Functions & modules

Functions

Python can be both *procedural* (using functions) and *object oriented* (using classes)

[We do objects tomorrow, but much of the function stuff now will also be applicable.]

Functions looks like:

```
def function_name(arg1,arg2,...,kw1=v1,kw2=v2,kw3=v3...)
```

argX are *arguments*

required

(and sequence is important)

kwX are *keywords*

optional

(sequence unimportant; vals act like defaults)

Functions

You can name a function anything you want as long as it:

- contains only numbers, letters, underscore
- does not start with a number
- is not the same name as a *built-in* function (like print)

There is no difference between *functions* and *procedures:*

unlike, say in, IDL, in Python
functions that return nothing
formally, still return None

```
>>> def addnums(x,y):
        return x + y
>>> addnums(2,3)
5
>>> print addnums(0x1f,3.3)
34.3
>>> print addnums("a","b")    # oh no!
ab
>>> print addnums("cat",23232)
TypeError: cannot concatenate 'str' and 'int' objects
```

Unlike in C, we cannot declare what type of variables are required by the function.

```
>>> def addnums(x,y):
    if (not (isinstance(x,float) or isinstance(x,int) or isinstance(x,long))) or \
       (not (isinstance(y,float) or isinstance(y,int) or isinstance(y,long))):
        print "I cannot add these types (" + str(type(x)) + "," + str(type(y)) + ")"
        return
    return x + y
>>> print addnums(2,3.0)
5.0
>>> print addnums(1,"a")
I cannot add these types (<type 'int'>,<type 'str'>) together
None
>>>
```

scope

```
>>> addnums
<function addnums at 0x103767848>
>>> type(addnums)
<type 'function'>
>>> x = 2
>>> print addnums(5,6)
11
>>> print x
2
```

Python has it's own local variables list.
x is not modified globally

```
>>> def numop(x,y):
       x *= 3.14
      return x + y
>>> x = 1
>>> print numop(x,3)
6.14
>>> print x
1
```

scope

...unless you specify that it's a global variable

```
>>> def numop(x,y):
       x *= 3.14
       global a
       a += 1
      return x + y, a
>>> a = 1
>>> numop(1,1)
(4.1400000000000006, 2)
>>> numop(1,1)
(4.1400000000000006, 3)
```

Note: we can return whatever we want (dictionary, tuple, lists, strings, etc.). This is really awesome...

keywords

```
>>> def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
...     if greetings is not None:
...          print greetings
...     return (x + y)*multiplier
>>> numop1(1,1)
Thanks for your inquiry.
2.0
>>> numop1(1,1,multiplier=-0.5,greetings=None)
-1.0
```

keywords are a natural way to grow new functionality without "breaking" old code

*arg, **kwargs captures unspecified args and keywords

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print "-" * 40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ":", keywords[kw]
```

```
>>> cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

http://docs.python.org/tutorial/controlflow.html#keyword-arguments

Documentation: Just the Right thing to Do

and Python makes it dead simple

Docstring: the first unassigned string in a function (or class, method, program, etc.)

```
def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
""" numop1 -- this does a simple operation on two numbers.
     We expect x,y are numbers and return x + y times the multiplier
     multiplier is also a number (a float is preferred) and is optional.
    It defaults to 1.0.
     You can also specify a small greeting as a string. """
       if greetings is not None:
            print greetings
       return (x + y)*multiplier
>>>
```

