Introduction

The software developer who needs to have an application parse command line arguments normally reaches straight for the trusty getopt(3) library. getopt has been a standard feature on development workstations for many years, and the current GNU version is installed on Linux workstations as a matter of course.

However, getopt is starting to look like a victim of its own success. Weighed down by new features, it has become somewhat unwieldy. It could be said that when a developer needs to add various punctuation marks to strings in order to invoke functionality, things have gotten a little out of hand.

For Python developers there is now a modern alternative to getopt. It's called optparse, and is available as part of the core language distribution from Python version 2.3 onwards. Written by Greg Ward, optparse claims to allow you to "add intelligent, sophisticated handling of command-line options to your scripts with very little overhead." This article looks at optparse, its usage and possibilities.

Basic Usage

optparse, like most Python modules, has an object based interface. Its usage involves three steps. Firstly, an option parser object is created. Next, method calls to that parser object are used to configure it to expect a set of command line options. Finally, the parser's parse_args() method is called. The parse_args() method examines the application's command line arguments to ensure they match the set expected, then builds a result object. This result object is returned containing a set of attributes, one for each command line argument expected. The value for each attribute is the value found on the command line.

Basic usage of optparse is easy to describe, but the easiest way to explore the details is to start looking at some code, so let's do that. Step one is to create a parser object, and although the class constructor takes some optional arguments, the simplest code is normally sufficient:

```
from optparse import *
optParser = OptionParser()
```

The next step is to tell the parser about the options it can expect to receive. This is done with the add_option() method. add_option() allows the developer to add a short format option (e.g. -v), a long format option (e.g. --verbose), or an option that takes both formats. A call to the add_option() method starts with the strings that represent the option on the command line:

```
optParser.add_option( "-f", "--file", …
```

The … represents the additional arguments that control the behaviour of the parser when it sees the option. It is best to use Python's named argument syntax for these additional arguments since they can be many and varied.

The next argument to specify should be the action, which tells the parser what action to take when it comes across the option in the command line. We'll look at some of the available actions in a moment, but for now we'll use the most often used one – "store":

```
optParser.add_option( "-f", "--file",
                      action="store", …
```

The store action tells the parser to take the next command line argument and store it in the parser's result ready for later use. The parser's result is an object which will have an attribute for each option that gets stored. For a store action, the name of the attribute is mandatory, and is therefore the next thing to specify with a dest argument:

```
optParser.add_option( "-f", "--file",
                      action="store", dest="filename", …
```

The parser also needs to be told what type of data to expect, with string, integer and float being the most popular. For this example, I need to specify a string:

```
optParser.add_option( "-f", "--file",
                      action="store", dest="filename", type="string",
                      …
```

The final argument, which should really be on every option's configuration, is some help text:

```
optParser.add_option( "-f", "--file",
                      action="store", dest="filename", type="string",
                      help="Input filename, or '-' for stdin" )
```

Help is required because, given a set of help strings, the optparse module is able to automatically build a nicely formatted help page from the options it knows about. This very handy feature prevents the developer having to maintain a separate "usage" string, which can easily get out of sync with the actual options accepted.
The add_option() method of the optParser object can be called as many times as is necessary to build up the complete set of command line options the application takes. Once they've all been added, the parser can be invoked with a call to its parse_args() method. By default this method assumes the command line arguments are in the argv list, which is almost always the case, so typically the parse is invoked with:

```
(options, remainingArgs) = optParser.parse_args()
```

When this is executed the parser will examine the command line, stopping with an appropriate error if the options provided don't match those expected. If everything parses correctly, the returned "options" object will contain details of the command line options found, and "remainingArgs" will be a list containing the rest of the arguments on the command line.
For the above example, if a command line like:

```
prog.py --file data.txt input.pdf
```

is parsed, the options object will have a `filename` attribute with the value `data.txt`, and remainingArgs will be a list with the single element `input.pdf`.
As a bonus, a help option is provided automatically for the "-h" and "--help" command line options:

```
> prog.py --help
usage: prog.py [options]

options:
  -h, --help            show this help message and exit
  -fFILENAME, --file=FILENAME
                        Input filename, or '-' for stdin
```

The help text can be fine tuned, but the default output is normally fine for most applications.
It can be seen from the above example that building sets of command line options with optparse is really quite easy. Let’s look at some more examples which show some more details, and a few of the more powerful options.

