Useful Python modules

Programming Bootcamp 2015

Day 7 – 6/23/15

Today's schedule

1. Review of modules
2. Useful modules
    a) `optparse` – fancy command line options
    b) `os` – doing things with file systems
    c) `glob` – getting lists of files
    d) `subprocess` – system commands from within python
    e) `time` - get the system time, create a timer
3. Other modules
4. Odds 'n ends

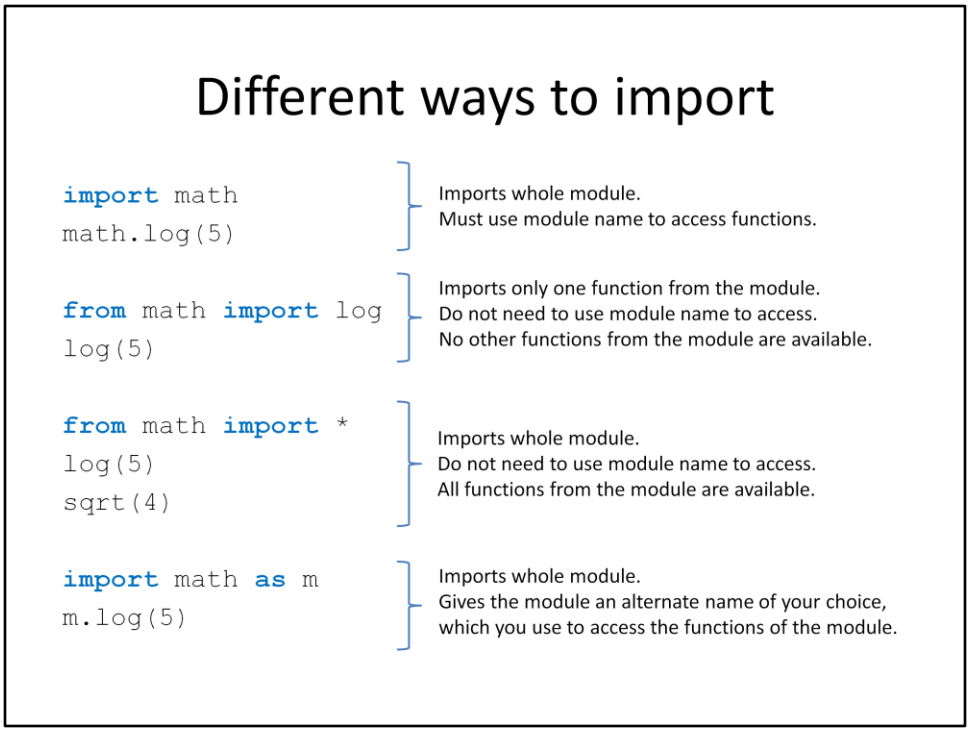
1. Review of modules

What is a module?

- In Python, a module is a separate file containing code and functions which can be imported into other scripts
- We've already used several built-in Python modules: `math, random, sys ...`
- If you did lab 6, you've actually already created your own module: `my_utils.py`

Built-in modules

- Built-in modules are those that come with every Python installation
- They must be imported before they can be used (unlike built-in *functions*, such as `int()`)
- Since anyone can make a module, there are also many non-built-in modules out there, but you'll have to download them separately.
  - Example: SciPy and NumPy – not built-in, but super useful and well-developed (http://www.scipy.org/)
- Today we'll go over a few useful built-in modules



Although option #3 (import *) seems the easiest, I generally don't recommend it. It makes your code very confusing to anyone else who has to read it later, especially if you are importing several modules. They'll be left wondering where this mysterious function you're using came from. Explicitly stating the module name next to the function clarifies this a lot. Also, when not using the module name, there's potential for problems when two functions have the same name.

2a. optparse

optparse

Purpose: Create nicer command line options

Example:

```
python myscript.py -a -b --file=data.txt --cpu=4
```

More info:

http://www.alexonlinux.com/pythons-optparse-for-human-beings

http://docs.python.org/2/library/optparse.html

optparse – Set up parser

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)
```

optparse – Set up parser

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)
```

A "usage message" – this will automatically be printed if the user uses the `-h` option. `%prog` is automatically replaced with the name of your script.