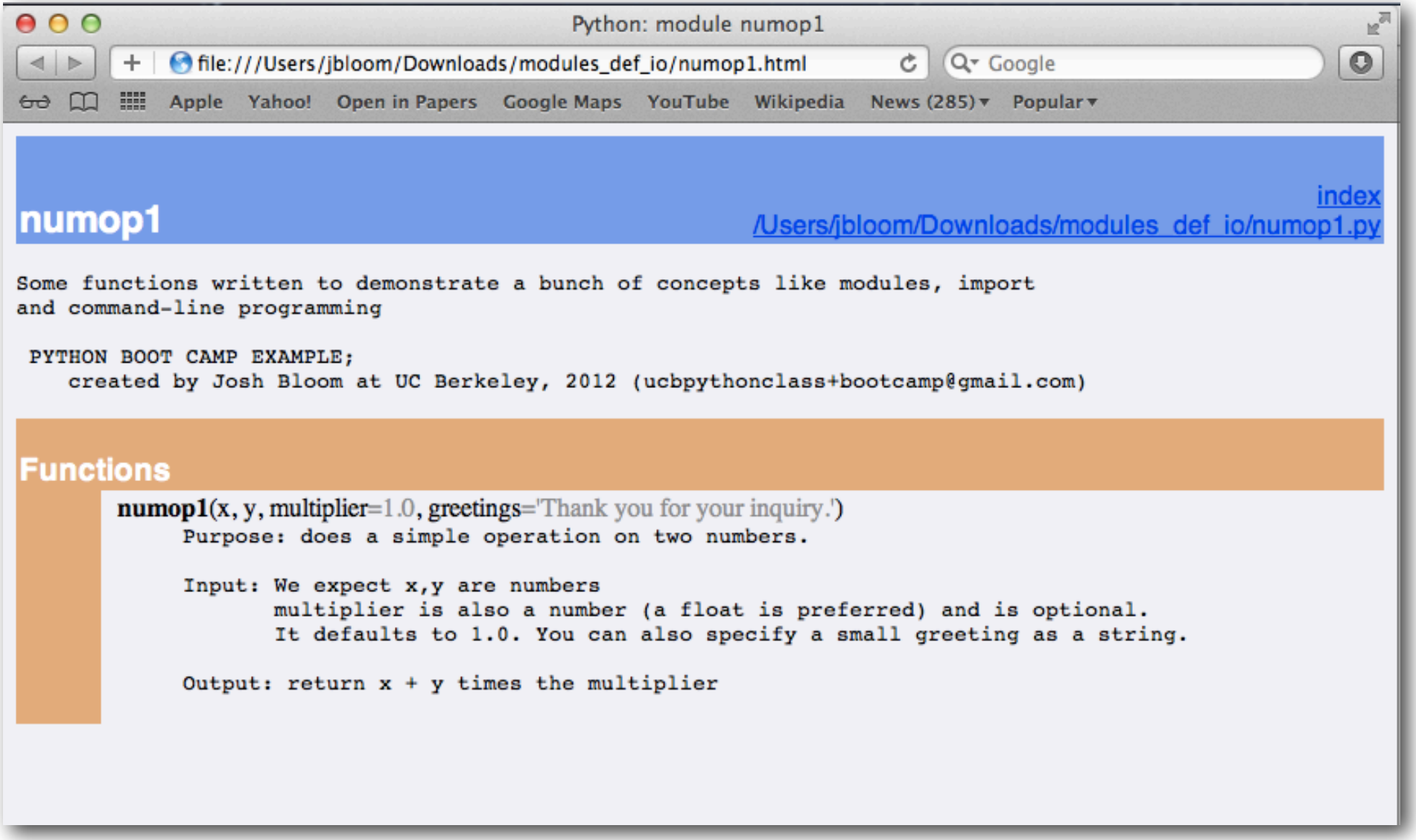
...accessing documentation within the interpreter

```
>>> help(numop1)   # or numop1? in ipython
Help on function numop1:

numop1(x, y, multiplier=1.0, greetings='Thank you for your inquiry.')
    Purpose: does a simple operation on two numbers.

    Input: We expect x,y are numbers
           multiplier is also a number (a float is preferred) and is optional.
           It defaults to 1.0. You can also specify a small greeting as a string.

    Output: return x + y times the multiplier
```

nice looking webpage documentation

assume that function is in file numop1.py

```
BootCamp> pydoc -w numop1
wrote numop1.html
BootCamp>
```


Python: module numop1
file:///Users/jbloom/Downloads/modules_def_io/numop1.html
Google
Apple
Yahoo!
Open in Papers
Google Maps
YouTube
Wikipedia
News (285)
Popular
numop1
index
/Users/jbloom/Downloads/modules_def_io/numop1.py
Some functions written to demonstrate a bunch of concepts like modules, import
and command-line programming
PYTHON BOOT CAMP EXAMPLE;
created by Josh Bloom at UC Berkeley, 2012 (ucbpythonclass+bootcamp@gmail.com)
Functions
numop1(x, y, multiplier=1.0, greetings='Thank you for your inquiry.')
Purpose: does a simple operation on two numbers.
Input: We expect x,y are numbers
multiplier is also a number (a float is preferred) and is optional.
It defaults to 1.0. You can also specify a small greeting as a string.
Output: return x + y times the multiplier

