cmd2 Documentation

Release 0.9

Catherine Devlin and Todd Leonhardt

Jun 15, 2019

Contents

A python package for building powerful command-line interpreter (CLI) programs. Extends the Python Standard Library's cmd package.

The basic use of `cmd2` is identical to that of cmd.

1. Create a subclass of `cmd2.Cmd`. Define attributes and `do_*` methods to control its behavior. Throughout this documentation, we will assume that you are naming your subclass `App`:

   ```
   from cmd2 import Cmd
   class App(Cmd):
       # customized attributes and methods here
   ```

2. Instantiate `App` and start the command loop:

   ```
   app = App()
   app.cmdloop()
   ```

Note: The tab-completion feature provided by cmd relies on underlying capability provided by GNU readline or an equivalent library. Linux distros will almost always come with the required library installed. For macOS, we recommend using the gnureadline Python module which includes a statically linked version of GNU readline. Alternatively on macOS the `conda` package manager that comes with the Anaconda Python distro can be used to install `readline` (preferably from conda-forge) or the Homebrew package manager can be used to to install the `readline` package. For Windows, we recommend installing the pyreadline Python module.

CHAPTER 1

Resources

- cmd
- cmd2 project page
- project bug tracker
- Florida PyCon 2017: slides, video

These docs will refer to `App` as your `cmd2.Cmd` subclass, and `app` as an instance of `App`. Of course, in your program, you may name them whatever you want.

Contents:

1.1 Installation Instructions

This section covers the basics of how to install, upgrade, and uninstall `cmd2`.

1.1.1 Installing

First you need to make sure you have Python 3.4+, pip, and setuptools. Then you can just use pip to install from PyPI.

Note: Depending on how and where you have installed Python on your system and on what OS you are using, you may need to have administrator or root privileges to install Python packages. If this is the case, take the necessary steps required to run the commands in this section as root/admin, e.g.: on most Linux or Mac systems, you can precede them with `sudo`:

```
sudo pip install <package_name>
```

Requirements for Installing

- If you have Python 3 >=3.4 installed from python.org, you will already have pip and setuptools, but may need to upgrade to the latest versions:

  On Linux or OS X:

  ```
  pip install -U pip setuptools
  ```

  On Windows:

  ```
  python -m pip install -U pip setuptools
  ```

Use pip for Installing

pip is the recommended installer. Installing packages from PyPI with pip is easy:

```
pip install cmd2
```

This should also install the required 3rd-party dependencies, if necessary.

Install from GitHub using pip

The latest version of `cmd2` can be installed directly from the master branch on GitHub using pip:

```
pip install -U git+git://github.com/python-cmd2/cmd2.git
```

This should also install the required 3rd-party dependencies, if necessary.

Install from Debian or Ubuntu repos

We recommend installing from pip, but if you wish to install from Debian or Ubuntu repos this can be done with apt-get.

For Python 3:

```
sudo apt-get install python3-cmd2
```

This will also install the required 3rd-party dependencies.

Warning: Versions of `cmd2` before 0.7.0 should be considered to be of unstable "beta" quality and should not be relied upon for production use. If you cannot get a version >= 0.7 from your OS repository, then we recommend installing from either pip or GitHub - see *Use pip for Installing* or *Install from GitHub using pip*.

Deploy cmd2.py with your project

`cmd2` is contained in a small number of Python files, which can be easily copied into your project. *The copyright and license notice must be retained.*

This is an option suitable for advanced Python users. You can simply include the files within your project's hierarchy. If you want to modify `cmd2`, this may be a reasonable option. Though, we encourage you to use stock `cmd2` and either composition or inheritance to achieve the same goal.

This approach will obviously NOT automatically install the required 3rd-party dependencies, so you need to make sure the following Python packages are installed:

- attrs
- colorama
- pyperclip
- wcwidth

On Windows, there is an additional dependency:

- pyreadline

1.1.2 Upgrading cmd2

Upgrade an already installed `cmd2` to the latest version from PyPI:

```
pip install -U cmd2
```

This will upgrade to the newest stable version of `cmd2` and will also upgrade any dependencies if necessary.

1.1.3 Uninstalling cmd2

If you wish to permanently uninstall `cmd2`, this can also easily be done with pip:

```
pip uninstall cmd2
```

1.1.4 Extra requirements for Python 3.4

`cmd2` requires the `contextlib2` module for Python 3.4. This is used to temporarily redirect stdout and stderr. Also when using Python 3.4, `cmd2` requires the `typing` module backport.

1.2 Extra requirement for macOS

macOS comes with the libedit library which is similar, but not identical, to GNU Readline. Tab-completion for `cmd2` applications is only tested against GNU Readline.

There are several ways GNU Readline can be installed within a Python environment on a Mac, detailed in the following subsections.

1.2.1 gnureadline Python module

Install the gnureadline Python module which is statically linked against a specific compatible version of GNU Readline:

```
pip install -U gnureadline
```

1.2.2 readline via conda

Install the **readline** package using the conda package manager included with the Anaconda Python distribution:

```
conda install readline
```

1.2.3 readline via brew

Install the **readline** package using the Homebrew package manager (compiles from source):

```
brew install openssl
brew install pyenv
brew install readline
```

Then use pyenv to compile Python and link against the installed readline

1.3 Overview

`cmd2` is an extension of cmd, the Python Standard Library's module for creating simple interactive command-line applications.

`cmd2` can be used as a drop-in replacement for cmd. Simply importing `cmd2` in place of cmd will add many features to an application without any further modifications.

Understanding the use of cmd is the first step in learning the use of `cmd2`. Once you have read the cmd docs, return here to learn the ways that `cmd2` differs from cmd.

Note: `cmd2` is not quite a drop-in replacement for cmd. The cmd.emptyline() function is called when an empty line is entered in response to the prompt. By default, in cmd if this method is not overridden, it repeats and executes the last nonempty command entered. However, no end user we have encountered views this as expected or desirable default behavior. Thus, the default behavior in `cmd2` is to simply go to the next line and issue the prompt again. At this time, cmd2 completely ignores empty lines and the base class cmd.emptyline() method never gets called and thus the emptyline() behavior cannot be overridden.

1.4 Features requiring no modifications

These features are provided "for free" to a cmd-based application simply by replacing `import cmd` with `import cmd2 as cmd`.

1.4.1 Script files

Text files can serve as scripts for your `cmd2`-based application, with the `load`, `_relative_load`, and `edit` commands.

Both ASCII and UTF-8 encoded unicode text files are supported.

Simply include one command per line, typed exactly as you would inside a `cmd2` application.

The `load` command loads commands from a script file into a queue and then the normal cmd2 REPL resumes control and executes the commands in the queue in FIFO order. A side effect of this is that if you redirect/pipe the output of a load command, it will redirect the output of the `load` command itself, but will NOT redirect the output of the command loaded from the script file. Of course, you can add redirection to the commands being run in the script file, e.g.:

```
# This is your script file
command arg1 arg2 > file.txt
```

`Cmd.`**`do_load`**(*args: argparse.Namespace*) → Optional[bool]

Run commands in script file that is encoded as either ASCII or UTF-8 text

Script should contain one command per line, just like the command would be typed in the console.

If the -r/–record_transcript flag is used, this command instead records the output of the script commands to a transcript for testing purposes.

`Cmd.`**`do__relative_load`**(*args: argparse.Namespace*) → Optional[bool]

Run commands in script file that is encoded as either ASCII or UTF-8 text

Script should contain one command per line, just like the command would be typed in the console.

If the -r/–record_transcript flag is used, this command instead records the output of the script commands to a transcript for testing purposes.

If this is called from within an already-running script, the filename will be interpreted relative to the already-running script's directory.

`Cmd.`**`do_edit`**(*args: argparse.Namespace*) → None

Edit a file in a text editor

The editor used is determined by a settable parameter. To set it:

set editor (program-name)

1.4.2 Comments

Any command line input where the first non-whitespace character is a # will be treated as a comment. This means any # character appearing later in the command will be treated as a literal. The same applies to a # in the middle of a multiline command, even if it is the first character on a line.

Comments can be useful in *Script files*, but would be pointless within an interactive session.

```
(Cmd) # this is a comment
(Cmd) this # is not a comment
```

1.4.3 Startup Initialization Script

You can load and execute commands from a startup initialization script by passing a file path to the `startup_script` argument to the `cmd2.Cmd.__init__()` method like so:

```
class StartupApp(cmd2.Cmd):
    def __init__(self):
        cmd2.Cmd.__init__(self, startup_script='.cmd2rc')
```

See the AliasStartup example for a demonstration.

1.4.4 Commands at invocation

You can send commands to your app as you invoke it by including them as extra arguments to the program. `cmd2` interprets each argument as a separate command, so you should enclose each command in quotation marks if it is more than a one-word command.

```
cat@eee:~/proj/cmd2/example$ python example.py "say hello" "say Gracie" quit
hello
Gracie
cat@eee:~/proj/cmd2/example$
```

Note: If you wish to disable cmd2's consumption of command-line arguments, you can do so by setting the `allow_cli_args` argument of your `cmd2.Cmd` class instance to `False`. This would be useful, for example, if you wish to use something like Argparse to parse the overall command line arguments for your application:

```
from cmd2 import Cmd
class App(Cmd):
    def __init__(self):
        super().__init__(allow_cli_args=False)
```

1.4.5 Output redirection

As in a Unix shell, output of a command can be redirected:

- sent to a file with `>`, as in `mycommand args > filename.txt`
- appended to a file with `>>`, as in `mycommand args >> filename.txt`
- piped (`|`) as input to operating-system commands, as in `mycommand args | wc`
- sent to the operating system paste buffer, by ending with a bare `>`, as in `mycommand args >`. You can even append output to the current contents of the paste buffer by ending your command with `>>`.

