Introduction to Python for HPC

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2014

Part I

Basic Python

Python is Not Just a Big Snake ...

Python

- created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see `http://www.cwi.nl/`) in the Netherlands
- Compiled, interpreted ("mid-level") language
- Not a cute acronym - rather a cute cultural reference (think Monty ...)
- Object-oriented and highly modular
- Designed to be “friendly” to other languages, as in easily call multi-language routines (`C/C++`,`Fortran`,...)
- Extensive set of standard libraries
- Lots of traction in a wide range of fields (embedded devices to HPC)
- Goal: programmer productivity

`www.python.org`

- Home page of all things python
- Portable (and Free!) ...
- Platforms: Windows, Linux/Unix, Mac OS X, OS/2 (also Amiga, Palm Handhelds, and Nokia mobile phones, ported to the Java and .NET virtual machines)
- Current version: **2.7.x** or **3.x**, but older versions still mostly to be found "in the wild" (watch out for that)
  - **3.x** series breaks backward-compatibility, so a lot of useful modules have yet to be ported to python 3.
  - Most major Linux distributions shipping with 3.x, some modules are starting to catch up
- Check your version: `python -V`

Both compiled and interpreted:

- “Compiled” in memory, not to machine assembly code, but “byte code”, machine independent
- Interpreted/executed by Python Virtual Machine (VM)
- Analogous to Java VM
- Faster than fully interpreted languages (e.g. Perl, shell)
- No object code, but byte code can be saved (typically `.pyc` suffix)

Running Python Interactively

Python is very easily invoked, our canonical example:

```
[rush:~]$ python
Python 2.6.6 (r266:84292, Feb 22 2013, 00:00:18)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello, world!"
Hello, world!
>>> 3.14159/2
1.5707949999999999
>>> ^D
```

Note the “CTRL-D” to exit the Python interpreter.

Python as Script

Python easily scripted using a file of Python commands:

```
[rush:~]$ cat hello.py
#!/usr/bin/python
print "Hello, world!"
print 3.14159/2
[rush:~]$
[rush:~]$ python hello.py
Hello, world!
1.570795
[rush:~]$
```

Note that Python scripts by convention get a `.py` suffix ...

or, of course, you can pass the interpreter directly to the shell,

```
[bono:~]$ which python
/usr/bin/python
[bono:~]$ cat hello.py
#!/usr/bin/python
print "Hello, world!"
print 3.14159/2
[rush:~]$ chmod u+x hello.py
[rush:~]$ ./hello.py
Hello, world!
1.570795
[rush:~]$
```

The `env` Trick

...or you can the `env` trick (this is a Linux/UNIX thing) - use the `env` command to load a default `python` interpreter:

```
#!/usr/bin/env python
print "Hello, world!"
print 3.14159/2
[rush:~]$ ./hello.py
Hello, world!
1.570795
[rush:~]$
```

This trick can be pretty handy for loading versions of python that are not the default version.

Python Variables

Python variables:

- Do not need to be explicitly declared
- Names begin with letter or underscore
- Case sensitive
- Everything in Python is really an object (as in object-oriented), but we will come back to that ...

Python Types

Python has five core data types:

- Numbers
- Lists
- Strings
- Dictionaries
- Tuples

Python Numbers

Numbers in python come in several (unsurprising) flavors:

Literal	(C) Interpretation
12345	Decimal Integers (C `long`)
12345L	Long Integers (unlimited size)
3.1, 1.e-10	Floating-point (C `double`)
0123,0x9fe	Octal/Hex
1+2j,3.0+4.0J	Complex

(Python numbers get as much precision as the C compiler used to build the interpreter)

Numerical Operators

Numeric operators are of the expected variety: **+x,-x** (unary), **x**y**, **x*y,x%y,x/y,x//y** (that last one is truncated division), **x+y,x-y**, in order of precedence.

The operators can be **overloaded**, so they also apply to other types.

Use parentheses to ensure subexpressions get evaluated the way that you expect them to.

Printing Out numerical Values

Echoing results in Python usually returns the default precision (e.g. 15 places for double precision floating-point):

```
>>> sq2 = 2.0**0.5
>>> print sq2
1.41421356237
>>> sq2
1.4142135623730951
>>> "square root of 2 is %12.8f" %(sq2)
'square root of 2 is   1.41421356'
```

but printing them defaults to something else. Note the overloading of the % (binary) operator, which for strings acts like C's `sprintf` function.

Complex Arithmetic

Not terribly surprising, but quite handy:

```
>>> (2+3J)+(4+5J) # addition
(6+8j)
>>> (2+3J)-(4+5J) # subtraction
(-2-2j)
>>> (2+3J)*(4+5J) # multiplication
(-7+22j)
>>> (2+3J)/(4+5J) # (complex) division
(0.5609756097560976+0.04878048780487809j)
>>> z1=1+2j
>>> abs(z1)
2.23606797749979
>>> z1.real
1.0
>>> z1.imag
2.0
```

Python Strings

Again, not very surprising - except that Python has no notion of *characters*, just single element strings.

Literal	Interpretation
s1=''	Empty String
s2="spam"	Double Quotes Same as Single
s3="Spam'r Us"	
s4=r'C:\temp\Spam'	Raw Strings suppress escapes
s5=u'spam'	Unicode (larger character set) Strings
s1[i:j]	Slice of String
s1[-1]	Negative indexes are from the end of the String
s2+s3	Concatenate
s4*3	Repeat
s2.find('pa')	String method calls
s2.replace('pa','XX')	
s1.split()	
len(s2)	length
b1 = ''' ... '''	triple quotes for multiline blocks

More String Methods

In Python, “methods” are just functions that are associated with a particular object - in this case strings. We exemplified a few in the preceding table, here is a somewhat more complete list:

S.capitalize()	S.ljust(width)
S.center(width()	S.lower()
S.count(sub [,start[,end]])	S.lstrip()
S.encode([encoding[,errors]])	S.replace(old,new[,maxsplit])
S.endswith(suffix[,start[,end]])	S.rfind(sub[,start[,end]])
S.expandtabs([tabsize])	S.rindex(sub[,start[,end]])
S.find(sub[,start[,end]])	S.rjust(width)
S.index(sub[,start[,end]])	S.rstrip()
S.isalnum()	S.split([sep[,maxsplit]])
S.isalpha()	S.splitlines([keepends])
S.isdigit()	S.startswith(prefix[,start[,end]])
S.islower()	S.strip()
S.isspace()	S.swapcase()
S.istitle()	S.title()
S.isupper()	S.translate(table[,delchars])
S.join(seq)	S.upper()

String Conversion

Functions for converting to/from strings:

Method	Interpretation
int(“42”)	convert string to integer
float(“42.0”)	convert string to floating-point
str(42)	convert to string
str(42.0)	ditto

Changing Strings

Strings are immutable, so they can not be changed directly in-place. Instead you have to construct new strings:

```
>>> S = 'spammy'
>>> S[0] = 'S'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> S = 'S' + S[1:]
>>> S
'Spammy'
```

Python Lists

Python's lists are ordered collections of arbitrary objects (hence heterogeneous), variable length, nestable and mutable:

