.)
3.0

>>> M = myfcns      # M points to module

>>> L = [f, M.f2]   # a list of functions

>>> L[0](2.)
3.0
```

Useful if you want to loop over a set of test functions.

Modules and functions are objects

Useful if you want to loop over a set of test functions.

```
>>> L = [myfcns.f1, myfcns.f2]

>>> for f in L:
...     print f, " evaluated at 2. is ", f(2.)
...


<function f1 at 0x1bf4b0>  evaluated at 2. is  3.0
<function f2 at 0x1bf4f0>  evaluated at 2. is  2980.9579
```

Lists aren't good as numerical arrays

Lists in Python are quite general, can have arbitrary objects as elements.

Addition and scalar multiplication are defined for lists, but not what we want for numerical computation, e.g.

Multiplication repeats:

```
>>> x = [2., 3.]
>>> 2*x
[2.0, 3.0, 2.0, 3.0]
```

Addition concatenates:

```
>>> y = [5., 6.]
>>> x+y
[2.0, 3.0, 5.0, 6.0]
```

NumPy module

Instead, use NumPy arrays:

```
>>> import numpy as np

>>> x = np.array([2., 3.])
>>> 2*x
array([ 4.,  6.])

>>> y = np.array([5., 6.])
>>> x+y
array([ 7.,  9.])
```

Other operations also apply component-wise:

```
>>> np.sqrt(x)
array([ 1.41421356,  1.73205081])
>>> np.cos(x)
array([-0.41614684, -0.9899925 ])
```

NumPy arrays

Unlike lists, all elements of an `np.array` have the same type

```
>>> np.array([1, 2, 3])     # all integers
array([1, 2, 3])

>>> np.array([1, 2, 3.])    # one float
array([ 1.,  2.,  3.])      # they're all floats!
```

Can explicitly state desired data type:

```
>>> x = np.array([1, 2, 3], dtype=complex)
>>> print x
[ 1.+0.j,  2.+0.j,  3.+0.j]

>>> (x + 1.j) * 2.j
array([-2.+2.j, -2.+4.j, -2.+6.j])
```

NumPy arrays for vectors and matrices

```
>>> A = np.array([[1.,2], [3,4], [5,6]])
>>> A
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])

>>> A.shape
(3, 2)

>>> A.T
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  6.]])

>>> x = np.array([1., 1.])
>>> x.T
array([ 1.,  1.])
```

NumPy arrays for vectors and matrices

```
>>> A
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])

>>> x
array([ 1.,  1.])

>>> np.dot(A,x)    # matrix-vector product
array([  3.,   7.,  11.])

>>> np.dot(A.T, A)   # matrix-matrix product
array([[ 35.,  44.],
       [ 44.,  56.]])
```