Note: If you wish to disable cmd2's output redirection and pipes features, you can do so by setting the `allow_redirection` attribute of your `cmd2.Cmd` class instance to `False`. This would be useful, for example, if you want to restrict the ability for an end user to write to disk or interact with shell commands for security reasons:

```
from cmd2 import Cmd
class App(Cmd):
    def __init__(self):
        self.allow_redirection = False
```

cmd2's parser will still treat the >, >>, and | symbols as output redirection and pipe symbols and will strip arguments after them from the command line arguments accordingly. But output from a command will not be redirected to a file or piped to a shell command.

If you need to include any of these redirection characters in your command, you can enclose them in quotation marks, `mycommand 'with > in the argument'`.

1.4.6 Python

The `py` command will run its arguments as a Python command. Entered without arguments, it enters an interactive Python session. The session can call "back" to your application through the name defined in `self.pyscript_name` (defaults to `app`). This wrapper provides access to execute commands in your cmd2 application while maintaining isolation.

You may optionally enable full access to to your application by setting `locals_in_py` to `True`. Enabling this flag adds `self` to the python session, which is a reference to your Cmd2 application. This can be useful for debugging your application. To prevent users from enabling this ability manually you'll need to remove `locals_in_py` from the `settable` dictionary.

The `app` object (or your custom name) provides access to application commands through raw commands. For example, any application command call be called with `app("<command>")`.

```
>>> app('say --piglatin Blah')
lahBay
```

More Python examples:

```
(Cmd) py print("-".join("spelling"))
s-p-e-l-l-i-n-g
(Cmd) py
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170118] on linux
Type "help", "copyright", "credits" or "license" for more information.
(CmdLineApp)

End with `Ctrl-D` (Unix) / `Ctrl-Z` (Windows), `quit()`, `exit()`.
Non-python commands can be issued with: app("your command")
Run python code from external script files with: run("script.py")

>>> import os
>>> os.uname()
('Linux', 'eee', '2.6.31-19-generic', '#56-Ubuntu SMP Thu Jan 28 01:26:53 UTC 2010',
↪'i686')
>>> app("say --piglatin {os}".format(os=os.uname()[0]))
inuxLay
>>> self.prompt
'(Cmd) '
>>> self.prompt = 'Python was here > '
>>> quit()
Python was here >
```

Using the `py` command is tightly integrated with your main `cmd2` application and any variables created or changed will persist for the life of the application:

```
(Cmd) py x = 5
(Cmd) py print(x)
5
```

The `py` command also allows you to run Python scripts via `py run('myscript.py')`. This provides a more complicated and more powerful scripting capability than that provided by the simple text file scripts discussed in *Script files*. Python scripts can include conditional control flow logic. See the **python_scripting.py** `cmd2` application and the **script_conditional.py** script in the `examples` source code directory for an example of how to achieve this in your own applications.

Using `py` to run scripts directly is considered deprecated. The newer `pyscript` command is superior for doing this in two primary ways:

- it supports tab-completion of file system paths
- it has the ability to pass command-line arguments to the scripts invoked

There are no disadvantages to using `pyscript` as opposed to `py run()`. A simple example of using `pyscript` is shown below along with the arg_printer script:

```
(Cmd) pyscript examples/scripts/arg_printer.py foo bar baz
Running Python script 'arg_printer.py' which was called with 3 arguments
arg 1: 'foo'
arg 2: 'bar'
arg 3: 'baz'
```

Note: If you want to be able to pass arguments with spaces to commands, then we strongly recommend using one of the decorators, such as `with_argument_list`. `cmd2` will pass your **do_*** methods a list of arguments in this case.

When using this decorator, you can then put arguments in quotes like so:

```
$ examples/arg_print.py
(Cmd) lprint foo "bar baz"
lprint was called with the following list of arguments: ['foo', 'bar baz']
```

1.4.7 IPython (optional)

If IPython is installed on the system **and** the `cmd2.Cmd` class is instantiated with `use_ipython=True`, then the optional `ipy` command will be present:

```
from cmd2 import Cmd
class App(Cmd):
    def __init__(self):
        Cmd.__init__(self, use_ipython=True)
```

The `ipy` command enters an interactive IPython session. Similar to an interactive Python session, this shell can access your application instance via `self` and any changes to your application made via `self` will persist. However, any local or global variable created within the `ipy` shell will not persist. Within the `ipy` shell, you cannot call "back" to your application with `cmd("")`, however you can run commands directly like so:

```
self.onecmd_plus_hooks('help')
```

IPython provides many advantages, including:

- Comprehensive object introspection
- Get help on objects with `?`
- Extensible tab completion, with support by default for completion of python variables and keywords

The object introspection and tab completion make IPython particularly efficient for debugging as well as for interactive experimentation and data analysis.

1.4.8 Searchable command history

All cmd-based applications have access to previous commands with the up- and down- arrow keys.

All cmd-based applications on systems with the `readline` module also provide Readline Emacs editing mode. With this you can, for example, use **Ctrl-r** to search backward through the readline history.

`cmd2` adds the option of making this history persistent via optional arguments to `cmd2.Cmd.__init__()`:

`Cmd.`**`__init__`**(*completekey: str = 'tab'*, *stdin=None*, *stdout=None*, ***, *persistent_history_file: str = ''*, *persistent_history_length: int = 1000*, *startup_script: Optional[str] = None*, *use_ipython: bool = False*, *allow_cli_args: bool = True*, *transcript_files: Optional[List[str]] = None*, *allow_redirection: bool = True*, *multiline_commands: Optional[List[str]] = None*, *terminators: Optional[List[str]] = None*, *shortcuts: Optional[Dict[str, str]] = None*) → None

An easy but powerful framework for writing line-oriented command interpreters, extends Python's cmd package.

Parameters

- **`completekey`** – (optional) readline name of a completion key, default to Tab
- **`stdin`** – (optional) alternate input file object, if not specified, sys.stdin is used
- **`stdout`** – (optional) alternate output file object, if not specified, sys.stdout is used
- **`persistent_history_file`** – (optional) file path to load a persistent cmd2 command history from
- **`persistent_history_length`** – (optional) max number of history items to write to the persistent history file
- **`startup_script`** – (optional) file path to a a script to load and execute at startup
- **`use_ipython`** – (optional) should the "ipy" command be included for an embedded IPython shell
- **`allow_cli_args`** – (optional) if True, then cmd2 will process command line arguments as either commands to be run or, if -t is specified, transcript files to run. This should be set to False if your application parses its own arguments.
- **`transcript_files`** – (optional) allows running transcript tests when allow_cli_args is False
- **`allow_redirection`** – (optional) should output redirection and pipes be allowed
- **`multiline_commands`** – (optional) list of commands allowed to accept multi-line input
- **`shortcuts`** – (optional) dictionary containing shortcuts for commands

`cmd2` makes a third type of history access available with the `history` command. Each time the user enters a command, `cmd2` saves the input. The `history` command lets you do interesting things with that saved input. The examples to follow all assume that you have entered the following commands:

```
(Cmd) alias create one !echo one
Alias 'one' created
(Cmd) alias create two !echo two
Alias 'two' created
(Cmd) alias create three !echo three
Alias 'three' created
(Cmd) alias create four !echo four
Alias 'four' created
```

In it's simplest form, the `history` command displays previously entered commands. With no additional arguments, it displays all previously entered commands:

```
(Cmd) history
    1  alias create one !echo one
    2  alias create two !echo two
```

(continues on next page)

(continued from previous page)

```
    3  alias create three !echo three
    4  alias create four !echo four
```

If you give a positive integer as an argument, then it only displays the specified command:

```
(Cmd) history 4
    4  alias create four !echo four
```

If you give a negative integer *N* as an argument, then it display the *Nth* last command. For example, if you give `-1` it will display the last command you entered. If you give `-2` it will display the next to last command you entered, and so forth:

```
(Cmd) history -2
    3  alias create three !echo three
```

You can use a similar mechanism to display a range of commands. Simply give two command numbers separated by `..` or `:`, and you will see all commands between, and including, those two numbers:

```
(Cmd) history 1:3
    1  alias create one !echo one
    2  alias create two !echo two
    3  alias create three !echo three
```

If you omit the first number, it will start at the beginning. If you omit the last number, it will continue to the end:

```
(Cmd) history :2
    1  alias create one !echo one
    2  alias create two !echo two
(Cmd) history 2:
    2  alias create two !echo two
    3  alias create three !echo three
    4  alias create four !echo four
```

If you want to display the last three commands entered:

```
(Cmd) history -- -3:
    2  alias create two !echo two
    3  alias create three !echo three
    4  alias create four !echo four
```

Notice the double dashes. These are required because the history command uses `argparse` to parse the command line arguments. As described in the argparse documentation , `-3:` is an option, not an argument:

> If you have positional arguments that must begin with - and don't look like negative numbers, you can insert the pseudo-argument '–' which tells parse_args() that everything after that is a positional argument:

There is no zeroth command, so don't ask for it. If you are a python programmer, you've probably noticed this looks a lot like the slice syntax for lists and arrays. It is, with the exception that the first history command is 1, where the first element in a python array is 0.

Besides selecting previous commands by number, you can also search for them. You can use a simple string search:

```
(Cmd) history two
    2  alias create two !echo two
```

Or a regular expression search by enclosing your regex in slashes:

```
(Cmd) history '/te\ +th/'
    3  alias create three !echo three
```

If your regular expression contains any characters that `argparse` finds interesting, like dash or plus, you also need to enclose your regular expression in quotation marks.

