All I ever needed to know about Python scripting

Matt Harrison
matthewharrison@gmail.com
http://panela.blog-city.com/
SCALE 2009

File Organization

Script

A python file that is run. Ideally found in `PATH`.

Module

A python library that is "imported". Usually a .py file, though can be c extension (.so,.dll) or .pyc. (lowercase no underscores(_))

Package

A directory (the package name) containing *packages* and/or *modules*. Each package must have a `__init__.py` file in it. (lowercase, underscores (_) ok)

Egg

A tarball of python files (similar to .jar files) used by the 3rd party `setuptools` package.

File Layout

Script style (install into `PATH`):

```
ScriptProject/ (Project name)
  script (file containing python code)
  setup.py (specifies packaging)
```

Your script can be a module (install into `PYTHONPATH`):

```
ScriptProject/
  script.py
  setup.py
```

Compromise layout placing logic in a *module/package* and providing a script too:

```
ScriptProject/
  bin/
    script(.py) (loads logic from scriptlib)
  scriptlib/
    __init__.py
  setup.py
  README
  INSTALL
```

Non .py import hack

To import non `.py` files

```
>>> from types import ModuleType
>>> my_script = ModuleType('my_script')
>>> exec open('bin/my_script') in my_script.__dict__
>>> my_script.main(['-t', 'bar'])
```

see http://mail.python.org/pipermail/python-list/2007-February/424702.html

(Intra) File Organization

- `#!/usr/bin/env python`
- `# -*- coding: utf-8 -*-` (PEP 263)
- (vim/emacs settings)
- Copyright/License (in comment)
- Module docstring
- Bookkeeping
  - Version
  - Author
- Imports
  - stdlib
  - 3rd party
  - module
- Globals
- Classes/functions
- `main`
- Conditional main
  - Exit code

Conditional main

```
>>> import sys
>>> def main(prog_args):
...   # implementation
...   pass
...   # return exit code
>>> if __name__ == "__main__":
...   sys.exit(main(sys.argv))
```

Passing in `sys.argv` into `main` lets you "call" it again without monkeypatching `sys.argv`. Enables scripting/testing of command line interface.

Exit code is 0 by default. 0 - success, non-zero - error.

No global execution

Try to break up code into functions. Lowers code complexity. Makes testing easier. Makes reuse easier (can import logic without it executing).

Configuration

There are many types of configuration. A Unix hierarchy looks like this (lowest priority first):

- `/etc/` rc (run control) files
- User's rc files
- User's environment variables
- Command line switches

see http://www.faqs.org/docs/artu/ch10s02.html

.ini File configuration

`ConfigParser` file configuration for rc files using ini-style.

```
>>> import ConfigParser
>>> import StringIO
>>> import os
```

Default location is usually in home directory (or in `~/.config/script`)

```
>>> SCRIPT_INI = os.path.expanduser('~/.script.ini')
>>> cfg = ConfigParser.ConfigParser()
>>> cfg.read(SCRIPT_INI)
[]
```

Could also embed into code (or for testing)

```
>>> ini = StringIO.StringIO('''
... [Properties]
... author = Matt
... email = matthewharrison at gmail.com
... food: pizza
... ; comment
... # other comment
... ''')

>>> cfg =
ConfigParser.ConfigParser(defaults={'author':'Dickens
', 'book':'Two Cities'})
>>> cfg.readfp(ini)

>>> cfg.get('Properties', 'author')
'Matt'
>>> cfg.get('Properties', 'book') # Gets default
'Two Cities'
>>> cfg.get('Properties', 'bad') # Non-existant
Traceback (most recent call last):
...
NoOptionError: No option 'bad' in section:
'Properties'
```

If you want per `get` default values.

```
>>> def getd(cfg, section, option, default,
raw=False, vars=None):
...   ' subclass ConfigParser for OO interface '
...   try:
...     value = cfg.get(section, option, raw=raw,
vars=vars)
...   except ConfigParser.NoOptionError, e:
...     value = default
...   return value

>>> getd(cfg, 'Properties', 'bad', 'not here')
'not here'
```