Operation	Interpretation
L1=[]	Empty list
L2=[0,1,2,3]	Four item list
L3=['a',['b','c']]	Nested sublists
L2[i]	Index
L3[i][j]	
L2[i:j]	Slice
len(L2)	Length
L1+L2	Concatenate
L2*3	Repeat

Dynamics of lists:

Operation	Interpretation
L2.append(4)	Grow list by one
L2.extend([5,6,7])	grow some more
L2.sort()	sort list
L2.index(1)	search
L2.reverse()	reverse list
L2.insert(1,0)	insert 0 before index 1
del L2[k]	shrink
del L2[i:j]	
L2.pop()	
L2[i:j] = []	

Python Dictionaries

Like hashes in Perl, dictionaries in Python are *unordered* collections, stored by **key**:

Operation	Interpretation
D1 = {}	Empty dictionary
D2 = {'spam' : 2, 'eggs' : 3}	2-item dictionary
D3 = {'food' : {'egg' : 1, 'cheese' : 2}}	Nested
D2['eggs']	Indexing by key
D3['food']['egg']	
D2.has_key('eggs')	Membership
'eggs' in D2	
D2.keys()	list of keys
D2.values()	list of values
len(D1)	number stored entries

Python Tuples

Tuples construct simple groups of objects in Python, like a list, but of fixed length (and immutable, so no *methods* to grow or shrink them:

Operation	Interpretation
()	Empty tuple
t1 = (0,)	1-item tuple
t2 = (0,'He',3.4,5)	4-item tuple
t2 = 0,'He',3.4,5	same as above
t3 = ('ab',('bc','de'))	Nested tuples
t1[i],t3[i][j]	Indexing
t1[i:j]	slice
len(t1)	length
t1+t2	concatenate
t2*3	repeat

Why do we need tuples? Basically just lists that are unchanging, whose integrity is guaranteed.

print

We have already made use of it, but the `print` statement outputs a sequence of strings to standard output (items separated by commas get a space, and the newline is automatic unless you end with a comma after the last string

Formatted output just uses:

```
print format_string %(variable_list)
```

where the format string is exactly the same usage as C's `printf` (c.f. **man printf**) function.

Input

The most basic Python input is provided by the `raw_input` function, which reads a line of input from standard input as a string:

```
>>> x=raw_input("enter x-value:")
enter x-value:1
>>> x
'1'
```

Converting them to other types can be done using the conversion functions. The `input` function is very similar, but does not necessarily treat the input as a string:

```
>>> y=input("enter y-value:")
enter y-value:2.0
>>> y
2.0
```

Simple File Operations

Summary of common file operations:

Operation	Interpretation
output=open('data_out','w')	Create output file in cwd, write mode
input=open('data_in','r')	Open input file, read mode
S = input.read()	reads entire file into string S
S = input.read(N)	reads N bytes into string S
S = input.readline()	reads next line (through EOL)
L = input.readlines()	read entire file into list L
output.write(S)	writes string S into file output
output.writelines(L)	write all line strings in L to file output
output.close()	manual file closure

This is just a subset of available file handling, of course (e.g. `seek`, `flush`, ...)

Output Redirects

Some more trickery with output:

```
'hello world!'                      # easy way - echo

import sys
sys.stdout.write('hello world!\n')  # hard way to do the same

log = open('log.txt','a')
print >> log,x,y,z                  # prints directly to log.txt
print x,y,z                         # back to stdout

sys.stdout = open('log.txt','a')    # redirect stdout to a file
print x,y,z                         # now shows up in the file
```

Is There in Truth No Beauty?

How does Python evaluate truth or falsity of expressions (like Perl, really)?

- Numbers are true if nonzero (otherwise false)
- Other types are true if nonempty (otherwise false)

Type comparisons:

- Numbers are straightforward, but complex numbers can not use built-in relational operators
- Strings compared element by element, then by ASCII collating sequence
- Lists (and tuples) compared member by member
- Dictionaries compared using sorted (key,value) lists

Relational Operators

Operation	Interpretation
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to
a **in** B	a is an element of B
a **not in** B	a is not an element of B

Boolean Operators

Operator	Interpretation
not boo	negates logical value of boo
alp **and** boo	logical and
alp **or** boo	logical or

Python Syntax Rules

Python syntax is a bit different:

- There are no braces (or other structure) to delimit "blocks" of code - Python uses indentation following a header, forcing the programmer to adopt sensible rules for indentation
- Python code "blocks" consist of a header followed by ":" (colon), and then indented code (exemplified by `if-elif-else` structure)
- Lines terminated by end of line/carriage returns (exception - lines can be continued with backslash, or automatically continued if you have open parentheses, braces, brackets, etc.)
- Blank lines, extra whitespace, comments (anything following #) ignored (blank lines can be significant to the interactive interpreter, though)
- *Docstrings* are an additional comment form that appear at the top of Python program files - retained as part of Python's internal documentation tools

`if` Statements

Basic conditional syntax:

```
if <test1>:              # blocks typically start with :
  <statement block 1>
elif <test2>:            # optional elif
  <statement block 2>
else:                    # optional else
  <statement block 3>
```

In the interactive interpreter, you need to leave a blank line after the last statement to indicate the end of the last block (unnecessary in a script, where it keys off the indentation).

There is no equivalent to the `switch/case` statement, instead nested `if-elif-else`

The while Loop

Typical structure, controlled by Boolean expression:

```
while <entry_boolean>:
    <statements1>
else:               # optional else
    <statements2>   # run if we did not exit loop with a break
```

The else is optional, and you can use a break statement to bail out of the loop early:

- break - jump out of closest enclosing loop
- continue - jump to top of closest enclosing loop
- pass - does nothing, empty placeholder
- else block - runs only if loop exited normally, without breaks

simple `pass` example:

```
>>> while 1:      # have to ctrl-C to interrupt
...   pass
...
KeyboardInterrupt
```

The for Loop

```
for <target> in <object>:      # assign object items to target
    <statements>
    if <test1>: break          # bail out, skip else
    if <test2>: continue       # next iteration
else:
    <statements>               # if we didn't hit break
```

```
>>> for i in range(11):
...     total += i
...
>>> print total
55
>>>
```

Note the extra blank line to delimit the statements in the `for` block, which would not be necessary in a script.

def Statements

Python function objects are assigned names using the `def` statement:

```
def <name>(arg1,arg2,...argN):
    <statements>
return <value>        # entirely optional
```

- `def` statements are executable code - can be nested, etc. (i.e. not compiled)
- Arguments are passed by (object) reference
- `global` needs to be used to make variables inside `def` global in scope
- Arguments, variables, return values generally not declared

A very simple function indeed:

```
>>> def times(x,y):
...     return x*y
...
>>> times(2,5)
10
>>> times('Foo',3)
'FooFooFoo'
```

but still very flexible - this type-dependent behavior is *polymorphism*, and is intentional on Python's part.

