Using Modules

Python's basic unit of reusable code is the *module*. You can access the functions and classes inside a module using the `import` statement. One of the most important modules is the `sys` module:

```
In [89]: import sys
         print sys
```

```
<module 'sys' (built-in)>
```

The `import` statement can also be used to *alias* a module:

```
In [90]: import sys as mysys
         print mysys
```

```
<module 'sys' (built-in)>
```

We can also `import` one or more names from a module:

```
In [91]: from sys import path
         print path
```

```
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-linux2', '/usr/lib/python2.7/lib-tk', '/usr/
```

```
In [92]: from sys import path as mypath
         print mypath
```

```
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-linux2', '/usr/lib/python2.7/lib-tk', '/usr/
```

One of Python's nicest features is its *introspection* capabilities. For instance, to get a list of the properties available on an object, we can use the builtin function `dir()`:

```
In [93]: dir(sys)
```

```
Out[93]: ['__displayhook__',
          '__doc__',
          '__excepthook__',
          '__name__',
          '__package__',
          '__stderr__',
          '__stdin__',
          '__stdout__',
          '_clear_type_cache',
          '_current_frames',
          '_getframe',
          '_mercurial',
          'api_version',
          'argv',
          'builtin_module_names',
          'byteorder',
          'call_tracing',
          'callstats',
          'copyright',
          'displayhook',
          'dont_write_bytecode',
          'exc_clear',
          'exc_info',
          'exc_type',
          'excepthook',
          'exec_prefix',
          'executable',
          'exit',
          'exitfunc',
          'flags',
          'float_info',
          'float_repr_style',
          'getcheckinterval',
          'getdefaultencoding',
          'getdlopenflags',
          'getfilesystemencoding',
          'getprofile',
          'getrecursionlimit',
          'getrefcount',
          'getsizeof',
          'gettrace',
          'hexversion',
          'last_traceback',
          'last_type',
          'last_value',
          'long_info',
          'maxint',
          'maxsize',
          'maxunicode',
          'meta_path',
          'modules',
          'path',
          'path_hooks',
          'path_importer_cache',
          'platform',
          'prefix',
          'py3kwarning',
          'pydebug',
          'setcheckinterval',
          'setdlopenflags',
          'setprofile',
          'setrecursionlimit',
          'settrace',
          'stderr',
```

If we want more information about something, we can also use the `help()` builtin function:

```
In [94]: help(sys.setprofile)
```

```
Help on built-in function setprofile in module sys:

setprofile(...)
    setprofile(function)

    Set the profiling function.  It will be called on each function call
    and return.  See the profiler chapter in the library manual.
```

The `sys` module

`sys` contains functions and variables related to the running Python program. In particular, if you wish to use command-line arguments, these are accessed via `sys.argv`.

```
In [95]: print sys.argv
```

```
['-c', '-f', '/home/vagrant/.ipython/profile_default/security/kernel-d6b2e81a-770d-4c48-912b-a11
```

You can also see the exact executable file containing your Python interpreter:

```
In [96]: print sys.executable
```

```
/usr/bin/python
```

Access to the standard input, output, and error streams is also through the `sys` module:

```
In [97]: print sys.stdin, sys.stdout, sys.stderr
```

```
<open file '<stdin>', mode 'r' at 0x7f951289e150> <IPython.zmq.iostream.OutStream object at 0x15
```

```
In [98]: sys.stderr.write('This is written to the stderr stream\n')
```

```
This is written to the stderr stream
```

```
In [99]: sys.stderr.write('This is written to the stdout stream\n')
```

```
This is written to the stdout stream
```

The `sys.path` variable gives you access to the search path the Python interpreter uses to find modules to `import`:

```
In [100]: sys.path
```

```
Out[100]: ['',
           '/usr/lib/python2.7',
           '/usr/lib/python2.7/plat-linux2',
           '/usr/lib/python2.7/lib-tk',
           '/usr/lib/python2.7/lib-old',
           '/usr/lib/python2.7/lib-dynload',
           '/usr/local/lib/python2.7/dist-packages',
           '/usr/lib/python2.7/dist-packages',
           '/usr/lib/pymodules/python2.7',
           '/usr/lib/python2.7/dist-packages/IPython/extensions']
```

You can also access various constants describing your system such as the largest integer:

```
In [101]: sys.maxint

Out[101]: 9223372036854775807
```

The os module

Where the sys module gives access to information about the current Python process, the os module provides several functions for accessing low-level operating system information:

