Modules

Introduction: Modules

Modules are files that contain reusable Python code: we often refer to them as "libraries" because they contain code that can be used in other scripts.

It is possible to *import* such library code directly into our programs through the **import** statement -- this simply means that the functions in the module are made available to our program.

Modules consist principally of functions that do useful things and are grouped together by subject. Here are some examples:

- The **sys** module has functions that let us work with python's interpreter and how it interacts with the operating system
- The **os** module has functions that let us work with the operating system's files, folders and other processes
- The **datetime** module has functions that let us easily calculate date into the future or past, or compare two dates
- The **urllib2** module has functions that let us easily make HTTP requests over the internet So when we **import** a module in our program, we're simply making *other Python code* (in the form of functions) available to our own programs. In a sense we're creating an assemblage of Python code -- some written by us, some by other people -- and putting it together into a single program. The imported code doesn't literally become part of our script, but it is part of our program in the sense that our script can call it and use it. We can also define *our own modules* -- collections of Python functions and/or other variables that we would like to make available to our other Python programs. We can even prepare modules designed for others to use, if we feel they might be useful. In this way we can collaborate with other members of our team, or even the world, by using code written by others and by providing code for others to use.

Objectives for the Unit: Modules

- save a file of Python code and import it into and use it in another Python program
- import individual functions from a module into our Python program
- access a module's functions as its attributes
- manipulate the module's *search path*
- write Python code that can act both as a module and a script
- raise exceptions in our module code, to be handled by the calling code
- design modules with an eye toward reuse by others

- install modules with **pip** or **easy_install**

Summary Statement: import *modulename*

Using **import**, we can import an entire Python module into our own code.

messages.py: a Python *module* that prints messages

```
import sys

def print_warning(msg):
    """write a message to STDOUT"""
    sys.stdout.write('warning:  {}\n'.format(msg))

def log_message(msg):
    """write a message to the log file"""
    try:
        fh = open('log.txt', 'a')
        fh.write(str(msg) + '\n')
    except IOError:
        print_warning('log file not readable')
```

test.py: a Python *script* that imports **messages.py**

```
#!/usr/bin/env python

import messages

print "test program running..."

messages.log_message('this is an important message')
messages.print_warning("I think we're in trouble.")
```

The *global variables* in the module become *attributes* of the module. The module's variables are accessible through the name of the module, as its attributes.

Summary statement: import *modulename* as *convenientname*

A module can be renamed at the point of import.

```
import pandas as pd
import datetime as dt

users = pd.read_table('myfile.data', sep=',', header=None)

print "yesterday's date:  {}".format(dt.date.today() - dt.timedelta(days=
```

Summary statement: from *modulename* import *variablename*

Individual variables can be imported by name from a module.

```
#!/usr/bin/env python

from messages import print_warning, log_message

print "test program running..."

log_message('this is an important message')
print_warning("I think we're in trouble.")
```

Summary: module search path

Python must be told where to find our own custom modules.

Python's standard module directories

When it encounters an **import**, Python searches for the module in a selected list of standard module directories. It does not search through the entire filesystem for modules. Modules like **sys** and **os** are located in one of these standard directories.

Our own custom module directories

Modules that we create should be placed in one or more directories that we designate for this purpose. In order to let Python know about our own module directories, we have a couple of options:

PYTHONPATH environment variable

The standard approach to adding our own module directories to the list of those that Python searches is to create or modify the PYTHONPATH environment variable. This colon-separated list of paths indicates any paths to search *in addition* to the ones Python normally searches.

