# Writing elegant command line scripts in Python

Vineet Naik

Mumpy meet, July 21, 2013

## About me

Developer at Kodeplay

We use Python/Django and a bunch of other technologies to build and run KodeCRM - A Customer Service solution for online businesses.

Writing command line scripts was my “gateway drug” to Python!

*@naiquevin*

## About you

- Have basic knowledge of Python
- Have some command line experience (preferably Linux)

## Overview of the talk

1. Why use Python for command line interfaces (CLIs)
2. Building elegant and extensible commands
3. Command parsers in Python
4. Best practices; DOs and DONTs
5. Some other handy libs/utilities
6. Providing scripts from python packages

Section 1

## Why use Python for CLI?

## Why write command line scripts?

- To automate tasks that are tedious or/and need to be repeated.
- To automate tasks that are impractical to do manually
- Can be run on remote boxes with no Desktop environments.
- After a point, GUIs get frustrating to work with particularly if you are a developer.

## Why use python for CLI?

- Easier to read, write and maintain
- Provides access to a lot of useful libraries (eg. requests, BeautifulSoup, command parsers)
- Easier to write tests and document
- Works across platforms (mostly)
- Makes Python programmers feel at home

Section 2

## Building elegant and extensible commands

## Elegant CLI

- Intuitive and consistent to use
- End users' familiarity with the language (here Python) should be a non-requirement
- Well documented for the both end users and developers
- Work well with other commands and tools
- Safe

## The Unix philosophy

- Write simple parts connected by clean interfaces.
- Complex front ends should be cleanly separated from complex back ends.
- Always do the least surprising thing
- When you must fail, fail noisily and as soon as possible
- Value developer time over machine time
- Design for future because it will be here sooner than you think

Read “The Art of Unix Programming” by Eric Raymond. Too much wisdom to fit in here

## Anatomy of a command

- $ ls
- $ ls -a
- $ ls ./Downloads
- $ ls ./Downloads -lah
- $ git commit -m "Fix README"
- $ git log --author=vineet
- $ cat /etc/passwd | cut -d ":" -f 1 > usernames.txt

Command, Options, Positional Arguments, Sub-command, Not a part of command

## Anatomy of a command

- $ ls
- $ ls -a
- $ ls ./Downloads
- $ ls ./Downloads -lah
- $ git commit -m "Fix README"
- $ git log --author=vineet
- $ cat /etc/passwd | cut -d ":" -f 1 > usernames.txt

Command, Options, Positional Arguments, Sub-command, Not a part of command

## Anatomy of a command

- $ ls
- $ ls -a
- $ ls ./Downloads
- $ ls ./Downloads -lah
- $ git commit -m "Fix README"
- $ git log --author=vineet
- $ cat /etc/passwd | cut -d ":" -f 1 > usernames.txt

Command, Options, Positional Arguments, Sub-command, Not a part of command

## Anatomy of a command

- $ ls
- $ ls -a
- $ ls ./Downloads
- $ ls ./Downloads -lah
- $ git commit -m "Fix README"
- $ git log --author=vineet
- $ cat /etc/passwd | cut -d ":" -f 1 > usernames.txt

Command, Options, Positional Arguments, Sub-command, Not a part of command

## Anatomy of a command

- $ ls
- $ ls -a
- $ ls ./Downloads
- $ ls ./Downloads -lah
- $ git commit -m "Fix README"
- $ git log --author=vineet
- $ cat /etc/passwd | cut -d ":" -f 1 > usernames.txt

Command, Options, Positional Arguments, Sub-command, Not a part of command

## Anatomy of a command

- $ ls
- $ ls -a
- $ ls ./Downloads
- $ ls ./Downloads -lah
- $ git commit -m "Fix README"
- $ git log --author=vineet
- $ cat /etc/passwd | cut -d ":" -f 1 > usernames.txt

Command, Options, Positional Arguments, Sub-command, Not a part of command

## Anatomy of a command

- $ ls
- $ ls -a
- $ ls ./Downloads
- $ ls ./Downloads -lah
- $ git commit -m "Fix README"
- $ git log --author=vineet
- $ cat /etc/passwd | cut -d ":" -f 1 > usernames.txt