Modules

Organized units (written as files) which contain functions, statements and other definitions

Any file ending in .py is treated as a module
(e.g., numop1.py, which names and defines a function numop1)

Modules: own global names/functions so you can name things whatever you want there and not conflict with the names in other modules

file: numfun1.py

```
"""
small demo of modules
"""
def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
""" numop1 -- this does a simple operation on two numbers.
     We expect x,y are numbers and return x + y times the multiplier
     multiplier is also a number (a float is preferred) and is optional.
    It defaults to 1.0.
     You can also specify a small greeting as a string.
        if greetings is not None:
            print greetings
        return (x + y)*multiplier
```

import *module_name*

gives us access to that module's functions

```
>>> import numfun1
>>> numfun1.numop1(2,3,2,greetings=None)
10
>>> numop1(2,3,2,greetings=None)
NameError: name 'numop1' is not defined
>>>
```

file: numfun2.py

```
"""
small demo of modules
"""
print "numfun2 in the house"
x    = 2
s    = "spamm"

def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
    """
Purpose: does a simple operation on two numbers.

Input: We expect x,y are numbers
       multiplier is also a number (a float is preferred) and is optional.
       It defaults to 1.0. You can also specify a small greeting as a string.

Output: return x + y times the multiplier
    """
    if greetings is not None:
          print greetings
    return (x + y)*multiplier
```

} do some stuff and set some variables

```
>>> import numfun2
numfun2 in the house
>>> import numfun2          # numfun2 is already imported...do nothing
>>>
>>> print numfun2.x, numfun2.s
2, 'spamm'
>>> s = "eggs" ; print s, numop2.s
'eggs', 'spamm'
>>> numop2.s = s
>>> print s, numop2.s
'eggs', 'eggs'
>>> exit()
```

bring some of module's functions into the current namespace:

```
from module_name import function_name
from module_name import variable
from module_name import variable, function_name1,
function_name2, ...
```

```
>>> from numfun2 import x, numop1
numfun2 in the house
>>> x == 2
True
>>> numop1(2,3,2,greetings=None)
5
>>> s
NameError: name 's' is not defined
>>> numfun2.x
NameError: name 'numfun2' is not defined
```

Renaming a function (or variable) for your namespace:

`from module_name import name as my_name`

```
>>> from numfun2 import s as my_fav_food
>>> from numfun2 import numop1 as wicked_awesome_adder
>>> print my_fav_food
'spamm'
>>> wicked_awesome_adder(2,3,1)
5
```

Kitchen-Sinking It

`from module_name import *`

```
>>> from numfun2 import *
>>> print numop1(x,3,1)
5
```

This is convenient in the interpreter, but considered bad coding style. It pollutes your namespace.

Built-In Modules

give access to the full range of what Python can do

For example,

sys	*exposes interpreter stuff & interactions (like environment and file I/O)*
os	*exposes platform-specific OS functions (like file statistics, directory services)*
math	*basic mathematical functions & constants*

These are super battle tested and close to the optimal way for doing things within Python

```
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://www.python.org/doc/2.7.2/lib/module-sys.html

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

    displayhook -- called to show results in an interactive session
    excepthook -- called to handle any uncaught exception other than SystemExit
      To customize printing in an interactive session or to install a custom
      top-level exception handler, assign other functions to replace these.
```