In [102]:
```
import os
dir(os)
```

Out[102]:
```
['EX_CANTCREAT',
 'EX_CONFIG',
 'EX_DATAERR',
 'EX_IOERR',
 'EX_NOHOST',
 'EX_NOINPUT',
 'EX_NOPERM',
 'EX_NOUSER',
 'EX_OK',
 'EX_OSERR',
 'EX_OSFILE',
 'EX_PROTOCOL',
 'EX_SOFTWARE',
 'EX_TEMPFAIL',
 'EX_UNAVAILABLE',
 'EX_USAGE',
 'F_OK',
 'NGROUPS_MAX',
 'O_APPEND',
 'O_ASYNC',
 'O_CREAT',
 'O_DIRECT',
 'O_DIRECTORY',
 'O_DSYNC',
 'O_EXCL',
 'O_LARGEFILE',
 'O_NDELAY',
 'O_NOATIME',
 'O_NOCTTY',
 'O_NOFOLLOW',
 'O_NONBLOCK',
 'O_RDONLY',
 'O_RDWR',
 'O_RSYNC',
 'O_SYNC',
 'O_TRUNC',
 'O_WRONLY',
 'P_NOWAIT',
 'P_NOWAITO',
 'P_WAIT',
 'R_OK',
 'SEEK_CUR',
 'SEEK_END',
 'SEEK_SET',
 'ST_APPEND',
 'ST_MANDLOCK',
 'ST_NOATIME',
 'ST_NODEV',
 'ST_NODIRATIME',
 'ST_NOEXEC',
 'ST_NOSUID',
 'ST_RDONLY',
 'ST_RELATIME',
 'ST_SYNCHRONOUS',
 'ST_WRITE',
 'TMP_MAX',
 'UserDict',
 'WCONTINUED',
 'WCOREDUMP',
 'WEXITSTATUS',
 'WIFCONTINUED',
 'WIFEXITED',
 'WIFSIGNALED',
In [103]: os.listdir('/usr')

Out[103]: ['include', 'src', 'local', 'lib', 'sbin', 'share', 'bin', 'games']
```

```
In [104]: fd = os.popen('ls -l')
```

```
In [105]: print fd.read()

total 112
-rw-r--r-- 1 vagrant vagrant 37639 Oct  5 00:16 Python Basic Syntax.ipynb
-rw-r--r-- 1 vagrant vagrant 17385 Oct  5 00:32 String Processing.ipynb
-rw-r--r-- 1 vagrant vagrant 45232 Oct  5 00:06 Using Modules.ipynb
-rw-r--r-- 1 vagrant vagrant  4008 Oct  3 03:20 Vagrantfile
```

Besides normal modules in Python, there are also modules containing other modules. These are called *packages*. The `os` module is such a package; inside it is the `os.path` module, used for manipulating filesystem pathnames:

```
In [106]: os.path

Out[106]: <module 'posixpath' from '/usr/lib/python2.7/posixpath.pyc'>
```

```
In [107]: os.path.abspath('.')

Out[107]: '/vagrant'
```

```
In [108]: os.path.dirname(sys.executable)

Out[108]: '/usr/bin'
```

```
In [109]: os.path.basename(sys.executable)

Out[109]: 'python'
```

```
In [110]: os.path.join('/usr/local', 'bin', 'foo')

Out[110]: '/usr/local/bin/foo'
```

```
In [111]: os.path.normpath('/usr/local/bin/../../bin')

Out[111]: '/usr/bin'
```

```
In [112]: os.path.expanduser('~')

Out[112]: '/home/vagrant'
```

```
In [113]: os.path.expandvars('$HOME')

Out[113]: '/home/vagrant'
```

In the `os` module, `os.path` is always available. This is not always the case. In some cases, you must import a submodule directly using a dotted import notation. (In the case of `os.path`, this is not necessary, but it will serve for illustration:

```
In [114]: import os.path
```

The `math` module

Although simple arithmetic operations are supported by Python's syntax, whenever you need to perform more complex math, you'll need to `import` the `math` module:

In [115]:

```python
import math
help(math)
```

```
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the hyperbolic arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)

        Return the hyperbolic arc sine (measured in radians) of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.

    atan2(...)
        atan2(y, x)

        Return the arc tangent (measured in radians) of y/x.
        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(...)
        atanh(x)

        Return the hyperbolic arc tangent (measured in radians) of x.

    ceil(...)
        ceil(x)

        Return the ceiling of x as a float.
        This is the smallest integral value >= x.

    copysign(...)
        copysign(x, y)

        Return x with the sign of y.

    cos(...)
        cos(x)

        Return the cosine of x (measured in radians)
```

```
In [116]: math.sqrt(2)

Out[116]: 1.4142135623730951

In [117]: math.pi

Out[117]: 3.141592653589793

In [118]: math.sin(math.pi / 4)

Out[118]: 0.7071067811865475
```

Exercises

- Create a python script that prints out its command-line arguments
- Update `sys.path` in a Python script to be the empty list. What happens when you try to `import time`?
- Create a Python script that prints out its own absolute path when run using `sys.argv` and `os.path.abspath`
- Calculate the value of e raised to the (j * pi) power