This all sounds great, but doesn't it seem like a bit of overkill to have all these ways to select commands if all we can do is display them? Turns out, displaying history commands is just the beginning. The history command can perform many other actions:

- running previously entered commands
- saving previously entered commands to a text file
- opening previously entered commands in your favorite text editor
- running previously entered commands, saving the commands and their output to a text file
- clearing the history of entered commands

Each of these actions is invoked using a command line option. The `-r` or `--run` option runs one or more previously entered commands. To run command number 1:

```
(Cmd) history --run 1
```

To rerun the last two commands (there's that double dash again to make argparse stop looking for options):

```
(Cmd) history -r -- -2:
```

Say you want to re-run some previously entered commands, but you would really like to make a few changes to them before doing so. When you use the `-e` or `--edit` option, `history` will write the selected commands out to a text file, and open that file with a text editor. You make whatever changes, additions, or deletions, you want. When you leave the text editor, all the commands in the file are executed. To edit and then re-run commands 2-4 you would:

```
(Cmd) history --edit 2:4
```

If you want to save the commands to a text file, but not edit and re-run them, use the `-o` or `--output-file` option. This is a great way to create *Script files*, which can be loaded and executed using the `load` command. To save the first 5 commands entered in this session to a text file:

```
(Cmd) history :5 -o history.txt
```

The `history` command can also save both the commands and their output to a text file. This is called a transcript. See *Transcript based testing* for more information on how transcripts work, and what you can use them for. To create a transcript use the `-t` or `--transcription` option:

```
(Cmd) history 2:3 --transcript transcript.txt
```

The `--transcript` option implies `--run`: the commands must be re-run in order to capture their output to the transcript file.

The last action the history command can perform is to clear the command history using `-c` or `--clear`:

```
(Cmd) history -c
```

In addition to these five actions, the `history` command also has some options to control how the output is formatted. With no arguments, the `history` command displays the command number before each command. This is great when displaying history to the screen because it gives you an easy reference to identify previously entered commands. However, when creating a script or a transcript, the command numbers would prevent the script from loading properly. The

-s or --script option instructs the history command to suppress the line numbers. This option is automatically set by the --output-file, --transcript, and --edit options. If you want to output the history commands with line numbers to a file, you can do it with output redirection:

```
(Cmd) history 1:4 > history.txt
```

You might use -s or --script on it's own if you want to display history commands to the screen without line numbers, so you can copy them to the clipboard:

```
(Cmd) history -s 1:3
```

cmd2 supports both aliases and macros, which allow you to substitute a short, more convenient input string with a longer replacement string. Say we create an alias like this, and then use it:

```
(Cmd) alias create ls shell ls -aF
Alias 'ls' created
(Cmd) ls -d h*
history.txt     htmlcov/
```

By default, the history command shows exactly what we typed:

```
(Cmd) history
    1  alias create ls shell ls -aF
    2  ls -d h*
```

There are two ways to modify that display so you can see what aliases and macros were expanded to. The first is to use -x or --expanded. These options show the expanded command instead of the entered command:

```
(Cmd) history -x
    1  alias create ls shell ls -aF
    2  shell ls -aF -d h*
```

If you want to see both the entered command and the expanded command, use the -v or --verbose option:

```
(Cmd) history -v
    1  alias create ls shell ls -aF
    2  ls -d h*
    2x shell ls -aF -d h*
```

If the entered command had no expansion, it is displayed as usual. However, if there is some change as the result of expanding macros and aliases, then the entered command is displayed with the number, and the expanded command is displayed with the number followed by an x.

1.4.9 Quitting the application

cmd2 pre-defines a quit command for you. It's trivial, but it's one less thing for you to remember.

1.4.10 Misc. pre-defined commands

Several generically useful commands are defined with automatically included do_ methods.

Cmd.**do_quit**(*_: argparse.Namespace*) → bool
Exit this application

Cmd.**do_shell**(*args: argparse.Namespace*) → None
Execute a command as if at the OS prompt

(! is a shortcut for `shell`; thus `!ls` is equivalent to `shell ls`.)

1.4.11 Transcript-based testing

A transcript is both the input and output of a successful session of a `cmd2`-based app which is saved to a text file. The transcript can be played back into the app as a unit test.

```
$ python example.py --test transcript_regex.txt
.
----------------------------------------------------------------------
Ran 1 test in 0.013s

OK
```

See *Transcript based testing* for more details.

1.4.12 Tab-Completion

`cmd2` adds tab-completion of file system paths for all built-in commands where it makes sense, including:

- `edit`
- `load`
- `pyscript`
- `shell`

`cmd2` also adds tab-completion of shell commands to the `shell` command.

Additionally, it is trivial to add identical file system path completion to your own custom commands. Suppose you have defined a custom command `foo` by implementing the `do_foo` method. To enable path completion for the `foo` command, then add a line of code similar to the following to your class which inherits from `cmd2.Cmd`:

```
complete_foo = self.path_complete
```

This will effectively define the `complete_foo` readline completer method in your class and make it utilize the same path completion logic as the built-in commands.

The built-in logic allows for a few more advanced path completion capabilities, such as cases where you only want to match directories. Suppose you have a custom command `bar` implemented by the `do_bar` method. You can enable path completion of directories only for this command by adding a line of code similar to the following to your class which inherits from `cmd2.Cmd`:

```
# Make sure you have an "import functools" somewhere at the top
complete_bar = functools.partialmethod(cmd2.Cmd.path_complete, path_filter=os.path.
↪isdir)
```

1.5 Features requiring only parameter changes

Several aspects of a `cmd2` application's behavior can be controlled simply by setting attributes of `App`. A parameter can also be changed at runtime by the user *if* its name is included in the dictionary `app.settable`. (To define your own user-settable parameters, see *Other user-settable parameters*)

1.5.1 Shortcuts

Command shortcuts for long command names and common commands can make life more convenient for your users. Shortcuts are used without a space separating them from their arguments, like `!ls`. By default, the following shortcuts are defined:

> **?** help
>
> **!** shell: run as OS-level command
>
> **@** load script file
>
> **@@** load script file; filename is relative to current script location

To define more shortcuts, update the dict `App.shortcuts` with the {'shortcut': 'command_name'} (omit `do_`):

```python
class App(Cmd2):
    def __init__(self):
      shortcuts = dict(self.DEFAULT_SHORTCUTS)
      shortcuts.update({'*': 'sneeze', '~': 'squirm'})
      cmd2.Cmd.__init__(self, shortcuts=shortcuts)
```

Warning: Shortcuts need to be created by updating the `shortcuts` dictionary attribute prior to calling the `cmd2.Cmd` super class `__init__()` method. Moreover, that super class init method needs to be called after updating the `shortcuts` attribute This warning applies in general to many other attributes which are not settable at runtime.

1.5.2 Aliases

In addition to shortcuts, `cmd2` provides a full alias feature via the `alias` command. Aliases work in a similar fashion to aliases in the Bash shell.

The syntax to create an alias is: `alias create name command [args]`.

> Ex: `alias create ls !ls -lF`

For more details run: `help alias create`

Use `alias list` to see all or some of your aliases. The output of this command displays your aliases using the same command that was used to create them. Therefore you can place this output in a `cmd2` startup script to recreate your aliases each time you start the application

> Ex: `alias list`

For more details run: `help alias list`

Use `alias delete` to remove aliases

For more details run: `help alias delete`

1.5.3 Macros

`cmd2` provides a feature that is similar to aliases called macros. The major difference between macros and aliases is that macros can contain argument placeholders. Arguments are expressed when creating a macro using {#} notation where {1} means the first argument.

The following creates a macro called my_macro that expects two arguments:

macro create my_macro make_dinner -meat {1} -veggie {2}

When the macro is called, the provided arguments are resolved and the assembled command is run. For example:

my_macro beef broccoli —> make_dinner -meat beef -veggie broccoli

For more details run: `help macro create`

The macro command has `list` and `delete` subcommands that function identically to the alias subcommands of the same name. Like aliases, macros can be created via a `cmd2` startup script to preserve them across application sessions.

For more details on listing macros run: `help macro list`

For more details on deleting macros run: `help macro delete`

1.5.4 Default to shell

Every `cmd2` application can execute operating-system level (shell) commands with `shell` or a `!` shortcut:

```
(Cmd) shell which python
/usr/bin/python
(Cmd) !which python
/usr/bin/python
```

However, if the parameter `default_to_shell` is `True`, then *every* command will be attempted on the operating system. Only if that attempt fails (i.e., produces a nonzero return value) will the application's own `default` method be called.

```
(Cmd) which python
/usr/bin/python
(Cmd) my dog has fleas
sh: my: not found
*** Unknown syntax: my dog has fleas
```

1.5.5 Quit on SIGINT

On many shells, SIGINT (most often triggered by the user pressing Ctrl+C) only cancels the current line, not the entire command loop. By default, a `cmd2` application will quit on receiving this signal. However, if `quit_on_sigint` is set to `False`, then the current line will simply be cancelled.

```
(Cmd) typing a comma^C
(Cmd)
```

Warning: The default SIGINT behavior will only function properly if **cmdloop** is running in the main thread.

1.5.6 Timing

Setting `App.timing` to `True` outputs timing data after every application command is executed. The user can `set` this parameter during application execution. (See *Other user-settable parameters*)

1.5.7 Echo

If `True`, each command the user issues will be repeated to the screen before it is executed. This is particularly useful when running scripts.

1.5.8 Debug

Setting `App.debug` to `True` will produce detailed error stacks whenever the application generates an error. The user can `set` this parameter during application execution. (See *Other user-settable parameters*)

1.5.9 Other user-settable parameters

A list of all user-settable parameters, with brief comments, is viewable from within a running application with:

```
(Cmd) set --long
colors: Terminal               # Allow colorized output
continuation_prompt: >         # On 2nd+ line of input
debug: False                   # Show full error stack on error
echo: False                    # Echo command issued into output
editor: vim                    # Program used by ``edit``
feedback_to_output: False      # include nonessentials in `|`, `>` results
locals_in_py: False            # Allow access to your application in py via self
prompt: (Cmd)                  # The prompt issued to solicit input
quiet: False                   # Don't print nonessential feedback
timing: False                  # Report execution times
```

Any of these user-settable parameters can be set while running your app with the `set` command like so:

```
set colors Never
```

1.6 Features requiring application changes

1.6.1 Multiline commands

Command input may span multiple lines for the commands whose names are listed in the `multiline_commands` argument to `cmd2.Cmd.__init__()`. These commands will be executed only after the user has entered a *terminator*. By default, the command terminator is `;`; specifying the `terminators` optional argument to `cmd2.Cmd.__init__()` allows different terminators. A blank line is *always* considered a command terminator (cannot be overridden).