Variable Scope

- Names in Python come into being when assigned - not before
- Functions have their own namespace - names defined inside a `def` are entirely local to the `def`
- Enclosing module is a global scope (spans a single file only)
- Each call to a function is a new local scope
- Names are local unless explicitly declared `global`
- LEGB rule - local, enclosing functions (if any), global, built-in

Simple Example

Simple example of `global` usage:

```
>>> X = 11
>>> def func():
...     global X
...     X = 22
...
>>> func()
>>> print X
22
```

Argument Matching Modes

Python has a few options when matching arguments:

- Position - what we have used so far (left to right)
- Keyword - uses argument name, `name=value`
- Varargs - special arguments preceded with * or ** to collect arbitrary extra arguments
- Defaults - can specify default values for arguments in case too few values are passed (`name=value` syntax)

Argument Matching Illustrated

Positional and keyword argument matching illustrated:

```
>>> def posf(a,b=2,c=3): print a,b,c
...
>>> posf(1)
1 2 3
>>> posf(a=2)
2 2 3
>>> posf(5,6)
5 6 3
>>> posf(5,c=6)
5 2 6
```

Arbitrary Argument Matching

You can get pretty crazy with the arbitrary argument matching. Generally,

- * collects all positional arguments into a new tuple:

```
>>> def arbf(*args): print args
...
>>> arbf()
()
>>> arbf(1)
(1,)
>>> arbf('foo',1,2,3)
('foo', 1, 2, 3)
```

- ** similar, but only for keyword arguments into a new dictionary:

```
>>> def arbf2(**args): print args
...
>>> arbf2()
{}
>>> arbf2(a='cat',b='dog',c=3.14)
{'a': 'cat', 'c': 3.1400000000000001, 'b': 'dog'}
```

Anonymous Functions: `lambda`

In addition to the `def` statement, Python has a way to construct functions on-the-fly as *expressions* - they end up being unnamed, hence are called **anonymous** functions. In honor of a similar construct in Lisp, they are called `lambda`:

```
lambda: arg1, arg2, ... argN : expression using arguments
```

Note that this object is an expression, not a statement, so it can be used in places where a `def` can not. Must be a single expression, not a block of statements (think of it like the `return` statement in a `def`).

lambda Examples

```
>>> f = lambda x,y,z : x+y+z
>>> f(2,3,4)
9
>>> f = (lambda a='fee', b='fie', c='foe', d='fum' : a+b+c+d)
>>> f()
'feefiefoefum'
>>> f('foo')
'foofiefoefum'
>>> f('foo','42')
'foo42foefum'
```

The scope rules and argument defaults are the same as for a `def`

List Comprehensions

`lambda` programming has become deprecated in the Python-verse, replaced by list comprehensions, e.g.,

```
[f(x) for x in generator]
```

and you can add conditionals to the list as well. More examples:

```
>>> def squares(lastterm):
...     for n in range(lastterm):
...         yield n**2
...
>>> [i for i in squares(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [i for i in squares(10) if i%2==0]
[0, 4, 16, 36, 64]
>>> [i for i in squares(10) if i%2==0 and i%3==1]
[4, 16, 64]
>>>
```

Common idea of making a new list based on an old one.

Exception Handling

There are a number of built-in exceptions (e.g., syntax), but you can also trap them yourself using a **try** statement:

```
>>> try:
... 1/0
... except ZeroDivisionError:
... print "Oops! divide by zero!"
... except:
... print "some other exception!"
```

and you can define your own exceptions through a Python class.

Modules

We have inadvertently already used modules in Python (they are pretty inescapable). Basically modules are Python program files (or C extensions) processed by:

- **import** Allows fetching of a whole module
- **from** Allows fetching of particular names from module (or * for all)
- **reload** Reload a module's code without stopping Python

Modules provide a way to structure namespaces (and scope) in more complex Python programs

Module Steps

Python takes three primary steps when importing a module:

1. Find the module's file using the search path
2. Compile it to byte code (if needed)
3. Run the module code to produce defined objects

Let's go through these ...

Module Search Path

Python's module search path concatenates the following:

- The home directory of the top-level file (i.e. usually your working directory)
- `PYTHONPATH` directories - environment variable containing list of directories to search
- Standard library directories - where python libraries were installed (not usually part of `PYTHONPATH`)
- .pth file directories - e.g. a sys.pth file

```
>>> import sys
>>> print sys.path
['', '/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python27.zip',
'/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7',
'/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/plat-linux2',
'/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/lib-tk',
'/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/lib-old',
'/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/lib-dynload',
'/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/site-packages',
'/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/site-packages/PIL']
>>>
```

More Module Fun

Another useful module example:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2',
'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod',
'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>> math.pi
3.1415926535897931
>>> math.sqrt(3**2+4**2)
5.0
```

Standard Module Library

There are **many** available modules for Python, we have touched upon only a few, and we will sample a few more.

`http://docs.python.org`

should have a reasonably complete list for the latest versions of Python (you can also browse your installation's documentation, as we will soon explore).

Sources of Internal Python Documentation

Summarized in following table:

# Comments	In-file/code documentation
`dir`	Function for listing attributes of objects
`__doc__`	Docstrings, in-file info attached to objects
`help`	Help function, interactive help for objects
HTML reports	Module documentation in a browser

Plenty of external documentation available, of course (web, books, etc.)

dir Function

The intrinsic `dir` function can be used to show all the attributes of an object:

```
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__',
'__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__str__', 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__', '__stdin__',
'__stdout__', '_getframe', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'exc_clear', 'exc_info',
'exc_type', 'excepthook', 'exec_prefix', 'executable', 'exit', 'getcheckinterval',
'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding', 'getrecursionlimit',
'getrefcount', 'hexversion', 'last_type', 'last_value', 'maxint', 'maxunicode', 'meta_path',
'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace',
'stderr', 'stdin', 'stdout', 'version', 'version_info', 'warnoptions']
```

Docstrings

Docstrings are Python's way of attaching documentation that can be accessed at runtime - it will be put into a string attached to the object as the `__doc__` attribute:

```
[rush:~]$ cat sample.py
"""
    Module docs
    Add some verbiage here
"""

def squareit(x):
    """
    function documentation
    It was the best of times, it was the worst of times ...
    """
    return x*x

class testclass:
    "triple quotes for multiline, but any string will do for one line"
    pass

print squareit(8)
print squareit.__doc__
```

```
>>> import sample
64

    function documentation
    It was the best of times, it was the worst of times ...

>>> print sample.__doc__

    Module docs
    Add some verbiage here

>>> print sample.testclass.__doc__
triple quotes for multiline, but any string will do for one line
```

Most (if not all) of the built-in modules and objects in Python have such documentation. There is not much of a standard for it, however.

help Function

The Python `help` function (part of the PyDoc tool) formats the Docstrings much like Linux/Unix man pages:

```
>>> import sys
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.

