cmd2 Documentation

Release 0.7.5

Catherine Devlin and Todd Leonhardt

Jul 19, 2017

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 Homebrew package manager to install the `readline` package; alternatively for macOS the `conda` package manager that comes with the Anaconda Python distro can be used to install `readline` (preferably from conda-forge). For Windows, we recommend installing the pyreadline Python module.

CHAPTER 1

Resources

- cmd
- cmd2 project page
- project bug tracker
- PyCon 2010 presentation, *Easy Command-Line Applications with cmd and cmd2*: 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:

Installation Instructions

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

Installing

First you need to make sure you have Python 2.7 or Python 3.3+, 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 2 >=2.7.9 or 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 2:

```
sudo apt-get install python-cmd2
```

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 only one Python file (**cmd2.py**), so it 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 this file 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:

- six
- pyparsing
- pyperclip

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.

Uninstalling cmd2

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

```
pip uninstall cmd2
```

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.

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`.

Script files

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

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; you may change this by overriding `app.commentGrammars` with a different pyparsing grammar.

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!
```

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
```

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`
- piped (`|`) as input to operating-system commands, as in `mycommand args | wc`
- sent to the paste buffer, ready for the next Copy operation, by ending with a bare >, as in `mycommand args >`.. Redirecting to paste buffer requires software to be installed on the operating system, pywin32 on Windows or xclip on *nix.

If your application depends on mathematical syntax, > may be a bad choice for redirecting output - it will prevent you from using the greater-than sign in your actual user commands. You can override your app's value of `self.redirector` to use a different string for output redirection:

```
class MyApp(cmd2.Cmd):
    redirector = '->'
```

```
(Cmd) say line1 -> out.txt
(Cmd) say line2 ->-> out.txt
(Cmd) !cat out.txt
line1
line2
```

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.

Python

The `py` command will run its arguments as a Python command. Entered without arguments, it enters an interactive Python session. That session can call "back" to your application with `cmd("")`. Through `self`, it also has access to your application instance itself which can be extremely useful for debugging. (If giving end-users this level of introspection is inappropriate, the `locals_in_py` parameter can be set to `False` and removed from the settable dictionary. See see *Other user-settable parameters*)

```
(Cmd) py print("-".join("spelling"))
s-p-e-l-l-i-n-g
(Cmd) py
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(CmdLineApp)
```

```
    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 `cmd("your command")`.

>>> import os
>>> os.uname()
('Linux', 'eee', '2.6.31-19-generic', '#56-Ubuntu SMP Thu Jan 28 01:26:53 UTC 2010',
↪'i686')
>>> cmd("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 setting the cmd2 global variable `USE_ARG_LIST` to `True` in your application using the `set_use_arg_list` function. This passes all arguments to `@options` commands as a list of strings instead of a single string.

Once this option is set, you can then put arguments in quotes like so:

```
(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'
```

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.

Searchable command history

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

All cmd-based applications on systems with the `readline` module also provide bash-like history list editing.

`cmd2` makes a third type of history access available, consisting of these commands:

Quitting the application

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

Abbreviated commands

`cmd2` apps will accept shortened command names so long as there is no ambiguity if the `abbrev` settable parameter is set to `True`. Thus, if `do_divide` is defined, then `divid`, `div`, or even `d` will suffice, so long as there are no other commands defined beginning with *divid*, *div*, or *d*.

This behavior is disabled by default, but can be turned on with `app.abbrev` (see *Other user-settable parameters*)

Warning: Due to the way the parsing logic works for multiline commands, abbreviations will not be accepted for multiline commands.

Misc. pre-defined commands

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

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

Transcript-based testing

If the entire transcript (input and output) of a successful session of a `cmd2`-based app is copied from the screen and pasted into a text file, `transcript.txt`, then a transcript test can be run against it:

```
python app.py --test transcript.txt
```

Any non-whitespace deviations between the output prescribed in `transcript.txt` and the actual output from a fresh run of the application will be reported as a unit test failure. (Whitespace is ignored during the comparison.)

Regular expressions can be embedded in the transcript inside paired `/` slashes. These regular expressions should not include any whitespace expressions.

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()
```

Tab-Completion

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

- `edit`
- `load`
- `pyscript`
- `save`
- `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`:

```
# Assuming you have an "import cmd2" somewhere at the top
complete_foo = cmd2.Cmd.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 build-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)
```

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*)

Case-insensitivity

By default, all `cmd2` command names are case-insensitive; `sing the blues` and `SiNg the blues` are equivalent. To change this, set `App.case_insensitive` to False.

Whether or not you set `case_insensitive`, *please do not* define command method names with any uppercase letters. `cmd2` expects all command methods to be lowercase.

Shortcuts

Special-character shortcuts for 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):
    Cmd2.shortcuts.update({'*': 'sneeze', '~': 'squirm'})
```

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
```

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*)

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.