In your Unix .bash_profile file in your home directory, you would place the following command:

```
export PYTHONPATH=$PYTHONPATH:/path/to/my/pylib:/path/to/my/other/pylib
```

...where **/path/to/my/pylib** and **/path/to/my/other/pylib** are paths

In Winows, you can set the **PYTHONPATH** environment variable through the Windows GUI.

manipulating *sys.path*

```
import sys

print sys.path

    # ['', '/Users/dblaikie/lib', '//anaconda/lib/python2.7', '//anaconda
    #  '//anaconda/lib/python2.7/plat-mac', '//anaconda/lib/python2.7/pla
    #  '//anaconda/lib/python2.7/lib-tk', '//anaconda/lib/python2.7/lib-o
    #  '//anaconda/lib/python2.7/lib-dynload', '//anaconda/lib/python2.7/
    #  '//anaconda/lib/python2.7/site-packages/PIL',
    #  '//anaconda/lib/python2.7/site-packages/Sphinx-1.3.1-py2.7.egg',
    #  '//anaconda/lib/python2.7/site-packages/aeosa',
    #  '//anaconda/lib/python2.7/site-packages/setuptools-19.1.1-py2.7.eg

sys.path.append('/path/to/my/pylib')
```

Once a python script is running, Python makes the PYTHONPATH search path available in a list called **sys.path**. Since it is a list, it can be manipulated; you are free to add whatever paths you wish

Summary: coding all Python scripts as modules

All Python scripts should be coded as modules: able to be both imported and executed.

```
#!/usr/bin/env python


""" helloworld.py:  print the "Hello! message """

def print_hello(arg):
    print "Hello, {}!".format(arg)

def main():
    print_hello('World')

if __name__ == '__main__':        # value is '__main__' if this script is
                                  # value is 'helloworld' if this script i
    main()
```

If we run the above script, we'll see "Hello, World!". But if we *import* the above script, we won't see anything. Why? Because the **if** statement above will be true *only if the script is executed*.

This is important behavior, because there are some scripts that we may want to run directly, but also allow others to import (in order to use the script's functions).

Whether we intend to import a script or not, it is considered a "best practice" to build **all of our programs** in this way -- with a "main body" of statements collected under function **main()** and the call to ain() inside the **if __name__ == '__main__'** gate.

Summary: raising exceptions

Causing an exception to be raised is the principal way a module signals an error to the importing script.

A file called **mylib.py**

```
def get_yearsum(user_year):

    user_year = int(user_year)
    if user_year < 1929 or user_year > 2013:
        raise ValueError('year {} out of range'.format(user_year))

    # calculate value for the year

    return 5.9             # returning a sample value (for testing purpos
```

An exception raised by us is indistinguishable from one raised by Python, and we can raise any exception type we wish.

This allows the user of our function to handle the error if needed (rather than have the script fail):

```
import mylib

while True:

    year = raw_input('please enter a year:  ')

    try:
        mysum = mylib.get_yearsum(year)
        break
    except ValueError:
        print 'invalid year:  try again'

print 'mysum is', mysum
```

Summary: installing modules

Third-party modules must be downloaded and installed into our Python distribution.

Unix

```
$ sudo pip search pandas              # searches for pandas in the PyPI re
$ sudo pip install pandas             # installs pandas
```

Installation on Unix requires something called *root permissions*, which are permissions that the Unix system administrator uses to make changes to the system. The below commands include **sudo**, which is a way to temporarily be granted root permissions.

Windows

```
C:\Windows > pip search pandas        # searches for pandas in the PyPI re
C:\Windows > pip install pandas       # installs pandas
```

PyPI: the Python Package Index

The Python Package Index at https://pypi.python.org/pypi (https://pypi.python.org/pypi) is a repository of software for the Python programming language. There are more than 70,000 projects uploaded there, from serious modules used by millions of developers to half-baked ideas that someone decided to share prematurely.

Usually, we encounter modules in the field -- shared through blog posts and articles, word of mouth and even other Python code. But the PPI can be used directly to try to find modules that support a particular purpose.

Summary: the Python *standard distribution* of modules

Modules included with Python are installed when Python is installed -- they are always available.

Python provides hundreds of supplementary modules to perform myriad tasks. The modules do not need to be installed because they come bundled in the Python distribution, that is they are installed at the time that Python itself is installed.

The documentation for the standard library is part of the official Python docs (https://docs.python.org/2/library/index.html).

