cmd2 Documentation

Release 0.9.3

Catherine Devlin and Todd Leonhardt

Jul 17, 2018

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:

- pyparsing
- pyperclip

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 requirement for Python 3.4

`cmd2` requires the `contextlib2` module for Python 3.4. This is used to temporarily redirect stdout and stderr.

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.

Cmd.**do_load**(*arglist: List[str]*) → None
Runs commands in script file that is encoded as either ASCII or UTF-8 text.

Usage: load <file_path>

- file_path - a file path pointing to a script

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

Cmd.**do__relative_load**(*arglist: List[str]*) → None
Runs commands in script file that is encoded as either ASCII or UTF-8 text.

Usage: _relative_load <file_path>

optional argument: file_path a file path pointing to a script

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

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

NOTE: This command is intended to only be used within text file scripts.

Cmd.**do_edit**(*arglist: List[str]*) → None
Edit a file in a text editor.

Usage: edit [file_path]

Where:

- file_path - path to a file to open in editor

The editor used is determined by the `editor` settable parameter. "set editor (program-name)" to change or set the EDITOR environment variable.

1.4.2 Comments

Comments are omitted from the argument list before it is passed to a `do_` method. By default, both Python-style and C-style comments are recognized. Comments can be useful in *Script files*, but would be pointless within an interactive session.

```
def do_speak(self, arg):
    self.stdout.write(arg + '\n')
```

```
(Cmd) speak it was /* not */ delicious! # Yuck!
it was  delicious!
```

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` attribute 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):
        self.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. That session can call

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

All application commands are accessible as python objects and functions matching the command name. For example, the following are equivalent:

```
>>> app('say --piglatin Blah')
lahBay
>>> app.say("Blah", piglatin=True)
lahBay
```

Sub-commands are also supported. The following pairs are equivalent:

```
>>> app('command subcmd1 subcmd2 param1 --myflag --otherflag 3')
>>> app.command.subcmd1.subcmd2('param1', myflag=True, otherflag=3)

>>> app('command subcmd1 param1 subcmd2 param2 --myflag --otherflag 3')
>>> app.command.subcmd1('param1').subcmd2('param2', myflag=True, otherflag=3)
```

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)

    Invoke python command, shell, or script

    py <command>: Executes a Python command.
    py: Enters interactive Python mode.
    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 **examples/arg_printer.py** script:

```
(Cmd) pyscript examples/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 scripts, 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 (NOTE: the `do_pyscript` method uses this decorator:

```
(Cmd) pyscript examples/arg_printer.py hello '23 fnord'
Running Python script 'arg_printer.py' which was called with 2 arguments
arg 1: 'hello'
arg 2: '23 fnord'
```

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 readline 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: Union[str, NoneType] = None, use_ipython: bool = False, transcript_files: Union[List[str], NoneType] = 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 readline history from
- **persistent_history_length** – (optional) max number of lines which will be written to the 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
- **transcript_files** – (optional) allows running transcript tests when allow_cli_args is False

`cmd2` makes a third type of history access available with the **history** command:

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

usage: history [-h] [-r | -e | -s | -o FILE | -t TRANSCRIPT | -c] [arg]

View, run, edit, save, or clear previously entered commands.

positional arguments:

arg empty all history items a one history item by number a..b, a:b, a:, ..b items by indices (inclusive) [string] items containing string /regex/ items matching regular expression

optional arguments:

-h, --help	show this help message and exit
-r, --run	run selected history items
-e, --edit	edit and then run selected history items
-s, --script	script format; no separation lines
-o FILE, --output-file FILE	output commands to a script file
-t TRANSCRIPT, --transcript TRANSCRIPT	output commands and results to a transcript file
-c, --clear	clears all history

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**(*_: str*) → bool
Exits this application.

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

Usage: shell <command> [arguments]

(! 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, dir_only=True)
```

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_`):

```
class App(Cmd2):
    def __init__(self):
      # Make sure you update the shortcuts attribute before calling the super class __
↪init__
      self.shortcuts.update({'*': 'sneeze', '~': 'squirm'})

      # Make sure to call this super class __init__ after updating shortcuts
      cmd2.Cmd.__init__(self)
```

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 such as `multiline_commands`, etc.

