Command Line Option Parsing

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

If I look back, an overwhelming number of the Python programs I have written have been simple scripts and tools meant to be executed from the command line. Sure, I've created the occasional Web application, but the command line has always been where the real action is. However, I also have a confession—despite my reliance on the command line, I just can't bring myself to use any of Python's built-in libraries for processing command line options. Should I use the `getopt` module? Nope. Not for me. What about `optparse` or `argparse`? Bah! Get out! No, most of my programs look something like this:

```
#!/usr/bin/env python
# program.py
...
... Something or another
...
if __name__ == '__main__':
   import sys
   if len(sys.argv) != 3:
      raise SystemExit('Usage: %s infile outfile' % sys.argv[0])
   infile = sys.argv[1]
   outfile = sys.argv[2]
   main(infile, outfile)
```

Sure, the exact details of the options themselves might change from program to program, but, generally speaking, the programs all look about like that. Should things start to get more complicated, I'll ponder the situation a bit before usually concluding that I should probably just keep it simple. Again, I'm not proud of this, but it's a fairly accurate description of my day-to-day coding. In this article, I'm going to visit the topic of command line option parsing. First, I'll quickly review Python's built-in modules and then look at some newer third-party libraries that aim to simplify the problem in a more sane manner.

Command Line Parsing in the Standard Library

As background, let's consider the options for command line option parsing in the standard library. First, suppose your program had a main function like this:

```
def main(infiles, outfile=None, debug=False):
   # Imagine real code here. We'll just print the args for an example
   print(infiles)
   print(outfile)
   print(debug)
```

In its most basic use, suppose you wanted the program to simply take a list of input files to be provided as the `infiles` argument. For example:

```
% python prog.py infile1 ... infileN
```

In addition, suppose you wanted the command line interface to have an optional outfile argument provided by an -o or --outfile= option like this:

```
% python prog.py -o outfile infile1 ... infileN
% python prog.py --outfile=outfile infile1 ... infileN
%
```

Finally, suppose the debug argument is provided by an optional -d or --debug option. For example:

```
% python prog.py --debug -o outfile infile1 ... infileN
%
```

At the lowest level, the getopt module provides C-style command line parsing. Here is an example of how it is used:

```
# prog.py
...

if __name__ == '__main__':
   import getopt
   usage = '''\
Usage: prog.py [options]

Options:
  -h, --help       show this help message and exit
  -o OUTFILE
  --output=OUTFILE
  -d
  --debug
'''
   try:
      optlist, args = getopt.getopt(sys.argv[1:], 'dho:',
['output=', 'debug', 'help'])
   except getopt.GetoptError as err:
      print(err, file=sys.stderr)
      print(usage)
      raise SystemExit(1)

   debug = False
   outfile = None
   for opt, value in optlist:
      if opt in ['-d', '--debug']:
         debug = True
      elif opt in ['-o', '--output']:
         outfile = value
      elif opt in ['-h', '--help']:
         print(usage)
         raise SystemExit(0)

   main(args, outfile, debug)
```

If that's a bit too low-level for your tastes, you can move up to the optparse module instead. For example:

```
# prog.py
...

if __name__ == '__main__':
   import optparse
   parser = optparse.OptionParser()

   parser.add_option('-o', action='store', dest='outfile')
   parser.add_option('--output', action='store', dest='outfile')
   parser.add_option('-d', action='store_true', dest='debug')
   parser.add_option('--debug', action='store_true',
dest='debug')
   parser.set_defaults(debug=False)

   opts, args = parser.parse_args()
   main(args, opts.outfile, opts.debug)
```

Or, if you prefer, you can use the more recent argparse module. For example:

```
# prog.py
...

if __name__ == '__main__':
   import argparse
   parser = argparse.ArgumentParser()

   parser.add_argument('infiles', metavar='INFILE', nargs='*')
   parser.add_argument('-o', action='store', dest='outfile')
   parser.add_argument('--output', action='store',
dest='outfile')
   parser.add_argument('-d', action='store_true', dest='debug',
default=False)
   parser.add_argument('--debug', action='store_true',
dest='debug', default=False)

   args = parser.parse_args()
   main(args.infiles, args.outfile, args.debug)
```

Both the optparse and argparse modules hide the details of command line parsing through extra layers of abstraction. They also provide error checking and the ability to create nice help messages. For example:

```
$ python prog.py --help
usage: prog.py [-h] [-o OUTFILE] [--output OUTFILE] [-d]
[--debug]
           [INFILE [INFILE ...]]

positional arguments:
  INFILE
```

```
optional arguments:
  -h, --help       show this help message and exit
  -o OUTFILE
  --output OUTFILE
  -d
  --debug
```

