termtool Documentation

Release 1.0

Mark Paschal

March 27, 2013

CONTENTS

termtool helps you write subcommand-based command line tools in Python.

CHAPTER

ONE

TERMTOOL GUIDE

termtool helps you write subcommand-based command line tools in Python. It collects several Python libraries into a declarative syntax:

- `argparse`, the argument parsing module with subcommand support provided in the standard library in Python 2.7 and later.
- prettytable, an easy module for building tables of information.
- progressbar, a handy module for displaying progress bars.
- `logging`, the simple built-in module for logging messages.

1.1 Making your script

Each `termtool` script is defined as a `Termtool` subclass. Each subcommand in the script is an instance method marked with the `subcommand()` decorator.

A script like this:

```
#!/usr/bin/env python

from termtool import Termtool, subcommand, argument


class Example(Termtool):

    description = 'A script that frobs or displays bazzes'

    @subcommand(help='frobs a baz')
    @argument('baz', help='the baz to frob')
    def frob(self, args):
        # do the work to frob a baz
        pass

    @subcommand(help='displays a baz')
    @argument('baz', help='the baz to display')
    @argument('--csv', action='store_true', help='sets display mode to CSV')
    def display(self, args):
        # display the baz
        pass
```

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

creates a command like this:

```
$ example --help
usage: example [-h] [-v] [-q] COMMAND ...

A script that frobs or displays bazzes

optional arguments:
  -h, --help  show this help message and exit
  -v          be more verbose (stackable)
  -q          be less verbose (stackable)

subcommands:
  COMMAND
    frob      frobs a baz
    display   displays a baz

$ example display --help
usage: example.py display [-h] [--csv] baz

positional arguments:
  baz         the baz to display

optional arguments:
  -h, --help  show this help message and exit
  --csv       sets display mode to CSV

$
```

The arguments to the `termtool.subcommand()` decorator describe the subcommand itself. Subcommands are created using argparse subcommands, so any argument you can pass to the `ArgumentParser` constructor is valid for `subcommand()`.

Arguments themselves are declared with the `termtool.argument()` decorator. Subcommand arguments are declared with `ArgumentParser.add_argument`, so all its arguments are valid for the `argument()` decorator.

Even though decorators are evaluated closest-first, arguments are added in the order they appear in your source file (that is, in reverse order of how they evaluate). Declare positional arguments in reading order, first argument first.

Arguments that should be available in general to all commands can be specified as class decorators (in scripts for Python 2.6 and greater):

```
@argument('baz', help='the baz to frob or display')
class Example(Termtool):

    description = 'A script that frobs or displays bazzes'
```

or by invoking `argument()` afterward, for compatibility with Python 2.5:

```
class Example(Termtool):

    description = 'A script that frobs or displays bazzes'

    ...

argument('baz', help='the baz to frob or display')(Example)
```

1.2 Logging

`termtool` tools provide automatic support for configuring the `logging` module. Log messages are formatted simply with the level and the message, and are printed to standard error.

People using your tool can use the *-v* and *-q* arguments to change the log level. By default, messages at *WARN* and lower logging levels are displayed. Each *-v* argument adds one more verbose level of logging, and each *-q* argument removes one level, down to *CRITICAL* level. Critical errors are always displayed.

For example, given the command:

```
@subcommand()
def loglevel(self, args):
    logging.critical('critical')
    logging.error('error')
    logging.warn('warning')
    logging.info('info')
    logging.debug('debug')
```

you would see output such as:

```
$ example loglevel
CRITICAL: critical
ERROR: error
WARNING: warning

$ example -v -v loglevel
INFO: Set log level to DEBUG
CRITICAL: critical
ERROR: error
WARNING: warning
INFO: info
DEBUG: debug

$ example -q -q loglevel
CRITICAL: critical

$ example -qqqqq loglevel
CRITICAL: critical

$ example -qqqqqvvvvvqvqvqqv loglevel
CRITICAL: critical
ERROR: error
```

1.3 Displaying tables

`termtool` tools can present information to people using them in tables for easy reading. Tables can be created using the `table()` callable on `termtool.Termtool` instances.

The `table()` callable is really the `prettytable.PrettyTable` class, so all arguments to the `prettytable.PrettyTable` constructor are valid arguments to the `table()` callable.

```
@subcommand(help='display the songs')
def display(self, args):
    song = self.get_songs()

    table = self.table(['ID', 'Title', 'Artist', 'Album'])
```

```
    for song in songs:
        table.add_row([song.id, song.title, song.artist, song.album])
    print table
```

1.4 Displaying progress bars

`termtool` tools can show people using them when they're busy performing long or multistep operations with a progress bar. Use the `progressbar()` callable on `termtool.Termtool` instances to create one.