Arguments that are absolutely required should not be added as options, but rather as "bare arguments". This is indicated to the user by listing such requirements here.

Creates a "parser". Right now, this parser will not recognize any command line options--we have to explicitly define which options we expect to see.

optparse – Add options

```
parser.add_option("--out", action="store", dest="OUTFILE",
default=None, help="Optional file to print output.")
```

optparse – Add options

```
parser.add_option("--out", action="store", dest="OUTFILE",
default=None, help="Optional file to print output.")
```

If the user specifies this option, the parser will store the indicated value in a variable called `OUTFILE`

If the option is **not** specified by the user, this default value will be stored in `OUTFILE`. The default can be pretty much anything you want.

If the user types `-h` or `--help` after the script name, a help message will be printed that contains info about each possible option. The help message for each command is defined here.

This allows the parser to recognize an option called `--out`.
The user can specify the option like so:

```
python script.py --out=outFile.txt
```

optparse – Add options

```
parser.add_option("--out", action="store", dest="OUTFILE",
default=None, help="Optional file to print output.")
parser.add_option("--cpu", action="store", type='int',
default=1, dest="MAX_CPU", help="Max number of CPUs to
use. Default is %default.")
```

optparse – Add options

```
parser.add_option("--out", action="store", dest="OUTFILE",
default=None, help="Optional file to print output.")
parser.add_option("--cpu", action="store", type='int',
default=1, dest="MAX_CPU", help="Max number of CPUs to
use. Default is %default.")
```

Indicates that the type must be an int. If we didn't include this, anything would be accepted and converted to a string.

The `%default` placeholder is automatically replaced by the default value when the help message is invoked.

For this option, we specify that the value supplied to `--cpu` must be an integer. If the user provides a non-integer, a useful error message will be printed and the program will not run.

optparse – Add options

```
parser.add_option("--out", action="store", dest="OUTFILE",
default=None, help="Optional file to print output.")
parser.add_option("--cpu", action="store", type='int',
default=1, dest="MAX_CPU", help="Max number of CPUs to
use. Default is %default.")
parser.add_option("--e-val", action="store", type='float',
default=1.0, dest="E_THRESH", help="E-value threshold.
Default is %default.")
```

optparse – Add options

```
parser.add_option("--out", action="store", dest="OUTFILE",
default=None, help="Optional file to print output.")
parser.add_option("--cpu", action="store", type='int',
default=1, dest="MAX_CPU", help="Max number of CPUs to
use. Default is %default.")
parser.add_option("--e-val", action="store", type='float',
default=1.0, dest="E_THRESH", help="E-value threshold.
Default is %default.")
parser.add_option("--verbose", action="store_true",
default=False, dest="VERBOSE", help="Print progress
messages.")
```

optparse – Add options

```
parser.add_option("--out", action="store", dest="OUTFILE",
default=None, help="Optional file to print output.")
parser.add_option("--cpu", action="store", type='int',
default=1, dest="MAX_CPU", help="Max number of CPUs to
use. Default is %default.")
parser.add_option("--e-val", action="store", type='float',
default=1.0, dest="E_THRESH", help="E-value threshold.
Default is %default.")
parser.add_option("--verbose", action="store_true",
default=False, dest="VERBOSE", help="Print progress
messages.")
```

Indicates that this option is a "flag". If it is used, the flag is set to `True`, otherwise, `False`.

optparse – Read user's choices

After all the options have been added to the parser, we parse the actual command line options that the user specified:

```
(opts, args) = parser.parse_args()
```

This adds all the option values to their respective variable inside of an object called `opts`. Any additional "bare" arguments will get put in a list called `args`.

optparse – All together

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store", dest="OUTFILE", default=None, help="Optional file to
print output.")
parser.add_option("--cpu", action="store", type='int', default=1, dest="MAX_CPU", help="Max
number of CPUs to use. Default is %default.")
parser.add_option("--e-val", action="store", type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose", action="store_true", default=False, dest="VERBOSE", help="Print
progress messages.")