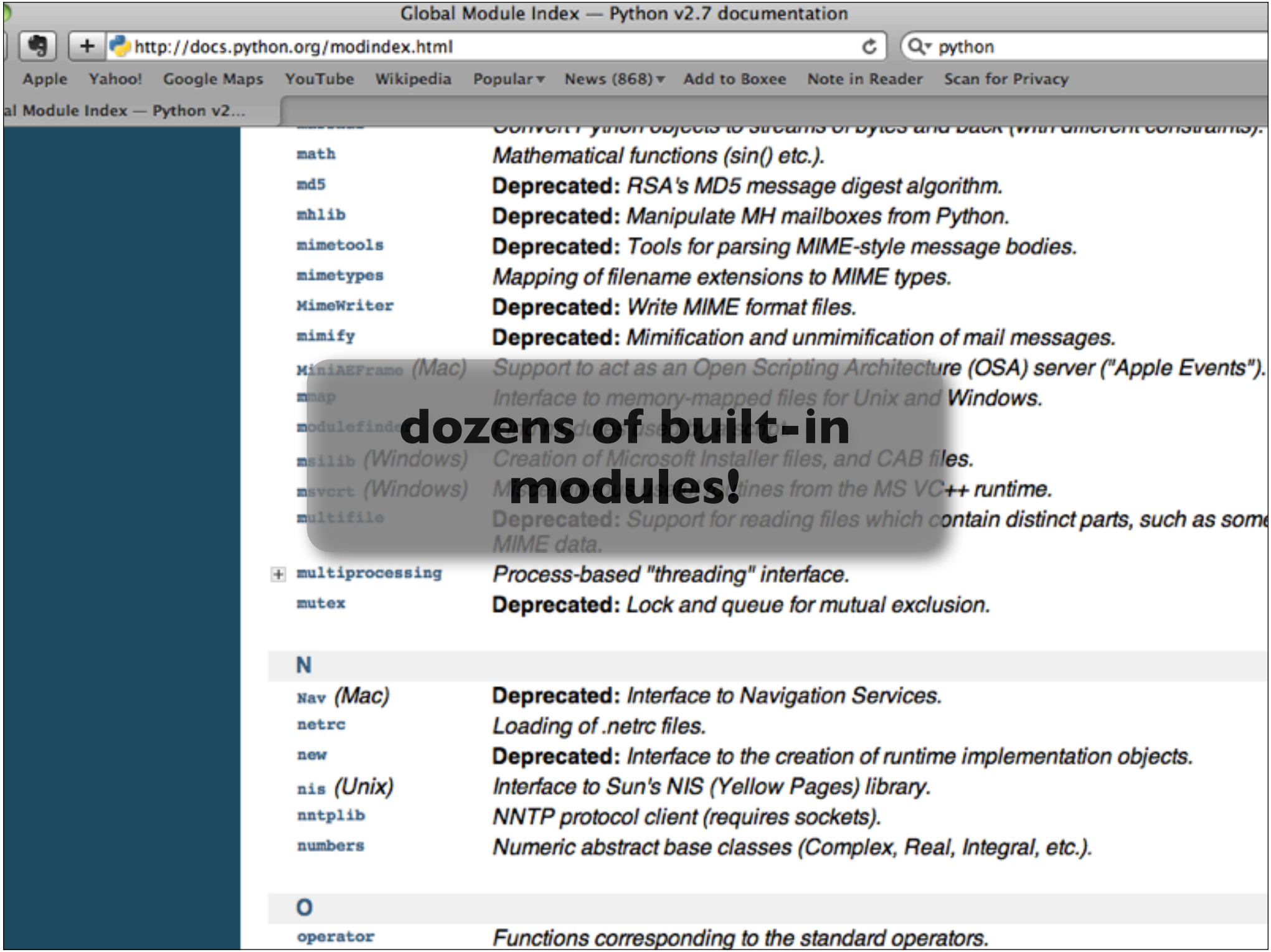
file: getinfo.py

```
import os
import sys

def getinfo(path="."):
    """
Purpose: make simple use of os and sys modules
Input: path (default = "."), the directory you want to list
    """
    print "You are using Python version ",
    print sys.version
    print "-" * 40
    print "Files in the directory " + str(os.path.abspath(path)) + ":"
    for f in os.listdir(path): print f
```

os.listdir() - return a dictionary of all the file names in the specified directory

sys.version() - string representation of the Python (and gcc) version

os.path.abspath() - translation of given pathname to the absolute path (operating system-specific)



dozens of built-in modules!