Command, Options, Positional Arguments, Sub-command, Not a part of command

Section 3

## Command Parsers in Python

## Command Parsers in Python

- sys.argv*
- optparse
- argparse
- docopt

But there are a few others which I haven't tried (eg. getopt, clint)

* sys.argv is not a parser but the basic mechanism in Python to collect command line args

## sys.argv

- Most basic and easy to get started with
- Only collects tokens
- We need to handle different combinations of args and options
- Leads to ugly code (nested try..except and if..else blocks)

## Example

```python
import sys

script = sys.argv[0]
args = sys.argv[1:]
print(script)
print(args)
exit(0)
```

```
$ python manage.py startapp poll
manage.py
['startapp', 'poll']
```

## optparse

- Stdlib module for parsing options
- No support for advanced functionality eg. subcommands, grouped commands etc.
- Generates help message/summary

Warning! Deprecated since version 2.7

## Examples

```
from optparse import OptionParser

p = OptionParser()
p.add_option('-p', '--port', dest='port', default=9000,
             help='Port to use for localhost (0.0.0.0)')

(options, args) = p.parse_args()

print(options.port) # access as attributes
```

Warning! Deprecated since version 2.7

## argparse

- Stdlib module. Replaces *optparse* in newer versions of Python
- Generates help message/summary
- Very powerful. Supports advanced configurations
- Verbose code and complex API

*"The D3.js of command parsers!"*

Warning! New in version 2.7

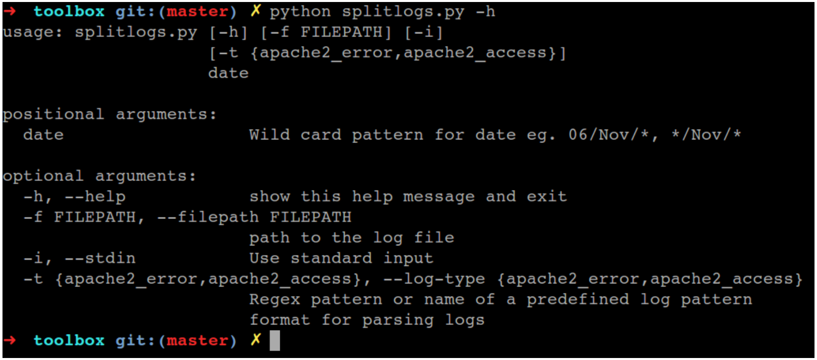
## Examples

```python
import argparse

p = argparse.ArgumentParser()
p.add_argument('date',
               help='Wild card pattern for date eg. 06/Nov/*, */Nov/*')
p.add_argument('-f', '--filepath', help='path to the log file')
p.add_argument('-i', '--stdin',
               help='Use standard input', action='store_true')
p.add_argument('-t', '--log-type',
               help=(
                   'Regex pattern or name of a '
                   'predefined log pattern format for parsing logs'
               ), default='apache2_access',
               choices=LOG_PATTERN_FORMATS.keys())

args = p.parse_args()
print(args.date) # access as attributes
```

## argpase help message



```
➜  toolbox git:(master) ✗ python splitlogs.py -h
usage: splitlogs.py [-h] [-f FILEPATH] [-i]
                    [-t {apache2_error,apache2_access}]
                    date

positional arguments:
  date                  Wild card pattern for date eg. 06/Nov/*, */Nov/*

optional arguments:
  -h, --help            show this help message and exit
  -f FILEPATH, --filepath FILEPATH
                        path to the log file
  -i, --stdin           Use standard input
  -t {apache2_error,apache2_access}, --log-type {apache2_error,apache2_access}
                        Regex pattern or name of a predefined log pattern
                        format for parsing logs
➜  toolbox git:(master) ✗ 
```