- various string-related services
- specialized containers (type-specific lists and dicts, pseudohashes, etc.)
- math calculations and number generation
- file and directory manipulation
- persistence (saving data on disk)
- data compression and archiving (e.g., creating zip files)
- encryption
- networking and interprocess (program-to-program) communication
- internet tasks: web server, web client, email, file transfer, etc.
- XML and HTML parsing
- multimedia: audio and image file manipulation
- GUI (graphical user interface) development
- code testing
- etc.

From the Standard Distribution: the time module

time gives us simple access to the current time or any other time in terms of *epoch seconds*, or seconds since the *epoch* began, which was set by Unix developers at January 1, 1970 at midnight!

The module has a simple interface - it generally works with seconds (which are often a float in order to specify milliseconds) and returns a float to indicate the time -- this float value can be manipulated and then passed back into **time** to see the time altered and can be formatted into any display.

```
import time

epochsecs = time.time()       # 1450818009.925441 (current time in epoch

print time.ctime(epochsecs)   # 'Tue Apr 12 13:29:19 2016'

epochsecs = epochsecs + (60 * 60 *24)    # adding one day!

print time.ctime(epochsecs)   # 'Wed Apr 13 13:29:19 2016'

time.sleep(10)                # pause execution 10 seconds
```

datetime.date, datetime.datetime and datetime.timedelta

Python uses the datetime library to allow us to work with dates. Using it, we can convert string representations of date and time (like "4/1/2001" or "9:30") to datetime objects, and then compare them (see how far apart they are) or change them (advance a date by a day or a year).

```
import datetime as dt            # load the datetime library
dt = dt.datetime.now()           # create a new date object set to now
dt = dt.date.today()             # same

dt = datetime(2011, 7, 14, 14, 22, 29)
                                 # create a new date object by setting
                                 # the year, month, day, hour, minute,
                                 # second (milliseconds if desired)

dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
                                 # create a new date object
                                 # by giving formatted date and time,
                                 # then telling datetime what format we u
```

Once a datetime object has been created, we can view the date in a number of ways

```
print dt                          # print formatted (ISO) date
                                  # 2011-07-14 14:22:29.045814

print dt.strftime("%m/%d/%Y %H:%M:%S")
                                  # print formatted date/time
                                  # using string format tokens
                                  # '07/14/2011 14:22:29'

print dt.year                     # 2011
print dt.month                    # 7
print dt.day                      # 14
print dt.hour                     # 14
print dt.minute                   # 22
print dt.second                   # 29
print dt.microsecond
print dt.weekday()                # 'Tue'
```

Comparing datetime objects

Often we may want to compare two dates. It's easy to see whether one date comes before another by comparing two date objects as if they were numbers:

```
d1 = datetime(2011, 7, 14, 9, 40, 15)
                                  # new date object:  July 14, 2011  9:40:

d2 = datetime(2011, 6, 14, 9, 30, 00)
                                  # new date object:  June 14, 2011  9:30:

print d1 < d2                     # True

print d2 > d1                     # False
```

"Delta" means change. If we want to measure the difference between two dates, we can subtract one from the other. The result is a timedelta object:

```
td = d1 - d2        # a new timedelta object
print td.days       # 30
print td.seconds    # 615
```

Between June 14, 2011 at 9:30am and July 14, 2011 at 9:45am, there is a difference of 30 days, 10 minutes and 15 seconds. timedelta doesn't show difference in minutes, however: instead, it shows the number of seconds â€“ seconds can easily be converted between seconds and minutes/hours with simple arithmetic.

Changing a date using the timedelta object

Timedelta can also be constructed and used to change a date. Here we create timedelta objects for 1 day, 1 hour, 1 minute, 1 second, and then change d1 to subtract one day, one hour, one minute and one second from the date.

```
from datetime import timedelta
add_day = timedelta(days=1)
add_hour = timedelta(hours=1)
add_minute = timedelta(minutes=1)
add_second = timedelta(seconds=1)

print d1                                  # 2011-07-14 09:40:15.678331

d1 = d1 - add_day
d1 = d1 - add_hour
d1 = d1 - add_minute
d1 = d1 - add_second

print d1.strftime("%m/%d/%Y %H:%M:%S")    # 2011-07-13 08:39:14.678331
```