    Dynamic objects:

    argv -- command line arguments; argv[0] is the script pathname if known
    path -- module search path; path[0] is the script directory, else ''
    modules -- dictionary of loaded modules
```

and it works on functions as well (it parses through structural parts of the code to show you the syntax as well as the developer's comments):

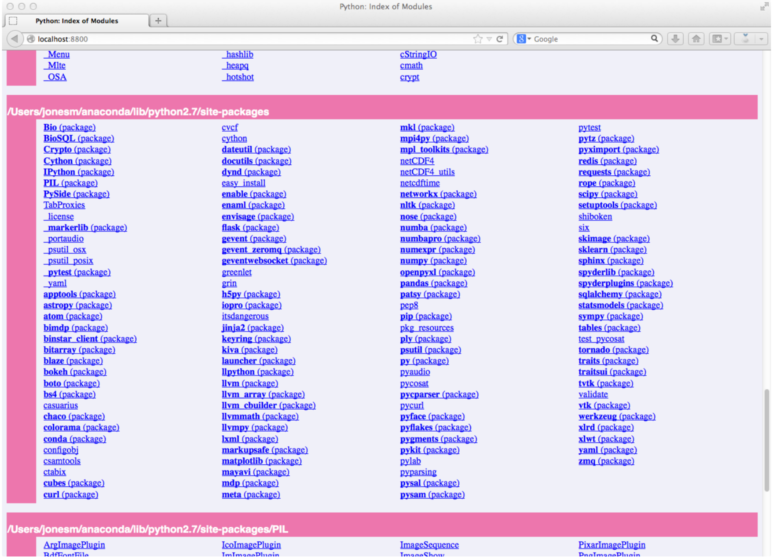
```
>>> help(range)
Help on built-in function range:

range(...)
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    ...
>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, count]) -> string

    Return a copy of string S with all occurrences of substring
    ...
>>>
```

HTML Reports in PyDoc

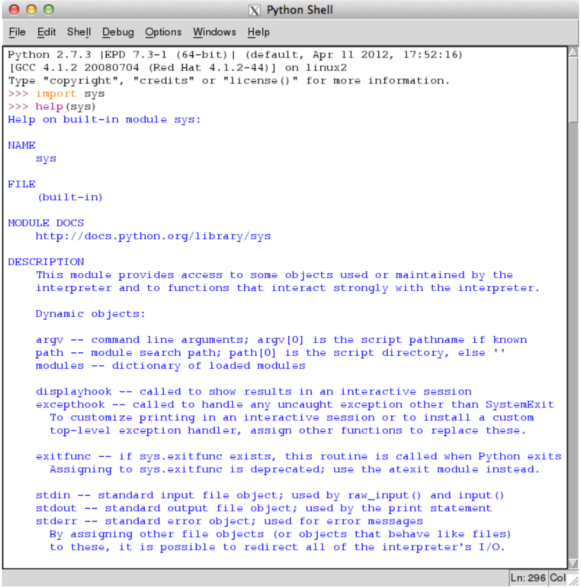
You can also browse HTML documentation for Python in a couple of ways. The first is through the use of the `pydoc` command (on some systems it is available under HTTP):

```
[cash:~]$ pydoc -p 8800
pydoc server ready at http://localhost:8800/
```

and then open in your favorite web browser ...


Python: Index of Modules
Python: Index of Modules
localhost:8800
Google
_Menu
_Mlte
_OSA
_hashlib
_heapq
_hotshot
cStringIO
cmath
crypt
/Users/jonesm/anaconda/lib/python2.7/site-packages
Bio (package)
BioSQL (package)
Crypto (package)
Cython (package)
IPython (package)
PIL (package)
PySide (package)
TabProxies
_license
_markerlib (package)
_portaudio
_psutil_osx
_psutil_posix
_pytest (package)
_yaml
apptools (package)
astropy (package)
atom (package)
bimdp (package)
binstar_client (package)
bitarray (package)
blaze (package)
bokeh (package)
boto (package)
bs4 (package)
casuarius
chaco (package)
colorama (package)
conda (package)
configobj
csamtools
ctabix
cubes (package)
curl (package)
cvcf
cython
dateutil (package)
docutils (package)
dynd (package)
easy_install
enable (package)
enaml (package)
envisage (package)
flask (package)
gevent (package)
gevent_zeromq (package)
geventwebsocket (package)
greenlet
grin
h5py (package)
iopro (package)
itsdangerous
jinja2 (package)
keyring (package)
kiva (package)
launcher (package)
llpython (package)
llvm (package)
llvm_array (package)
llvm_cbuilder (package)
llvmmath (package)
llvmpy (package)
lxml (package)
markupsafe (package)
matplotlib (package)
mayavi (package)
mdp (package)
meta (package)
mkl (package)
mpi4py (package)
mpl_toolkits (package)
netCDF4
netCDF4_utils
netcdftime
networkx (package)
nltk (package)
nose (package)
numba (package)
numbapro (package)
numexpr (package)
numpy (package)
openpyxl (package)
pandas (package)
patsy (package)
pep8
pip (package)
pkg_resources
ply (package)
psutil (package)
py (package)
pyaudio
pycosat
pycparser (package)
pycurl
pyface (package)
pyflakes (package)
pygments (package)
pykit (package)
pylab
pyparsing
pysal (package)
pysam (package)
pytest
pytz (package)
pyximport (package)
redis (package)
requests (package)
rope (package)
scipy (package)
setuptools (package)
shiboken
six
skimage (package)
sklearn (package)
sphinx (package)
spyderlib (package)
spyderplugins (package)
sqlalchemy (package)
statsmodels (package)
sympy (package)
tables (package)
test_pycosat
tornado (package)
traits (package)
traitsui (package)
tvtk (package)
validate
vtk (package)
werkzeug (package)
xlrd (package)
xlwt (package)
yaml (package)
zmq (package)
/Users/jonesm/anaconda/lib/python2.7/site-packages/PIL
ArgImagePlugin
IcoImagePlugin
ImageSequence
PixarImagePlugin

You can also use the “Help” button in the `IDLE` IDE window ...



In Python, Everything is an Object

Everything in Python is an object (instantiation) of some class - under the covers Python is all about OOP (Object Oriented Programming)

- Types in Python are actually class definitions - hence very flexible
- Class definitions contain data members and function members (also known in Python lingo as “methods”) - any member get accessed using the “dot” notation that we have seen (e.g. `math.sqrt`)
- You don’t have to use the OOP aspects of Python to be productive - but you can use it to write some very flexible code (and leverage a lot of already written flexible code)

Class Syntax

How to define your own Python class:

```
class class_name:
    def method1(self,arg1,...)
    def method2(self,arg2,...)
```

Note that the `self` keyword needs to be the first argument of every class method (a kin to the C++ `this` pointer).

Operator Overloading

Another key element of OOP is operator overloading - the ability to have operators work with multiple types of operands.

- For example, addition we have already used in this way - adds numbers, concatenates strings
- Python reserves names just for overloading operators - e.g. `__len__` to overload the `len` operator
- It is considered bad coding style to make operators behave in “unexpected” ways

Part II

Selected Python Modules

A Breif History of SciPY (not time)

Brief (but tangled!) history of SciPy/NumPy:

- Originally conceived under several nicknames, **Numeric**, **NumPy**, **Numerical Python** ... circa 1995, early days of SourceForge (Jim Hugunin, plus Jim Fulton, David Ascher, Paul DuBois, and Konrad Hinsen)
- **SciPy** based on **Numeric** began development 2001 under Travis Oliphant, Eric Jones and Pearu Peterson
- **numarray** developed as a replacement for **numeric** by Perry Greenfield, Todd Miller and Rick White at the Space Science Telescope Institute
- **NumPy** reborn under Travis Oliphant to solidify core for **SciPy** in 2005. Headed for inclusion in Python standard libraries as *N*-dimensional array interface

So **SciPy** has been around for a while - but the underlying libraries have been shifting a bit. Sometimes finding a coherent installation can be tricky on older systems ...

SciPy Base

SciPy base functions:

array	NumPy Array construction
zeros	Return an array of all zeros
empty	Return an unitialized array
shape	Return shape of sequence or array
rank	Return number of dimensions
size	Return number of elements in entire array or a certain dimension
fromstring	Construct array from (byte) string
take	Select sub-arrays using sequence of indices
put	Set sub-arrays using sequence of 1-D indices
putmask	Set portion of arrays using a mask
reshape	Return array with new shape
repeat	Repeat elements of array
choose	Construct new array from indexed array tuple
cross_correlate	Correlate two 1-d arrays
searchsorted	Search for element in 1-d array
sum	Total sum over a specified dimension
average	Average, possibly weighted, over axis or array
cumsum	Cumulative sum over a specified dimension
product	Total product over a specified dimension
cumproduct	Cumulative product over a specified dimension
alltrue	Logical and over an entire axis
sometrue	Logical or over an entire axis
allclose	Tests if sequences are essentially equal

More base functions:

arrayrange (arange)	Return regularly spaced array
asarray	Guarantee NumPy array
sarray	Guarantee a NumPy array that keeps precision
convolve	Convolve two 1-d arrays
swapaxes	Exchange axes
concatenate	Join arrays together
transpose	Permute axes
sort	Sort elements of array
argsort	Indices of sorted array
argmax	Index of largest value
argmin	Index of smallest value
innerproduct	Innerproduct of two arrays
dot	Dot product (matrix multiplication)
outerproduct	Outerproduct of two arrays
resize	Return array with arbitrary new shape
indices	Tuple of indices
fromfunction	Construct array from universal function
diagonal	Return diagonal array
trace	Trace of array
dump	Dump array to file object (pickle)
dumps	Return pickled string representing data
load	Return array stored in file object
loads	Return array from pickled string
ravel	Return array as 1-D
nonzero	Indices of nonzero elements for 1-D array
shape	Shape of array
where	Construct array from binary result
compress	Elements of array where condition is true
clip	Clip array between two values
ones	Array of all ones
identity	2-D identity array (matrix)

Math functions:

absolute,add, arccos, arccosh, arcsin, arcsinh, arctan, arctan2,arctanh,around, bitwise_and, bitwise_or, bitwise_xor, ceil, conjugate, cos, cosh, divide, divide_safe, equal, exp, fabs, floor, fmod, greater, greater_equal, hypot, invert, left_shift, less, less_equal, log, log10, logical_and, logical_not, logical_or, logical_xor, maximum, minimum, multiply, negative, not_equal, power, right_shift, sign, sin, sinh, sqrt, subtract, tan, tanh,

Type handling functions:

iscomplexobj	Test for complex object, scalar result
isrealobj	Test for real object, scalar result
iscomplex	Test for complex elements, array result
isreal	Test for real elements, array result
imag	Imaginary part
real	Real part
real_if_close	complex with tiny imaginary part to real
isneginf	Tests for negative infinity
isposinf	Tests for positive infinity
isnan	Tests for nans
isinf	Tests for infinity
isfinite	Tests for finite numbers
isscalar	True if argument is a scalar
nan_to_num	Replaces NaN/inf with 0/large numbers
cast	Dictionary of functions to force cast to each type
common_type	minimum common type code
mintypecode	Return minimal allowed common typecode

Indexing and shape manipulation:

mgrid	construction of N-d 'mesh-grids'
r_	append and construct arrays
index_exp	building complicated slicing syntax
squeeze	length-one dimensions removed
atleast_1d	Force arrays to be > 1D
atleast_2d	Force arrays to be > 2D
atleast_3d	Force arrays to be > 3D
vstack	Stack arrays vertically (row on row)
hstack	Stack arrays horizontally (column on column)
column_stack	Stack 1D arrays as columns into 2D array
dstack	Stack arrays depthwise (along third dimension)
split	Divide array into a list of sub-arrays
hsplit	Split into columns
vsplit	Split into rows
dsplit	Split along third dimension
who	print numeric arrays and memory utilization

Matrix and polynomial functions:

fliplr	2D array with columns flipped
flipud	2D array with rows flipped
rot90	Rotate a 2D array a multiple of 90 degrees
eye	Return a 2D array with ones down a given diagonal
diag	Construct 2D array from vector or return diagonal from 2D array
mat	Construct a Matrix
bmat	Build a Matrix from blocks
poly1d	A one-dimensional polynomial class
poly	Return polynomial coefficients from roots
roots	Find roots of polynomial given coefficients
polyint	Integrate polynomial
polyder	Differentiate polynomial
polyadd	Add polynomials
polysub	Substract polynomials
polymul	Multiply polynomials
polydiv	Divide polynomials
polyval	Evaluate polynomial at given argument

SciPy Packages

There are a bunch of packages in the `scipy` namespace, organized by subject area:

Name	Description
Cluster	vector quantization/kmeans
Fftpack	discrete fourier transform algorithms
Integrate	integration routines
Interpolate	interpolation tools
Linalg	linear algebra routines
Misc	various utilities (including Python Imaging Library)
dimage	n-dimensional image tools
Optimize	optimization tools
Signal	signal processing tools
Sparse	sparse matrices
Stats	statistical functions
Io	data input and output
Lib	wrappers to external libraries (BLAS, LAPACK)
Sandbox	incomplete, poorly-tested, or experimental code
Special	definitions of many usual math functions
Weave	C/C++ integration

There is way too much detail to cover them all - resort to the DocStrings and `help` function, as well as documentation at `www.scipy.org`

Using Interactive Help

```
>>> import scipy
>>> help(scipy)
Help on package scipy:

NAME
    scipy

FILE
    /util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/site-packages/scipy/__init__.py

DESCRIPTION
    SciPy: A scientific computing package for Python
    ================================================

    Documentation is available in the docstrings and
    online at http://docs.scipy.org.

    Contents
    --------
    SciPy imports all the functions from the NumPy namespace, and in
    addition provides:
...
```

Getting more help from SciPy itself:

```
>>> import scipy
>>> from scipy import optimize,integrate         # imports selected SciPy modules
>>> help(scipy)                                  # info on SciPy module
>>> scipy.info(scipy)                            # ditto
>>> help(scipy.optimize)                         # detailed help on optimize
>>> help(scipy.integrate)                        # ditto for integrate
```

Drilling Down Into A SciPy Package

Ok, how about a concrete example - we want to optimize, so what kind of optimization methods are available?