## docopt

- Not in Stdlib
- Uses a well formed help message (from docstring) to parse the command
- Lightweight and minimal
- Generates a dictionary of args and options
- Doesn't handle types. All collected args/opts are strings
- Sometimes fails with hard to debug error messages

## Example

```
"""A simple CSV to JSON converter

Usage: csv2json.py ( -i | FILE ) [ -q QUOTECHAR -d DELIMITER ]
       csv2json.py -h | --help | --version

Options:
    -i              Read from stdin
    -d DELIMITER    Specify csv delimiter [default: ,]
    -q QUOTECHAR    Specify csv quotechar [default: |]
    -h --help       Show help
    --version       Show version

"""

from docopt import docopt

args = docopt(__doc__, version='1.0')
```

## Which one to use?

- sys.argv if it's too simple (no options etc.)
- Choose between argparse and docopt as per complexity of the command and style preference
- Donot use optparse as far as possible since it's deprecated
- What I use:

  sys.argv – docopt – argparse

Section 4

## Best Practices; DOs and DONTs

## Separation of concerns and Reusability

- Keep command parsing logic separate from the implementation of the command
- Define helper functions
- Pass in arguments to functions instead of having global variables
- Have the functions "return" things rather than "doing" things
- Treat scripts as modules with import-able code

## Example script template

```
"""A script to ...

Usage: ...

"""
## imports

## constants

## functions

## tests

if __name__ == '__main__':
    ## command parsing logic and calls to functions
    pass
```

## Document code and write tests

- Documentation helps when you have to fix something or extend the script three weeks after writing it
- Same with tests. Simple assert statements in the same file are sufficient.
- *nose* makes it convenient to run tests

% myscript.py

```
def test_something():
    assert 2 + 2 == 4
```

% Running all the test* functions in myscript.py from terminal

```
$ nosetests -v myscript.py
```

## Write composable scripts

```
$ cat /etc/passwd | cut -d : -f 1 > users.txt
```

```
$ cat ./access.log.gz \
   | gunzip \
   | python splitlogs.py "18/Jul/*" -i \
   | python log2json.py -i \
   | python logan.py -i -p ./.config/dynurls.json \
   > ./18-07-analysis.txt
```

Such composable scripts play well with other commands so that complex commands can be composed using smaller ones that do one thing well.

## Reading from either file or stdin

```
import os
import sys
from contextlib import contextmanager

@contextmanager
def read_input(filepath, stdin):
    if filepath is not None:
        f = open(os.path.abspath(filepath))
        yield f
        f.close()
    elif stdin:
        yield sys.stdin
    else:
        raise Exception('Either filepath or stdin required')

## calling code
with read_input(args.filepath, args.stdin) as f:
    do_something(f)
```

## Keep debug messages separate from stdout

Writing debug messages to *stderr* is a better alternative as even if stdout is redirected, debug messages will still be printed on the screen.

```
print 'I am here' # bad, will pollute stdout

print >> sys.stderr, 'I am here'     # python 2.x
print('I am here', file=sys.stderr) # python 3.x
sys.stderr.write('I am here')
```

## Return correct exit codes

This means your program communicates well with other programs

```
try:
    do_something()
    exit(0)  # 0 means successful exit
except Exception:
    exit(1)  # non-zero means abnormal exit
```

eg. Fabric stops if any of the command that it runs returns 1 exit code such as when tests fail

## Avoid writing redundant code

eg. Having your script save output to a file is redundant,

```
if args.outfile is not None:
    with open(args.outfile, 'w') as f:
        json.dump(data, f)
else:
    sys.stdout.write(json.dumps(data))
```

```
$ python myscript.py --outfile=output.json
```

Redirect output to file instead,

```
sys.stdout.write(json.dumps(data))
```

```
$ python myscript.py > output.json
```

Often, this also results in lesser options

## Ensure safety

Take care to avoid doing stupid things on behalf of the user

- Warn users and ask for confirmation. “Danger zone. Proceed? [Y/N]“
- Beware of “shell injection” when invoking system calls using user input

```
from subprocess import call
call('ls -l' + ' ' + args.dirpath, shell=True) # unsafe
```

```
$ python myscript --dirpath="nothing; rm -rf /" # oops!
```

```
call(['ls', '-l'] + [args.dirpath]) # much safer
```