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*)

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
abbrev: False                 # Accept abbreviated commands
autorun_on_edit: False        # Automatically run files after editing
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: True            # 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 abbrev True
```

Features requiring application changes

Multiline commands

Command input may span multiple lines for the commands whose names are listed in the parameter `app.multilineCommands`. These commands will be executed only after the user has entered a *terminator*. By default, the command terminators is `;`; replacing or appending to the list `app.terminators` allows different terminators. A blank line is *always* considered a command terminator (cannot be overridden).

Parsed statements

`cmd2` passes `arg` to a `do_` method (or `default`) as a ParsedString, a subclass of string that includes an attribute `parsed`. `parsed` is a `pyparsing.ParseResults` object produced by applying a pyparsing grammar applied to `arg`. It may include:

command Name of the command called

raw Full input exactly as typed.

terminator Character used to end a multiline command

suffix Remnant of input after terminator

```
def do_parsereport(self, arg):
    self.stdout.write(arg.parsed.dump() + '\n')
```

```
(Cmd) parsereport A B /* C */ D; E
['parsereport', 'A B  D', ';', 'E']
- args: A B  D
- command: parsereport
- raw: parsereport A B /* C */ D; E
- statement: ['parsereport', 'A B  D', ';']
    - args: A B  D
    - command: parsereport
    - terminator: ;
- suffix: E
- terminator: ;
```

If `parsed` does not contain an attribute, querying for it will return `None`. (This is a characteristic of `pyparsing.ParseResults`.)

The parsing grammar and process currently employed by cmd2 is stable, but is likely significantly more complex than it needs to be. Future `cmd2` releases may change it somewhat (hopefully reducing complexity).

(Getting `arg` as a `ParsedString` 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* `arg.parsed`.)

Environment parameters

Your application can define user-settable parameters which your code can reference. Create them as class attributes with their default values, and add them (with optional documentation) to `settable`.

```
from cmd2 import Cmd
class App(Cmd):
    degrees_c = 22
    sunny = False
```

```
    settable = Cmd.settable + '''degrees_c temperature in Celsius
        sunny'''
    def do_sunbathe(self, arg):
        if self.degrees_c < 20:
            result = "It's {temp} C - are you a penguin?".format(temp=self.degrees_c)
        elif not self.sunny:
            result = 'Too dim.'
        else:
            result = 'UV is bad for your skin.'
        self.stdout.write(result + '\n')
app = App()
app.cmdloop()
```

```
(Cmd) set --long
degrees_c: 22                   # temperature in Celsius
sunny: False                    #
(Cmd) sunbathe
Too dim.
(Cmd) set sunny yes
sunny - was: False
now: True
(Cmd) sunbathe
UV is bad for your skin.
(Cmd) set degrees_c 13
degrees_c - was: 22
now: 13
(Cmd) sunbathe
It's 13 C - are you a penguin?
```

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 optparse to parse the flags, and they work the same way as for that module.

Flags are defined with the `options` decorator, which is passed a list of optparse-style options, each created with `make_option`. The method should accept a second argument, `opts`, in addition to `args`; the flags will be stripped from `args`.

```
@options([make_option('-p', '--piglatin', action="store_true", help="atinLay"),
    make_option('-s', '--shout', action="store_true", help="N00B EMULATION MODE"),
    make_option('-r', '--repeat', type="int", help="output [n] times")
])
def do_speak(self, arg, opts=None):
    """Repeats what you tell me to."""
    arg = ''.join(arg)
    if opts.piglatin:
        arg = '%s%say' % (arg[1:].rstrip(), arg[0])
    if opts.shout:
        arg = arg.upper()
    repetitions = opts.repeat or 1
    for i in range(min(repetitions, self.maxrepeats)):
        self.stdout.write(arg)
        self.stdout.write('\n')
```

```
(Cmd) say goodnight, gracie
goodnight, gracie
```

```
(Cmd) say -sp goodnight, gracie
OODNIGHT, GRACIEGAY
(Cmd) say -r 2 --shout goodnight, gracie
GOODNIGHT, GRACIE
GOODNIGHT, GRACIE
```

`options` takes an optional additional argument, `arg_desc`. If present, `arg_desc` will appear in place of `arg` in the option's online help.

```
@options([make_option('-t', '--train', action='store_true', help='by train')],
         arg_desc='(from city) (to city)')
def do_travel(self, arg, opts=None):
    'Gets you from (from city) to (to city).'
```

```
(Cmd) help travel
Gets you from (from city) to (to city).
Usage: travel [options] (from-city) (to-city)

Options:
  -h, --help   show this help message and exit
  -t, --train  by train
```

Controlling how arguments are parsed for commands with flags

There are three functions which can globally effect how arguments are parsed for commands with flags:

Note: Since optparse has been deprecated since Python 3.2, the `cmd2` developers plan to replace optparse with argparse at some point in the future. We will endeavor to keep the API as identical as possible when this change occurs.

---

poutput, pfeedback, perror

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')`, and `self.perror('errmsg')` instead. These methods have these advantages:

- **More concise**
  - `.pfeedback()` destination is controlled by *quiet* parameter.

color

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

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.

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`).

```
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!
```

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.

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__':
    app = Cmd2EventBased()
    app.preloop()
```

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

    app.postloop()
```

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

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.

Application Lifecycle Hook Methods

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

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

Command Processing Hooks

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

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.

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 2.7 and 3.3+.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search