(opts, args) = parser.parse_args()
```

optparse – All together

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store", dest="OUTFILE", default=None, help="Optional file to
print output.")
parser.add_option("--cpu", action="store", type='int', default=1, dest="MAX_CPU", help="Max
number of CPUs to use. Default is %default.")
parser.add_option("--e-val", action="store", type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default ")
parser.add_option("--verbose", action="store_true", default=False, dest="VERBOSE", help="Print
progress messages.")

(opts, args) = parser.parse_args()
inFile = args[0]
```

We are requiring the user to input the name of an input file (that's why it's a bare argument, not an option).

As indicated in our usage message, this will be the FIRST bare argument; therefore we can expect it to be put into the first position of the `args` list.

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py
```

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py

Traceback (most recent call last):
  File "opt_test.py", line 13, in <module>
    inFile = args[0]
IndexError: list index out of range
```

We must specify at least the INFILE!

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py infile.txt
```

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py infile.txt

Opts: {'OUTFILE': None, 'MAX_CPU': 1,
'E_THRESH': 1.0, 'VERBOSE': False}
Args: ['infile.py']
```

This works, and sets all options to their default values.

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py infile.txt
--out=outfile.txt
```

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py infile.txt
--out=outfile.txt

Opts: {'OUTFILE': 'outfile.txt',
'MAX_CPU': 1, 'E_THRESH': 1.0, 'VERBOSE':
False}
Args: ['infile.py']
```

This also works, and changes the output file name to outfile.txt.

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py infile.txt --cpu=X
```

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py infile.txt --cpu=X

Usage: opt_test.py INFILE [options]

opt_test.py: error: option --cpu: invalid
integer value: 'X'
```

Since --cpu is defined to take an integer, this gives an error.

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py --help
```

optparse – Usage

opt_test.py

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store",
dest="OUTFILE", default=None, help="Optional
file to print output.")
parser.add_option("--cpu", action="store",
type='int', default=1, dest="MAX_CPU",
help="Max number of CPUs to use. Default is
%default.")
parser.add_option("--e-val", action="store",
type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose",
action="store_true", default=False,
dest="VERBOSE", help="Print progress
messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

print "Opts:", opts
print "Args:", args
```

Command line:

```
>>> python opt_test.py --help
Usage: opt_test.py INFILE [options]

Options:
  -h, --help          show this help message and exit
  --out=OUTFILE       Optional file to print output.
  --cpu=MAX_CPU       Max number of CPUs to use. Default is 1
  --e-val=E_THRESH    E-value threshold. Default is 1.0
  --verbose           Print progress messages.
```

This prints out a nice help message. -h also works. We do not need to define this option, it is automatically created by the parser.

optparse – Using provided data

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store", dest="OUTFILE", default=None, help="Optional file to
print output.")
parser.add_option("--cpu", action="store", type='int', default=1, dest="MAX_CPU", help="Max
number of CPUs to use. Default is %default.")
parser.add_option("--e-val", action="store", type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose", action="store_true", default=False, dest="VERBOSE", help="Print
progress messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

# example of using the option values
if opts.VERBOSE == True:
    print "Reading in file"
ins = open(inFile, 'r')
for line in ins:
    cols = line.split()
    if cols[2] <= opts.E_THRESH:
        print cols[0] #print only if below threshold
ins.close()
```

optparse – Using provided data

```
import optparse

msg = "Usage: %prog INFILE [options]"
parser = optparse.OptionParser(usage=msg)

parser.add_option("--out", action="store", dest="OUTFILE", default=None, help="Optional file to
print output.")
parser.add_option("--cpu", action="store", type='int', default=1, dest="MAX_CPU", help="Max
number of CPUs to use. Default is %default.")
parser.add_option("--e-val", action="store", type='float', default=1.0, dest="E_THRESH",
help="E-value threshold. Default is %default.")
parser.add_option("--verbose", action="store_true", default=False, dest="VERBOSE", help="Print
progress messages.")

