CommandLineApp Documentation

Release 3.0.7

Doug Hellmann

July 20, 2012

CONTENTS

Contents:

CHAPTER

ONE

COMMANDLINEAPP – COMMAND LINE APPLICATION BUILDER

1.1 Application Base Class

CHAPTER

TWO

COMMAND LINE PROGRAMS ARE CLASSES, TOO!

Note: This article was originally published in the November 2007 issue of Python Magazine. It has been updated to match the more recent versions of `CommandLineApp`.

Most OOP discussions focus on GUI or domain-specific development areas, completely ignoring the workhorse of computing: command line programs. This article examines CommandLineApp, a base class for creating command line programs as objects, with option and argument validation, help text generation, and more.

Although many of the hot new development topics are centered on web technologies like AJAX, regular command line programs are still an important part of most systems. Many system administration tasks still depend on command line programs, for example. Often, a problem is simple enough that there is no reason to build a graphical or web user interface when a straightforward command line interface will do the job. Command line programs are less glamorous than programs with fancy graphics, but they are still the workhorses of modern computing.

The Python standard library includes two modules for working with command line options. The **getopt** module presents an API that has been in use for decades on some platforms and is commonly available in many programming languages, from C to bash. The **optparse** module is more modern than **getopt**, and offers features such as type validation, callbacks, and automatic help generation. Both modules elect to use a procedural-style interface, though, and as a result neither has direct support for treating your command line application as a first class object. There is no facility for sharing common options between related programs using **getopt**. And, while it is possible to reuse `optparse.OptionParser` instances in different programs, it is not as natural as inheritance.

CommandLineApp is a base class for command line programs. It handles the repetitive aspects of interacting with the user on the command line such as parsing options and arguments, generating help messages, error handling, and printing status messages. To create your application, just make a subclass of `CommandLineApp` and concentrate on your own code. All of the information about switches, arguments, and help text necessary for your program to run is derived through introspection. Common options and behavior can be shared by applications through inheritance.

2.1 csvcat Requirements

Recently, I needed to combine data from a few different sources, including a database and a spreadsheet, to summarize the results. I wanted to import the merged data into a spreadsheet where I could perform the analysis. All of the sources were able to save data to comma-separated-value (CSV) files; the challenge was merging the files together. Using the **csv** module in the Python standard library, and `CommandLineApp`, I wrote a small program to read multiple CSV files and concatenate them into a single output file. The program, csvcat, is a good illustration of how to create applications with `CommandLineApp`.

The requirements for **csvcat** were fairly simple. It needed to read one or more CSV files and combine them, without repeating the column headers that appeared in each input source. In some cases, the input data included columns I did not want, so I needed to be able to select the columns to include in the output. No sort feature was needed, since I was going to import it into a spreadsheet when I was done and I could sort the data after importing it. To make the program more generally useful, I also included the ability to select the output format using a **csv** module feature called "dialects".

2.2 Analyzing the Help

Listing 1 shows the help output for the final version of **csvcat**, produced by running `csvcat --help`. Listing 2 shows the source for the program. All of the information in the help output is derived from the **csvcat** class through introspection. The help text follows a fairly standard layout. It begins with a description of the application, followed by increasingly more detailed descriptions of the syntax, arguments, and options. Application-specific help such as examples and argument ranges appears at the end.

2.2.1 Listing 1

```
$ python docs/source/PyMagArticle/Listing2.py --help
Concatenate comma separated value files.


SYNTAX:

  csvcat [<options>] filename [filename...]

    -c col[,col...], --columns=col[,col...]
    -d name, --dialect=name
    --debug
    -h
    --help
    --quiet
    --skip-headers
    -v
    --verbose=level


ARGUMENTS:

    The names of comma separated value files, such as might be
    exported from a spreadsheet or database program.


OPTIONS:

    -c col[,col...], --columns=col[,col...]
        Limit the output to the specified columns. Columns are
        identified by number, starting with 0.

    -d name, --dialect=name
        Specify the output dialect name. Defaults to "excel".

    --debug
        Set debug mode to see tracebacks.
```

```
    -h
        Displays abbreviated help message.

    --help
        Displays verbose help message.

    --quiet
        Turn on quiet mode.

    --skip-headers
        Treat the first line of each file as a header, and only
        include one copy in the output.

    -v
        Increment the verbose level.

        Higher levels are more verbose. The default is 1.

    --verbose=level
        Set the verbose level.

EXAMPLES:


To concatenate 2 files, including all columns and headers:

  $ csvcat file1.csv file2.csv

To concatenate 2 files, skipping the headers in the second file:

  $ csvcat --skip-headers file1.csv file2.csv

To concatenate 2 files, including only the first and third columns:

  $ csvcat --col 0,2 file1.csv file2.csv
```

2.2.2 Listing 2

```
#!/usr/bin/env python
"""Concatenate csv files.
"""

import csv
import sys
import commandlineapp

class csvcat(commandlineapp.CommandLineApp):
    """Concatenate comma separated value files.
    """

    _app_name = 'csvcat'

    EXAMPLES_DESCRIPTION = '''
To concatenate 2 files, including all columns and headers:

  $ csvcat file1.csv file2.csv

