cmd2 Documentation

Release 1.3

Catherine Devlin and Todd Leonhardt

Sep 18, 2020

Contents

1 Getting Started 3

2 Migrating from cmd 13

3 Features 19

4 Examples 83

5 Plugins 85

6 Testing 87

7 API Reference 89

8 Meta 133

Python Module Index 137

Index 139

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()
```

CHAPTER 1

Getting Started

Building a new REPL or Command Line Interface application?

Already built an application that uses cmd from the python standard library and want to add more functionality with very little work?

`cmd2` is a powerful python library for building command line applications. Start here to find out if this library is a good fit for your needs.

- *Installation Instructions* - how to install `cmd2` and associated optional dependencies
- *First Application* - a sample application showing 8 key features of `cmd2`
- *Integrate cmd2 Into Your Project* - adding `cmd2` to your project
- *Alternatives* - other python packages that might meet your needs
- *Resources* - related links and other materials

1.1 Getting Started

1.1.1 Installation Instructions

`cmd2` works on Linux, macOS, and Windows. It requires Python 3.5 or higher, pip, and setuptools. If you've got all that, then you can just:

```
$ pip install cmd2
```

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>
```

Prerequisites

If you have Python 3 >=3.5 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
```

Install from PyPI

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

```
$ pip install cmd2
```

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

Install from GitHub

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
```

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 0.8.9 should be considered to be of unstable "beta" quality and should not be relied upon for production use. If you cannot get a version >= 0.8.9 from your OS repository, then we recommend installing from either pip or GitHub - see *Install from PyPI* or *Install from GitHub*.

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
```

macOS Considerations

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.

gnureadline Python module

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

```
$ pip install -U gnureadline
```

readline via conda

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

```
$ conda install readline
```

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

1.1.2 First Application

Here's a quick walkthrough of a simple application which demonstrates 8 features of `cmd2`:

- *Settings*
- *Commands*
- *Argument Processing*

- *Generating Output*
- *Help*
- *Shortcuts*
- *Multiline Commands*
- *History*

If you don't want to type as we go, you can download the complete source for this example.

Basic Application

First we need to create a new `cmd2` application. Create a new file `first_app.py` with the following contents:

```
#!/usr/bin/env python
"""A simple cmd2 application."""
import cmd2


class FirstApp(cmd2.Cmd):
    """A simple cmd2 application."""


if __name__ == '__main__':
    import sys
    c = FirstApp()
    sys.exit(c.cmdloop())
```

We have a new class `FirstApp` which is a subclass of *cmd2.Cmd*. When we tell python to run our file like this:

```
$ python first_app.py
```

it creates an instance of our class, and calls the *cmdloop()* method. This method accepts user input and runs commands based on that input. Because we subclassed *cmd2.Cmd*, our new app already has a bunch of features built in.

Congratulations, you have a working `cmd2` app. You can run it, and then type `quit` to exit.

Create a New Setting

Before we create our first command, we are going to add a setting to this app. `cmd2` includes robust support for *Settings*. You configure settings during object initialization, so we need to add an initializer to our class:

```
def __init__(self):
    super().__init__()

    # Make maxrepeats settable at runtime
    self.maxrepeats = 3
    self.add_settable(cmd2.Settable('maxrepeats', int, 'max repetitions for speak␣
↪command'))
```

In that initializer, the first thing to do is to make sure we initialize `cmd2`. That's what the `super().__init__()` line does. Next create an attribute to hold the setting. Finally, call the *add_settable()* method with a new instance of a *Settable()* class. Now if you run the script, and enter the `set` command to see the settings, like this:

```
$ python first_app.py
(Cmd) set
```

you will see our `maxrepeats` setting show up with it's default value of `3`.

Create A Command

Now we will create our first command, called `speak` which will echo back whatever we tell it to say. We are going to use an *argument processor* so the `speak` command can shout and talk piglatin. We will also use some built in methods for *generating output*. Add this code to `first_app.py`, so that the `speak_parser` attribute and the `do_speak()` method are part of the `CmdLineApp()` class:

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

@cmd2.with_argparser(speak_parser)
def do_speak(self, args):
    """Repeats what you tell me to."""
    words = []
    for word in args.words:
        if args.piglatin:
            word = '%s%say' % (word[1:], word[0])
        if args.shout:
            word = word.upper()
        words.append(word)
    repetitions = args.repeat or 1
    for _ in range(min(repetitions, self.maxrepeats)):
        # .poutput handles newlines, and accommodates output redirection too
        self.poutput(' '.join(words))
```

Up at the top of the script, you'll also need to add:

```
import argparse
```

There's a bit to unpack here, so let's walk through it. We created `speak_parser`, which uses the argparse module from the Python standard library to parse command line input from a user. There is nothing thus far that is specific to `cmd2`.

