

Tool oriented python

Making you own and using others

Morgan Goose

February 2010

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

This presentation

Want the source? it's all wrapped up for you here:

http://morgangoose.com/p/tool_oriented_python.tar.gz

So you've got a python script

What you'll most likely hear next:

- Can you do much with it?
- What options does it have?
- Where are its logs?
- How do I install it?
- Does it work?
- Any documentation?
- Got a website for all of this?

What we'll use to answer these questions

Most of these modules/tools are going to make you life simpler.

Promise.

- [multiprocessing](#) - a forked answer for threading
- [optparse](#) - cli option and argument parser
- [logging](#) - logging toolset
- [distutils](#) - install scripts
- [nose](#) - unittesting framework
- [sphinx](#) - documentation generation
- [fabric](#) - for project deployment

Python, yeah the GIL is a buzz kill

The global interpreter lock, is what keeps (c)python down. It make us unable to do real parallel threading in python. The [multiprocessing](#) module makes it possible. kinda.

It give a simple and convinent threading library, that instead of threads forks off the process. So we miss out on the lightweighted and simple intercommunication of real threads. We do however get to make parallel code again, and the os sorts out the hard stuff for us.

Let's side step some things

```
from multiprocessing import Process
import os

def info(title):
    print title
    print 'module name:', __name__
    print 'parent process:', os.getppid()
    print 'process id:', os.getpid()

def f(name):
    info('function f')
    print 'hello', name

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

That'll give us this

```
$ python multi.py
main line
module name: __main__
parent process: 11430
process id: 15419
function f
module name: __main__
parent process: 15419
process id: 15420
hello bob
```

Not super exciting perhaps

But you can loop over that function like this, and get a whole lot out of it:

```
for host in hosts:
    p = multiprocessing.Process(
        target=run_commands,
        args=(host, get_commands(), module_name, verbose),
    )

    jobs.append(p)
    p.start()
```

This really needs its own talk

Not only does `multiprocessing` give simple forked process control, but it also provides; Queue and Pipe objects for communication between procs and Process pools, like apache instances waiting for data.

```

from multiprocessing import Pool

def f(x): return x*x

if __name__ == '__main__':
    # start 4 worker processes
    pool = Pool(processes=4)

    # evaluate "f(10)" asynchronously
    result = pool.apply_async(f, [10])

    # prints "100" unless your computer is *very* slow
    print result.get(timeout=1)

    # prints "[0, 1, 4, ..., 81]"
    print pool.map(f, range(10))

```

Lets move onto the command line

So you've got you script or program. You call it and it asks you questions. You hate this.

We will fix this with [optparse](#).

It's a "convenient, flexible, and powerful library for parsing command-line options"

You "create an instance of OptionParser, populate it with options, and parse the command line."

And [optparse](#) "additionally generates usage and help messages for you."

Neat! let's see it

```

from optparse import OptionParser

parser = OptionParser()

parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")

parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()

```

This makes the parser. Populates it with two options. Then processes the command lines args and outputs a modified dict of options and a list of args.

What else?

In `add_option()` you can do some neat things:

- set what type it'll be: *string* (default), *int*, *float*, and more
- have the action of a flag to *store_true/false*, *count*, *append*
- there is also a *callback* action that links a flag to a function
- define how many args a flag takes, default being 1

Optparse will also you to:

- Have groups of options
- use `parser.error(..)` to throw exceptions one can follow to parsing
- `SURPRESS_HELP` on a command flag if you want it to be hidden
- extend it to take custom actions and types

So this helps us how?

You now put the control in the users' hands while giving yourself an out by being able to set sensible defaults.

This should make, most, everyone happy.

When you change options, your help output changes, and is there by default.

Once you use it, you'll use it again, and your scripts will develop a common look and feel to them. This being a good thing if you want people to adopt them.

Now we need to get some logs going

```
import logging
import sys

LEVELS = {'debug': logging.DEBUG,
          'info': logging.INFO,
          'warning': logging.WARNING,
          'error': logging.ERROR,
          'critical': logging.CRITICAL}

if len(sys.argv) > 1:
    level_name = sys.argv[1]
    level = LEVELS.get(level_name, logging.NOTSET)
    logging.basicConfig(level=level)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical error message')
```

Some fun bits of logging

```
import logging

logging.basicConfig(level=logging.DEBUG,
                  format='%(asctime)s %(levelname)-8s %(message)s',
                  datefmt='%a, %d %b %Y %H:%M:%S',
                  filename='/temp/myapp.log',
                  filemode='w')

logging.error('Pack my box with %d dozen %s', 5, 'liquor jugs')
```

Looks like:

Tie that into the optparsing

Remember those nice flags we got with optparse. We can tie those into log levels now. Super useful tidbit.