For non-stdlib versions see
http://wiki.python.org/moin/ConfigParserShootout

xml Configuration

```
>>> from xml.etree.ElementTree import ElementTree
>>> xml_conf = StringIO.StringIO("""<Properties
author="Matt">
... <Email value="mattharrison at gmail.com"/>
... <pet>dog</pet>
... <!-- xml comment -->
... </Properties>""")

>>> tree = ElementTree()
>>> props = tree.parse(xml_conf)
>>> props.attrib['author']
'Matt'

>>> props.find('pet').text
'dog'
```

Reading Environment variables

```
>>> os.environ['HOME']
'/home/matt'

>>> os.environ.get('PET', 'cat')
'cat'
```

Call `os.putenv` for temporary manipulation while launching child processes. They don't affect `os.environ`

```
>>> os.putenv('WIERDO', 'value')
>>> os.environ['WIERDO']
Traceback (most recent call last):
...
KeyError: 'WIERDO'

>>> os.environ['OTHER'] = 'value2'
>>> import subprocess
>>> p = subprocess.Popen('echo $WIERDO $OTHER',
shell=True, stdout=subprocess.PIPE)
>>> p.stdout.read()
```

```
'value value2\n'
```

optparse Command line configuration

```
>>> import optparse
>>> __version__ = '0.1'
>>> usage = "A script that illustrates scripting"
```

Passing in *version* gives you free `--version` (see .print_help below) Passing in *usage* allows user specified help.

```
>>> parser = optparse.OptionParser(usage=usage,
version=__version__)
```

"store" is default action for options. The default for *dest* is the long option (with underscores).

```
>>> parser.add_option('-f', '--other-file',
#action='store' # doctest: +ELLIPSIS
...                    help='specify file to process')
<Option at ...: -f/--other-file>

>>> opt, args = parser.parse_args(['script.py', '--
other-file', 'some_file'])
>>> opt.other_file # long option name converted if
'dest' not specified
'some_file'
```

Can add *groups* to organize options

```
>>> group = optparse.OptionGroup(parser, "Group
Name", "Some details about the group")
>>> group.add_option('--be-noisy',
action='store_true', help='make noise') #doctest:
+ELLIPSIS
<Option at ...: --be-noisy>
>>> parser.add_option_group(group) #doctest:
+ELLIPSIS
<optparse.OptionGroup instance at ...>

>>> parser.print_help()
Usage: A script that illustrates scripting
<BLANKLINE>
Options:
  --version            show program's version number
and exit
  -h, --help           show this help message and
exit
  -f OTHER_FILE, --other-file=OTHER_FILE
                       specify file to process
<BLANKLINE>
  Group Name:
    Some details about the group
<BLANKLINE>
    --be-noisy         make noise
```

Chaining Configuration

Ugly code to cascade configuration

```
>>> class Unset(object): pass
>>> def cascade_value(opt=None, opt_name=None,
env_name=None, cfg=None, cfg_section=None,
cfg_name=None, default=None):
...    """
...    opt - result of OptionParser.parse_args
...    opt_name - string of opt name you want to
access
...    """
...    # get from cmd line
...    value = Unset()
...    if opt and opt_name:
...        try:
...            value = opt.__getattr__(opt_name)
...        except AttributeError, e:
...            pass
...    if not isinstance(value, Unset):
...        return value
...    # get from ENV
...    if env_name:
...        try:
...            value = os.environ[env_name]
...        except KeyError, e:
...            pass
...    if not isinstance(value, Unset):
...        return value
...    # get from config file
...    if cfg and cfg_section and cfg_name:
...        try:
...            value = cfg.get(cfg_section, cfg_name)
...        except ConfigParser.NoOptionError, e:
...            pass
...    if not isinstance(value, Unset):
...        return value
...    return default

>>> cascade_value(opt=opt, opt_name='author',
cfg=cfg, cfg_section='Properties', cfg_name='author')
'Matt'
```

Composite Scripts

Tools like `svn` have options without -- or -, such as `svn status`. One such way to do this in python is with "composite" scripts. Implement a "status" script in a module by itself, then dispatch to it (and other commands) from the "svn" script based on `sys.argv[1]`.