```
>>> help(scipy.optimize)
Help on package scipy.optimize in scipy:

NAME
    scipy.optimize
FILE
/util/python-epd/epd-7.3-1-rh5-x86_64/lib/python2.7/site-packages/scipy/optimize/__init__.py

DESCRIPTION
    =====================================================
    Optimization and root finding (:mod:`scipy.optimize`)
    =====================================================

    .. currentmodule:: scipy.optimize

    Optimization
    ============

    General-purpose
    ---------------
    .. autosummary::
       :toctree: generated/

       fmin - Nelder-Mead Simplex algorithm
       fmin_powell - Powell's (modified) level set method
       fmin_cg - Non-linear (Polak-Ribiere) conjugate gradient algorithm
       fmin_bfgs - Quasi-Newton method (Broydon-Fletcher-Goldfarb-Shanno)
       fmin_ncg - Line-search Newton Conjugate Gradient
       leastsq - Minimize the sum of squares of M equations in N unknowns
```

```
    Constrained (multivariate)
    --------------------------

    .. autosummary::
       :toctree: generated/

       fmin_l_bfgs_b - Zhu, Byrd, and Nocedal's constrained optimizer
       fmin_tnc - Truncated Newton code
       fmin_cobyla - Constrained optimization by linear approximation
       fmin_slsqp - Minimization using sequential least-squares programming
       nnls - Linear least-squares problem with non-negativity constraint

    Global
    ------

    .. autosummary::
       :toctree: generated/

       anneal - Simulated annealing
       brute - Brute force searching optimizer

    Scalar function minimizers
    --------------------------

    .. autosummary::
       :toctree: generated/

       fminbound - Bounded minimization of a scalar function
       brent - 1-D function minimization using Brent method
       golden - 1-D function minimization using Golden Section method
       bracket - Bracket a minimum, given two starting points
```

```
    Fitting
    =======

    .. autosummary::
       :toctree: generated/

       curve_fit -- Fit curve to a set of points

    Root finding
    ============

    Scalar functions
    ----------------

    .. autosummary::
       :toctree: generated/

       brentq - quadratic interpolation Brent method
       brenth - Brent method, modified by Harris with hyperbolic extrapolation
       ridder - Ridder's method
       bisect - Bisection method
       newton - Secant method or Newton's method

    Fixed point finding:

    .. autosummary::
       :toctree: generated/

       fixed_point - Single-variable fixed-point solver
```

```
    Multidimensional
    ----------------

    General nonlinear solvers:

    .. autosummary::
       :toctree: generated/

       fsolve - Non-linear multi-variable equation solver
       broyden1 - Broyden's first method
       broyden2 - Broyden's second method

    Large-scale nonlinear solvers:

    .. autosummary::
       :toctree: generated/

       newton_krylov
       anderson

    Simple iterations:

    .. autosummary::
       :toctree: generated/

       excitingmixing
       linearmixing
       diagbroyden

    :mod:`Additional information on the nonlinear solvers <scipy.optimize.nonlin>`
```

```
    Utility Functions
    =================

    .. autosummary::
       :toctree: generated/

       line_search - Return a step that satisfies the strong Wolfe conditions
       check_grad - Check the supplied derivative using finite differences

PACKAGE CONTENTS
    _cobyla
    _lbfgsb
    _minpack
    _nnls
    _slsqp
    _tstutils
    _zeros
    anneal
    cobyla
```

and the first method is just our old friend, simplex:

```
>>> from scipy import optimize        # imports all of scipy.optimize
>>> help(optimize.fmin)               # optimize.fmin
Help on function fmin in module scipy.optimize.optimize:

fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None, full_output=0,
disp=1, retall=0, callback=None)
    Minimize a function using the downhill simplex algorithm.

    This algorithm only uses function values, not derivatives or second
    derivatives.

    Parameters
    ----------
    func : callable func(x,*args)
        The objective function to be minimized.
    x0 : ndarray
        Initial guess.
    args : tuple
        Extra arguments passed to func, i.e. ``f(x,*args)``.
    callback : callable
        Called after each iteration, as callback(xk), where xk is the
        current parameter vector.
```

```
    Returns
    -------
    xopt : ndarray
        Parameter that minimizes function.
    fopt : float
        Value of function at minimum: ``fopt = func(xopt)``.
    iter : int
        Number of iterations performed.
    funcalls : int
        Number of function calls made.
    warnflag : int
        1 : Maximum number of function evaluations made.
        2 : Maximum number of iterations reached.
    allvecs : list
        Solution at each iteration.

    Other parameters
    ----------------
    xtol : float
        Relative error in xopt acceptable for convergence.
    ftol : number
        Relative error in func(xopt) acceptable for convergence.
    maxiter : int
        Maximum number of iterations to perform.
    maxfun : number
        Maximum number of function evaluations to make.
    full_output : bool
        Set to True if fopt and warnflag outputs are desired.
    disp : bool
        Set to True to print convergence messages.
    retall : bool
        Set to True to return list of solutions at each iteration.
```

```
Notes
-----
Uses a Nelder-Mead simplex algorithm to find the minimum of function of
one or more variables.

This algorithm has a long history of successful use in applications.
But it will usually be slower than an algorithm that uses first or
second derivative information. In practice it can have poor
performance in high-dimensional problems and is not robust to
minimizing complicated functions. Additionally, there currently is no
complete theory describing when the algorithm will successfully
converge to the minimum, or how fast it will if it does.

References
----------
Nelder, J.A. and Mead, R. (1965), "A simplex method for function
minimization", The Computer Journal, 7, pp. 308-313
Wright, M.H. (1996), "Direct Search Methods: Once Scorned, Now
Respectable", in Numerical Analysis 1995, Proceedings of the
1995 Dundee Biennial Conference in Numerical Analysis, D.F.
Griffiths and G.A. Watson (Eds.), Addison Wesley Longman,
Harlow, UK, pp. 191-208.
```

SciPy Arrays

Basic array construction:

```
>>> a1=array([1,2,3])
>>> a1c=array([1,2,3],dtype=complex)
>>> a1c
array([ 1.+0.j,  2.+0.j,  3.+0.j])
>>> a2=arange(1,9,2)     # start, stop, increment
>>> a2
array([1, 3, 5, 7])
>>> myarray, stepsize = linspace(0,5,num=10,endpoint=False,retstep=True)
>>> myarray
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
>>> stepsize
0.5
>>> r_[1:10:4]           # same as arange(1,10,4) (row-wise concatenation)
array([1, 5, 9])
>>> a3=ones(9)
>>> a3
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> zeros(4)
array([ 0.,  0.,  0.,  0.])
```

SciPy Matrices

```
>>> from numpy import *
>>> m1=mat('1 2; 3 4')
>>> m1
matrix([[1, 2],
        [3, 4]])
>>> m1.T                    # transpose
matrix([[1, 3],
        [2, 4]])
>>> m1.I                    # inverse
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
>>> m1.H                    # conjugate transpose
matrix([[1, 3],
        [2, 4]])
>>> m1*m1                   # matrix multiplication
matrix([[ 7, 10],
        [15, 22]])
>>> m1**2                   # and powers
matrix([[ 7, 10],
        [15, 22]])
```

Mesh grids:

```
>>> mgrid[0:5:4j,0:5:4j]
array([[[ 0.        ,  0.        ,  0.        ,  0.        ],
        [ 1.66666667,  1.66666667,  1.66666667,  1.66666667],
        [ 3.33333333,  3.33333333,  3.33333333,  3.33333333],
        [ 5.        ,  5.        ,  5.        ,  5.        ]],