In multiline commands, output redirection characters like `>` and `|` are part of the command arguments unless they appear after the terminator.

1.6.2 Parsed statements

`cmd2` passes `arg` to a `do_` method (or `default`) as a Statement, a subclass of string that includes many attributes of the parsed input:

command Name of the command called

args The arguments to the command with output redirection or piping to shell commands removed

command_and_args A string of just the command and the arguments, with output redirection or piping to shell commands removed

argv A list of arguments a-la `sys.argv`, including the command as `argv[0]` and the subsequent arguments as additional items in the list. Quotes around arguments will be stripped as will any output redirection or piping portions of the command

raw Full input exactly as typed.

terminator Character used to end a multiline command

If `Statement` does not contain an attribute, querying for it will return `None`.

(Getting `arg` as a `Statement` is technically "free", in that it requires no application changes from the cmd standard, but there will be no result unless you change your application to *use* any of the additional attributes.)

1.6.3 Environment parameters

Your application can define user-settable parameters which your code can reference. First create a class attribute with the default value. Then update the `settable` dictionary with your setting name and a short description before you initialize the superclass. Here's an example, from `examples/environment.py`:

```
#!/usr/bin/env python
# coding=utf-8
"""
A sample application for cmd2 demonstrating customized environment parameters
"""
import cmd2


class EnvironmentApp(cmd2.Cmd):
    """ Example cmd2 application. """

    degrees_c = 22
    sunny = False

    def __init__(self):
        super().__init__()
        self.settable.update({'degrees_c': 'Temperature in Celsius'})
        self.settable.update({'sunny': 'Is it sunny outside?'})

    def do_sunbathe(self, arg):
        if self.degrees_c < 20:
            result = "It's {} C - are you a penguin?".format(self.degrees_c)
        elif not self.sunny:
            result = 'Too dim.'
        else:
            result = 'UV is bad for your skin.'
        self.poutput(result)

    def _onchange_degrees_c(self, old, new):
        # if it's over 40C, it's gotta be sunny, right?
        if new > 40:
            self.sunny = True


if __name__ == '__main__':
    import sys
```

(continues on next page)

(continued from previous page)

```
    c = EnvironmentApp()
    sys.exit(c.cmdloop())
```

If you want to be notified when a setting changes (as we do above), then define a method `_onchange_{setting}()`. This method will be called after the user changes a setting, and will receive both the old value and the new value.

```
(Cmd) set --long | grep sunny
sunny: False                # Is it sunny outside?
(Cmd) set --long | grep degrees
degrees_c: 22               # Temperature in Celsius
(Cmd) sunbathe
Too dim.
(Cmd) set degrees_c 41
degrees_c - was: 22
now: 41
(Cmd) set sunny
sunny: True
(Cmd) sunbathe
UV is bad for your skin.
(Cmd) set degrees_c 13
degrees_c - was: 41
now: 13
(Cmd) sunbathe
It's 13 C - are you a penguin?
```

1.6.4 Commands with flags

All `do_` methods are responsible for interpreting the arguments passed to them. However, `cmd2` lets a `do_` methods accept Unix-style *flags*. It uses argparse to parse the flags, and they work the same way as for that module.

`cmd2` defines a few decorators which change the behavior of how arguments get parsed for and passed to a `do_` method. See the section *Argument Processing* for more information.

1.6.5 poutput, pfeedback, perror, ppaged

Standard `cmd` applications produce their output with `self.stdout.write('output')` (or with `print`, but `print` decreases output flexibility). `cmd2` applications can use `self.poutput('output')`, `self.pfeedback('message')`, `self.perror('errmsg')`, and `self.ppaged('text')` instead. These methods have these advantages:

- Handle output redirection to file and/or pipe appropriately
- **More concise**
    - `.pfeedback()` destination is controlled by *Suppressing non-essential output* parameter.
- Option to display long output using a pager via `ppaged()`

`Cmd.`**`poutput`**(*msg: Any*, *end: str = '\n'*, *color: str = ''*) → None

Smarter self.stdout.write(); color aware and adds newline of not present.

Also handles BrokenPipeError exceptions for when a commands's output has been piped to another process and that process terminates before the cmd2 command is finished executing.

Parameters

- **`msg`** – message to print to current stdout (anything convertible to a str with '{}'.format() is OK)
- **`end`** – (optional) string appended after the end of the message if not already present, default a newline
- **`color`** – (optional) color escape to output this message with

`Cmd.`**`perror`**(*err: Union[str, Exception], traceback_war: bool = True, err_color: str = '\x1b[91m', war_color: str = '\x1b[93m'*) → None

Print error message to sys.stderr and if debug is true, print an exception Traceback if one exists.

Parameters

- **`err`** – an Exception or error message to print out
- **`traceback_war`** – (optional) if True, print a message to let user know they can enable debug
- **`err_color`** – (optional) color escape to output error with
- **`war_color`** – (optional) color escape to output warning with

`Cmd.`**`pfeedback`**(*msg: str*) → None

For printing nonessential feedback. Can be silenced with *quiet*. Inclusion in redirected output is controlled by *feedback_to_output*.

`Cmd.`**`ppaged`**(*msg: str*, *end: str = '\n'*, *chop: bool = False*) → None

Print output using a pager if it would go off screen and stdout isn't currently being redirected.

Never uses a pager inside of a script (Python or text) or when output is being redirected or piped or when stdout or stdin are not a fully functional terminal.

Parameters

- **`msg`** – message to print to current stdout (anything convertible to a str with '{}'.format() is OK)
- **`end`** – string appended after the end of the message if not already present, default a newline
- **`chop`** –

  True -> causes lines longer than the screen width to be chopped (truncated) rather than wrapped

  - truncated text is still accessible by scrolling with the right & left arrow keys
  - chopping is ideal for displaying wide tabular data as is done in utilities like pgcli

  False -> causes lines longer than the screen width to wrap to the next line

  - wrapping is ideal when you want to keep users from having to use horizontal scrolling

WARNING: On Windows, the text always wraps regardless of what the chop argument is set to

1.6.6 Colored Output

The output methods in the previous section all honor the `colors` setting, which has three possible values:

Never poutput(), pfeedback(), and ppaged() strip all ANSI escape sequences which instruct the terminal to colorize output

Terminal (the default value) poutput(), pfeedback(), and ppaged() do not strip any ANSI escape sequences when the output is a terminal, but if the output is a pipe or a file the escape sequences are stripped. If you want colorized output you must add ANSI escape sequences, preferably using some python color library like *plumbum.colors*, *colorama*, *blessings*, or *termcolor*.

Always poutput(), pfeedback(), and ppaged() never strip ANSI escape sequences, regardless of the output destination

1.6.7 Suppressing non-essential output

The `quiet` setting controls whether `self.pfeedback()` actually produces any output. If `quiet` is `False`, then the output will be produced. If `quiet` is `True`, no output will be produced.

This makes `self.pfeedback()` useful for non-essential output like status messages. Users can control whether they would like to see these messages by changing the value of the `quiet` setting.

1.6.8 select

Presents numbered options to user, as bash `select`.

`app.select` is called from within a method (not by the user directly; it is `app.select`, not `app.do_select`).

`Cmd.`**`select`** (*opts: Union[str, List[str], List[Tuple[Any, Optional[str]]]], prompt: str = 'Your choice? '*) → str

Presents a numbered menu to the user. Modeled after the bash shell's SELECT. Returns the item chosen.

Argument `opts` can be:

a single string -> will be split into one-word options

a list of strings -> will be offered as options

a list of tuples -> interpreted as (value, text), so that the return value can differ from the text advertised to the user

```
def do_eat(self, arg):
    sauce = self.select('sweet salty', 'Sauce? ')
    result = '{food} with {sauce} sauce, yum!'
    result = result.format(food=arg, sauce=sauce)
    self.stdout.write(result + '\n')
```

```
(Cmd) eat wheaties
    1. sweet
    2. salty
Sauce? 2
wheaties with salty sauce, yum!
```

1.6.9 Exit code to shell

The `self.exit_code` attribute of your `cmd2` application controls what exit code is returned from `cmdloop()` when it completes. It is your job to make sure that this exit code gets sent to the shell when your application exits by calling `sys.exit(app.cmdloop())`.

1.6.10 Asynchronous Feedback

`cmd2` provides two functions to provide asynchronous feedback to the user without interfering with the command line. This means the feedback is provided to the user when they are still entering text at the prompt. To use this functionality, the application must be running in a terminal that supports VT100 control characters and readline. Linux, Mac, and Windows 10 and greater all support these.

async_alert() Used to display an important message to the user while they are at the prompt in between commands. To the user it appears as if an alert message is printed above the prompt and their current input text and cursor location is left alone.

async_update_prompt() Updates the prompt while the user is still typing at it. This is good for alerting the user to system changes dynamically in between commands. For instance you could alter the color of the prompt to indicate a system status or increase a counter to report an event.

`cmd2` also provides a function to change the title of the terminal window. This feature requires the application be running in a terminal that supports VT100 control characters. Linux, Mac, and Windows 10 and greater all support these.

set_window_title() Sets the terminal window title

The easiest way to understand these functions is to see the AsyncPrinting example for a demonstration.

