Scraping and Command Line Arguments

Let's talk a little bit about how we write scripts as they're an important part of making your scripts usable to people without Python knowledge or in deployment / automated settings.

- Command line arguments with `sys`
- Using `argparse` for more complex scrips
- `__main__` for dual purpose Python modules
- Using `cron` to run scripts in a virtualenv

Command line args with `sys`

The `sys` object has a number of things packed inside:

In [2]:
```
import sys

print "Our PATH: %s" % sys.path
print "Min/max int values: %d, %d" % ( -sys.maxint - 1, sys.maxint)
print "Command line arguments: %s" % sys.argv
```

```
Our PATH: ['', '/usr/local/lib/python2.7/site-packages/Cython-0.23.4-py
2.7-macosx-10.9-x86_64.egg', '/Users/will/Code/develop_intelligence/pyt
hon/notebooks', '/Users/will/Code/caffe/caffe/python', '/usr/local/Cell
ar/python/2.7.10_2/Frameworks/Python.framework/Versions/2.7/lib/python2
7.zip', '/usr/local/Cellar/python/2.7.10_2/Frameworks/Python.framework/
Versions/2.7/lib/python2.7', '/usr/local/Cellar/python/2.7.10_2/Framewo
rks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin', '/usr/loc
al/Cellar/python/2.7.10_2/Frameworks/Python.framework/Versions/2.7/lib/
python2.7/plat-mac', '/usr/local/Cellar/python/2.7.10_2/Frameworks/Pyth
on.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages',
 '/usr/local/Cellar/python/2.7.10_2/Frameworks/Python.framework/Version
s/2.7/lib/python2.7/lib-tk', '/usr/local/Cellar/python/2.7.10_2/Framewo
rks/Python.framework/Versions/2.7/lib/python2.7/lib-old', '/usr/local/C
ellar/python/2.7.10_2/Frameworks/Python.framework/Versions/2.7/lib/pyth
on2.7/lib-dynload', '/usr/local/lib/python2.7/site-packages', '/usr/loc
al/Cellar/numpy/1.10.4/libexec/nose/lib/python2.7/site-packages', '/us
r/local/Cellar/protobuf/2.6.1/libexec/lib/python2.7/site-packages', '/L
ibrary/Python/2.7/site-packages', '/usr/local/lib/python2.7/site-packag
es/IPython/extensions', '/Users/will/.ipython']
Min/max int values: -9223372036854775808, 9223372036854775807
Command line arguments: ['/usr/local/lib/python2.7/site-packages/ipyker
nel/__main__.py', '-f', '/Users/will/Library/Jupyter/runtime/kernel-fa1
31e23-e579-473d-aacf-1dabb2c91a71.json']
```

A first command line args script

```
#!/Users/will/.virtualenvs/di/bin/python
import sys

for i, arg in enumerate(sys.argv):
    print "arg=%s" % arg

print "There were %d arguments!" % (i + 1,)
```

This is quite simple - we have access to a list of arguments in order on the command line.

If we want to start adding more complex arguments that are optional, flags, and variable length arguments, we'll quickly find ourselves doing a lot of work.

Enter `argparse`

A much more powerful way to write command line scripts:

In [6]:
```
import argparse

"""
    $ python your_script.py --foo mystring --bar 5
    $ python your_script.py --foo otherstring
    $ python your_script.py                                  # ERROR! Req
uired argument needed
    $ python your_script.py --foo string --bar otherstring  # ERROR! Wro
ng type
"""
parser = argparse.ArgumentParser(description='Description of your progra
m')
parser.add_argument('-f','--foo', help='Description for foo argument', r
equired=True, type=str)
parser.add_argument('-b','--bar', help='Description for bar argument', r
equired=False, type=int)
args = parser.parse_args()

# print args.foo, args.bar
```

The `__main__` block & dual purpose files

Sometimes we might want to have module code (reusable & extensible) alongside scripting code (run for a specific task).

The problem is every time we import a script, we'd print or write to disk (which is not what we want when we import). Solution is much like Java - we designate a main method!

In [9]:
```
import sys

def my_important_function(arg1, arg2):
    # ...
    return {'something' : 'cool'}

if __name__ == "__main__":
    # __name__ is set to the module name otherwise
    print "This script was run directly from the command line!"
    print my_important_function(1, 2)
```

```
This script was run directly from the command line!
{'something': 'cool'}
```

Using `cron` to run scripts

If you're familiar with `cron`, you know that the setup is fairly easy to have a predetermined script run at certain intervals.

This script runs every **12am and 12pm on the 1st day of every 2nd month** and puts the output of the command into the log file /var/log/pinglog.

```
$ crontab -e

# inside crontab
# min   hours   day    day-of-month   day-of-week    COMMAND
0       0,12    1      */2            *              /sbin/ping -c 192.168.
0.1 >> /var/log/pinglog
```

(So long as you get your environment and executable locations right, we've all debugged those before...)

Lab: Writing Your Own Script

1) Complete the script at `excercises/script.py`. You'll make use of your previous scraping code and set up a way to query and compute statistics on it.

2) Next, make it run as a cron job every minute (with whatever arguments you like), and put the output into a file called `/tmp/frequencies-log.txt`.

3) Wait a couple minutes (more than 120 seconds) and verify you see more than one result in file with the command: `cat /tmp/frequencies-log.txt`.

Wrap-Up

- Command line arguments with `sys`
- Using `argparse` for more complex scrips
- `__main__` for dual purpose Python modules
- Using `cron` to run scripts in a virtualenv