The `progressbar()` callable is really the `progressbar.ProgressBar` class, so all arguments to the `progressbar.ProgressBar` constructor are valid arguments to the `progressbar()` callable.

```
@subcommand(help='upload the files')
def upload(self, args):
    files = self.get_files()

    progress = self.progressbar()
    for somefile in progress(files):
        somefile.upload()
```

1.5 Configuration files

`termtool` tools automatically load options from "rc" style configuration files.

The tool will look for a configuration file in the user's home directory, named after the tool's class. Configuration files are simply command line elements separated each on one line. That is, each argument element that would be separated by spaces should be on a separate line; specifically, arguments that take values should be on separate lines from their values. Because configuration files are always loaded, only command-level arguments that are valid for all subcommands should be added.

For example, for a tool declared as:

```
@argument('--consumer-key', help='the API consumer key')
@argument('--consumer-secret', help='the API consumer secret')
@argument('--access-token', help='the API access token')
class Example(Termtool):
    ...
```

a configuration file specifying these API tokens would be a file named *~/.example* that contains:

```
--consumer-token
b5e53e6601cbdcc02b24
--consumer-secret
a8e5df863e
--access-token
uo9lctpryiscvujgab0cvns860xlg3
```

If your tool has specific arguments you may want people using it to save for later, you can use `write_config_file()` in another command (such as *configure*) to write one out. Pass all the arguments you'd like to write out, and `Termtool` will overwrite the config file with the new settings. The file is created with umask 077 so that it's readable only by the owner.

```
@subcommand()
def configure(self, args):
    if not args.access_token:
```

```
        args.access_token = self.request_access_token(args)

    self.write_config_file(
        '--consumer-key', args.consumer_key,
        '--consumer-secret', args.consumer_secret,
        '--access-token', args.access_token,
    )

    logging.info("Configured!")
```

CHAPTER

TWO

TERMTOOL API REFERENCE

termtool.**subcommand**([*help*], ***kwargs*)
: A decorator (that is, *returns* a decorator) to mark a function as a `Termtool` subcommand. Arguments are passed to the `argparse.ArgumentParser` constructor, so any of its arguments are valid keyword arguments.

termtool.**argument**(*name or flags...*[, *help*], ***kwargs*)
: A decorator (that is, *returns* a decorator) to declare an argument of a subcommand method or command class. Arguments are passed to the `argparse.ArgumentParser.add_argument()` method of the command's `argparse.ArgumentParser` instance, so any of its arguments are valid.

class termtool.**Termtool**
: Creates a new `Termtool` instance. Make your own command line tool by subclassing this class and defining new subcommands with the `subcommand()` decorator.

  table([*field_names*], ***kwargs*)
  : Returns a new `prettytable.PrettyTable` instance.

    Any arguments for the `prettytable.PrettyTable` constructor are valid arguments to `table()`. See the prettytable documentation for more information.

  progressbar([*max_val*], ***kwargs*)
  : Returns a new `progressbar.ProgressBar` instance.

    Any arguments for the `progressbar.ProgressBar` constructor are valid arguments to `progressbar()`. See the progressbar documentation for more information.

  read_config_file()
  : Reads any additional arguments saved in the user's configuration file for the tool and returns them as a list.

    Configuration files are files in the user's home directory named *.toolname* where *toolname* is the name of the tool class in lower case. The file if present should contain arguments one per line.

  write_config_file(*config_args*)
  : Replaces the user's configuration file with the arguments present in *config_args*.

    This method will overwrite any unexpected changes the user has made to the configuration file, so only use it in response to an explicit instruction by the user, such as in a `configure` command. If the file is created, it is created with umask 077 so that it is neither group nor world readable.

  main(*argv*)
  : Invokes the tool with the specified command line arguments, returning the appropriate exit code.

    `main()` first reads the "rc" style configuration file, prepending its arguments to *argv* before the other arguments. The arguments are then parsed and the `logging` module is first configured. `main()` then dispatches to the instance method matching the subcommand specified by the first positional argument in *argv*.

run()

Invokes the tool as run from a script.

Command line arguments are read from `sys.argv`. When the tool's run is complete, `run()` exits the interpreter using `sys.exit()` with an appropriate exit code (*0* if the run completed normally and a non-zero value otherwise) when complete.

Use this method in your `if __name__ == '__main__'` block.

CHAPTER

THREE

INDICES AND TABLES

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

PYTHON MODULE INDEX

t