Details and Features

You may have noticed in the help text generated for the above example that the word “FILENAME” appeared automatically. It actually came from the name given for the destination attribute, which is converted to uppercase, then used by default. This often makes sense as long as the destination attribute has a sensible name, but sometimes it needs specifying. You use the metavar argument for this. For example, this option takes an integer as a weight, but prints KILOS in the help text:

```
optParser.add_option( "-w", "--weight",
                      action="store", type="int", dest="weight",
                      metavar="KILOS",
                      help="Required weight" )
```

Another occasionally useful option is nargs, which specifies the number of arguments on the command line which the option “consumes”. For example:

```
optParser.add_option( "-r", "--range",
                      action="store", type="int", dest="range",
                      nargs=2,
                      help="Range (2 values, start and end)" )
```

This example expects to see a --range argument on the command line followed by two integers. Both those integers will be consumed by the parser and the result attribute will contain them in a two element list.
So far we’ve only looked at the “store” action. Another useful action is “store_true” which is used to handle boolean arguments, such as the traditional –v for verbose output:

```
optParser.add_option( "-v", "--verbose",
                      action="store_true", dest="verbose",
                      default=False,
                      help="Provide extra information" )
```

No type is needed here since store_true can only work with boolean values. There is, however, another argument in here we haven’t seen before: the default value, which works pretty much as you’d expect. Without it, an command line argument which is either not supplied, or which doesn’t have an associated value, gets set to the Python built in value of “None”.
The “store_true” action has an inverse called “store_false”, which is used to switch off a normally on boolean value. Coupling two options together with the same destination allows an interesting tri-state sort of logic value:

```
optParser.add_option( "-c", "--colour",
                      action="store_true", dest="colour",
                      help="Use full colour settings" )
optParser.add_option( "-m", "--monochrome",
                      action="store_false", dest="colour",
                      help="Disable full colour" )
```

With these two options, specifying a "-c" on the command line sets the colour result to True, while specifying "-m" sets it to False. Since there is no default in either option, specifying neither "-c" nor "-m" on the command line leaves the colour result at None.
The "count" action counts the number of times an option is seen on the command line. This would typically be used for increasing something like a debug level. The more times the option is specified, the higher the level should be:

```
optParser.add_option( "-d", "--debug",
                      action="count", dest="debug",
                      help="Increase debugging level" )
```

Given this option and a command line with "-d -d -d" on it, the result object would have a debug attribute with a value of 3.
Alternatively, the "store_const" action can be used to set the result object's attribute to a predefined constant value based on an option. For example, to set a optimisation level, you might use:

```
optParser.add_option( "-n", "--no-optimisation",
                      action="store_const", dest="optlevel",
                      const=0,
                      help="Turn off optimisation" )
optParser.add_option( "-o", "--optimise",
                      action="store_const", dest="optlevel",
                      const=1,
                      help="Turn on optimisation" )
optParser.add_option( "-O", "--full-optimisation",
                      action="store_const", dest="optlevel",
                      const=2,
                      help="Use maximum optimisation(experimental)" )
```

which would set the optlevel attribute to 0, 1 or 2, based on the command line containing –n, -o or –O.
The "append" action takes a number of command line arguments of the same type and builds a list from them. Here's an example that builds a list of user names, as specified on the command line:

```
optParser.add_option( "-u", "--user",
                      action="append", dest="user_list",
                      help="Effected user name(repeat as required)" )
```

Given a command line containing "--user fred --user bert --user john", this will return a result attribute of user_list with a value of ['fred', 'bert', 'john'].
There is one more feature optparse has which is worth discussing, but this one isn't an action - it's a type called "choice", and is paired with a standard store action. The choice allows you to specify a number of choices, and the user will receive an error if they supply the option with an argument that isn't in the allowed choice selection:

```
optParser.add_option( "-U", "--unit",
                      action="store", type="choice", dest="unit",
                      choices=["in", "mm"],
                      help="Unit to use: in=inches, mm=millimetres" )
```

The choice type only works with strings.