`sys.argv` is a list starting with the filename and then any options passed to it

```
>>> def main(pargs): # pargs = ['script.py',
'status', '--some-option']
...   if pargs[1] == 'status':
...     status.main(pargs[2:])
```

3 layers of I/O

I favor a 3 layer abstraction, which eases testing, provids useful interfaces and can make python speedy.

- Filename interface (usually through `main` function)
- File object interface
- Generator interface

File interface

Below `main` function, try to pass around filelike instances instead of filenames, ie `sys.stdin`, `StringIO.StringIO()`, `open()`.

Input Filename/File inteface

```
>>> def process_data(file_instance):
...   ''' file_instance could be sys.stdin, StringIO
or file '''
...   pass # call file.write/read
```

Assume the following is in the body of `main`. Often scripts read from either a `stdin` or a file specified on the command line.

```
>>> pargs = ['script.py', '--input', 'handout.rst']
>>> parser.add_option('--input', help='Specify input
file (default stdin)') # doctest: +ELLIPSIS
<Option at ...: --input>
>>> opt, args = parser.parse_args(pargs)
>>> fin = sys.stdin
>>> if opt.input:
...     fin = open(opt.input)
>>> process_data(fin)
```

Use Generators

Don't accumulate data if you don't need to, `yield`, reduce or drop it.

```
>>> process_line = process_data
```

Don't do:

```
>>> input = fin.readlines()
>>> for line in input:
...   process_line(line)
```

Do:

```
>>> for line in fin:
...   process_line(line)
```

see http://www.dabeaz.com/generators/

Reading a password

Use `getpass` function from the `getpass` module to read data without echoing to terminal

Output

```
>>> parser.add_option('--output', help='Specify
output file (default stdout)') #doctest: +ELLIPSIS
<Option at ...: --output>
>>> opt, args = parser.parse_args(pargs)
>>> fout = sys.stdout
>>> if opt.output:
...   fout = open(opt.output, 'w')

>>> process_data(fout)
```

Temporary files

```
>>> import tempfile
>>> fd, filename = tempfile.mkstemp() # fd is a file
descriptor
>>> fout = open(filename, 'w')
>>> # Use fout
>>> fout.close()
```

Remember to clean up when done

```
>>> os.remove(filename)
```

Executing other scripts

Reading output

```
>>> import subprocess
>>> p = subprocess.Popen('id -u', shell=True,
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
>>> p.stdout.read()
'1000\n'
>>> p.returncode # None means not done
>>> print p.wait()
0
```

Feeding `stdin`

Can use `communicate` or `p2.stdin.write` w/ `flush/close`.

```
>>> p2 = subprocess.Popen('wc -l', shell=True,
stdout=subprocess.PIPE, stdin=subprocess.PIPE,
stderr=subprocess.PIPE)
>>> out, err = p2.communicate('foo\nbar\n')
#p.stdin.flush()
```

```
>>> out
'2\n'
>>> p2.returncode
0
```

Chaining scripts

Chaining is pretty straightforward make sure to `close stdin`.

```
>>> p3 = subprocess.Popen('sort', shell=True,
...                          stdout=subprocess.PIPE,
...                          stdin=subprocess.PIPE)
>>> p4 = subprocess.Popen('uniq', shell=True,
...                          stdout=subprocess.PIPE,
...                          stdin=p3.stdout,
...                          close_fds=True) # hangs w/o
close_fds

>>> p3.stdin.write('1\n2\n1\n') # hangs
>>> p3.stdin.flush()
>>> p3.stdin.close()
>>> p4.stdout.read()
'1\n2\n'
```

Chaining scripts and python

`cat` 0-2, add 10 to then (in python) and `wc -l` results.

```
>>> p5 = subprocess.Popen('cat', shell=True,
stdout=subprocess.PIPE, stdin=subprocess.PIPE,
close_fds=True)
>>> def p6(input):
...   ''' add 10 to line in input '''
...   for line in input:
...     yield '%d%s' %(int(line.strip())+10,
os.linesep)
>>> p7 = subprocess.Popen('wc -l', shell=True,
stdout=subprocess.PIPE, stdin=subprocess.PIPE,
close_fds=True)
>>> ignore = [p5.stdin.write(str(x)+os.linesep) for x
in xrange(3)]
>>> p5.stdin.close()
>>> ignore = [p7.stdin.write(x) for x in
p6(p5.stdout.xreadlines())]
>>> p7.stdin.close()
>>> p7.stdout.read()
'3\n'
```