Making a Script Executable

When a script/module is run from the command line, a special variable called `__name__` is set to `"__main__"`

```
# all your module stuff here

# at the bottom stick...
if __name__ == "__main__":
    """only executed if this module is called from the command line"""
    print "I was called from the command line!"
```

On the first line of a script, say what to run the script with (as with Perl):

```
#!/usr/bin/env python
"""doctring for this module"""
# all your module stuff here
```

set execute permissions of that script

```
BootCamp> chmod a+x script_name.py  ## this works in UNIX, Mac OSX
BootCamp> ./script_name.py
I was called from the command line!
```

file: modfun.py

```
#!/usr/bin/env python
"""
Some functions written to demonstrate a bunch of concepts like modules, import
and command-line programming
"""
import os
import sys

def getinfo(path=".",show_version=True):
    """
Purpose: make simple us of os and sys modules
Input: path (default = "."), the directory you want to list
    """
    if show_version:
        print "-" * 40
        print "You are using Python version ",
        print sys.version
        print "-" * 40

    print "Files in the directory " + str(os.path.abspath(path)) + ":"
    for f in os.listdir(path): print "   " + f
    print "*" * 40

if __name__ == "__main__":
    """
Executed only if run from the command line.
call with
  modfun.py <dirname> <dirname> ...
If no dirname is given then list the files in the current path
    """
    if len(sys.argv) == 1:
        getinfo(".",show_version=True)
    else:
        for i,dir in enumerate(sys.argv[1:]):
            if os.path.isdir(dir):
                # if we have a directory then operate on it
                # only show the version info if it's the first directory
                getinfo(dir,show_version=(i==0))
            else:
                print "Directory: " + str(dir) + " does not exist."
```

```
BootCamp> ./modfun.py
----------------------------------------
You are using Python version  2.7.2 |EPD 7.2-2 (32-bit)| (r265:79063, Jan 11 2012, 15:13:03)
[GCC 4.0.1 (Apple Inc. build 5488)]
----------------------------------------
Files in the directory /Users/jbloom/Classes/BootCamp:
  basic training.key
  data structures.key
  modfun.html
  modfun.py
  modfun.pyc
   ...
****************************************
BootCamp> ./modfun.py . MySpamDir /tmp/
----------------------------------------
You are using Python version  2.7.2 |EPD 6.2-2 (32-bit)| (r265:79063, Jan 11 2012, 15:13:03)
[GCC 4.0.1 (Apple Inc. build 5488)]
----------------------------------------
Files in the directory /Users/jbloom/Classes/BootCamp:
  basic training.key
  data structures.key
  modfun.html
  modfun.py
  modfun.pyc
  modfun.py~
  modules_def_io.key
 ...
****************************************
Directory: MySpamDir does not exist.
****************************************
Files in the directory /tmp:
  .font-unix
  .ICE-unix
  .X0-lock
  .X11-unix
  dao.param
 ...
****************************************
BootCamp>
```

If you make changes to a (module) file and want to reload it into the name space:

`reload(module_name)`

this is also true if you want to reload a module that was imported from an (unchanged) module

```
>>> import os ; os.system("cat josh1.py josh2.py")
# josh1.py
import josh2
x = 1
# josh2.py
y = 2
>>> import josh1 ; print josh1.josh2.y
2
>>> ### now edit josh2
>>> os.system("cat josh1.py josh2.py")
import josh2
x = 1
# josh2.py
y = True
>>> reload(josh1.josh2) ; print josh1.josh2.y
True
```

Breakout Session

exploring some modules

remember: help()

A. create and edit a new file called **age.py**

B. within age.py, import the `datetime` module

- use `datetime.datetime()` to create a variable representing when you were born
- use `datetime.datetime.now()` to create a variable representing now
- subtract the two, forming a new variable, which will be a `datetime.timedelta()` object. Print that variable.

1. how many days have you been alive? How many hours?

2. What will be the date in 1000 days from now?

C. create and edit a new file called **age1.py**

when run from the command line with 1 argument, age1.py should print out the date in days from now. If run with three arguments print the time in days since then

```
BootCamp> ./age1.py 1000
date in 1000 days 2014-10-09 07:40:49.682973
BootCamp> ./age1.py 1980 1 8
days since then... 11699
```