There is also a new method called `do_speak()`. In both cmd and `cmd2`, methods that start with `do_` become new commands, so by defining this method we have created a command called `speak`.

Note the *`with_argparser()`* decorator on the `do_speak()` method. This decorator does 3 useful things for us:

1. It tells `cmd2` to process all input for the `speak` command using the argparser we defined. If the user input doesn't meet the requirements defined by the argparser, then an error will be displayed for the user.
2. It alters our `do_speak` method so that instead of receiving the raw user input as a parameter, we receive the namespace from the argparser.
3. It creates a help message for us based on the argparser.

You can see in the body of the method how we use the namespace from the argparser (passed in as the variable `args`). We build an array of words which we will output, honoring both the `--piglatin` and `--shout` options.

At the end of the method, we use our `maxrepeats` setting as an upper limit to the number of times we will print the output.

The last thing you'll notice is that we used the `self.poutput()` method to display our output. `poutput()` is a method provided by `cmd2`, which I strongly recommend you use anytime you want to *generate output*. It provides the following benefits:

1. Allows the user to redirect output to a text file or pipe it to a shell process
2. Gracefully handles `BrokenPipeWarning` exceptions for redirected output
3. Makes the output show up in a *transcript*
4. Honors the setting to *strip embedded ansi sequences* (typically used for background and foreground colors)

Go run the script again, and try out the `speak` command. Try typing `help speak`, and you will see a lovely usage message describing the various options for the command.

With those few lines of code, we created a *command*, used an *Argument Processor*, added a nice *help message* for our users, and *generated some output*.

Shortcuts

`cmd2` has several capabilities to simplify repetitive user input: *Shortcuts, Aliases, and Macros*. Let's add a shortcut to our application. Shortcuts are character strings that can be used instead of a command name. For example, `cmd2` has support for a shortcut `!` which runs the `shell` command. So instead of typing this:

```
(Cmd) shell ls -al
```

you can type this:

```
(Cmd) !ls -al
```

Let's add a shortcut for our `speak` command. Change the `__init__()` method so it looks like this:

```
def __init__(self):
    shortcuts = cmd2.DEFAULT_SHORTCUTS
    shortcuts.update({'&': 'speak'})
    super().__init__(shortcuts=shortcuts)

    # Make maxrepeats settable at runtime
    self.maxrepeats = 3
    self.add_settable(cmd2.Settable('maxrepeats', int, 'max repetitions for speak␣
↪command'))
```

Shortcuts are passed to the `cmd2` initializer, and if you want the built-in shortcuts of `cmd2` you have to pass them. These shortcuts are defined as a dictionary, with the key being the shortcut, and the value containing the command. When using the default shortcuts and also adding your own, it's a good idea to use the `.update()` method to modify the dictionary. This way if you add a shortcut that happens to already be in the default set, yours will override, and you won't get any errors at runtime.

Run your app again, and type:

```
(Cmd) shortcuts
```

to see the list of all of the shortcuts, including the one for speak that we just created.

Multiline Commands

Some use cases benefit from the ability to have commands that span more than one line. For example, you might want the ability for your user to type in a SQL command, which can often span lines and which are terminated with a semicolon. Let's add a *multiline command* to our application. First we'll create a new command called `orate`. This code shows both the definition of our `speak` command, and the `orate` command:

```
@cmd2.with_argparser(speak_parser)
def do_speak(self, args):
    """Repeats what you tell me to."""
    words = []
    for word in args.words:
        if args.piglatin:
            word = '%s%say' % (word[1:], word[0])
        if args.shout:
            word = word.upper()
        words.append(word)
    repetitions = args.repeat or 1
    for _ in range(min(repetitions, self.maxrepeats)):
        # .poutput handles newlines, and accommodates output redirection too
        self.poutput(' '.join(words))

# orate is a synonym for speak which takes multiline input
do_orate = do_speak
```

With the new command created, we need to tell `cmd2` to treat that command as a multi-line command. Modify the super initialization line to look like this:

```
super().__init__(multiline_commands=['orate'], shortcuts=shortcuts)
```

Now when you run the example, you can type something like this:

```
(Cmd) orate O for a Muse of fire, that would ascend
> The brightest heaven of invention,
> A kingdom for a stage, princes to act
> And monarchs to behold the swelling scene! ;
```

Notice the prompt changes to indicate that input is still ongoing. `cmd2` will continue prompting for input until it sees an unquoted semicolon (the default multi-line command termination character).