1.6.11 Grouping Commands

By default, the `help` command displays:

```
Documented commands (type help <topic>):
========================================
alias    findleakers  pyscript    sessions             status       vminfo
config   help         quit        set                  stop         which
connect  history      redeploy    shell                thread_dump
deploy   list         resources   shortcuts            unalias
edit     load         restart     sslconnectorciphers  undeploy
expire   py           serverinfo  start                version
```

If you have a large number of commands, you can optionally group your commands into categories. Here's the output from the example `help_categories.py`:

```
Documented commands (type help <topic>):

Application Management
======================
deploy  findleakers  redeploy  sessions  stop
expire  list         restart   start     undeploy

Connecting
==========
connect  which

Server Information
==================
resources  serverinfo  sslconnectorciphers  status  thread_dump  vminfo

Other
=====
alias   edit  history  py        quit  shell      unalias
config  help  load     pyscript  set   shortcuts  version
```

There are 2 methods of specifying command categories, using the `@with_category` decorator or with the `categorize()` function. Once a single command category is detected, the help output switches to a categorized mode of display. All commands with an explicit category defined default to the category *Other*.

Using the `@with_category` decorator:

```
@with_category(CMD_CAT_CONNECTING)
def do_which(self, _):
    """Which command"""
    self.poutput('Which')
```

Using the `categorize()` function:

You can call with a single function:

```
def do_connect(self, _):
    """Connect command"""
    self.poutput('Connect')

# Tag the above command functions under the category Connecting
categorize(do_connect, CMD_CAT_CONNECTING)
```

Or with an Iterable container of functions:

```
def do_undeploy(self, _):
    """Undeploy command"""
    self.poutput('Undeploy')

def do_stop(self, _):
    """Stop command"""
    self.poutput('Stop')

def do_findleakers(self, _):
    """Find Leakers command"""
    self.poutput('Find Leakers')

# Tag the above command functions under the category Application Management
categorize((do_undeploy,
            do_stop,
            do_findleakers), CMD_CAT_APP_MGMT)
```

The `help` command also has a verbose option (`help -v` or `help --verbose`) that combines the help categories with per-command Help Messages:

```
Documented commands (type help <topic>):

Application Management
================================================================================
deploy              Deploy command
expire              Expire command
findleakers         Find Leakers command
list                List command
redeploy            Redeploy command
restart             usage: restart [-h] {now,later,sometime,whenever}
sessions            Sessions command
start               Start command
stop                Stop command
undeploy            Undeploy command

Connecting
================================================================================
connect             Connect command
which               Which command
```

(continues on next page)

(continued from previous page)

```
Server Information
================================================================================
resources              Resources command
serverinfo             Server Info command
sslconnectorciphers    SSL Connector Ciphers command is an example of a command that␣
↪contains
                       multiple lines of help information for the user. Each line of␣
↪help in a
                       contiguous set of lines will be printed and aligned in the␣
↪verbose output
                       provided with 'help --verbose'
status                 Status command
thread_dump            Thread Dump command
vminfo                 VM Info command

Other
================================================================================
alias                Define or display aliases
config               Config command
edit                 Edit a file in a text editor
help                 List available commands with "help" or detailed help with "help␣
↪cmd"
history              usage: history [-h] [-r | -e | -s | -o FILE | -t TRANSCRIPT] [arg]
load                 Runs commands in script file that is encoded as either ASCII or␣
↪UTF-8 text
py                   Invoke python command, shell, or script
pyscript             Runs a python script file inside the console
quit                 Exits this application
set                  usage: set [-h] [-a] [-l] [settable [settable ...]]
shell                Execute a command as if at the OS prompt
shortcuts            Lists shortcuts available
unalias              Unsets aliases
version              Version command
```

1.6.12 Disabling Commands

`cmd2` supports disabling commands during runtime. This is useful if certain commands should only be available when the application is in a specific state. When a command is disabled, it will not show up in the help menu or tab complete. If a user tries to run the command, a command-specific message supplied by the developer will be printed. The following functions support this feature.

enable_command() Enable an individual command

enable_category() Enable an entire category of commands

disable_command() Disable an individual command and set the message that will print when this command is run or help is called on it while disabled

disable_category() Disable an entire category of commands and set the message that will print when anything in this category is run or help is called on it while disabled

See the definitions of these functions for descriptions of their arguments.

See the `do_enable_commands()` and `do_disable_commands()` functions in the HelpCategories example for a demonstration.

1.7 Transcript based testing

A transcript is both the input and output of a successful session of a `cmd2`-based app which is saved to a text file. With no extra work on your part, your app can play back these transcripts as a unit test. Transcripts can contain regular expressions, which provide the flexibility to match responses from commands that produce dynamic or variable output.

1.7.1 Creating a transcript

Automatically from history

A transcript can automatically generated based upon commands previously executed in the *history* using `history -t`:

```
(Cmd) help
...
(Cmd) help history
...
(Cmd) history 1:2 -t transcript.txt
2 commands and outputs saved to transcript file 'transcript.txt'
```

This is by far the easiest way to generate a transcript.

Warning: Make sure you use the **poutput()** method in your `cmd2` application for generating command output. This method of the `cmd2.Cmd` class ensure that output is properly redirected when redirecting to a file, piping to a shell command, and when generating a transcript.

Automatically from a script file

A transcript can also be automatically generated from a script file using `load -t`:

```
(Cmd) load scripts/script.txt -t transcript.txt
2 commands and their outputs saved to transcript file 'transcript.txt'
(Cmd)
```

This is a particularly attractive option for automatically regenerating transcripts for regression testing as your `cmd2` application changes.

Manually

Here's a transcript created from `python examples/example.py`:

```
(Cmd) say -r 3 Goodnight, Gracie
Goodnight, Gracie
Goodnight, Gracie
Goodnight, Gracie
(Cmd) mumble maybe we could go to lunch
like maybe we ... could go to hmmm lunch
(Cmd) mumble maybe we could go to lunch
well maybe we could like go to er lunch right?
```

This transcript has three commands: they are on the lines that begin with the prompt. The first command looks like this:

```
(Cmd) say -r 3 Goodnight, Gracie
```

Following each command is the output generated by that command.

The transcript ignores all lines in the file until it reaches the first line that begins with the prompt. You can take advantage of this by using the first lines of the transcript as comments:

```
# Lines at the beginning of the transcript that do not
; start with the prompt i.e. '(Cmd) ' are ignored.
/* You can use them for comments. */

All six of these lines before the first prompt are treated as comments.

(Cmd) say -r 3 Goodnight, Gracie
Goodnight, Gracie
Goodnight, Gracie
Goodnight, Gracie
(Cmd) mumble maybe we could go to lunch
like maybe we ... could go to hmmm lunch
(Cmd) mumble maybe we could go to lunch
maybe we could like go to er lunch right?
```

In this example I've used several different commenting styles, and even bare text. It doesn't matter what you put on those beginning lines. Everything before:

```
(Cmd) say -r 3 Goodnight, Gracie
```

will be ignored.

1.7.2 Regular Expressions

If we used the above transcript as-is, it would likely fail. As you can see, the `mumble` command doesn't always return the same thing: it inserts random words into the input.

Regular expressions can be included in the response portion of a transcript, and are surrounded by slashes:

```
(Cmd) mumble maybe we could go to lunch
/.*\bmaybe\b.*\bcould\b.*\blunch\b.*/
(Cmd) mumble maybe we could go to lunch
/.*\bmaybe\b.*\bcould\b.*\blunch\b.*/
```

Without creating a tutorial on regular expressions, this one matches anything that has the words `maybe`, `could`, and `lunch` in that order. It doesn't ensure that `we` or `go` or `to` appear in the output, but it does work if mumble happens to add words to the beginning or the end of the output.

Since the output could be multiple lines long, `cmd2` uses multiline regular expression matching, and also uses the `DOTALL` flag. These two flags subtly change the behavior of commonly used special characters like `.`, `^` and `$`, so you may want to double check the Python regular expression documentation.

If your output has slashes in it, you will need to escape those slashes so the stuff between them is not interpred as a regular expression. In this transcript:

```
(Cmd) say cd /usr/local/lib/python3.6/site-packages
/usr/local/lib/python3.6/site-packages
```

the output contains slashes. The text between the first slash and the second slash, will be interpreted as a regular expression, and those two slashes will not be included in the comparison. When replayed, this transcript would therefore fail. To fix it, we could either write a regular expression to match the path instead of specifying it verbatim, or we can escape the slashes:

```
(Cmd) say cd /usr/local/lib/python3.6/site-packages
\/usr\/local\/lib\/python3.6\/site-packages
```

Warning: Be aware of trailing spaces and newlines. Your commands might output trailing spaces which are impossible to see. Instead of leaving them invisible, you can add a regular expression to match them, so that you can see where they are when you look at the transcript:

```
(Cmd) set prompt
prompt: (Cmd)/ /
```

Some terminal emulators strip trailing space when you copy text from them. This could make the actual data generated by your app different than the text you pasted into the transcript, and it might not be readily obvious why the transcript is not passing. Consider using *Output redirection* to the clipboard or to a file to ensure you accurately capture the output of your command.

If you aren't using regular expressions, make sure the newlines at the end of your transcript exactly match the output of your commands. A common cause of a failing transcript is an extra or missing newline.

If you are using regular expressions, be aware that depending on how you write your regex, the newlines after the regex may or may not matter. `\Z` matches *after* the newline at the end of the string, whereas `$` matches the end of the string *or* just before a newline.

1.7.3 Running a transcript

Once you have created a transcript, it's easy to have your application play it back and check the output. From within the `examples/` directory:

```
$ python example.py --test transcript_regex.txt
.
----------------------------------------------------------------------
Ran 1 test in 0.013s

OK
```

The output will look familiar if you use `unittest`, because that's exactly what happens. Each command in the transcript is run, and we `assert` the output matches the expected result from the transcript.