Advanced Features and Callbacks

There are other advanced features of optparse which can be useful in special circumstances. Two of the most useful are option groups and callbacks.
Option groups allow you to logically group a set of options together. Although all options are actually created equal, and none get any preferential treatment at parse time, it sometimes makes sense to group a few of them. This grouping only happens on the help screen, but it can help to show related features, or isolate dangerous or rarely used options. To use the grouping facility, create an OptionGroup object, add your options to it in the normal way, then add the group to the parser:

```
sGroup = OptionGroup( optParser, "Specialised options",
                      "The following options are specialised. "
                      "Don't change them unless you know what "
                      "you are doing" )
sGroup.add_option( "-p", "--pi",
                   action="store", type="float", dest="pi",
                   default=3.141592,
                   help="Value for pi" )
optParser.add_option_group( sGroup )
```

This causes the help screen to look like this:

```
usage: prog.py [options]

options:
  -h, --help            show this help message and exit
  -fFILENAME, --file=FILENAME
                        Input filename, or '-' for stdin
                        Unit to use: in=inches, mm=millimetres

  Specialised options:
    The following options are specialised. Don't change them unless
    you know what you are doing
    -pPI, --pi=PI       Value for pi
```

The other advanced feature of optparse that is sometimes useful is the idea of callbacks. Callbacks are used when the standard argument parsing features aren't quite what you need. The principle is the same as all callback style operations: you supply a piece of code that the parser can call back to when it encounters something it can't deal with. Callbacks can get pretty complicated, and the fine details are beyond the scope of this article. However, a simple example might be where two options are related – say maximum and minimum values. The requirement might be that if the minimum is supplied, the maximum must be supplied after it, and that the minimum value must be lower than the maximum. The options to handle this might look like this:

```
optParser.add_option( "-M", "--min",
                      action="callback", type="int", dest="min",
                      callback=checkMinMax,
                      help="Minimum. Maximum must follow" )
optParser.add_option( "-X", "--max",
                      action="callback", type="int", dest="max",
                      callback=checkMinMax,
                      help="Maximum. Must be preceded by minimum" )
```

Here, the action is “callback” and the callback argument names a function called checkMinMax. The checkMinMax function is called with a set of parameters that allows it to examine the options provided. For my example criteria, the checkMinMax function might look like this:

```
def checkMinMax( option, opt, value, parser ):

    if option.dest == "min":
        for nextOpt in parser.rargs:
            if nextOpt == "--max" or nextOpt == "-X": break
        else:
            raise OptionValueError, \
                  "Maximum value required when minimum″
                  ‶ value is specified"

    if option.dest == "max":
        if parser.values.min == None:
            raise OptionValueError, \
                  "Minimum value expected before maximum value"
        if parser.values.min >= value:
            raise OptionValueError, \
                  "Minimum value must be less than maximum value"

    setattr( parser.values, option.dest, value )
```

This code handles both the maximum and minimum options. When it is called with “--min” it scans the remaining options on the command line to ensure a “--max” is following along somewhere. When it is called with “--max” it ensures a “--min” has already been seen, and that the minimum value supplied was less than the provided maximum value.

Conclusion

The venerable getopt library is still the standard option parsing solution for most languages, and still works with Python where it is needed for legacy code. However, for new Python applications, the power, convenience and relative simplicity of the optparse library should ensure that it replaces getopt as the foremost solution to the option parsing problem. It has been designed to be extensible and so shouldn’t suffer from the feature creep which has bogged down its predecessor. It is surely only a matter of time before software libraries similar to optparse start replacing getopt in other development languages.

Sidebar – Extending the optparse Module

The built in functionality of optparse makes it flexible enough to deal with the majority of option parsing requirements the average application might have. Its callback mechanism provides a comprehensive way of providing extra functionality to suit the unusual cases. However, there are always applications that require extraordinary (or just plain odd) features which a standard library like optparse can’t fulfil out of the box. To meet these requirements optparse can be extended using Python code that implements new features and adds them to the module structure so they can be used as, and alongside, the standard features. This is an advanced topic that is way beyond the scope of this article, but it’s worth having a quick look at the idea in order to understand the possibilities.

The most common extending requirements are the additions of new data types and new actions which dictate what to do with the data supplied. For example, you could add a data type of “url” to the list of standard types, then provide a function that examines the supplied option string to ensure it fits your application’s idea of a valid URL. Once installed, your url data type could be used in add_option() method calls just like the standard types.

There are other reasons to extend the module. Perhaps you’d like to replace the layout of the generated help page? Perhaps you’d like to deal with option errors in some way other than just exiting the program? Perhaps you’d like to replace the hyphen option specifier (e.g. –d) with a more DOS-like slash specifier (e.g. /d). All of these are possible by subclassing the optparse classes and reimplementing the areas which need adjusting to meet your application requirements.