History

`cmd2` tracks the history of the commands that users enter. As a developer, you don't need to do anything to enable this functionality, you get it for free. If you want the history of commands to persist between invocations of your application, you'll need to do a little work. The *History* page has all the details.

Users can access command history using two methods:

- the readline library which provides a python interface to the GNU readline library
- the `history` command which is built-in to `cmd2`

From the prompt in a `cmd2`-based application, you can press `Control-p` to move to the previously entered command, and `Control-n` to move to the next command. You can also search through the command history using `Control-r`. The GNU Readline User Manual has all the details, including all the available commands, and instructions for customizing the key bindings.

The `history` command allows a user to view the command history, and select commands from history by number, range, string search, or regular expression. With the selected commands, users can:

- re-run the commands
- edit the selected commands in a text editor, and run them after the text editor exits
- save the commands to a file
- run the commands, saving both the commands and their output to a file

Learn more about the `history` command by typing `history -h` at any `cmd2` input prompt, or by exploring *Command History For Users*.

Conclusion

You've just created a simple, but functional command line application. With minimal work on your part, the application leverages many robust features of `cmd2`. To learn more you can:

- Dive into all of the *Features* that `cmd2` provides
- Look at more *Examples*
- Browse the *API Reference*

1.1.3 Integrate cmd2 Into Your Project

Once installed, you will want to ensure that your project's dependencies include `cmd2`. Make sure your `setup.py` includes the following:

```
install_requires=[
  'cmd2>=1,<2',
]
```

The `cmd2` project uses Semantic Versioning, which means that any incompatible API changes will be release with a new major version number. The public API is documented in the *API Reference*.

We recommend that you follow the advice given by the Python Packaging User Guide related to install_requires. By setting an upper bound on the allowed version, you can ensure that your project does not inadvertently get installed with an incompatible future version of `cmd2`.

Windows Considerations

If you would like to use *Completion*, and you want your application to run on Windows, you will need to ensure you install the `pyreadline` package. Make sure to include the following in your `setup.py`:

```
install_requires=[
  'cmd2>=1,<2',
  ":sys_platform=='win32'": ['pyreadline'],
]
```

1.1.4 Alternatives

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.

1.1.5 Resources

Project related links and other resources:

- cmd
- cmd2 project page
- project bug tracker
- PyOhio 2019: slides, video, examples

Building a new REPL or Command Line Interface application?

Already built an application that uses cmd from the python standard library and want to add more functionality with very little work?

`cmd2` is a powerful python library for building command line applications. Start here to find out if this library is a good fit for your needs.

- *Installation Instructions* - how to install `cmd2` and associated optional dependencies
- *First Application* - a sample application showing 8 key features of `cmd2`
- *Integrate cmd2 Into Your Project* - adding `cmd2` to your project
- *Alternatives* - other python packages that might meet your needs
- *Resources* - related links and other materials

CHAPTER 2

Migrating from cmd

If you're thinking of migrating your cmd app to `cmd2`, this section will help you decide if it's right for your app, and show you how to do it.

- *Why cmd2* - we try and convince you to use `cmd2` instead of cmd
- *Incompatibilities* - `cmd2` is not quite 100% compatible with cmd.
- *Minimum Required Changes* - the minimum changes required to move from cmd to `cmd2`. Start your migration here.
- *Next Steps* - Once you've migrated, here a list of things you can do next to add even more functionality to your app.

2.1 Migrating From cmd

2.1.1 Why cmd2

cmd

cmd is the Python Standard Library's module for creating simple interactive command-line applications. cmd is an extremely bare-bones framework which leaves a lot to be desired. It doesn't even include a built-in way to exit from an application!