Note: If you have set `allow_cli_args` to False in order to disable parsing of command line arguments at invocation, then the use of `-t` or `--test` to run transcript testing is automatically disabled. In this case, you can alternatively provide a value for the optional `transcript_files` when constructing the instance of your `cmd2.Cmd` derived class in order to cause a transcript test to run:

```
from cmd2 import Cmd
class App(Cmd):
  # customized attributes and methods here

if __name__ == '__main__':
    app = App(transcript_files=['exampleSession.txt'])
    app.cmdloop()
```

1.8 Argument Processing

`cmd2` makes it easy to add sophisticated argument processing to your commands using the `argparse` python module. `cmd2` handles the following for you:

1. Parsing input and quoted strings like the Unix shell
2. Parse the resulting argument list using an instance of `argparse.ArgumentParser` that you provide
3. Passes the resulting `argparse.Namespace` object to your command function. The `Namespace` includes the `Statement` object that was created when parsing the command line. It is stored in the `__statement__` attribute of the `Namespace`.
4. Adds the usage message from the argument parser to your command.
5. Checks if the `-h/--help` option is present, and if so, display the help message for the command

These features are all provided by the `@with_argparser` decorator which is importable from `cmd2`.

See the either the argprint or decorator example to learn more about how to use the various `cmd2` argument processing decorators in your `cmd2` applications.

1.8.1 Decorators provided by cmd2 for argument processing

`cmd2` provides the following decorators for assisting with parsing arguments passed to commands:

`cmd2.`**`with_argument_list`**`(`*, *preserve_quotes: bool = False*`)` → Callable[[List[T]], Optional[bool]]

A decorator to alter the arguments passed to a do_* cmd2 method. Default passes a string of whatever the user typed. With this decorator, the decorated method will receive a list of arguments parsed from user input.

Parameters

- **`args`** – Single-element positional argument list containing do_* method this decorator is wrapping
- **`preserve_quotes`** – if True, then argument quotes will not be stripped

Returns function that gets passed a list of argument strings

`cmd2.`**`with_argparser`**`(`*, *ns_provider: Optional[Callable[[...], argparse.Namespace]] = None, preserve_quotes: bool = False*`)` → Callable[[argparse.Namespace], Optional[bool]]

A decorator to alter a cmd2 method to populate its `args` argument by parsing arguments with the given instance of argparse.ArgumentParser.

Parameters

- **`argparser`** – unique instance of ArgumentParser
- **`ns_provider`** – An optional function that accepts a cmd2.Cmd object as an argument and returns an argparse.Namespace. This is useful if the Namespace needs to be prepopulated with state data that affects parsing.
- **`preserve_quotes`** – if True, then arguments passed to argparse maintain their quotes

Returns function that gets passed the argparse-parsed args in a Namespace A member called __statement__ is added to the Namespace to provide command functions access to the Statement object. This can be useful if the command function needs to know the command line.

`cmd2.`**`with_argparser_and_unknown_args`**`(`**, ns_provider: Optional[Callable[[...], argparse.Namespace]] = None, preserve_quotes: bool = False*`)` → Callable[[argparse.Namespace, List[T]], Optional[bool]]

A decorator to alter a cmd2 method to populate its `args` argument by parsing arguments with the given instance of argparse.ArgumentParser, but also returning unknown args as a list.

Parameters

- **`argparser`** – unique instance of ArgumentParser
- **`ns_provider`** – An optional function that accepts a cmd2.Cmd object as an argument and returns an argparse.Namespace. This is useful if the Namespace needs to be prepopulated with state data that affects parsing.
- **`preserve_quotes`** – if True, then arguments passed to argparse maintain their quotes

Returns function that gets passed argparse-parsed args in a Namespace and a list of unknown argument strings A member called __statement__ is added to the Namespace to provide command functions access to the Statement object. This can be useful if the command function needs to know the command line.

All of these decorators accept an optional **preserve_quotes** argument which defaults to `False`. Setting this argument to `True` is useful for cases where you are passing the arguments to another command which might have its own argument parsing.

1.8.2 Using the argument parser decorator

For each command in the `cmd2` subclass which requires argument parsing, create a unique instance of `argparse.ArgumentParser()` which can parse the input appropriately for the command. Then decorate the command method with the `@with_argparser` decorator, passing the argument parser as the first parameter to the decorator. This changes the second argument to the command method, which will contain the results of `ArgumentParser.parse_args()`.

Here's what it looks like:

```python
import argparse
from cmd2 import with_argparser

argparser = argparse.ArgumentParser()
argparser.add_argument('-p', '--piglatin', action='store_true', help='atinLay')
argparser.add_argument('-s', '--shout', action='store_true', help='N00B EMULATION MODE
↪')
argparser.add_argument('-r', '--repeat', type=int, help='output [n] times')
argparser.add_argument('word', nargs='?', help='word to say')

@with_argparser(argparser)
def do_speak(self, opts)
   """Repeats what you tell me to."""
   arg = opts.word
   if opts.piglatin:
      arg = '%s%say' % (arg[1:], arg[0])
   if opts.shout:
      arg = arg.upper()
   repetitions = opts.repeat or 1
   for i in range(min(repetitions, self.maxrepeats)):
      self.poutput(arg)
```

Warning: It is important that each command which uses the `@with_argparser` decorator be passed a unique instance of a parser. This limitation is due to bugs in CPython prior to Python 3.7 which make it impossible to make a deep copy of an instance of a `argparse.ArgumentParser`.

See the table_display example for a work-around that demonstrates how to create a function which returns a unique instance of the parser you want.

Note: The `@with_argparser` decorator sets the `prog` variable in the argument parser based on the name of the method it is decorating. This will override anything you specify in `prog` variable when creating the argument parser.

1.8.3 Help Messages

By default, cmd2 uses the docstring of the command method when a user asks for help on the command. When you use the `@with_argparser` decorator, the docstring for the `do_*` method is used to set the description for the `argparse.ArgumentParser`.

With this code:

```
import argparse
from cmd2 import with_argparser

argparser = argparse.ArgumentParser()
argparser.add_argument('tag', help='tag')
argparser.add_argument('content', nargs='+', help='content to surround with tag')
@with_argparser(argparser)
def do_tag(self, args):
    """create a html tag"""
    self.stdout.write('<{0}>{1}</{0}>'.format(args.tag, ' '.join(args.content)))
    self.stdout.write('\n')
```

the `help tag` command displays:

```
usage: tag [-h] tag content [content ...]

create a html tag

positional arguments:
  tag         tag
  content     content to surround with tag

optional arguments:
  -h, --help  show this help message and exit
```

If you would prefer you can set the `description` while instantiating the `argparse.ArgumentParser` and leave the docstring on your method empty:

```
import argparse
from cmd2 import with_argparser

argparser = argparse.ArgumentParser(description='create an html tag')
argparser.add_argument('tag', help='tag')
argparser.add_argument('content', nargs='+', help='content to surround with tag')
@with_argparser(argparser)
```

(continues on next page)

(continued from previous page)

```
def do_tag(self, args):
    self.stdout.write('<{0}>{1}</{0}>'.format(args.tag, ' '.join(args.content)))
    self.stdout.write('\n')
```

Now when the user enters `help tag` they see:

```
usage: tag [-h] tag content [content ...]

create an html tag

positional arguments:
  tag         tag
  content     content to surround with tag

optional arguments:
  -h, --help  show this help message and exit
```

To add additional text to the end of the generated help message, use the `epilog` variable:

```
import argparse
from cmd2 import with_argparser

argparser = argparse.ArgumentParser(description='create an html tag',
                                    epilog='This command can not generate tags with␣
↪no content, like <br/>.')
argparser.add_argument('tag', help='tag')
argparser.add_argument('content', nargs='+', help='content to surround with tag')
@with_argparser(argparser)
def do_tag(self, args):
    self.stdout.write('<{0}>{1}</{0}>'.format(args.tag, ' '.join(args.content)))
    self.stdout.write('\n')
```

Which yields:

```
usage: tag [-h] tag content [content ...]

create an html tag

positional arguments:
  tag         tag
  content     content to surround with tag

optional arguments:
  -h, --help  show this help message and exit

This command can not generate tags with no content, like <br/>
```

Warning: If a command **foo** is decorated with one of cmd2's argparse decorators, then **help_foo** will not be invoked when `help foo` is called. The argparse module provides a rich API which can be used to tweak every aspect of the displayed help and we encourage `cmd2` developers to utilize that.

1.8.4 Receiving an argument list

The default behavior of `cmd2` is to pass the user input directly to your `do_*` methods as a string. The object passed to your method is actually a `Statement` object, which has additional attributes that may be helpful, including `arg_list` and `argv`:

```
class CmdLineApp(cmd2.Cmd):
    """ Example cmd2 application. """

    def do_say(self, statement):
        # statement contains a string
        self.poutput(statement)

    def do_speak(self, statement):
        # statement also has a list of arguments
        # quoted arguments remain quoted
        for arg in statement.arg_list:
            self.poutput(arg)

    def do_articulate(self, statement):
        # statement.argv contains the command
        # and the arguments, which have had quotes
        # stripped
        for arg in statement.argv:
            self.poutput(arg)
```

If you don't want to access the additional attributes on the string passed to you‘‘do_*‘‘ method you can still have `cmd2` apply shell parsing rules to the user input and pass you a list of arguments instead of a string. Apply the `@with_argument_list` decorator to those methods that should receive an argument list instead of a string:

```
from cmd2 import with_argument_list

class CmdLineApp(cmd2.Cmd):
    """ Example cmd2 application. """

    def do_say(self, cmdline):
        # cmdline contains a string
        pass

    @with_argument_list
    def do_speak(self, arglist):
        # arglist contains a list of arguments
        pass
```

1.8.5 Using the argument parser decorator and also receiving a list of unknown positional arguments

If you want all unknown arguments to be passed to your command as a list of strings, then decorate the command method with the `@with_argparser_and_unknown_args` decorator.