```

```
To concatenate 2 files, skipping the headers in the second file:

  $ csvcat --skip-headers file1.csv file2.csv

To concatenate 2 files, including only the first and third columns:

  $ csvcat --col 0,2 file1.csv file2.csv
'''

    def showVerboseHelp(self):
        commandlineapp.CommandLineApp.showVerboseHelp(self)
        print
        print 'OUTPUT DIALECTS:'
        print
        for name in csv.list_dialects():
            print '\t%s' % name
        print
        return

    skip_headers = False
    def option_handler_skip_headers(self):
        """Treat the first line of each file as a header,
        and only include one copy in the output.
        """
        self.skip_headers = True
        return

    dialect = "excel"
    def option_handler_dialect(self, name):
        """Specify the output dialect name.
        Defaults to "excel".
        """
        self.dialect = name
        return
    option_handler_d = option_handler_dialect

    columns = []
    def option_handler_columns(self, *col):
        """Limit the output to the specified columns.
        Columns are identified by number, starting with 0.
        """
        self.columns.extend([int(c) for c in col])
        return
    option_handler_c = option_handler_columns

    def getPrintableColumns(self, row):
        """Return only the part of the row which should be printed.
        """
        if not self.columns:
            return row

        # Extract the column values, in the order specified.
        response = ()
        for c in self.columns:
            response += (row[c],)
        return response

    def getWriter(self):
```

```
78          return csv.writer(sys.stdout, dialect=self.dialect)
79
80      def main(self, *filename):
81          """
82          The names of comma separated value files, such as might be
83          exported from a spreadsheet or database program.
84          """
85          headers_written = False
86
87          writer = self.getWriter()
88
89          # process the files in order
90          for name in filename:
91              f = open(name, 'rt')
92              try:
93                  reader = csv.reader(f)
94
95                  if self.skip_headers:
96                      if not headers_written:
97                          # This row must include the headers for the output
98                          headers = reader.next()
99                          writer.writerow(self.getPrintableColumns(headers))
100                         headers_written = True
101                     else:
102                         # We have seen headers before, and are skipping,
103                         # so do not write the first row of this file.
104                         ignore = reader.next()
105
106                 # Process the rest of the file
107                 for row in reader:
108                     writer.writerow(self.getPrintableColumns(row))
109             finally:
110                 f.close()
111         return
112
113 if __name__ == '__main__':
114     csvcat().run()
```

The program description is taken from the docstring of the **csvcat** class. Before it is printed, the text is split into paragraphs and reformatted using **textwrap**, to ensure that it is no wider than 80 columns of text.

The program description is followed by a syntax summary for the program. The options listed in the syntax section correspond to methods with names that begin with `option_handler_`. For example, `option_handler_skip_headers()` indicates that **csvcat** should accept a `--skip-headers` option on the command line.

The names of any non-optional arguments to the program appear in the syntax summary. In this case, **csvcat** needs the names of the files containing the input data. At least one file name is necessary, and multiple names can be given, as indicated by the fact that the `filename` argument to `main()` uses the variable argument notation: `*filename`. A longer description of the arguments, taken from the docstring of the `main()` method (lines 79-82), follows the syntax summary. As with the general program summary, the description of the arguments is reformatted with **textwrap** to fit the screen.

2.3 Options and Their Arguments

Following the argument description is a detailed explanation of all of the options to the program. `CommandLineApp` examines each option handler method to build the option description, including the name of the option, alternative names for the same option, and the name and description of any arguments the option accepts. There are three variations of option handlers, based on the arguments used by the option.

The simplest kind of option does not take an argument at all, and is used as a "switch" to turn a feature on or off. The method `option_handler_skip_headers` (lines 38-43) is an example of such a switch. The method takes no argument, so `CommandLineApp` recognizes that the option being defined does not take an argument either. To create the option name, the prefix is stripped from the method name, and the underscore is converted to a dash (`-`); `option_handler_skip_headers` becomes `--skip-headers`.

Other options accept a single argument. For example, the `--dialect` option requires the name of the CSV output dialect. The method `option_handler_dialect` (lines 46-51) takes one argument, called `name`. The suggested syntax for the option, as seen in Listing 1, is `--dialect=name`. The name of the method's argument is used as the name of the argument to the option in the help text.

The `-d` option has the same meaning as `--dialect`, because `option_handler_d` is an alias for `option_handler_dialect`. `CommandLineApp` recognizes aliases, and combines the forms in the documentation so the alternative forms `-d name` and `--dialect=name` are described together.

It is often useful for an option to take multiple arguments, as with `--columns`. The user could repeat the option on the command line, but it is more compact to allow them to list multiple values in one argument list. When `CommandLineApp` sees an option handler method that takes a variable argument list, it treats the corresponding option as accepting a list of arguments. When the option appears on the command line, the string argument is split on any commas and the resulting list of strings is passed to the option handler method.

For example, `option_handler_columns` (lines 55-60) takes a variable length argument named `col`. The option `--columns` can be followed by several column numbers, separated by commas. The option handler is called with the list of values pre-parsed. In the syntax description, the argument is shown repeating: `--columns=col[,col...]`.

For all cases, the docstring from the option handler method serves as the help text for the option. The text of the docstring is reformatted using **textwrap** so both the code and help output are easy to read without extra effort on the part of the developer.

2.4 Application-specific Detailed Help

The general syntax and option description information is produced in the same way for all `CommandLineApp` programs. There are times when an application needs to include additional information in the help output, though, and there are two ways to add such information.

The first way is by providing examples of how to use the program on the command line. Although it is optional, including examples of how to apply different combinations of arguments to your program to achieve various results enhances the usefulness of the help as a reference manual. When the `EXAMPLES_DESCRIPTION` class attribute is set, it is used as the source for the examples. Unlike the other documentation strings, the `EXAMPLES_DESCRIPTION` is printed directly without being reformatted. This preserves the indentation and other formatting of the examples, so the user sees an accurate representation of the program's inputs and outputs.

Occasionally, a program may need to include information in its help output which cannot be statically defined in a docstring or derived by `CommandLineApp`. At the very end of its help, **csvcat** includes a list of available CSV dialects which can be used with the `--dialect` option. Since the list of dialects must be constructed at runtime based on what dialects have been registered with the **csv** module, **csvcat** overrides `showVerboseHelp()` to print the list itself (lines 27-35).