1.5.2 Aliases

In addition to shortcuts, `cmd2` provides a full alias feature via the `alias` command which is similar to the `alias` command in Bash.

The syntax to create an alias is `alias <name> <value>`. `value` can contain spaces and does not need to be quoted. Ex: `alias ls !ls -lF`

If `alias` is run without arguments, then a list of all aliases will be printed to stdout and are in the proper `alias` command syntax, meaning they can easily be reused.

The `unalias` is used to clear aliases. Using the `-a` flag will clear all aliases. Otherwise provide a list of aliases to clear. Ex: `unalias ls cd pwd` will clear the aliases called ls, cd, and pwd.

1.5.3 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
```

1.5.4 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)
```

1.5.5 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*)

1.5.6 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.

1.5.7 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*)

1.5.8 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: True                    # Colorized output (*nix only)
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 False
```

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 parameter `app.multiline_commands`. These commands will be executed only after the user has entered a *terminator*. By default, the command terminator is `;`; replacing or appending to the list `app.terminators` 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):
        self.settable.update({'degrees_c': 'Temperature in Celsius'})
        self.settable.update({'sunny': 'Is it sunny outside?'})
        super().__init__()

    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__':
    c = EnvironmentApp()
    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
```

(continues on next page)

(continued from previous page)

```
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 *quiet* parameter.
- Option to display long output using a pager via `ppaged()`

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

Convenient shortcut for self.stdout.write(); by default adds newline to end if not already 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`** – string appended after the end of the message if not already present, default a newline

`Cmd.`**`perror`**(*err: Union[str, Exception]*, *traceback_war: bool = True*) → 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

Returns

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 avoid users having to use horizontal scrolling

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

1.6.6 color

Text output can be colored by wrapping it in the `colorize` method.

Cmd.**colorize**(*val: str*, *color: str*) → str
Given a string (`val`), returns that string wrapped in UNIX-style special characters that turn on (and then off) text color and style. If the `colors` environment parameter is `False`, or the application is running on Windows, will return `val` unchanged. `color` should be one of the supported strings (or styles): red/blue/green/cyan/magenta, bold, underline

1.6.7 quiet

Controls whether `self.pfeedback('message')` output is suppressed; useful for non-essential feedback that the user may not always want to read. `quiet` is only relevant if `app.pfeedback` is sometimes used.

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[str, Union[str, NoneType]]]]*, *prompt: str = 'Your choice? '*) → str
Presents a numbered menu to the user. Modelled 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!
```

1.7 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

A transcript can automatically generated based upon commands previously executed in the *history*:

```
(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.

Manually

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

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

(continues on next page)

(continued from previous page)

```
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
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 argparse example to learn more about how to use the various cmd2 argument processing decorators in your cmd2 applications.

1.8.1 Using the argument parser decorator

For each command in the cmd2 subclass which requires argument parsing, create an 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 argumen to the command method, which will contain the results of ArgumentParser.parse_args().

Here's what it looks like:

```
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:
```

(continues on next page)

(continued from previous page)

```
        arg = arg.upper()
    repetitions = opts.repeat or 1
    for i in range(min(repetitions, self.maxrepeats)):
        self.poutput(arg)
```

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.

1.8.2 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` is 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)
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/>
```

1.8.3 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')
```

(continues on next page)

(continued from previous page)

```
# 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

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
```

(continues on next page)

(continued from previous page)

```
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 (aliases) available.
unalias             Unsets aliases
version             Version command
```

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. If you don't want to use the full argument parser support outlined above, 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 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()
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
```

(continues on next page)

(continued from previous page)

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

    ...
```

1.8.6 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

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.