Here's what it looks like:

```
import argparse
from cmd2 import with_argparser_and_unknown_args

dir_parser = argparse.ArgumentParser()
```

(continues on next page)

(continued from previous page)

```
dir_parser.add_argument('-l', '--long', action='store_true', help="display in long␣
↪format with one item per line")

@with_argparser_and_unknown_args(dir_parser)
def do_dir(self, args, unknown):
    """List contents of current directory."""
    # No arguments for this command
    if unknown:
        self.perror("dir does not take any positional arguments:", traceback_
↪war=False)
        self.do_help('dir')
        self._last_result = CommandResult('', 'Bad arguments')
        return

    # Get the contents as a list
    contents = os.listdir(self.cwd)

    ...
```

1.8.6 Using custom argparse.Namespace with argument parser decorators

In some cases, it may be necessary to write custom `argparse` code that is dependent on state data of your application. To support this ability while still allowing use of the decorators, both `@with_argparser` and `@with_argparser_and_unknown_args` have an optional argument called `ns_provider`.

`ns_provider` is a Callable that accepts a `cmd2.Cmd` object as an argument and returns an `argparse.Namespace`:

```
Callable[[cmd2.Cmd], argparse.Namespace]
```

For example:

```
def settings_ns_provider(self) -> argparse.Namespace:
    """Populate an argparse Namespace with current settings"""
    ns = argparse.Namespace()
    ns.app_settings = self.settings
    return ns
```

To use this function with the argparse decorators, do the following:

```
@with_argparser(my_parser, ns_provider=settings_ns_provider)
```

The Namespace is passed by the decorators to the `argparse` parsing functions which gives your custom code access to the state data it needs for its parsing logic.

1.8.7 Sub-commands

Sub-commands are supported for commands using either the `@with_argparser` or `@with_argparser_and_unknown_args` decorator. The syntax for supporting them is based on argparse sub-parsers.

You may add multiple layers of sub-commands for your command. Cmd2 will automatically traverse and tab-complete sub-commands for all commands using argparse.

See the subcommands and tab_autocompletion example to learn more about how to use sub-commands in your cmd2 application.

1.9 Integrating cmd2 with external tools

1.9.1 Integrating cmd2 with the shell

Typically you would invoke a cmd2 program by typing:

```
$ python mycmd2program.py
```

or:

```
$ mycmd2program.py
```

Either of these methods will launch your program and enter the cmd2 command loop, which allows the user to enter commands, which are then executed by your program.

You may want to execute commands in your program without prompting the user for any input. There are several ways you might accomplish this task. The easiest one is to pipe commands and their arguments into your program via standard input. You don't need to do anything to your program in order to use this technique. Here's a demonstration using the examples/example.py included in the source code of cmd2:

```
$ echo "speak -p some words" | python examples/example.py
omesay ordsway
```

Using this same approach you could create a text file containing the commands you would like to run, one command per line in the file. Say your file was called somecmds.txt. To run the commands in the text file using your cmd2 program (from a Windows command prompt):

```
c:\cmd2> type somecmds.txt | python.exe examples/example.py
omesay ordsway
```

By default, cmd2 programs also look for commands pass as arguments from the operating system shell, and execute those commands before entering the command loop:

```
$ python examples/example.py help

Documented commands (type help <topic>):
========================================
alias  help     load    orate  pyscript  say  shell      speak
edit   history  mumble  py     quit      set  shortcuts  unalias

(Cmd)
```

You may need more control over command line arguments passed from the operating system shell. For example, you might have a command inside your cmd2 program which itself accepts arguments, and maybe even option strings. Say you wanted to run the speak command from the operating system shell, but have it say it in pig latin:

```
$ python example/example.py speak -p hello there
python example.py speak -p hello there
usage: speak [-h] [-p] [-s] [-r REPEAT] words [words ...]
speak: error: the following arguments are required: words
*** Unknown syntax: -p
```

(continues on next page)

(continued from previous page)

```
*** Unknown syntax: hello
*** Unknown syntax: there
(Cmd)
```

Uh-oh, that's not what we wanted. `cmd2` treated `-p`, `hello`, and `there` as commands, which don't exist in that program, thus the syntax errors.

There is an easy way around this, which is demonstrated in `examples/cmd_as_argument.py`. By setting `allow_cli_args=False` you can so your own argument parsing of the command line:

```
$ python examples/cmd_as_argument.py speak -p hello there
ellohay heretay
```

Check the source code of this example, especially the `main()` function, to see the technique.

1.9.2 Integrating cmd2 with event loops

Throughout this documentation we have focused on the **90%** use case, that is the use case we believe around **90+%** of our user base is looking for. This focuses on ease of use and the best out-of-the-box experience where developers get the most functionality for the least amount of effort. We are talking about running cmd2 applications with the `cmdloop()` method:

```
from cmd2 import Cmd
class App(Cmd):
    # customized attributes and methods here
app = App()
app.cmdloop()
```

However, there are some limitations to this way of using `cmd2`, mainly that `cmd2` owns the inner loop of a program. This can be unnecessarily restrictive and can prevent using libraries which depend on controlling their own event loop.

Many Python concurrency libraries involve or require an event loop which they are in control of such as asyncio, gevent, Twisted, etc.

`cmd2` applications can be executed in a fashion where `cmd2` doesn't own the main loop for the program by using code like the following:

```
import cmd2

class Cmd2EventBased(cmd2.Cmd):
    def __init__(self):
        cmd2.Cmd.__init__(self)

    # ... your class code here ...

if __name__ == '__main__':
    app = Cmd2EventBased()
    app.preloop()

    # Do this within whatever event loop mechanism you wish to run a single command
    cmd_line_text = "help history"
    app.runcmds_plus_hooks([cmd_line_text])

    app.postloop()
```

The **runcmds_plus_hooks()** method is a convenience method to run multiple commands via **onecmd_plus_hooks()**. It properly deals with `load` commands which under the hood put commands in a FIFO queue as it reads them in from a script file.

The **onecmd_plus_hooks()** method will do the following to execute a single `cmd2` command in a normal fashion:

1. Parse user input into *Statement* object
2. Call methods registered with *register_postparsing_hook()*
3. Redirect output, if user asked for it and it's allowed
4. Start timer
5. Call methods registered with *register_precmd_hook()*
6. Call *precmd()* - for backwards compatibility with `cmd.Cmd`
7. Add statement to history
8. Call *do_command* method
9. Call methods registered with *register_postcmd_hook()*
10. Call *postcmd(stop, statement)* - for backwards compatibility with `cmd.Cmd`
11. Stop timer and display the elapsed time
12. Stop redirecting output if it was redirected
13. Call methods registered with *register_cmdfinalization_hook()*

Running in this fashion enables the ability to integrate with an external event loop. However, how to integrate with any specific event loop is beyond the scope of this documentation. Please note that running in this fashion comes with several disadvantages, including:

- Requires the developer to write more code
- Does not support transcript testing
- Does not allow commands at invocation via command-line arguments

Here is a little more info on `runcmds_plus_hooks`:

`Cmd.`**`runcmds_plus_hooks`**(*cmds: List[Union[cmd2.history.HistoryItem, str]]*) → bool

Used when commands are being run in an automated fashion like text scripts or history replays. The prompt and command line for each command will be printed if echo is True.

Parameters **`cmds`** – commands to run

Returns True if running of commands should stop

1.10 cmd2 Application Lifecycle and Hooks

The typical way of starting a cmd2 application is as follows:

```python
import cmd2
class App(cmd2.Cmd):
    # customized attributes and methods here

if __name__ == '__main__':
    app = App()
    app.cmdloop()
```

There are several pre-existing methods and attributes which you can tweak to control the overall behavior of your application before, during, and after the command processing loop.

1.10.1 Application Lifecycle Hooks

You can register methods to be called at the beginning of the command loop:

```
class App(cmd2.Cmd):
    def __init__(self, *args, *kwargs):
        super().__init__(*args, **kwargs)
        self.register_preloop_hook(self.myhookmethod)

    def myhookmethod(self):
        self.poutput("before the loop begins")
```

To retain backwards compatibility with *cmd.Cmd*, after all registered preloop hooks have been called, the `preloop()` method is called.

A similar approach allows you to register functions to be called after the command loop has finished:

```
class App(cmd2.Cmd):
    def __init__(self, *args, *kwargs):
        super().__init__(*args, **kwargs)
        self.register_postloop_hook(self.myhookmethod)

    def myhookmethod(self):
        self.poutput("before the loop begins")
```

To retain backwards compatibility with *cmd.Cmd*, after all registered postloop hooks have been called, the `postloop()` method is called.

Preloop and postloop hook methods are not passed any parameters and any return value is ignored.

1.10.2 Application Lifecycle Attributes

There are numerous attributes of and arguments to `cmd2.Cmd` which have a significant effect on the application behavior upon entering or during the main loop. A partial list of some of the more important ones is presented here:

- **intro**: *str* - if provided this serves as the intro banner printed once at start of application, after `preloop` runs
- **allow_cli_args**: *bool* - if True (default), then searches for -t or –test at command line to invoke transcript testing mode instead of a normal main loop and also processes any commands provided as arguments on the command line just prior to entering the main loop
- **echo**: *bool* - if True, then the command line entered is echoed to the screen (most useful when running scripts)
- **prompt**: *str* - sets the prompt which is displayed, can be dynamically changed based on application state and/or command results

1.10.3 Command Processing Loop

When you call *.cmdloop()*, the following sequence of events are repeated until the application exits:

1. Output the prompt
2. Accept user input

3. Parse user input into *Statement* object
4. Call methods registered with *register_postparsing_hook()*
5. Redirect output, if user asked for it and it's allowed
6. Start timer
7. Call methods registered with *register_precmd_hook()*
8. Call *precmd()* - for backwards compatibility with `cmd.Cmd`
9. Add statement to history
10. Call *do_command* method
11. Call methods registered with *register_postcmd_hook()*
12. Call *postcmd(stop, statement)* - for backwards compatibility with `cmd.Cmd`
13. Stop timer and display the elapsed time
14. Stop redirecting output if it was redirected
15. Call methods registered with *register_cmdfinalization_hook()*

By registering hook methods, steps 4, 8, 12, and 16 allow you to run code during, and control the flow of the command processing loop. Be aware that plugins also utilize these hooks, so there may be code running that is not part of your application. Methods registered for a hook are called in the order they were registered. You can register a function more than once, and it will be called each time it was registered.