2.5 Using csvcat

The inputs to **csvcat** are any number of CSV files, and the output is CSV data printed to standard output. To test **csvcat** during development, I created two small files with test data. Each file contains three columns of data: a number, a string, and a date.

```
$ cat testdata1.csv
"Title 1","Title 2","Title 3"
1,"a",08/18/07
2,"b",08/19/07
3,"c",08/20/07
```

The second file does not include quotes around any of the string fields. I chose to include this variation because **csvcat** does not quote its output, so using unquoted test data simulates re-processing the output of **csvcat**.

```
$ cat testdata2.csv
Title 1,Title 2,Title 3
40,D,08/21/07
50,E,08/22/07
60,F,08/23/07
```

The simplest use of **csvcat** is to print the contents of an input file to standard output. Notice that the output does not include quotes around the string fields.

```
$ csvcat testdata1.csv
Title 1,Title 2,Title 3
1,a,08/18/07
2,b,08/19/07
3,c,08/20/07
```

It is also possible to select which columns should be included in the output using the `--columns` option. Columns are identified by their number, beginning with `0`. Column numbers can be listed in any order, so it is possible to reorder the columns of the input data, if needed.

```
$ csvcat --columns 2,0 testdata1.csv
Title 3,Title 1
08/18/07,1
08/19/07,2
08/20/07,3
```

Switching to tab-separated columns instead of comma-separated is easily accomplished by using the `--dialect` option. There are only two dialects available by default, but the the **csv** module API supports registering additional dialects.

```
$ csvcat --dialect excel-tab testdata1.csv
Title 1 Title 2 Title 3
1       a       08/18/07
2       b       08/19/07
3       c       08/20/07
```

For my project, there were input files with several columns, but only two of them needed to be included in the output. Each file had a single row of column headers. I only wanted one set of headers in the output, so the headers from subsequent files needed to be skipped. And the output had to be in a format I could import into a spreadsheet, for which the default "excel" dialect worked fine. The data was merged with a command like this:

```
$ csvcat --skip-headers --columns 2,0 testdata1.csv testdata2.csv
Title 3,Title 1
08/18/07,1
08/19/07,2
```

```
08/20/07,3
08/21/07,40
08/22/07,50
08/23/07,60
```

2.6 Running a CommandLineApp Program

Most of the work for **csvcat** is being done in the `main()` method. To invoke the application, however, the caller does not invoke `main()` directly. The program should be started by calling `run()`, so the options are validated and exceptions from `main()` are handled. The `run()` method is one of several methods that are not intended to be overridden by derived classes, since they implement the core features of a command line program. The source for `CommandLineApp` appears in Listing 3.

2.6.1 Listing 3

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# Copyright 2007 Doug Hellmann.
#
#
#                         All Rights Reserved
#
# Permission to use, copy, modify, and distribute this software and
# its documentation for any purpose and without fee is hereby
# granted, provided that the above copyright notice appear in all
# copies and that both that copyright notice and this permission
# notice appear in supporting documentation, and that the name of Doug
# Hellmann not be used in advertising or publicity pertaining to
# distribution of the software without specific, written prior
# permission.
#
# DOUG HELLMANN DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
# INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
# NO EVENT SHALL DOUG HELLMANN BE LIABLE FOR ANY SPECIAL, INDIRECT OR
# CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
# NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
# CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
#

"""Base class for building command line applications.

:class:`CommandLineApp` makes creating command line applications as
simple as defining callbacks to handle options when they appear in
``sys.argv``.
"""

#
# Import system modules
#
import getopt
import inspect
import os
```

```
try:
    from cStringIO import StringIO
except:
    from StringIO import StringIO
import sys
import textwrap

#
# Import Local modules
#

#
# Module
#

class OptionDef(object):
    """Definition for a command line option.

    Attributes:

      method_name - The name of the option handler method.
      option_name - The name of the option.
      switch      - Switch to be used on the command line.
      arg_name    - The name of the argument to the option handler.
      is_variable - Is the argument expected to be a sequence?
      default     - The default value of the option handler argument.
      help        - Help text for the option.
      is_long     - Is the option a long value (--) or short (-)?
    """

    # Option handler method names start with this value
    OPTION_HANDLER_PREFIX = 'option_handler_'

    # For *args arguments to option handlers, how to split the argument values
    SPLIT_PARAM_CHAR = ','

    def __init__(self, method_name, method):
        self.method_name = method_name
        self.option_name = method_name[len(self.OPTION_HANDLER_PREFIX):]
        self.is_long = len(self.option_name) > 1

        self.switch_base = self.option_name.replace('_', '-')
        if len(self.switch_base) == 1:
            self.switch = '-' + self.switch_base
        else:
            self.switch = '--' + self.switch_base

        argspec = inspect.getargspec(method)

        self.is_variable = False
        args = argspec[0]
        if len(args) > 1:
            self.arg_name = args[-1]
        elif argspec[1]:
            self.arg_name = argspec[1]
            self.is_variable = True
        else:
            self.arg_name = None
```

```
98
99          if argspec[3]:
100             self.default = argspec[3][0]
101         else:
102             self.default = None
103
104         self.help = inspect.getdoc(method)
105         return
106
107     def get_switch_text(self):
108         """Return the description of the option switch.
109
110         For example: --switch=arg or -s arg or --switch=arg[,arg]
111         """
112         parts = [ self.switch ]
113         if self.arg_name:
114             if self.is_long:
115                 parts.append('=')
116             else:
117                 parts.append(' ')
118             parts.append(self.arg_name)
119             if self.is_variable:
120                 parts.append('[%s%s...]' % (self.SPLIT_PARAM_CHAR, self.arg_name))
121         return ''.join(parts)
122
123
124     def invoke(self, app, arg):
125         """Invoke the option handler.
126         """
127         method = getattr(app, self.method_name)
128         if self.arg_name:
129             if self.is_variable:
130                 opt_args = arg.split(self.SPLIT_PARAM_CHAR)
131                 method(*opt_args)
132             else:
133                 method(arg)
134         else:
135             method()
136         return
137
138
139 class CommandLineApp(object):
140     """Base class for building command line applications.
141
142     Define a docstring for the class to explain what the program does.
143
144     Include descriptions of the command arguments in the docstring for
145     ''main()''.
146
147     When the ''EXAMPLES_DESCRIPTION'' class attribute is not empty, it
148     will be printed last in the help message when the user asks for
149     help.
150     """
151
152     EXAMPLES_DESCRIPTION = ''
153
154     # If true, always ends run() with sys.exit()
155     force_exit = True
```