```
import optparse, logging

def callback(*args, **kwargs):
    logging.root.setLevel(logging.root.level - 10)

parser = optparse.OptionParser()
parser.add_option('-v', '--verbose', dest="verbose",
                  action='callback', callback=callback,
                  help='Increase verbosity')

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

Now as you add v's to the command you're log level verbosity will increase.

Other ways of logging

Logging can also have different handlers. Here's a list of standard options

- *StreamHandler* instances send error messages to streams.
- *FileHandler* instances send error messages to disk files.
- *RotatingFileHandler* instances send error messages to disk files, with support for maximum log file sizes and log file rotation.
- *SocketHandler* instances send error messages to TCP/IP sockets.
- *SMTPHandler* instances send error messages to a designated email address.
- *SysLogHandler* instances send error messages to a Unix syslog daemon, possibly on a remote machine.
- *HTTPHandler* instances send error messages to an HTTP server using either GET or POST semantics.

And you can add multiple handlers to a logging instance, so you can log to file and email if you feel so inclined.

You want people to use this?

Help them out, make an installer. It isn't to hard with distutils.

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Some observations:

- most information that you supply to the Distutils is supplied as keyword arguments to the *setup* function

- those keyword arguments fall into two categories: package metadata and information about what's in the package.
- modules are specified by module name, not filename

How this helps

Now we can run simple commands to setup and install our scripts as we've configured. This will make a tarball of our simple project.

```
$ python setup.py sdist
```

If an end-user wishes to install your foo module, all she has to do is download foo-1.0.tar.gz (or .zip), unpack it, and—from the foo-1.0 directory—run

```
$ python setup.py install
```

If you want to make things really easy for your users, you can create one or more built distributions for them, rpms for eg.

```
$ python setup.py bdist_rpm
```

to find out some other bdist options use: *python setup.py bdist --help-formats*

Read reddit? you know about unit tests

If you write unit tests nose will find them for you, and run them.

"Any function or class that matches the configured testMatch regular expression ((?:^|[\b_\-.])[Tt]est) by default – that is, has test or Test at a word boundary or following a - or _) and lives in a module that also matches that expression will be run as a test."

```
def test_evens():
    for i in range(0, 5):
        yield check_even, i, i*3

def check_even(n, mn):
    assert n % 2 == 0 or mn % 2 == 0
```

Helper functions nose gives you

```
@raises(TypeError, ValueError)
def test_raises_type_error():
    raise TypeError("This test passes")

@raises(Exception)
def test_that_fails_by_passing():
    pass

@timed(.1)
def test_that_fails():
    time.sleep(.2)

@with_setup(setup, teardown)
def test_something():
    ...
```

If you follow pep you've already written some

Docstrings are where some people, hopefully you, put info about a module, function, or class. They are simple to add, and are stripped out for documentation purposes by a number of python doc utils.

```
def f(foo):
    """
    function f() takes in foo and so that it can double its
    contents, before returning.
    """

    return int(foo)*2
```

Sphinx and pydocs are two such tools that can strip these out. If you are a fan of ipython, or the pytohn >>> command line. When you type help(f), the info you get back is from these doc strings.

Sphinx stuff

The root directory of a documentation collection is called the source directory. Normally, this directory also contains the Sphinx configuration file conf.py, but that file can also live in another directory, the configuration directory.

New in version 0.3: Support for a different configuration directory.

Sphinx comes with a script called sphinx-quickstart that sets up a source directory and creates a default conf.py from a few questions it asks you. Just run

```
$ sphinx-quickstart
```

and answer the questions.

Going all out for doc

So once you get the sphinx build started, you populate it with rst (ReSTructured text) files that hold your documentation.

To use the docstrings you so arduously made you'll use their automodule command.

```
Some doc heading
=====

:command:`check_foo` -- checks foo
-----
.. automodule:: bar.check_foo

.. program:: checks
```

Lets stitch this all together

A good tool for putting all our data out there with simple recipes is [fabric](#).

```
from fabric.api import *

env.user = 'username'
env.hosts = ['host1.com', 'host2.com']

def pack():
    local('tar czf /tmp/project_foo.tgz project_foo/', capture=False)

def deploy():
    pack()
    put('/tmp/project_foo.tgz', '/tmp/')

    with cd('/var/www/foo/'):
        run('tar xzf /tmp/project_foo.tgz')
```

Thats all I have

Any questions?