Postparsing, precommand, and postcommand hook methods share some common ways to influence the command processing loop.

If a hook raises a `cmd2.EmptyStatement` exception: - no more hooks (except command finalization hooks) of any kind will be called - if the command has not yet been executed, it will not be executed - no error message will be displayed to the user

If a hook raises any other exception: - no more hooks (except command finalization hooks) of any kind will be called - if the command has not yet been executed, it will not be executed - the exception message will be displayed for the user.

Specific types of hook methods have additional options as described below.

Postparsing Hooks

Postparsing hooks are called after the user input has been parsed but before execution of the command. These hooks can be used to:

- modify the user input
- run code before every command executes
- cancel execution of the current command
- exit the application

When postparsing hooks are called, output has not been redirected, nor has the timer for command execution been started.

To define and register a postparsing hook, do the following:

```
class App(cmd2.Cmd):
    def __init__(self, *args, *kwargs):
        super().__init__(*args, **kwargs)
        self.register_postparsing_hook(self.myhookmethod)

    def myhookmethod(self, params: cmd2.plugin.PostparsingData) -> cmd2.plugin.
↪PostparsingData:
        # the statement object created from the user input
        # is available as params.statement
        return params
```

`register_postparsing_hook()` checks the method signature of the passed callable, and raises a `TypeError` if it has the wrong number of parameters. It will also raise a `TypeError` if the passed parameter and return value are not annotated as `PostparsingData`.

The hook method will be passed one parameter, a `PostparsingData` object which we will refer to as `params`. `params` contains two attributes. `params.statement` is a `Statement` object which describes the parsed user input. There are many useful attributes in the `Statement` object, including `.raw` which contains exactly what the user typed. `params.stop` is set to `False` by default.

The hook method must return a `PostparsingData` object, and it is very convenient to just return the object passed into the hook method. The hook method may modify the attributes of the object to influece the behavior of the application. If `params.stop` is set to true, a fatal failure is triggered prior to execution of the command, and the application exits.

To modify the user input, you create a new `Statement` object and return it in `params.statement`. Don't try and directly modify the contents of a `Statement` object, there be dragons. Instead, use the various attributes in a `Statement` object to construct a new string, and then parse that string to create a new `Statement` object.

`cmd2.Cmd()` uses an instance of `cmd2.StatementParser` to parse user input. This instance has been configured with the proper command terminators, multiline commands, and other parsing related settings. This instance is available as the `self.statement_parser` attribute. Here's a simple example which shows the proper technique:

```
def myhookmethod(self, params: cmd2.plugin.PostparsingData) -> cmd2.plugin.
↪PostparsingData:
    if not '|' in params.statement.raw:
        newinput = params.statement.raw + ' | less'
        params.statement = self.statement_parser.parse(newinput)
    return params
```

If a postparsing hook returns a `PostparsingData` object with the `stop` attribute set to `True`:

- no more hooks of any kind (except command finalization hooks) will be called
- the command will not be executed
- no error message will be displayed to the user
- the application will exit

Precommand Hooks

Precommand hooks can modify the user input, but can not request the application terminate. If your hook needs to be able to exit the application, you should implement it as a postparsing hook.

Once output is redirected and the timer started, all the hooks registered with `register_precmd_hook()` are called. Here's how to do it:

```
class App(cmd2.Cmd):
    def __init__(self, *args, *kwargs):
        super().__init__(*args, **kwargs)
        self.register_precmd_hook(self.myhookmethod)

    def myhookmethod(self, data: cmd2.plugin.PrecommandData) -> cmd2.plugin.
↪PrecommandData:
        # the statement object created from the user input
        # is available as data.statement
        return data
```

`register_precmd_hook()` checks the method signature of the passed callable, and raises a `TypeError` if it has the wrong number of parameters. It will also raise a `TypeError` if the parameters and return value are not annotated as `PrecommandData`.

You may choose to modify the user input by creating a new `Statement` with different properties (see above). If you do so, assign your new `Statement` object to `data.statement`.

The precommand hook must return a `PrecommandData` object. You don't have to create this object from scratch, you can just return the one passed into the hook.

After all registered precommand hooks have been called, `self.precmd(statement)` will be called. To retain full backward compatibility with `cmd.Cmd`, this method is passed a `Statement`, not a `PrecommandData` object.

Postcommand Hooks

Once the command method has returned (i.e. the `do_command(self, statement) method` has been called and returns, all postcommand hooks are called. If output was redirected by the user, it is still redirected, and the command timer is still running.

Here's how to define and register a postcommand hook:

```
class App(cmd2.Cmd):
    def __init__(self, *args, *kwargs):
        super().__init__(*args, **kwargs)
        self.register_postcmd_hook(self.myhookmethod)

    def myhookmethod(self, data: cmd2.plugin.PostcommandData) -> cmd2.plugin.
↪PostcommandData:
        return data
```

Your hook will be passed a `PostcommandData` object, which has a `statement` attribute that describes the command which was executed. If your postcommand hook method gets called, you are guaranteed that the command method was called, and that it didn't raise an exception.

If any postcommand hook raises an exception, the exception will be displayed to the user, and no further postcommand hook methods will be called. Command finalization hooks, if any, will be called.

After all registered postcommand hooks have been called, `self.postcmd(statement)` will be called to retain full backward compatibility with `cmd.Cmd`.

If any postcommand hook (registered or `self.postcmd()`) returns a `PostcommandData` object with the stop attribute set to `True`, subsequent postcommand hooks will still be called, as will the command finalization hooks, but once those hooks have all been called, the application will terminate. Likewise, if `self.postcmd()` returns `True`, the command finalization hooks will be called before the application terminates.

Any postcommand hook can change the value of the `stop` parameter before returning it, and the modified value will be passed to the next postcommand hook. The value returned by the final postcommand hook will be passed to the command finalization hooks, which may further modify the value. If your hook blindly returns `False`, a prior hook's requst to exit the application will not be honored. It's best to return the value you were passed unless you have a compelling reason to do otherwise.

Command Finalization Hooks

Command finalization hooks are called even if one of the other types of hooks or the command method raise an exception. Here's how to create and register a command finalization hook:

```
class App(cmd2.Cmd):
    def __init__(self, *args, *kwargs):
        super().__init__(*args, **kwargs)
        self.register_cmdfinalization_hook(self.myhookmethod)

    def myhookmethod(self, stop, statement):
        return stop
```

Command Finalization hooks must check whether the statement object is `None`. There are certain circumstances where these hooks may be called before the statement has been parsed, so you can't always rely on having a statement.

If any prior postparsing or precommand hook has requested the application to terminate, the value of the `stop` parameter passed to the first command finalization hook will be `True`. Any command finalization hook can change the value of the `stop` parameter before returning it, and the modified value will be passed to the next command finalization hook. The value returned by the final command finalization hook will determine whether the application terminates or not.

This approach to command finalization hooks can be powerful, but it can also cause problems. If your hook blindly returns `False`, a prior hook's requst to exit the application will not be honored. It's best to return the value you were passed unless you have a compelling reason to do otherwise.

If any command finalization hook raises an exception, no more command finalization hooks will be called. If the last hook to return a value returned `True`, then the exception will be rendered, and the application will terminate.

1.11 Alternatives to cmd and cmd2

For programs that do not interact with the user in a continuous loop - programs that simply accept a set of arguments from the command line, return results, and do not keep the user within the program's environment - all you need are sys.argv (the command-line arguments) and argparse (for parsing UNIX-style options and flags). Though some people may prefer docopt or click to argparse.

The curses module produces applications that interact via a plaintext terminal window, but are not limited to simple text input and output; they can paint the screen with options that are selected from using the cursor keys. However, programming a curses-based application is not as straightforward as using cmd.

Several Python packages exist for building interactive command-line applications approximately similar in concept to cmd applications. None of them share `cmd2`'s close ties to cmd, but they may be worth investigating nonetheless. Two of the most mature and full featured are:

- Python Prompt Toolkit
- Click

Python Prompt Toolkit is a library for building powerful interactive command lines and terminal applications in Python. It provides a lot of advanced visual features like syntax highlighting, bottom bars, and the ability to create fullscreen apps.

Click is a Python package for creating beautiful command line interfaces in a composable way with as little code as necessary. It is more geared towards command line utilities instead of command line interpreters, but it can be used for either.

Getting a working command-interpreter application based on either Python Prompt Toolkit or Click requires a good deal more effort and boilerplate code than `cmd2`. `cmd2` focuses on providing an excellent out-of-the-box experience with as many useful features as possible built in for free with as little work required on the developer's part as possible. We believe that `cmd2` provides developers the easiest way to write a command-line interpreter, while allowing a good experience for end users. If you are seeking a visually richer end-user experience and don't mind investing more development time, we would recommend checking out Python Prompt Toolkit.

In the future, we may investigate options for incorporating the usage of Python Prompt Toolkit and/or Click into `cmd2` applications.

CHAPTER 2

Compatibility

Tested and working with Python 3.4+ on Windows, macOS, and Linux.

CHAPTER 3

Index

Index