```
156
157     # The name of this application
158     _app_name = os.path.basename(sys.argv[0])
159
160     _app_version = None
161
162     def __init__(self, command_line_options=None):
163         "Initialize CommandLineApp."
164         if command_line_options is None:
165             command_line_options = sys.argv[1:]
166         self.command_line_options = command_line_options
167         self.before_options_hook()
168         self.supported_options = self.scan_for_options()
169         self.after_options_hook()
170         return
171
172     def before_options_hook(self):
173         """Hook to initialize the app before the options are processed.
174
175         Overriding __init__() requires special handling to make sure the
176         arguments are still passed to the base class.  Override this method
177         instead to create local attributes or do other initialization before
178         the command line options are processed.
179         """
180         return
181
182     def after_options_hook(self):
183         """Hook to initialize the app after the options are processed.
184
185         Overriding __init__() requires special handling to make sure the
186         arguments are still passed to the base class.  Override this method
187         instead to create local attributes or do other initialization after
188         the command line options are processed.
189         """
190         return
191
192     def main(self, *args):
193         """Main body of your application.
194
195         This is the main portion of the app, and is run after all of
196         the arguments are processed.  Override this method to implment
197         the primary processing section of your application.
198         """
199         pass
200
201     def handle_interrupt(self):
202         """Called when the program is interrupted via Control-C
203         or SIGINT.  Returns exit code.
204         """
205         sys.stderr.write('Canceled by user.\n')
206         return 1
207
208     def handle_main_exception(self, err):
209         """Invoked when there is an error in the main() method.
210         """
211         if self.debugging:
212             import traceback
213             traceback.print_exc()
```

```
214             else:
215                 self.error_message(str(err))
216             return 1
217 
218         ## HELP
219 
220         def show_help(self, error_message=None):
221             "Display help message when error occurs."
222             print
223             if self._app_version:
224                 print '%s version %s' % (self._app_name, self._app_version)
225             else:
226                 print self._app_name
227             print
228 
229             #
230             # If they made a syntax mistake, just
231             # show them how to use the program.  Otherwise,
232             # show the full help message.
233             #
234             if error_message:
235                 print ''
236                 print 'ERROR: ', error_message
237                 print ''
238                 print ''
239                 print '%s\n' % self._app_name
240                 print ''
241 
242             txt = self.get_simple_syntax_help_string()
243             print txt
244             print 'For more details, use --help.'
245             print
246             return
247 
248         def show_verbose_help(self):
249             "Display the full help text for the command."
250             txt = self.get_verbose_syntax_help_string()
251             print txt
252             return
253 
254         ## STATUS MESSAGES
255 
256         def _status_message(self, msg, output):
257             if isinstance(msg, unicode):
258                 to_print = msg.encode('ascii', 'replace')
259             else:
260                 to_print = unicode(msg, 'utf-8').encode('ascii', 'replace')
261             output.write(to_print)
262             return
263 
264         def status_message(self, msg='', verbose_level=1, error=False, newline=True):
265             """Print a status message to output.
266 
267             msg
268                 The status message string to be printed.
269             verbose_level
270                 The verbose level to use.  The message
271                 will only be printed if the current verbose
```

```
272             level is >= this number.
273         error
274             If true, the message is considered an error and
275             printed as such.
276         newline
277             If true, print a newline after the message.
278
279         """
280         if self.verbose_level >= verbose_level:
281             if error:
282                 output = sys.stderr
283             else:
284                 output = sys.stdout
285             self._status_message(msg, output)
286             if newline:
287                 output.write('\n')
288             # some log mechanisms don't have a flush method
289             if hasattr(output, 'flush'):
290                 output.flush()
291         return
292
293     def error_message(self, msg=''):
294         'Print a message as an error.'
295         self.status_message('ERROR: %s\n' % msg, verbose_level=0, error=True)
296         return
297
298     ## DEFAULT OPTIONS
299
300     debugging = False
301     def option_handler_debug(self):
302         "Set debug mode to see tracebacks."
303         self.debugging = True
304         return
305
306     _run_main = True
307     def option_handler_h(self):
308         "Displays abbreviated help message."
309         self.show_help()
310         self._run_main = False
311         return
312
313     def option_handler_help(self):
314         "Displays verbose help message."
315         self.show_verbose_help()
316         self._run_main = False
317         return
318
319     def option_handler_quiet(self):
320         'Turn on quiet mode.'
321         self.verbose_level = 0
322         return
323
324     verbose_level = 1
325     def option_handler_v(self):
326         """Increment the verbose level.
327
328         Higher levels are more verbose.
329         The default is 1.
```