PID file

As a mechanism for preventing concurrent runs of script. Be careful with file permissions (user write access).

pidfile example

```
>>> def pid_running(pid):
...   p = subprocess.Popen('ps auxww |grep %s | grep
-v grep' %pid,
...                          shell=True,
stdout=subprocess.PIPE)
...   return str(pid) in p.stdout.read()

>>> import os
>>> PID_FILE = os.path.expanduser('~/.script.pid')
>>> if os.path.exists(PID_FILE):
...   pid = open(PID_FILE).read()
...   if pid_running(pid):
...     raise AlreadyRunningError
>>> p_file = open(PID_FILE, 'w')
>>> p_file.write(str(os.getpid()))
```

Do stuff, remember to clean up when done

```
>>> import os
>>> os.remove(PID_FILE)
```

`atexit`

The `atexit` module provides `register(func, [,args [,kwargs]])` to perform `func` when the interpreter exits

Theft Packaging

setup.py example

This can be tedius, *copying* is recommended:

```
from distutils.core import setup
#from setuptools import setup # for setup.py develop
import scriptlib

setup(name="poachplate",
    version=scriptlib.__version__,
    author=scriptlib.__author__,
    description="FILL IN",
    scripts=["bin/script"],
    package_dir={"scriptlib":"scriptlib"},
    packages=['scriptlib'],
)
```

Uncomment the `setuptools` line if you want to do `python setup.py develop` (which allows you to develop in place, which having the script installed)

`distutils` commands

`python setup.py sdist`

Create a source distribution in the `dist` directory

`setuptools` commands

`python setup.py develop`

Install the scripts/libraries using the developed versions. Further changes to source code changes installed versions.

Logging

Levels

`CRITICAL`, `ERROR`, `STATUS`, `INFO`, `DEBUG`

Basic Example

Setup a logger that rotates log at 500K and creates up to 2 older files when the first is full:

```
~/.script.log
~/.script.log.1
~/.script.log.2

>>> import logging
>>> from logging import handlers, Formatter
>>> LOGFILE = os.path.expanduser('~/.script.log')
>>> logger = logging.getLogger('ScriptLogger')
>>> logger.setLevel(logging.DEBUG)
>>> handler = handlers.RotatingFileHandler(LOGFILE,
maxBytes=500, backupCount=2)
>>> log_format = Formatter("%(asctime)s - %(name)s -
%(levelname)s - %(message)s")
>>> handler.setFormatter(log_format)
>>> logger.addHandler(handler)
```

Test log

```
>>> logger.debug('Test the logger')
>>> open(LOGFILE).read() # doctest: +SKIP
'2009-02-10 00:53:15,509 - ScriptLogger - DEBUG -
Test the logger\n'

>>> os.remove(LOGFILE)
```

Testing

`doctest`

Doctests can be placed in python docstrings at the module, class or function/method level. Also text files can contain doctests by having '>>>' in them.

This file happens to have many doctests, to execute `doctest` on a module do the following:

```
>>> import doctest
>>> doctest.testmod()
(0, 0)
```

To test a file use `doctest.testfile(filename)`

`unittest`

Execute unittests at the level of abstraction you want, filename (via `main`), file object or generator.

Coverage tools can be useful to see where tests are missing (see `figleaf` or `coverage.py`)

Useful methods are `setup`, `teardown`, `assert_(expr[, msg])`, `assertEqual(first, second[, msg])`, `assertNotEqual`, and `assertRaises(exception, callable)`.

```
>>> import unittest
>>> class TestScript(unittest.TestCase):
...   def test_num_lines(self):
...     self.assertEqual(list(num_lines(range(1))),
['0\n'])

>>> if __name__ == '__main__':
...   unittest.main()
```

No print

If you are using 3 layers of I/O and logging correctly, there will be no `print` statements in your code.

A cheat

The project *poachplate* is a simple tool to generate directory structure, `setup.py` and boiler plate content for scripts. Find it on pypi.