Carl Haddick
PO Box 1586
Mexia TX 76667
carl@carlhaddick.com

Visit https://www.carlhaddick.com for updates and more

December 11, 2020

Options

Parsing command line options with Python's getopt module is easy enough, but if you want to do the job right, you should catch typos and make sure that conflicting options aren't chosen.

The Options module takes care of housekeeping like that while reducing statements in your application.

Options

Command line options can be a pain. The idea sounds simple enough, take some command line parameters and act on them. Often, that's all there is to it.

Imagine, though, a Python application that serves different phases of a problem. Maybe you need to create a report from multiple data sources. You run the script several times to enroll new data in your database with options for data gathering. A final run is done with options for report production.

Some options won't be compatible with other options. Some options might mandate additional options, like output file names.

It would be nice to prescreen your command line arguments, and that's part of what the Options module does.

The Options module solves several problems and eliminates coding. It's a very easy module to use, but some context is in order. This document does not cover a complete application. Options is a support module.

Options is typically used as the first step in a Python application. it parses the command line or a string containing arguments. This document will explain how Options works without addressing any specific usage.

Arguments are passed in a string or on the command line in typical "getopt" style. Every option has two names, a single character and a verbose form.

For more information about the underlying options structure, please refer to documentation for the Python getopt module. Most of what you need to know to put Options to work will be contained here.

Single character commands are prefaced with a single hyphen. Here's a script called progname with a single parameter, o:

```
progname -o
```

Multiple single character commands can appear after a single hyphen. These two command lines are equal, and both invoke options o, p, and q:

```
progname -opq
progname -o -p -q
```

Each parameter has a verbose name. If -o has the long name “output”, it would appear with two hyphens. That makes --output different from -output, which would actually be six parameters, the single character o, u, t, p, u, and t options.

```
progname --output
```

Command line parameters can have an additional value. Single character commands use a colon. Verbose commands use an equals sign. These are equivalent:

```
progname -o:logfile.txt
progname --output=logfile.txt
```

Options.Param

Each instance of the Param class houses a single parameter. The primary purpose for Param objects is for the Options class, documented later.

Permissible command line options are defined in Param objects.

The Param object constructor recognizes these attributes:

- shortName - a single character name for a command line option.
- longName - the verbose name for this command.
- required - iterable of short option names that are required companions to the current option.
- optional - iterable of short option names that are compatible with the current option.
- extra non-keyword args for designated function (see below).
- paramFlag - True if this parameter accepts a value, as in -o:output.txt.
- sanityCheck - a function to all to check option sanity for this option. This function is called by the constructor to the Options class.
- paramFunc - a function called to service this option.
- keyword args for passing to paramFunc (see below).

There are certain manual aspects to defining lists of command line options. The easiest way to keep setup under control is to remember what the optional and required checks actually do.

Each option on a command line parsed by Options must be compatible with all other options.

For instance, imagine options a, b, and c that are all mutually compatible. In addition, if option b appears on the command line, option z is required.

Here’s a tuple of Param objects that satisfies those relationships:

```
opts = (Param(‘a’, ‘optionA’, ‘’, ‘bc’),
        Param(‘b’, ‘optionB’, ‘z’, ‘ac’),
        Param(‘c’, ‘optionC’, ‘’, ‘ab’),
        Param(‘z’, ‘optionZ’, ‘’, ‘abc’, paramFlag = True))
```

A and c can coexist. If option b is specified on the command line, it will have to be in the company of option z. These are legal command lines:

```
progname -ac
progname -c
prognamem -bz:logfile.txt
```

If you want functions to automatically execute, this will trigger the named functions for the command line arguments:

```
opts = (Param(‘a’, ‘optionA’, ‘’, ‘bc’, paramFunc=funcA),
        Param(‘b’, ‘optionB’, ‘z’, ‘ac’),
        Param(‘c’, ‘optionC’, ‘’, ‘ab’, paramFunc=funcC),
        Param(‘z’, ‘optionZ’, ‘’, ‘abc’, paramFunc=funcZ,
              paramFlag = True))
```

The only method in the Param class is the constructor, __init__. Param exists as a container for the attributes defining a parameter.