```
330         """
331         self.verbose_level = self.verbose_level + 1
332         self.status_message('New verbose level is %d' % self.verbose_level,
333                             3)
334         return
335 
336     def option_handler_verbose(self, level=1):
337         """Set the verbose level.
338         """
339         self.verbose_level = int(level)
340         self.status_message('New verbose level is %d' % self.verbose_level,
341                             3)
342         return
343 
344     ## INTERNALS (Subclasses should not need to override these methods)
345 
346     def run(self):
347         """Entry point.
348 
349         Process options and execute callback functions as needed.
350         This method should not need to be overridden, if the main()
351         method is defined.
352         """
353         # Process the options supported and given
354         options = {}
355         for info in self.supported_options:
356             options[ info.switch ] = info
357         parsed_options, remaining_args = self.call_getopt(self.command_line_options,
358                                                           self.supported_options)
359         exit_code = 0
360         try:
361             for switch, option_value in parsed_options:
362                 opt_def = options[switch]
363                 opt_def.invoke(self, option_value)
364 
365             # Perform the primary action for this application,
366             # unless one of the options has disabled it.
367             if self._run_main:
368                 main_args = tuple(remaining_args)
369 
370                 # We could just call main() and catch a TypeError,
371                 # but that would not let us differentiate between
372                 # application errors and a case where the user
373                 # has not passed us enough arguments.  So, we check
374                 # the argument count ourself.
375                 num_args_ok = False
376                 argspec = inspect.getargspec(self.main)
377                 defaults = argspec[3]
378                 # Arguments with defaults are not required, so subtract them
379                 expected_arg_count = len(argspec[0]) - 1 - len(defaults or [])
380 
381                 if argspec[1] is not None:
382                     num_args_ok = True
383                     if len(argspec[0]) > 1:
384                         num_args_ok = (len(main_args) >= expected_arg_count)
385                 elif len(main_args) == expected_arg_count:
386                     num_args_ok = True
387 
```

```
388                 if num_args_ok:
389                     exit_code = self.main(*main_args)
390                 else:
391                     self.show_help('Incorrect arguments.')
392                     exit_code = 1
393 
394         except KeyboardInterrupt:
395             exit_code = self.handle_interrupt()
396 
397         except SystemExit, msg:
398             exit_code = msg.args[0]
399 
400         except Exception, err:
401             exit_code = self.handle_main_exception(err)
402 
403         if self.force_exit:
404             sys.exit(exit_code)
405         return exit_code
406 
407     def scan_for_options(self):
408         "Scan through the inheritence hierarchy to find option handlers."
409         options = []
410 
411         methods = inspect.getmembers(self.__class__, inspect.ismethod)
412         for method_name, method in methods:
413             if method_name.startswith(OptionDef.OPTION_HANDLER_PREFIX):
414                 options.append(OptionDef(method_name, method))
415 
416         return options
417 
418     def call_getopt(self, command_line_options, supported_options):
419         "Parse the command line options."
420         short_options = []
421         long_options = []
422         for o in supported_options:
423             if len(o.option_name) == 1:
424                 short_options.append(o.option_name)
425                 if o.arg_name:
426                     short_options.append(':')
427             elif o.arg_name:
428                 long_options.append('%s=' % o.switch_base)
429             else:
430                 long_options.append(o.switch_base)
431 
432         short_option_string = ''.join(short_options)
433 
434         try:
435             parsed_options, remaining_args = getopt.getopt(
436                 command_line_options,
437                 short_option_string,
438                 long_options)
439         except getopt.error, message:
440             self.show_help(message)
441             if self.force_exit:
442                 sys.exit(1)
443             raise
444         return (parsed_options, remaining_args)
445 
```

```
446     def _group_option_aliases(self):
447         """Return a sequence of tuples containing
448         (option_names, option_defs)
449         """
450         # Figure out which options are aliases
451         option_aliases = {}
452         for option in self.supported_options:
453             method = getattr(self, option.method_name)
454             existing_aliases = option_aliases.setdefault(method, [])
455             existing_aliases.append(option)
456 
457         # Sort the groups in order
458         grouped_options = []
459         for options in option_aliases.values():
460             names = [ o.option_name for o in options ]
461             grouped_options.append( (names, options) )
462         grouped_options.sort()
463         return grouped_options
464 
465     def _get_option_identifier_text(self, options):
466         """Return the option identifier text.
467 
468         For example:
469 
470         -h
471 
472         -v, --verbose
473 
474         -f bar, --foo bar
475         """
476         option_texts = []
477         for option in options:
478             option_texts.append(option.get_switch_text())
479         return ', '.join(option_texts)
480 
481     def get_arguments_syntax_string(self):
482         """Look at the arguments to main to see what the program accepts,
483         and build a syntax string explaining how to pass those arguments.
484         """
485         syntax_parts = []
486         argspec = inspect.getargspec(self.main)
487         args = argspec[0]
488         if len(args) > 1:
489             for arg in args[1:]:
490                 syntax_parts.append(arg)
491         if argspec[1]:
492             syntax_parts.append(argspec[1])
493             syntax_parts.append('[' + argspec[1] + '...]')
494         syntax = ' '.join(syntax_parts)
495         return syntax
496 
497     def get_simple_syntax_help_string(self):
498         """Return syntax statement.
499 
500         Return a simplified form of help including only the
501         syntax of the command.
502         """
503         buffer = StringIO()
```

```
504
505         # Show the name of the command and basic syntax.
506         buffer.write('%s [<options>] %s\n\n' % \
507                      (self._app_name, self.get_arguments_syntax_string())
508                  )
509
510         grouped_options = self._group_option_aliases()
511
512         # Assemble the text for the options
513         for names, options in grouped_options:
514             buffer.write('    %s\n' % self._get_option_identifier_text(options))
515
516         return buffer.getvalue()
517
518     def _format_help_text(self, text, prefix):
519         if not text:
520             return ''
521         buffer = StringIO()
522         text = textwrap.dedent(text)
523         for para in text.split('\n\n'):
524             formatted_para = textwrap.fill(para,
525                                            initial_indent=prefix,
526                                            subsequent_indent=prefix,
527                                            )
528             buffer.write(formatted_para)
529             buffer.write('\n\n')
530         return buffer.getvalue()
531
532     def get_verbose_syntax_help_string(self):
533         """Return the full description of the options and arguments.
534
535         Show a full description of the options and arguments to the
536         command in something like UNIX man page format. This includes
537
538           - a description of each option and argument, taken from the
539             __doc__ string for the option_handler method for
540             the option
541
542           - a description of what additional arguments will be processed,
543             taken from the arguments to main()
544
545         """
546         buffer = StringIO()
547
548         class_help_text = self._format_help_text(inspect.getdoc(self.__class__),
549                                                  '')
550         buffer.write(class_help_text)
551
552         buffer.write('\nSYNTAX:\n\n  ')
553         buffer.write(self.get_simple_syntax_help_string())
554
555         main_help_text = self._format_help_text(inspect.getdoc(self.main), '    ')
556         if main_help_text:
557             buffer.write('\n\nARGUMENTS:\n\n')
558             buffer.write(main_help_text)
559
560         buffer.write('\nOPTIONS:\n\n')
561
```