Since the API provided by cmd provides the foundation on which `cmd2` is based, 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.

cmd2

`cmd2` is a batteries-included extension of cmd, which provides a wealth of functionality to make it quicker and easier for developers to create feature-rich interactive command-line applications which delight customers.

cmd2 can be used as a drop-in replacement for cmd with a few minor discrepancies as discussed in the *Incompatibilities* section. Simply importing cmd2 in place of cmd will add many features to an application without any further modifications. Migrating to cmd2 will also open many additional doors for making it possible for developers to provide a top-notch interactive command-line experience for their users.

Free Features

After switching from cmd to cmd2, your application will have the following new features and capabilities, without you having to do anything:

- More robust *History*. Both cmd and cmd2 have readline history, but cmd2 also has a robust `history` command which allows you to edit prior commands in a text editor of your choosing, re-run multiple commands at a time, and save prior commands as a script to be executed later.
- Users can redirect output to a file or pipe it to some other operating system command. You did remember to use `self.stdout` instead of `sys.stdout` in all of your print functions, right? If you did, then this will work out of the box. If you didn't, you'll have to go back and fix them. Before you do, you might consider the various ways cmd2 has of *Generating Output*.
- Users can load script files, which contain a series of commands to be executed.
- Users can create *Shortcuts, Aliases, and Macros* to reduce the typing required for repetitive commands.
- Embedded python shell allows a user to execute python code from within your cmd2 app. How meta.
- *Clipboard Integration* allows you to save command output to the operating system clipboard.
- A built-in *Timer* can show how long it takes a command to execute
- A *Transcript* is a file which contains both the input and output of a successful session of a cmd2-based app. The transcript can be played back into the app as a unit test.

Next Steps

In addition to the features you get with no additional work, cmd2 offers a broad range of additional capabilties which can be easily added to your application. *Next Steps* has some ideas of where you can start, or you can dig in to all the *Features*.

2.1.2 Incompatibilities

cmd2 strives to be drop-in compatible with cmd, however there are a few incompatibilities.

Cmd.emptyline()

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. cmd2 completely ignores empty lines and the base class `cmd.emptyline()` method never gets called and thus the empty line behavior cannot be overridden.

Cmd.identchars

In cmd, the Cmd.identchars attribute contains the string of characters accepted for command names. cmd uses those characters to split the first "word" of the input, without requiring the user to type a space. For example, if

`identchars` contained a string of all alphabetic characters, the user could enter a command like `L20` and it would be interpreted as the command `L` with the first argument of `20`.

Since version 0.9.0, `cmd2` has ignored `identchars`; the parsing logic in `cmd2` splits the command and arguments on whitespace. We opted for this breaking change because while cmd supports unicode, using non-ascii unicode characters in command names while simultaneously using `identchars` functionality can be somewhat painful. Requiring white space to delimit arguments also ensures reliable operation of many other useful `cmd2` features, including *Completion* and *Shortcuts, Aliases, and Macros*.

If you really need this functionality in your app, you can add it back in by writing a *Postparsing Hook*.

Cmd.cmdqueue

In cmd, the Cmd.cmdqueue attribute contains a list of queued input lines. The cmdqueue list is checked in `cmdloop()` when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

Since version 0.9.13 `cmd2` has removed support for `Cmd.cmdqueue`. Because `cmd2` supports running commands via the main `cmdloop()`, text scripts, Python scripts, transcripts, and history replays, the only way to preserve consistent behavior across these methods was to eliminate the command queue. Additionally, reasoning about application behavior is much easier without this queue present.

2.1.3 Minimum Required Changes

`cmd2.Cmd` subclasses `Cmd.cmd` from the standard library, and overrides most of the methods. Most apps based on the standard library can be migrated to `cmd2` in just a couple of minutes.

Import and Inheritance

You need to change your import from this:

```
import cmd
```

to this:

```
import cmd2
```

Then you need to change your class definition from:

```
class CmdLineApp(cmd.Cmd):
```

to:

```
class CmdLineApp(cmd2.Cmd):
```

Exiting

Have a look at the commands you created to exit your application. You probably have one called `exit` and maybe a similar one called `quit`. You also might have implemented a `do_EOF()` method so your program exits like many operating system shells. If all these commands do is quit the application, you may be able to remove them. See *Exiting*.

Distribution

If you are distributing your application, you'll also need to ensure that `cmd2` is properly installed. You will need to add this to your `setup()` method in `setup.py`:

```
install_requires=[
    'cmd2>=1,<2`
]
```

See *Integrate cmd2 Into Your Project* for more details.