(opts, args) = parser.parse_args()
inFile = args[0]

# example of using the option values
if opts.VERBOSE == True:
    print "Reading in file"
ins = open(inFile, 'r')
for line in ins:
    cols = line.split()
    if cols[2] <= opts.E_THRESH:
        print cols[0] #print only if below threshold
ins.close()
```

So the basic syntax for accessing these values is `opts.`*VAR_NAME*

2b. os

OS

Purpose: Useful functions for working with file names/directory paths.

Example:

```
>>> os.path.exists("test_file.txt")
True
>>> os.mkdir("newFolder")
```

More info:

http://docs.python.org/2/library/os.path.html

http://docs.python.org/2/library/os.html#module-os

os.path

```
>>> import os
>>> os.path.exists("test_file.txt") #checks if file/directory exists
True

>>> os.path.isfile("test_file.txt") #checks if it is a file
True

>>> os.path.isdir("test_file.txt") #checks if it is a directory
False

>>> os.path.getsize("test_file.txt") #gets size of file
18L

>>> os.path.abspath("test_file.txt") #gets absolute/full path of file
'/cygdrive/c/Users/Sarah/Dropbox/Python/PythonBootcamp2013/lab8/test_file.txt'

>>> fullPath = os.path.abspath("test_file.txt")
>>> os.path.basename(fullPath) #extracts file name from longer path
'test_file.txt'
>>> os.path.dirname(fullPath) #extracts path, removes file name
'/cygdrive/c/Users/Sarah/Dropbox/Python/PythonBootcamp2013/lab8'

>>> os.mkdir("newFolder") #makes a new directory
```

A note on file paths

- So far we've mostly worked with input/output files stored in the same directory as our script
- What if we want to work with files stored somewhere else?

```
# open a file in a directory contained
# inside the current directory:
inFile = open("data/input_file.txt", 'r')

# open a file in the directory that contains
# the current directory (parent directory)
inFile = open("../input_file2.txt", 'r')

# open a file using an absolute path (i.e.
# a path that will always work, regardless of the
# current directory location)
inFile = open("/home/sarah/lab7/data/input_file.txt", 'r')
inFile = open("/home/sarah/lab7/input_file2.txt", 'r')
```

2c. glob

glob

Purpose: Get list of files in a folder that match a certain pattern. Good for when you need to read in a large number of files but don't have a list of all their file names.

Example:

```
glob.glob("../data/sequences/*.fasta")
```

Important to note:

The `*` here is a wildcard. So this will match any file in `../data/sequences/` that ends in `.fasta`.

More info:

http://docs.python.org/2/library/glob.html

glob

```
>>> import glob

>>> glob.glob("sequences/*") #get list of everything in "sequences" folder
['sequences/abcde.fasta', 'sequences/asdas123.fasta',
'sequences/README.txt', 'sequences/seq1.fasta', 'sequences/seq2.fasta',
'sequences/seq3.fasta', 'sequences/temp_file.tmp']

>>> glob.glob("sequences/*.fasta") #get list of all with .fasta extension
['sequences/abcde.fasta', 'sequences/asdas123.fasta',
'sequences/seq1.fasta', 'sequences/seq2.fasta', 'sequences/seq3.fasta']

>>> glob.glob("sequences/seq*.fasta") #get everything named seq*.fasta
['sequences/seq1.fasta', 'sequences/seq2.fasta', 'sequences/seq3.fasta']

>>> glob.glob("*") #get list of everything in current folder
['data', 'lab8_useful_modules.pptx', 'newFolder', 'opt_test.py',
'sequences', 'test_file.txt']
```

The * is a wildcard -- it will match anything.

2d. subprocess

subprocess

Purpose: Launch another program or a shell command from within a Python script.

Example:

```
subprocess.Popen("python other_script.py")
```

More info:

http://docs.python.org/2/library/subprocess.html

http://stackoverflow.com/questions/89228/calling-an-external-command-in-python

subprocess

Basic command:

```
job = subprocess.Popen(command)
```

Recommended version:

```
job = subprocess.Popen(command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
```

subprocess

Basic command:

```
job = subprocess.Popen(command)
```

Recommended version:

```
job = subprocess.Popen(command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
```

Allows us to run shell (terminal) commands

If the command would normally output something to the terminal, this allows us to capture that output in a string variable. That way we can read through it in our code and use it, if necessary.

Allows us to capture the "standard error" stream of the command. In other words, this will allow us to check if our command succeeded or gave an error.

subprocess - an example

```
# create and run command, use variable 'job' to access results
command = "blastn -query seq1.fasta -db refseq_rna"
job = subprocess.Popen(command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)

# read whatever this command would have printed to the screen,
# and then actually print it (it's suppressed otherwise)
jobOutput = job.stdout.readlines()
for line in jobOutput:
    print line,

# check for error and ensure that the script does not continue
# until the command has finished executing.
result = job.wait()
if result != 0:
    print "There was an error running the command."
```

subprocess - in a custom function

(in useful_fns.py)

```
# A function that runs the given command using the system shell.
# Returns the output of the command in a list, the result variable, and whether there was an error.
def run_command(command, verbose=False):
    import subprocess
    error = False

    if verbose == True:
        print command
        print ""

    job = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    jobOutput = []
    if verbose == True:
        for line in job.stdout:
            print "   ", line,
            jobOutput.append(line)
    else:
        jobOutput = job.stdout.readlines()
    result = job.wait()
    if result != 0:
        error = True

    return (jobOutput, result, error)
```

subprocess - in a custom function

```
# Using the custom function:
import useful_fns as uf
command = "blastn -query seq1.fasta -db refseq_rna"
(output, result, error) = uf.run_command(command, verbose=True)

# check for error
if error:
    print "Error running command:", command
    print "Exiting."
    sys.exit()

# use output, or whatever
for line in output:
    ...
```

subprocess - a warning

Warning: Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of `shell=True` is **strongly discouraged** in cases where the command string is constructed from external input:

```
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly...
```

`shell=False` disables all shell based features, but does not suffer from this vulnerability; see the Note in the `Popen` constructor documentation for helpful hints in getting `shell=False` to work.

When using `shell=True`, `pipes.quote()` can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

If you set shell = True, this executes the command using the shell. This is good because it lets us do more things, but it's potentially dangerous because it essentially opens up a way for someone to run malicious shell commands (like in the example above, a command to delete all of your files...). Should you worry about this? Probably not, UNLESS you plan to run code **on your computer/server that accepts input from strangers over the internet**. If you're just running the code yourself, or letting other people run the code on their own computers themselves, this is a non-issue.

2e. time

time

Purpose: Get the current system time. Can be used to time your code.

Example:

```
import time

startTime = time.time()
...some code...
elapsedTime = startTime - time.time()
```

`elapsedTime` will hold the difference, in seconds, between the two calls to `time.time()`

More info:

http://docs.python.org/2/library/time.html

Important to note:

`time.time()` returns a `float` that indicates the time, in seconds, since the start of the "epoch" (this is operating system-dependent) at the current moment. It won't make much sense by itself, but we can use it to make simple timers as shown here.

3. Other modules

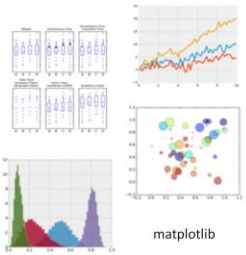
Other useful modules

Built-in:

- `multiprocessing` – functions for writing parallel code that utilizes multiple CPU cores
- `argparse` – the successor to optparse
- `re` – regular expressions (advanced pattern matching)
- `collections` – advanced data structures
- `logging` – facilitates the creation of log files
- `datetime` – for accessing/manipulating date & time info

Not built-in (must be installed before using)

- `SciPy` – scientific/mathematical algorithms
- `NumPy` – advanced math & linear algebra
- `matplotlib` – plotting module for python
- `pandas` – data structures and data analysis



matplotlib

4. Odds 'n ends

This is the end! (of our Python lessons)

- Next time: Overview of R with Derek Oldridge
- Some final odds 'n ends:
  - Nested dictionaries
  - Error handling with `try-except`
  - A whole world of built-in functions...
  - +=
  - Other ways to use and write Python

Nested dictionaries

A dictionary can store almost anything... including other dictionaries! This is useful for when you want to associate several pieces of info with a given key.

Example:

```
inFile = open("gene_positions.txt", 'r')

for line in inFile:
    (geneID, chr, start, end, strand) = line.split()
    geneDict[geneID] = {}       #set this equal to a new hash
    geneDict[geneID]['chromosome'] = chr
    geneDict[geneID]['startPos'] = int(start)
    geneDict[geneID]['endPosition'] = int(end)
    geneDict[geneID]['strand'] = strand

inFile.close()

print geneDict['Actb']['strand'] #example of accessing data
```

You can also make nested lists, or dictionaries in lists, or lists in dictionaries... you get the idea.

Each dictionary within the main dictionary is considered a completely separate entity, so it's fine to re-use keys as long as the keys within any single dictionary are unique.

Error handling with try-except

Purpose: catch a specific error before it causes the script to terminate, and handle the error in a manner of your choosing.

Syntax:

```
try:
    ...some code here...
    ...that might create an error...
except ErrorName:
    ...code to execute if error occurs...
else:
    ...(optional) code to execute if no error...
```

You must provide the specific name of the error type (e.g. `TypeError`, `ValueError`, `IOError`, etc)

Example:

```
try:
    inFile = open(fileName, 'r')
except IOError:
    print "Error: could not open", fileName, "--exiting."
    sys.exit()
for line in inFile:
...
```

You can do anything you want in the except block; you do not have to exit. However, in general it's good form to at least print some kind of message/warning.

The else is optional. You really don't need it if you're just going to exit if there's an error. It's mostly useful when you don't want to exit after the error, because then it lets you have code that will only be executed if there was no error.

You can check for multiple errors at once. You can also create a multi-except (kind of like a multi-elif). See the docs for more examples.

A whole world of built-in functions

- There's tons of stuff I didn't get a chance to tell you about
- In particular, there are several functions out that that automatically do things I made you do manually (sorry! It's for the sake of learning!)
  - String functions: https://docs.python.org/2/library/stdtypes.html#string-methods
    - string.count()
    - string.upper() / string.lower()
    - string.find()
    - string.join()
  - random.choice()
  - many more

+=

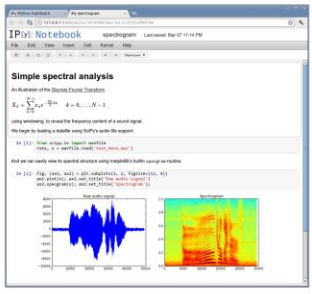
This is a shortcut for adding/concatenating onto a variable. Also valid (for ints): −= , *= , /=

Examples:

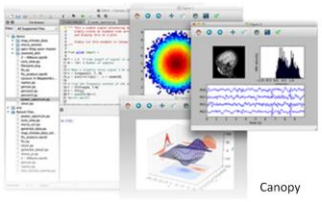
```
count = 0
while count < 100:
    count += 1 #same as count = count + 1

name = ""
for c in "Wilfred"
    name += c #same as name = name + c
```

Other ways to write & use Python

- iPython notebook (http://ipython.org/)– these are like a mixture of a script and the interpreter. Saves code output & figures inline with the code. Very good for exploratory data analysis!
- Enthought Canopy (IDE + modules) (www.enthought.com/products/canopy/) – an advanced coding environment that comes with many useful non-built-in modules pre-installed (e.g. SciPy, NumPy). Also comes with iPython notebooks built-in.



iPython


Canopy