## No sensitive data in code

Having sensitive data such as a password hard-coded in code is not just unsafe but it isn't a constant in the first place.

```
HOST = '123.456.789.01'
PASSWORD = 'is-a-top-secret'      # O RLY!!
```

Use the *getpass* module

```
from getpass import getpass

password = getpass()
# getpass prompts user for password while printing nothing in the
# terminal

print('Your password is safe with us')
```

## Filepaths are more than just strings

```
LOG_DIR = '/var/log'
# ...
# string concatenation is bad and unreliable
logfile_path = LOG_DIR + '/' + 'error.log'

# good
import os
logfile_path = os.path.join(LOG_DIR, 'error.log')
```

Section 5

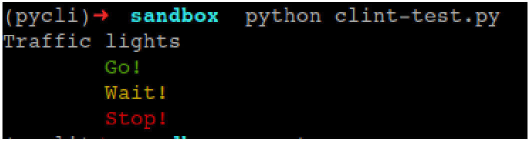
## Other useful utils

## Beautiful printing in terminal

Clint provides colored output and indentation.

```
from clint.textui import colored, indent, puts

print('Traffic lights')
with indent(8):
    puts(colored.green('Go!'))
    puts(colored.yellow('Wait!'))
    puts(colored.red('Stop!'))
```



```
(pycli)➜  sandbox  python clint-test.py
Traffic lights
        Go!
        Wait!
        Stop!
```

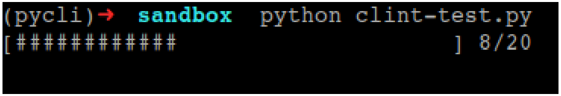
Other alternatives: curses, blessings, colorama

## Progress bar

Clint also provides progress bars

```
from clint.textui import progress
import time

data = range(20)
progb = progress.bar(data)
for d in data:
    time.sleep(0.1)
    progb.next()
```



```
(pycli)➜  sandbox  python clint-test.py
[############                    ] 8/20
```

Section 6

## Providing commands from packages

## Providing scripts from packages

What does that mean?

```
$ pip install Django

$ django-admin.py --version
```

*django-admin.py* is a command which is made available to us after we install Django

## Allowing a module to be run as a script

```
$ python -m json.tool
$ python -m SimpleHTTPServer 9000
```

```
def main(args):
    # do something here

if __name__ == '__main__':
    # get args using some method
    main(args)
```

## Using distutils

% Django/setup.py

```
setup(
    name = "Django",
    # ...
    scripts = ['django/bin/django-admin.py'],
    # ...
)
```

```
$ django-admin.py startproject
```

## Using Setuptools/Distribute

% myutil/setup.py

```
setup(
    name='MyUtil',
    # ...
    entry_points={
        'console_scripts': [
            'myutil = myutil.commands:main'
            ]
        }
    # ...
)
```

A file “myutil“ will be created in the *bin* directory of the environment with 755 permissions

## Which one to use?

There are various ways to do this because there are various ways to package a library in Python ie. using distutils (stdlib), setuptools/distribute

Comparing these is a topic of another talk!

## Summary

- Treat command line scripts as any other application or program
- Document code, write tests
- Embrace the Unix Philosophy
- Give importance to safety
- Stick to best practices as far as possible
- But sometimes there may be a good reason not to..

*"Every rule can be broken but none may be ignored"**

* Central rule of typography

## Thank You!

Questions?

## References

- The Art of Unix Programming - http://catb.org/esr/writings/taoup/
- optparse - http://docs.python.org/2/library/optparse.html
- argparse - http://docs.python.org/dev/library/argparse.html
- docopt - https://github.com/docopt/docopt
- getpass - http://docs.python.org/2/library/getpass.html
- clint - https://github.com/kennethreitz/clint
- Some examples are taken from these scripts - https://github.com/naiquevin/toolbox