       [[ 0.        ,  1.66666667,  3.33333333,  5.        ],
        [ 0.        ,  1.66666667,  3.33333333,  5.        ],
        [ 0.        ,  1.66666667,  3.33333333,  5.        ],
        [ 0.        ,  1.66666667,  3.33333333,  5.        ]]])
```

SciPy Namespace

There are too many functions and methods to list - best bet is to use Python module documentation, or browse on the web:

`http://docs.scipy.org`

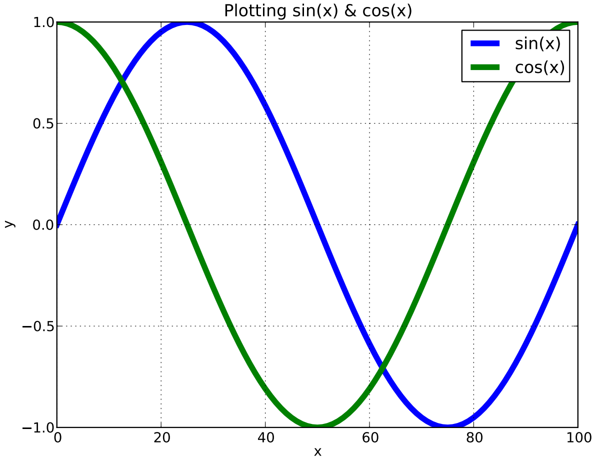
Matplotlib

Matplotlib is a 2D plotting library which produces publication quality figures in a variety of formats. Plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc.

`http://matplotlib.sourceforge.net`

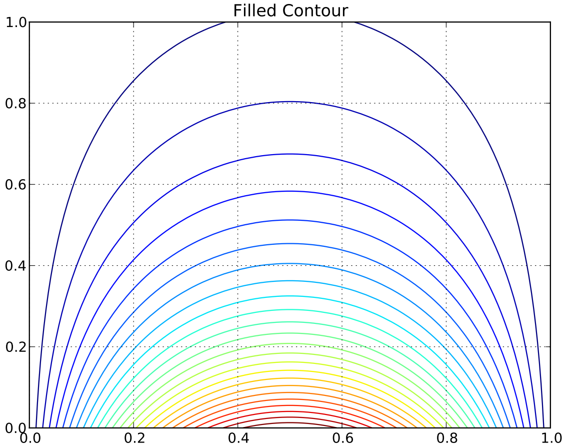
Designed to be easy-to-use (Matlab/Mathematica style) - most functions in the `pylab` Matlab-style interface.

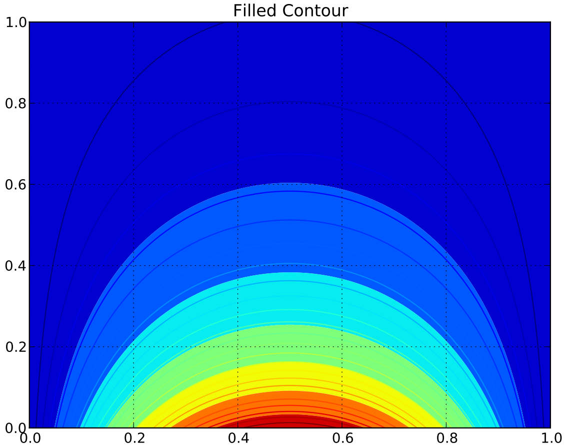
```
#!/usr/bin/env python
from scipy import *
from pylab import *
x = r_[0:101]
y01 = sin(2*pi*x/100)
y02 = cos(2*pi*x/100)
plot(x,y01,linewidth=5.0)
hold(True)
plot(x,y02,linewidth=5.0)
xlabel('x')
ylabel('y')
title('Plotting sin(x) & cos(x)')
legend(('sin(x)','cos(x)'))
grid(True)
show()
```


Plotting sin(x) & cos(x)
sin(x)
cos(x)
1.0
0.5
0.0
−0.5
−1.0
y
0
20
40
60
80
100
x

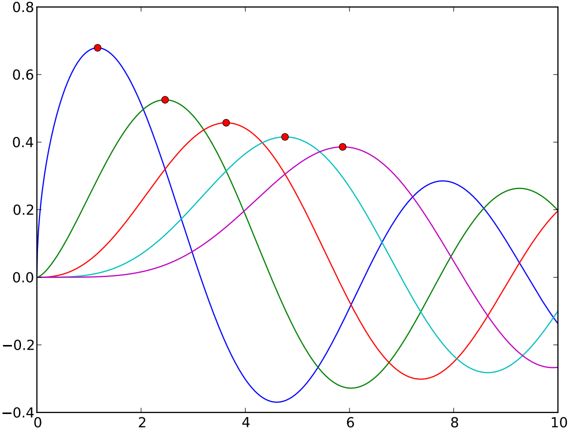
Remember this function on the unit square?

```python
#!/usr/bin/env python
from scipy import *
from pylab import *

x,y = meshgrid(r_[0:1:100j],r_[0:1:100j])

z = sin(math.pi*x)*exp(-math.pi*y)

contour(x,y,z,25)
grid(True)
#contourf(x,y,z)
#grid(True)
title('Filled Contour')
show()
```


Filled Contour
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Filled Contour
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

```
#!/usr/bin/env python
#
# Find (and plot) 1st maximum in some Bessel Functions
#
from scipy import arange, special, optimize
from pylab import *

x = arange(0,10,0.01)

for k in arange(0.5,5.5):
    y = special.jv(k,x)
    plot(x,y)
    hold(True)
    f = lambda x: -special.jv(k,x)
    x_max = optimize.fminbound(f,0,6)
    plot([x_max], [special.jv(k,x_max)],'ro')
show()
```


0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
2
4
6
8
10

Parallel Python

Approaches that you might take with parallel Python:

- threading - not suitable for parallel computation thanks to the infamous GIL (global interpreter lock)
- subprocess - low level control for dynamic process management
- multiprocessing - multiple Python instances
- MPI - e.g., **mpi4py** allows full use of local MPI implementation

Python & threads

Python uses threads already:

- POSIX threads, or `pthreads`
- Shares memory address space with parent processes
- Light weight, pretty low latency
- GIL - global interpreter lock, keeps memory coherent, but allows only a single thread to run at any given time. So Python is essentially serialized by the GIL.

The GIL does make Python safer and easier to code in, but it prevents much in the way of parallel performance gains

Thread Example

```python
from threading import Thread, Lock
import random

lock = Lock() # lock for making operations atomic

def calcInside(nsamples,rank):
    global inside # we need something everyone can share
    random.seed(rank)
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x*x)+(y*y)<1:
            lock.acquire() # GIL does not always save you
            inside += 1
            lock.release()

if __name__ == '__main__':
    nt=1 # thread count
    inside = 0 # you need to initialize this
    samples=int(12e6/nt)
    threads=[Thread(target=calcInside, args=(samples,i)) for i in range(nt)]

    for t in threads: t.start()
    for t in threads: t.join()

    print (4.0*inside)/(1.0*samples*nt)
```

(borrowed from SC10 Python tutorial).

Example Results

Results from running threaded version:

```
[i04n11:~/d_python]$ /usr/bin/time python pi-thread.py
3.14109
13.51user 0.20system 0:13.77elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+96347minor)pagefaults 0swaps
[i04n11:~/d_python]$ vi pi-thread.py
[i04n11:~/d_python]$ /usr/bin/time python pi-thread.py
3.14139066667
59.11user 36.94system 1:14.10elapsed 129%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+96352minor)pagefaults 0swaps
[i04n11:~/d_python]$ /usr/bin/time python pi-thread.py
3.14111366667
42.89user 83.36system 1:26.62elapsed 145%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+96360minor)pagefaults 0swaps
[i04n11:~/d_python]$ vi pi-thread.py
[i04n11:~/d_python]$ /usr/bin/time python pi-thread.py
3.141613
49.61user 71.33system 1:24.38elapsed 143%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+96372minor)pagefaults 0swaps
```

Results for 1,2,4,8 threads ...

Multiprocessing

`multiprocessing` module added in Python >=2.6:

- Sidesteps GIL
- Uses subprocesses (local and remote)
- IPC through pipes and queues, synchronization by locks

Multiprocessing Example

```
import multiprocessing as mp
import numpy as np
import random
import os

#processes = mp.cpu_count()
processes = `os.environ.get("OMP_NUM_THREADS")`  # keep it consistent with OpenMP usage
print "Nprocs = ",processes
processes = int(eval(processes))
nsamples = int(12e6/processes)

def calcInside(rank):
    inside = 0
    random.seed(rank)
    for i in range(nsamples):
        x = random.random();
        y = random.random();
        if (x*x)+(y*y)<1:
            inside += 1
    return (4.0*inside)/nsamples

if __name__ == '__main__':
    pool = mp.Pool(processes)
    result = pool.map(calcInside, range(processes))
    print np.mean(result)
```

Multiprocessing Example

```
[i01n02:~/d_python]$ /usr/bin/time python pi-mp.py
Nprocs =  '1'
3.14109
6.55user 0.22system 0:08.34elapsed 81%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (36major+101460minor)pagefaults 0swaps
[i01n02:~/d_python]$ export OMP_NUM_THREADS=2
[i01n02:~/d_python]$ /usr/bin/time python pi-mp.py
Nprocs =  '2'
3.14171066667
6.90user 0.15system 0:03.98elapsed 177%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+102046minor)pagefaults 0swaps
[i01n02:~/d_python]$ export OMP_NUM_THREADS=4
[i01n02:~/d_python]$ /usr/bin/time python pi-mp.py
Nprocs =  '4'
3.14133766667
6.62user 0.17system 0:02.01elapsed 338%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+103028minor)pagefaults 0swaps
[i01n02:~/d_python]$ export OMP_NUM_THREADS=8
[i01n02:~/d_python]$ /usr/bin/time python pi-mp.py
Nprocs =  '8'
3.14137566667
6.65user 0.20system 0:01.08elapsed 631%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+105212minor)pagefaults 0swaps
[i01n02:~/d_python]$
```

MPI and Python

There are several Python packages that interface with MPI (the following appear to the best supported and most stable):

- https://bitbucket.org/mpi4py
- http://pympi.sourceforge.net/
- http://sourceforge.net/projects/pypar

mpi4py MPI

- Still have task starting issue (which causes a significant dependency on your choice of MPI implementation)
- Result is that most python distributions lack built-in MPI support, and you will need to install your own based on the MPI flavor that you want to use.
- Works best with `numpy` data types, but can use any serialized object (does require `numpy`)
- CCR already has an install of `mpi4py` ...

mpi4py MPI Attributes

- `mpi4py` jobs still started with `mpiexec` (exact nature of startup depends on underlying MPI implementation)
- Each MPI process has its own python interpreter, access to files and libraries local to it (unless explicitly distributed)
- MPI handles communications
- Functions outside a conditional based on rank is assumed to be global
- Any limitations in underlying MPI implementation are inherited by `mpi4py`

More `mpi4py` Details

- The `import MPI` calls `MPI_Init`, generally you can skip calling `MPI_Init` or `MPI_Init_thread` (in fact calling `MPI_Init` more than once violates the standard and can cause failures) - use `Is_initialized` function to test if you need to
- Similarly `MPI_Finalize` is automatically run when python interpreter exits (can use `Is_finalized` if you want)
- Message buffers can be any serialized object - in python they will be pickled unless they are strings or integers
  - Pickling has significant overhead on sends and receives - use lower case methods `recv(), send()`
  - MPI datatypes are **not** pickled - near the speed/overhead of C use capitalized methods, `Recv(), Send()`
  - `numpy` datatypes are converted to MPI datatypes
  - Similar features in MPI collectives

mpi4py Example

```
from mpi4py import MPI
import numpy as np
import random

wt1 = MPI.Wtime()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()
nsamples = int(12e6/mpisize)

inside = 0
random.seed(rank)
for i in range(nsamples):
    x = random.random()
    y = random.random()
    if (x*x)+(y*y)<1:
        inside += 1

mypi = (4.0 * inside)/nsamples
pi = comm.reduce(mypi, op=MPI.SUM, root=0)
wt2 = MPI.Wtime()

if rank==0:
    print (1.0 / mpisize)*pi
    print "Elapsed wall time = ",wt2-wt1
```

Show `mpi4py` example ...

```
[i01n02:~/d_python]$ mpiexec -np 1 python pi-mpi.py
3.14109
Elapsed wall time =  10.1332399845
[i01n02:~/d_python]$ mpiexec -np 2 python pi-mpi.py
3.14171066667
Elapsed wall time =  5.03412413597
[i01n02:~/d_python]$ mpiexec -np 4 python pi-mpi.py
3.14133766667
Elapsed wall time =  2.54429721832
[i01n02:~/d_python]$ mpiexec -np 8 python pi-mpi.py
3.14137566667
Elapsed wall time =  1.27964806557
```

Python Packaging

Finding a python distribution is not difficult - most open-source operating systems come with one pre-built, or as in the case of Mac OS X a python distribution can be relatively easily installed (e.g., using ports). There are always a few interesting twists:

- Does it have all of the modules that you want? If not, installing them into the distribution may not be all that straightforward, even if you have the necessary privileges ...
- Are the modules built the way you need them to be? `numpy/scipy` are really nice when they utilize highly tuned LAPACK/BLAS libraries (e.g., Intel's MKL), but stock versions are often not linked against optimized libraries
- Are the modules sufficiently recent to have the features that you want? `ipython`, a Mathematica-like notebook environment for python, has been moving so fast recently that most OS-based python distributions are very far behind indeed ...

Python Distributions

Some (certainly not all) solutions to the packaging issues:

- Commercial - Anaconda, from the developer of `numpy`:
  `http://continuum.io`
- Commercial - Enthought Python Distribution (EPD), free for academic use:
  `http://www.enthought.com`
- `virtualenv`, lets you install and manage your own python distributions:
  `http://www.virtualenv.org`
- `modules`, on HPC environments (like CCR) you can find alternate python distributions installed in different locations
- Python for HPC, gathering place for python-enabled HPC (tutorials, workshops, etc.): `http://www.pyhpc.org`