The `time` and `datetime` modules

Working with dates and times in Python is performed using these two modules. The `time` module contains lower-level C-like timestamp manipulation functions (similar to what you would find in `<time.h>`). `datetime` contains higher-level objects for dealing with datetime components:

```
In [119]: import time
          time.time()

Out[119]: 1349368421.4513

In [120]: time.asctime()

Out[120]: 'Fri Oct  5 00:33:41 2012'

In [121]: time.ctime()

Out[121]: 'Fri Oct  5 00:33:41 2012'

In [122]: time.gmtime()

Out[122]: time.struct_time(tm_year=2012, tm_mon=10, tm_mday=4, tm_hour=16, tm_min=33, tm_sec=41, tm_wday=

In [123]: time.mktime(time.gmtime())

Out[123]: 1349339621.0

In [124]: time.localtime()

Out[124]: time.struct_time(tm_year=2012, tm_mon=10, tm_mday=5, tm_hour=0, tm_min=33, tm_sec=41, tm_wday=

In [125]: time.mktime(time.localtime())

Out[125]: 1349368421.0
```

```
In [126]: time.sleep(0.1)
```

```
In [127]: import datetime
          datetime.datetime.now()

Out[127]: datetime.datetime(2012, 10, 5, 0, 33, 41, 665122)
```

```
In [128]: datetime.datetime.utcnow()

Out[128]: datetime.datetime(2012, 10, 4, 16, 33, 41, 672292)
```

```
In [129]: now = datetime.datetime.utcnow()
          print repr(now.date())
          print repr(now.time())

        datetime.date(2012, 10, 4)
        datetime.time(16, 33, 41, 679374)
```

```
In [130]: now.month

Out[130]: 10
```

```
In [131]: now.ctime()

Out[131]: 'Thu Oct  4 16:33:41 2012'
```

```
In [132]: now.strftime('%Y-%m-%d')

Out[132]: '2012-10-04'
```

```
In [133]: datetime.datetime.strptime('2012-10-05', '%Y-%m-%d')

Out[133]: datetime.datetime(2012, 10, 5, 0, 0)
```

```
In [134]: now.timetuple()

Out[134]: time.struct_time(tm_year=2012, tm_mon=10, tm_mday=4, tm_hour=16, tm_min=33, tm_sec=41, tm_wday
```

```
In [135]: time.mktime(now.timetuple())

Out[135]: 1349339621.0
```

```
In [136]: datetime.date.today()

Out[136]: datetime.date(2012, 10, 5)
```

```
In [137]: datetime.date.min

Out[137]: datetime.date(1, 1, 1)
```

```
In [138]: datetime.date.max

Out[138]: datetime.date(9999, 12, 31)
```

```
In [139]: datetime.time.min
```

```
Out[139]: datetime.time(0, 0)
```

```
In [140]: datetime.time.max
```

```
Out[140]: datetime.time(23, 59, 59, 999999)
```

```
In [141]: local_now = datetime.datetime.now()
          utc_now = datetime.datetime.utcnow()
          difference = utc_now - local_now
          difference
```

```
Out[141]: datetime.timedelta(-1, 57600, 136)
```

Files and `StringIO`

We've touched a bit on files (`sys.stdin`, etc.) but not much. Files are opened using the `open` builtin:

```
In [142]: fp = open('Using Modules.ipynb')
```

```
In [143]: fp.read(100)
```

```
Out[143]: '{\n "metadata": {\n  "name": "Using Modules"\n }, \n "nbformat": 2, \n "worksheets": [\n  {\n
```

```
In [144]: fp.seek(10)
          fp.read(100)
```

```
Out[144]: 'ta": {\n  "name": "Using Modules"\n }, \n "nbformat": 2, \n "worksheets": [\n  {\n   "cells":
```

We can also treat a file as a sequence of lines:

```
In [145]: fp.seek(0)
          num_lines = 0
          for line in fp:
              num_lines += 1
          num_lines
```

```
Out[145]: 1856
```

Many places where we might want to use a file, it's actually more convenient to use a string. In those cases, we can create a *file-like object* using the `StringIO` module:

```
In [146]: import StringIO
          fp = StringIO.StringIO('This is a file-like object')
```

```
In [147]: fp.read()
```

```
Out[147]: 'This is a file-like object'
```

```
In [148]: fp.seek(4)
          fp.read(10)

Out[148]: ' is a file'
```

```
In [149]: fp.tell()

Out[149]: 14
```

We can also write to file-like objects:

```
In [150]: fp = StringIO.StringIO()
          fp.write('Hello, there')
```

```
In [151]: fp.seek(0)
          fp.read()

Out[151]: 'Hello, there'
```

We can also get the underlying buffer of the object using `getvalue()`:

```
In [152]: fp.getvalue()

Out[152]: 'Hello, there'
```

Debugging using `pdb`

We can enter an interactive debugger from a Python file by importing the `pdb` module and setting a breakpoint:

```
import pdb
pdb.set_trace()
```

Exercises

- Write a script that prints the current value of `time.time()` every second
- Update the script to print the value of `datetime.datetime.now()`
- Update to print the value of `datetime.datetime.utcnow()`
- Write a function to convert from a datetime object (as in `datetime.datetime.utcnow()` into a "seconds since the epoch" timestamp (as in `time.time()`)
- Create a `StringIO` object that contains several lines, separated by '\n' characters. Pass that object to your function that prints a file with line numbers.
- Add the following line to your file printing function at the beginning: `import pdb; pdb.set_trace()`. Step through the execution of the function using 'n'. (You can also (c)ontinue running the program to exit the debugger.)
- `import hashlib`. Use `dir()` to determine the contents of the hashlib module. Which hashing modules are available?