Interlude

Looking at the above examples, it might seem that either `optparse` or `argparse` should work well enough for parsing a command line. This is true. However, they are also modules that are difficult to remember—in fact, I always have to look at the manual (or at my own book). Moreover, if you look at the documentation, you'll find that both modules are actually massive frameworks that aim to solve every possible problem with command line options that might ever arise. It's often overkill for more simple projects—in fact, it usually just makes my overworked pea-brain throb. Thus, it's worth looking at some alternatives that might serve as a kind of middle ground.

docopt

One alternative to the standard libraries is to use `docopt` (http://docopt.org). The idea with `docopt` is that you simply write the help string that describes the usage. An option parser is then automatically generated from it. Here is an example:

```
# prog.py
'''
My program.

Usage:
  prog.py [-o OUTFILE] [-d] [INFILES ... ]
  prog.py [--outfile=OUTFILE] [--debug] [INFILES ...]
  prog.py (-h | --help)

Options:
  -h, --help                Show this screen.
  -o OUTFILE, --outfile=OUTFILE Set output file
  -d, --debug               Enable debugging
'''
...

if __name__ == '__main__':
    import docopt
    args = docopt.docopt(__doc__)
    main(args['INFILES'], args['--outfile'], args['--debug'])
```

In this example, the documentation string for the module describes the usage and command line options. The `docopt.docopt(__doc__)` statement then automatically parses the options directly from that. The result is simply a dictionary where values for the various options are found. It's a neat idea that flips option parsing on its head—instead of specifying the options through a complicated API, you simply write the usage string that you want and it figures it out.

Click

A newer entry to the command line argument game is Click (http://click.pocoo.org/). Click uses decorators to annotate program entry points with a command line interface. Here is an example:

```
import click

@click.command()
@click.argument('infiles', required=False, nargs=-1)
@click.option('-o', '--outfile')
@click.option('-d', '--debug', is_flag=True)
def main(infiles, outfile=None, debug=False):
    print(infiles)
    print(outfile)
    print(debug)

if __name__ == '__main__':
    main()
```

In this example, the `@click.command()` decorator declares a new command. The `@click.argument('infiles', required=False, nargs=-1)` decorator is declaring the infiles argument to be an optional argument that can take any number of values. The `@click.option()` decorators are declaring additional options that are tied to arguments on the decorated function.

Once decorated, the original function operates in a slightly different manner. If you call `main()` without arguments, `sys.argv` is parsed and used to supply the arguments. You can also call `main()` and provide the argument list yourself, which might be useful for testing. For example:

```
main(['--outfile=out.txt', 'foo', 'bar'])
```

One of the more interesting features of Click is that it allows different functions and parts of an application to be composed separately. Here is a more advanced example that defines two separate commands with different options:

```
# prog.py
import click

@click.group()
@click.option('-d', '--debug', is_flag=True)
def cli(debug=False):
    if debug:
        print('Debugging enabled')
```

```
@cli.command()
@click.argument('infiles', required=False, nargs=-1)
@click.option('-o', '--outfile')
def spam(infiles, outfile=None):
    print('spam', infiles, outfile)

@cli.command()
@click.argument('url')
@click.option('-t', '--timeout')
def grok(url, timeout=None):
    print('grok', url, timeout)

if __name__ == '__main__':
    cli()
```

In this example, two commands (`spam` and `grok`) are defined. Here is an interactive example showing their use and output:

```
% python prog.py spam -o out.txt foo bar
spam (u'foo', u'bar') out.txt
% python prog.py grok http://localhost:8080
grok http://localhost:8080 None
%
```

The debugging option (`-d`), being defined on the enclosing group, applies to both commands:

```
% python prog.py -d spam -o out.txt foo bar
Debugging enabled
spam (u'foo', u'bar') out.txt
%
```

Corresponding help screens are tailored to each command.

```
% python prog.py --help
Usage: prog.py [OPTIONS] COMMAND [ARGS]...

Options:
  -d, --debug
  --help      Show this message and exit.

Commands:
  grok
  spam

% python prog.py spam --help
Usage: prog.py spam [OPTIONS] [INFILES]...

Options:
  -o, --outfile TEXT
  --help           Show this message and exit.

%
```

The ability to easily compose commands and options is a powerful feature of Click. In many projects, you can easily have a large number of independent scripts, and it can be difficult to keep track of all of those scripts and their invocation options. As an alternative, Click might allow all of those scripts to be unified under a common command line interface that provides nice help functionality and simplified use for end users.

Final Words

If you write a lot of simple command line tools, looking at third-party alternatives such as `docopt` and Click might be worth your time. This article has only provided the most basic introduction, but both tools have a variety of more advanced functionality. One might ask if there is a clear winner. That, I don't know. However, for my own projects, the ability to compose command line interfaces with Click could be a big win. So I intend to give it a whirl.

Resources

http://docs.python.org/dev/library/getopt.html (getopt module documentation).

http://docs.python.org/dev/library/optparse.html (optparse module documentation).

http://docs.python.org/dev/library/argparse.html (argparse module documentation).

http://docopt.org (docopt homepage).

http://click.pocoo.org (Click homepage).