1.9.1 Integrating cmd2 with event loops

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__':
```

(continues on next page)

(continued from previous page)

```
    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 the command line text
2. Execute postparsing_precmd()
3. Add the command to the history
4. Apply output redirection, if present
5. Execute precmd()
6. Execute onecmd() - this is what actually runs the command
7. Execute postcmd()
8. Undo output redirection (if present) and perform piping, if present
9. Execute postparsing_postcmd()

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[str]*) → bool

Convenience method to run multiple commands by onecmd_plus_hooks.

This method adds the given cmds to the command queue and processes the queue until completion or an error causes it to abort. Scripts that are loaded will have their commands added to the queue. Scripts may even load other scripts recursively. This means, however, that you should not use this method if there is a running cmdloop or some other event-loop. This method is only intended to be used in "one-off" scenarios.

NOTE: You may need this method even if you only have one command. If that command is a load, then you will need this command to fully process all the subsequent commands that are loaded from the script file. This is an improvement over onecmd_plus_hooks, which expects to be used inside of a command loop which does the processing of loaded commands.

Example: cmd_obj.runcmds_plus_hooks(['load myscript.txt'])

Parameters **`cmds`** – command strings suitable for onecmd_plus_hooks.

Returns True implies the entire application should exit.

1.10 cmd2 Application Lifecycle and Hooks

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

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

1.10.1 Application Lifecycle Hook Methods

The `preloop` and `postloop` methods run before and after the main loop, respectively.

`Cmd.`**`preloop`**`()` → None
Hook method executed once when the cmdloop() method is called.

`Cmd.`**`postloop`**`()`
Hook method executed once when the cmdloop() method is about to return.

1.10.2 Application Lifecycle Attributes

There are numerous attributes (member variables of the `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 inst**
  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 Hooks

Inside the main loop, every time the user hits <Enter> the line is processed by the `onecmd_plus_hooks` method.

`Cmd.`**`onecmd_plus_hooks`**`(`*line: str*`)` → bool
Top-level function called by cmdloop() to handle parsing a line and running the command and all of its hooks.

Parameters **`line`** – line of text read from input

Returns True if cmdloop() should exit, False otherwise

As the `onecmd_plus_hooks` name implies, there are a number of *hook* methods that can be defined in order to inject application-specific behavior at various points during the processing of a line of text entered by the user. `cmd2` increases the 2 hooks provided by `cmd` (**precmd** and **postcmd**) to 6 for greater flexibility. Here are the various hook methods, presented in chronological order starting with the ones called earliest in the process.

`Cmd.`**`preparse`**`(`*raw: str*`)` → str
Hook method executed just before the command line is interpreted, but after the input prompt is generated.

Parameters **raw** – raw command line input

Returns potentially modified raw command line input

Cmd.**postparsing_precmd**(*statement: cmd2.parsing.Statement*) → Tuple[bool, cmd2.parsing.Statement]
This runs after parsing the command-line, but before anything else; even before adding cmd to history.

NOTE: This runs before precmd() and prior to any potential output redirection or piping.

If you wish to fatally fail this command and exit the application entirely, set stop = True.

If you wish to just fail this command you can do so by raising an exception:

- raise EmptyStatement - will silently fail and do nothing
- raise <AnyOtherException> - will fail and print an error message

Parameters **statement** –

- the parsed command-line statement as a Statement object

Returns (bool, statement) - (stop, statement) containing a potentially modified version of the statement object

Cmd.**precmd**(*statement: cmd2.parsing.Statement*) → cmd2.parsing.Statement
Hook method executed just before the command is processed by `onecmd()` and after adding it to the history.

Parameters **statement** – subclass of str which also contains the parsed input

Returns a potentially modified version of the input Statement object

Cmd.**postcmd**(*stop*, *line*)
Hook method executed just after a command dispatch is finished.

Cmd.**postparsing_postcmd**(*stop: bool*) → bool
This runs after everything else, including after postcmd().

It even runs when an empty line is entered. Thus, if you need to do something like update the prompt due to notifications from a background thread, then this is the method you want to override to do it.

Parameters **stop** – bool - True implies the entire application should exit.

Returns bool - True implies the entire application should exit.

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

Indices and tables

- genindex
- modindex
- search

Index

Symbols

C

D

O

P

R

S