```
562         grouped_options = self._group_option_aliases()
563
564         # Describe all options, grouping aliases together
565         for names, options in grouped_options:
566             buffer.write('    %s\n' % self._get_option_identifier_text(options))
567
568             help = self._format_help_text(options[0].help, '        ')
569             buffer.write(help)
570
571         if self.EXAMPLES_DESCRIPTION:
572             buffer.write('EXAMPLES:\n\n')
573             buffer.write(self.EXAMPLES_DESCRIPTION)
574         return buffer.getvalue()
575
576
577 if __name__ == '__main__':
578     CommandLineApp().run()
```

The available and supported options are examined when the instance is initialized. By default, the contents of `sys.argv` are used as the options and arguments passed in from the command line to the program. It is easy to pass a different list of options when writing automated tests for your program, by passing a list of strings to `__init__()` as `command_line_options`. The options supported by the program are determined by scanning the class for option handler methods. No options are actually evaluated until `run()` is called.

When the program is run, the first thing it does is use **getopt** to validate the options it has been given. In `callGetopt()`, the arguments needed by **getopt** are constructed based on the option handlers discovered for the class. Options are processed in the order they are passed on the command line, and the option handler method for each option encountered is called. When an option handler requires an argument that is not provided on the command line, **getopt** detects the error. When an argument is provided, the option handler is responsible for determining whether the value is the correct type or otherwise valid. When the argument is not valid, the option handler can raise an exception with an error message to be printed for the user.

After all of the options are handled, the remaining arguments to the program are checked to be sure there are enough to satisfy the requirements, based on the argspec of the `main()` function. The number of arguments is checked explicitly to avoid having to handle a `TypeError` if the user does not pass the right number of arguments on the command line. If `CommandLineApp` depended on catching a `TypeError` when it passed too few arguments to `main()`, it could not tell the difference between a coding error and a user error. If a mistake inside `main()` caused a `TypeError` to occur, it might look like the user had passed an incorrect number of arguments to the program.

2.7 Error Handling

When an exception is raised during option processing or inside `main()`, the exception is caught by one of the `except` clauses and given to an error handling method. Subclasses can change the error handling behavior by overriding these methods.

`KeyboardInterrupt` exceptions are handled by calling `handleInterrupt()`. The default behavior is to print a message that the program has been interrupted and cause the program to exit with an error code. A subclass could override the method to clean up an in-progress task, background thread, or other operation which otherwise might not be automatically stopped when the `KeyboardInterrupt` is received.

When a lower level library tries to exit the program, `SystemExit` may be raised. `CommandLineApp` traps the `SystemExit` exception and exits normally, using the exit status taken from the exception. If the `force_exit` attribute of the application is false, `run()` returns instead of exiting. Trapping attempts to exit makes it easier to integrate `CommandLineApp` programs with `unittest` or other testing frameworks. The test can instantiate the

application, set `force_exit` to a false value, then run it. If any errors occur, a status code is returned but the test process does not exit.

All other types of exceptions are handled by calling `handleMainException()` and passing the exception as an argument. The default implementation of `handleMainException()` (lines 62-70) prints a simple error message based on the exception, unless debugging mode is turned on. Debugging mode prints the entire traceback for the exception.

```
$ csvcat file_does_not_exist.csv
ERROR: [Errno 2] No such file or directory:
'file_does_not_exist.csv'
```

2.8 Option Definitions

The standard library module **inspect** provides functions for performing introspection operations on classes and objects at runtime. The API supports basic querying and type checking so it is possible, for example, to get a list of the methods of a class, including all inherited methods.

`CommandLineApp.scan_for_options()` uses **inspect** to scan an application class for option handler methods. All of the methods of the class are retrieved with `inspect.getmembers()`, and those whose name starts with `option_handler_` are added to the list of supported options. Since most command line options use dashes instead of underscores, but method names cannot contain dashes, the underscores in the option handler method names are converted to dashes when creating the option name.

The `__init__()` method of the **OptionDef** class does all of the work of determining the command line switch name and what type of arguments the switch takes. The option handler method is examined with `inspect.getargspec()`, and the result is used to initialize the **OptionDef**.

An "argspec" for a function is a tuple made up of four values: a list of the names of all regular arguments to the function, including `self` if the function is a method; the name of the argument to receive the variable argument values, if any; the name of the argument to receive the keyword arguments, if any; and a list of the default values for the arguments, in they order they appear in the list of option names.

The argspecs for the option handlers in **csvcat** illustrate the variations of interest to **OptionDef**. First, `option_handler_skip_headers`:

```
>>> import Listing2
>>> import inspect
>>> print inspect.getargspec(
... Listing2.csvcat.option_handler_skip_headers)
(['self'], None, None, None)
```

Since the only positional argument to the method is `self`, and there is no variable argument name given, the option handler is treated as a simple command line switch without any arguments.

The `option_handler_dialect`, on the other hand, does include an additional argument:

```
>>> print inspect.getargspec(
... Listing2.csvcat.option_handler_dialect)
(['self', 'name'], None, None, None)
```

The `name` argument is listed in the argspec as a single regular argument. The result, when a program is run, is that while the options are being processed by `CommandLineApp` and **OptionDef**, the value for `name` is passed directly to the option handler method.

The `option_handler_columns` method illustrates variable argument handling:

```
>>> print inspect.getargspec(
... Listing2.csvcat.option_handler_columns)
(['self'], 'col', None, None)
```

The `col` argument from `option_handler_columns` is named in the argspec as the variable argument identifier. Since `option_handler_columns` accepts variable arguments, the **OptionDef** splits the argument value into a list of strings, and the list is passed to the option handler method using the variable argument syntax.

The other variable argument configuration, using unidentified keyword arguments, does not make sense for an option handler. The user of the command line program has no standard way to specify named arguments to options, so they are not supported by **OptionDef**.

2.9 Status Messages

In addition to command line option and argument parsing, and error handling, `CommandLineApp` provides a "status message" interface for giving varying levels of feedback to the user. Status messages are printed by calling `self.status_message()`. Each message must indicate the verbose level setting at which the message should be printed. If the current verbose level is at or higher than the desired level, the message is printed. Otherwise, it is ignored. The `-v`, `--verbose`, and `--quiet` flags let the user control the `verbose_level` setting for the application, and are defined in the `CommandLineApp` so that all subclasses inherit them.

2.9.1 Listing 4

```
#!/usr/bin/env python
# Illustrate verbose level controls.

import commandlineapp

class verbose_app(commandlineapp.CommandLineApp):
    "Demonstrate verbose level controls."

    def main(self):
        for i in range(1, 10):
            self.status_message('Level %d' % i, i)
        return 0

if __name__ == '__main__':
    verbose_app().run()
```

Listing 4 contains another sample application which uses `status_message()` to illustrate how the verbose level setting is applied. The default verbose level is 1, so when the program is run without any additional arguments only a single message is printed:

```
$ python Listing4.py
Level 1
$
```

The `--quiet` option silences all status messages by setting the verbose level to `0`:

```
$ python Listing4.py --quiet
$
```

Using the `-v` option increases the verbose setting, one level at a time. The option can be repeated on the command line:

```
$ python Listing4.py -v
Level 1
Level 2
$ python Listing4.py -vv
New verbose level is 3
Level 1
Level 2
Level 3
$
```

And the `--verbose` option sets the verbose level directly to the desired value:

```
$ python Listing4.py --verbose 4
New verbose level is 4
Level 1
Level 2
Level 3
Level 4
$
```

Error messages can be printed to the standard error stream using the `error_message()` method. The message is prefixed with the word "ERROR", and error messages are always printed, no matter what verbose level is set. Most programs will not need to use `errorMessage()` directly, because raising an exception is sufficient to have an error message displayed for the user.

2.10 CommandLineApp and Inheritance

When creating a suite of related programs, it is usually desirable for all of the programs to use the same options and, in many cases, share other common behavior. For example, when working with a database the connection and transaction must be managed reliably. Rather than re-implementing the same database handling code in each program, by using `CommandLineApp`, you can create an intermediate base class for your programs and share a single implementation. Listing 5 includes a skeleton base class called **SQLiteAppBase** for working with an `sqlite3` database in this way.

2.10.1 Listing 5

```
#!/usr/bin/env
# Base class for sqlite programs.

import sqlite3
import commandlineapp

class SQLiteAppBase(commandlineapp.CommandLineApp):
    """Base class for accessing sqlite databases.
    """

    dbname = 'sqlite.db'
    def optionHandler_db(self, name):
        """Specify the database filename.
        Defaults to 'sqlite.db'.
        """
        self.dbname = name
        return

    def main(self):
```

```
        # Subclasses can override this to control the arguments
        # used by the program.
        self.db_connection = sqlite3.connect(self.dbname)
        try:
            self.cursor = self.db_connection.cursor()
            exit_code = self.takeAction()
        except:
            # throw away changes
            self.db_connection.rollback()
            raise
        else:
            # save changes
            self.db_connection.commit()
        return exit_code

    def takeAction(self):
        """Override this in the actual application.
        Return the exit code for the application
        if no exception is raised.
        """
        raise NotImplementedError('Not implemented!')

if __name__ == '__main__':
    SQLiteAppBase().run()
```

SQLiteAppBase defines a single option handler for the `--db` option to let the user choose the database file. The default database is a file in the current directory called "sqlite.db". The `main()` method establishes a connection to the database, opens a cursor for working with the connection, then calls `takeAction()` to do the work. When `takeAction()` raises an exception, all database changes it may have made are discarded and the transaction is rolled back. When there is no error, the transaction is committed and the changes are saved.

2.10.2 Listing 6

```
#!/usr/bin/env python
# Initialize the database

import time
from Listing5 import SQLiteAppBase

class initdb(SQLiteAppBase):
    """Initialize a database.
    """

    def takeAction(self):
        self.statusMessage('Initializing database %s' % self.dbname)
        # Create the table
        self.cursor.execute("CREATE TABLE log (date text, message text)")
        # Log the actions taken
        self.cursor.execute(
            "INSERT INTO log (date, message) VALUES (?, ?)",
            (time.ctime(), 'Created database'))
        self.cursor.execute(
            "INSERT INTO log (date, message) VALUES (?, ?)",
            (time.ctime(), 'Created log table'))
        return 0

```

```
24 if __name__ == '__main__':
25     initdb().run()
```

A subclass of **SQLiteAppBase** can override `takeAction()` to do some actual work using the database connection and cursor created in `main()`. Listing 6 contains one such program, called `initdb`. In `initdb`, the `takeAction()` method creates a "log" table using the database cursor established in the base class. It then inserts two rows into the new table, using the same cursor. There is no need for `initdb` to commit the transaction, since the base class will do that after `takeAction()` returns without raising an exception.