Options.OptionsError

This is the exception class raised by problems encountered in Options.Options.

OptionsError has two members of interest. OptionsError.idx is a numeric flag indicating both what happened to create the exception and where in the code the exception was raised.

The current values are:

- 1 - general error parsing the command line. An option not defined with paramFlag set to True but has an associated value will raise this error. For example, Param(‘z’, ‘optionZ’, ‘’, ‘’, paramFlag=False) with a command line argument of -z:someparameter.
- 2 - a duplicate shortName (single character option name) was encountered in the Param list handed to Options.__init__().
- 3 - a duplicate longName (verbose command line option name) was found in the Param list.
- 4 - a required companion option was missing in the command line.
- 5 - a companion option not listed as compatible was found in the command line.

OptionsError.errstr will contain a descriptive indication of what error was found. See the typical usage section for an example.

Options.Options

The Options class does the real work of handling command line parameters, including sanity checking and optional function dispatching.

Options.__init__

The Options constructor takes two arguments, args and opts. Args is a string of getopt style options, typically sys.argv[1:]. Opts is a tuple of Param objects detailing permissible command line parameters.

The tuple of permissible options is checked in by __init__. Each Param actually present on the command line is flagged as an active option.

Python’s getopt.getopt is called to parse the args string.

Each active option’s optional and required iterables are checked to make sure all other active options are either optional are required.

Any option not either optional or required will raise an exception.

Options.sanityCheck

This method is called after the options are checked in and the command line arguments parsed. The default implementation of sanityCheck confirms optional or required status for all active command line arguments.

Options.checkForReq

Used by sanityCheck to check for required companion options.

Options.checkValidCompanions

Used by sanityCheck to check to make sure all companions to each active command line option are either in required or optional lists.

Options.printOpts

Options.printOpts is a convenience function to print the active options from the command line. This is usually only used to confirm command line interpretation for debugging.

Options.activeOpts

This method returns an array of Param objects corresponding to active command line arguments.

Options.execParams

If called, this method will go through all active command line arguments in the order they appear in the constructor's opts parameter. For each active argument, the corresponding Param.paramFunc method will be called, passing *args and **kwargs from Param's constructor to paramFunc.

Options.fetchOpt

This function prompts for a replacement value for a command line parameter. FetchOpt understands these parameters:

- option - short name for a valid option
- prompt - an optional prompt. If it is None, the user won't be prompted for input.
- default - an optional default value to return.

If the option is active on the command line, its associated parameter will be returned. In the case of “-o:logfile.txt,” the return would be “logfile.txt.”

Otherwise, if the prompt parameter is non-None, the user will be prompted to enter a value.

If the parameter isn’t active and no prompt is set, the default value will be returned.

Typical usage

Here is a sample case for Options.

These command line parameters will be supported. Note that these options are simply hypothetical. Specific names are cited for the purpose of example and do not represent any specific function in the Options module.

- a - append. Compatible with option c. Requires the o argument
- b - bifurcate. Not compatible with other options.
- c - concatenate. Compatible with options a.
- o - output. Compatible with option c. Requires a.

Functions fA, fB, and fC will handle the command line operations. No function is needed for option o because it is tied to option a.

```
# declare acceptable parameters
params = (Param(‘a’, ‘append’, ‘o’, ‘c’, paramFunc=fA),
          Param(‘b’, ‘bifurcate’, ‘’, ‘’, paramFunc=fB),
          Param(‘c’, ‘concatenate’, ‘’, ‘ao’, paramFunc=fC),
          Param(‘o’, ‘output’, ‘a’, ‘c’, paramFlag=True))
try:
  opt = Options(sys.argv[1:], params) # parse command line
  opt.execParams() # execute paramFuncs
except OptionsError as e:
  print(e.errstr) # print error message
```

This setup will complain if any options appear with -b, or if -a isn’t accompanied by -o.

If -o is used by itself, that will also raise an exception. The -c option can appear with -ao.

These are all legal command lines in this example:

```
progname -ao:output.file -c
progname -b
progname --append --output=output.file
progname --concatenate
progname -c
```