```
$ python Listing6.py
Initializing database sqlite.db
```

2.10.3 Listing 7

```
1  #!/usr/bin/env python
2  # Initialize the database
3
4  from Listing5 import SQLiteAppBase
5
6  class showlog(SQLiteAppBase):
7      """Show the contents of the log.
8      """
9
10     substring = None
11     def optionHandler_message(self, substring):
12         """Look for messages with the substring.
13         """
14         self.substring = substring
15         return
16
17     def takeAction(self):
18         if self.substring:
19             pattern = '%' + self.substring + '%'
20             c = self.cursor.execute(
21                 "SELECT * FROM log WHERE message LIKE ?;",
22                 (pattern,))
23         else:
24             c = self.cursor.execute("SELECT * FROM log;")
25
26         for row in c:
27             print '%-30s %s' % row
28         return 0
29
30 if __name__ == '__main__':
31     showlog().run()
```

The `showlog` program in Listing 7 also uses **SQLiteAppBase**. It reads records from the log table and prints them out to the screen. When no options are given, it uses the cursor opened by the base class to find all of the records in the "log" table, and print them:

```
$ python Listing7.py
Sat Aug 25 19:09:41 2007       Created database
Sat Aug 25 19:09:41 2007       Created log table
```

The `--message` option to `showlog` can be used to filter the output to include only records whose message column matches the pattern given. When a message substring is specified, the select statement is altered to include only

messages containing the substring. In this example, only log messages with the word "table" in the message are printed:

```
$ python Listing7.py --message table
Sat Aug 25 19:09:41 2007       Created log table
```

The `updatelog` program in Listing 8 inserts new records into the database. Each time `updatelog` is called, the message passed on the command line is saved as an instance attribute by `main()` so it can be used later when a new row is inserted into the `log` table by `takeAction()`.

2.10.4 Listing 8

```
#!/usr/bin/env python
# Initialize the database

import time
from Listing5 import SQLiteAppBase

class updatelog(SQLiteAppBase):
    """Add to the contents of the log.
    """

    def main(self, message):
        """Provide the new message to add to the log.
        """
        # Save the message for use in takeAction()
        self.message = message
        return SQLiteAppBase.main(self)

    def takeAction(self):
        self.cursor.execute(
            "INSERT INTO log (date, message) VALUES (?, ?)",
            (time.ctime(), self.message))
        return 0

if __name__ == '__main__':
    updatelog().run()
```

```
$ python Listing8.py "another new message"
$ python Listing7.py
Sat Aug 25 19:09:41 2007       Created database
Sat Aug 25 19:09:41 2007       Created log table
Sat Aug 25 19:10:29 2007       another new message
```

As with `initdb`, because the base class commits changes to the database after `takeAction()` returns, `updatelog` does not need to manage the database connection in any way. Since all of the example programs use the database connection and cursor created by their base class, they could be updated to use a Postgresql or MySQL database by modifying the base class, without having to make those changes to each program separately.

2.11 Future Work

I have been using `CommandLineApp` in my own work for several years now, and continue to find ways to enhance it. The two primary features I would still like to add are the ability to print the help for a command in formats other than plain text, and automatic type conversion for arguments.

It is difficult to prepare attractive printed documentation from plain text help output like what is produced by the current version of `CommandLineApp`. Parsing the text output directly is not necessarily straightforward, since the embedded help may contain characters or patterns that would confuse a simple parser. A better solution is to use the option data gathered by introspection to generate output in a format such as DocBook, which could then be converted to PDF or HTML using other tool sets specifically designed for that purpose. There is a prototype of a program to create DocBook output from an application class, but it is not robust enough to be released - yet.

`CommandLineApp` is based on the older option parsing module, **getopt**, rather than the new **optparse**. This means it does not support some of the newer features available in **optparse**, such as type conversion for arguments. Type conversion could be added to `CommandLineApp` by inferring the types from default values for arguments. The **OptionDef** already discovers default values, but they are not used. The `OptionDef.invoke()` method needs to be updated to look at the default for an option before calling the option handler. If the default is a type object, it can be used to convert the incoming argument. If the default is a regular object, the type of the object can be determined using `type()`. Then, once the type is known, the argument can be converted.

2.11.1 Conclusion

I hope this article encourages you to think about your command line programs in a different light, and to treat them as first class objects. Using inheritance to share code is so common in other areas of development that it is hardly given a second thought in most cases. As has been shown with the **SQLiteAppBase** programs, the same technique can be just as powerful when applied to building command line programs, saving development time and testing effort as a result. `CommandLineApp` has been used as the foundation for dozens of types of programs, and could be just what you need the next time you have to write a new command line program.

CHAPTER

THREE

HISTORY

3.0.7

- Repackage the documentation

3.0.6

- Bug fix from Cezary Statkiewicz for handling default arguments.

3.0.5

- Fixed packaging problems that prevented installation with easy_install and pip.

3.0.4

- Switched to sphinx for documentation.

3.0.3

- Updated the build to work with Mercurial and migrated the source to bitbucket host. No code changes.

3.0.2

- source file encoding patch from Ben Finney

3.0.1

- replace the test script missing from the 3.0 release

3.0

- Ben Finney provided a patch to convert the names of the module, method, etc. to be PEP8-compliant. Thanks, Ben!

These changes are obviously backwards incompatible.

2.6

- Add initialization hooks to make application setup easier without overriding `__init__()`.

2.5

- Updated to handle Unicode status messages more reliably.

2.4

- Code clean up and error handling changes.

2.3

- Refine help output a little more.

2.2

- Handle missing docstrings for main() and the class.

2.1

- Add automatic detection and validation of main function arguments, including help text generation. Also includes the main function docstring in `--help` output.

2.0

- Substantial rewrite using inspect and with modified API.

1.0

- This is the old version, which was developed with and works under Python 1.5.4-2.5.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*