2.1.4 Next Steps

Once your current application is using `cmd2`, you can start to expand the functionality by levering other `cmd2` features. The three ideas here will get you started. Browse the rest of the *Features* to see what else `cmd2` can help you do.

Argument Parsing

For all but the simplest of commands, it's probably easier to use argparse to parse user input. `cmd2` provides a `@with_argparser()` decorator which associates an `ArgumentParser` object with one of your commands. Using this method will:

1. Pass your command a Namespace containing the arguments instead of a string of text.
2. Properly handle quoted string input from your users.
3. Create a help message for you based on the `ArgumentParser`.
4. Give you a big headstart adding *Completion* to your application.
5. Make it much easier to implement subcommands (i.e. `git` has a bunch of subcommands such as `git pull`, `git diff`, etc).

There's a lot more about *Argument Processing* if you want to dig in further.

Help

If you have lot of commands in your application, `cmd2` can categorize those commands using a one line decorator `@with_category()`. When a user types `help` the available commands will be organized by the category you specified.

If you were already using `argparse` or decided to switch to it, you can easily *standardize all of your help messages* to be generated by your argument parsers and displayed by `cmd2`. No more help messages that don't match what the code actually does.

Generating Output

If your program generates output by printing directly to `sys.stdout`, you should consider switching to *`poutput()`*, *`perror()`*, and *`pfeedback()`*. These methods work with several of the built in *Settings* to allow the user to view or suppress feedback (i.e. progress or status output). They also properly handle ansi colored output according to user preference. Speaking of colored output, you can use any color library you want, or use the included *`cmd2.ansi.style()`* function. These and other related topics are covered in *Generating Output*.

If you're thinking of migrating your cmd app to `cmd2`, this section will help you decide if it's right for your app, and show you how to do it.

- *Why cmd2* - we try and convince you to use `cmd2` instead of cmd
- *Incompatibilities* - `cmd2` is not quite 100% compatible with cmd.
- *Minimum Required Changes* - the minimum changes required to move from cmd to `cmd2`. Start your migration here.
- *Next Steps* - Once you've migrated, here a list of things you can do next to add even more functionality to your app.

CHAPTER 3

Features

3.1 Features

3.1.1 Argument Processing

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

1. Parsing input and quoted strings like the Unix shell
2. Parse the resulting argument list using an instance of `argparse.ArgumentParser` that you provide
3. Passes the resulting `argparse.Namespace` object to your command function. The `Namespace` includes the `Statement` object that was created when parsing the command line. It can be retrieved by calling `cmd2_statement.get()` on the `Namespace`.
4. Adds the usage message from the argument parser to your command.
5. Checks if the `-h/--help` option is present, and if so, display the help message for the command

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

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

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

- *cmd2.decorators.with_argparser()*
- *cmd2.decorators.with_argparser_and_unknown_args()*
- *cmd2.decorators.with_argument_list()*

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

Argument Parsing

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

Here's what it looks like:

```
import argparse
from cmd2 import with_argparser

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

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

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

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

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

Help Messages

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

With this code:

```
import argparse
from cmd2 import with_argparser
```

(continues on next page)

(continued from previous page)

```
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',
```

(continues on next page)

(continued from previous page)

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

Which yields:

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

create an html tag

positional arguments:
  tag         tag
  content     content to surround with tag

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

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

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

Argument List

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

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

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

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

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

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

```
from cmd2 import with_argument_list

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

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

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

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(dir_parser, with_unknown_args=True)
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:")
        self.do_help('dir')
        self.last_result = CommandResult('', 'Bad arguments')
        return

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

    ...
```

Using A Custom Namespace

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

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

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

For example:

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

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

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

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

Subcommands

Subcommands 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 subcommands for your command. `cmd2` will automatically traverse and tab complete subcommands for all commands using argparse.

See the subcommands example to learn more about how to use subcommands in your `cmd2` application.

Argparse Extensions

`cmd2` augments the standard `argparse.nargs` with range tuple capability:

- `nargs=(5,)` - accept 5 or more items
- `nargs=(8, 12)` - accept 8 to 12 items

`cmd2` also provides the *cmd2.argparse_custom.Cmd2ArgumentParser* class which inherits from `argparse.ArgumentParser` and improves error and help output.

Decorator Order

If you are using custom decorators in combination with either `@cmd2.with_argparser` or `@cmd2.with_argparser_and_unknown_args`, then the order of your custom decorator(s) relative to the `cmd2` decorator matters when it comes to runtime behavior and `argparse` errors. There is nothing `cmd2`-specific here, this is just a side-effect of how decorators work in Python. To learn more about how decorators work, see decorator_primer.

If you want your custom decorator's runtime behavior to occur in the case of an `argparse` error, then that decorator needs to go **after** the `argparse` one, e.g.:

```
@cmd2.with_argparser(foo_parser)
@my_decorator
def do_foo(self, args: argparse.Namespace) -> None:
    """foo docs"""
    pass
```

However, if you do NOT want the custom decorator runtime behavior to occur even in the case of an *argparse* error, then that decorator needs to go **before** the `arpgarse` one, e.g.:

```
@my_decorator
@cmd2.with_argparser(bar_parser)
def do_bar(self, args: argparse.Namespace) -> None:
    """bar docs"""
    pass
```

The help_categories example demonstrates both above cases in a concrete fashion.

Reserved Argument Names

`cmd2` argparse decorators add the following attributes to argparse Namespaces. To avoid naming collisions, do not use any of the names for your argparse arguments.

- `cmd2_statement` - `cmd2.Cmd2AttributeWrapper` object containing `cmd2.Statement` object that was created when parsing the command line.
- `__statement__` - `cmd2.Statement` object that was created when parsing the command line. (This is deprecated and will be removed in 2.0.0.) Use `cmd2_statement` instead.
- `__subcmd_handler__` - used by cmd2 to identify the handler for a subcommand created with `@cmd2.as_subcommand_to` decorator.
- `cmd2_handler` - `cmd2.Cmd2AttributeWrapper` object containing a subcommand handler function or `None` if one was not set.

3.1.2 Builtin Commands

Applications which subclass *cmd2.Cmd* inherit a number of commands which may be useful to your users. Developers can *Remove Builtin Commands* if they do not want them to be part of the application.

List of Builtin Commands

alias

This command manages aliases via subcommands `create`, `delete`, and `list`. See *Aliases* for more information.

edit

This command launches an editor program and instructs it to open the given file name. Here's an example:

```
(Cmd) edit ~/.ssh/config
```

The program to be launched is determined by the value of the *editor* setting.

help

This command lists available commands or provides detailed help for a specific command. When called with the `-v/--verbose` argument, it shows a brief description of each command. See *Help* for more information.

history

This command allows you to view, run, edit, save, or clear previously entered commands from the history. See *History* for more information.

ipy

This optional opt-in command enters an interactive IPython shell. See *IPython (optional)* for more information.

macro

This command manages macros via subcommands `create`, `delete`, and `list`. A macro is similar to an alias, but it can contain argument placeholders. See *Macros* for more information.

py

This command invokes a Python command or shell. See *Embedded Python Shells* for more information.

quit

This command exits the `cmd2` application.

run_pyscript

This command runs a Python script file inside the `cmd2` application. See *Python Scripts* for more information.

run_script

This command runs commands in a script file that is encoded as either ASCII or UTF-8 text. See *Command Scripts* for more information.

_relative_run_script

This command is hidden from the help that's visible to end users. It runs a script like *run_script* but does so using a path relative to the script that is currently executing. This is useful when you have scripts that run other scripts. See *Running Command Scripts* for more information.

set

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

```
(Cmd) set --verbose
allow_style: 'Terminal'   # Allow ANSI text style sequences in output (valid values:␣
↪Terminal, Always, Never)
always_show_hint: False   # Display tab completion hint even when completion␣
↪suggestions print
```

(continues on next page)

(continued from previous page)

```
debug: True                # Show full traceback on exception
echo: False                # Echo command issued into output
editor: 'vi'               # Program used by 'edit'
feedback_to_output: False # Include nonessentials in '|', '>' results
max_completion_items: 50  # Maximum number of CompletionItems to display during tab␣
↪completion
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:

```
(Cmd) set allow_style Never
```

See *Settings* for more information.

shell

Execute a command as if at the operating system shell prompt:

```
(Cmd) shell pwd -P
/usr/local/bin
```

shortcuts

This command lists available shortcuts. See *Shortcuts* for more information.

Remove Builtin Commands

Developers may not want to offer the commands builtin to *cmd2.Cmd* to users of their application. To remove a command you must delete the method implementing that command from the *cmd2.Cmd* object at runtime. For example, if you wanted to remove the *shell* command from your application:

```
class NoShellApp(cmd2.Cmd):
    """A simple cmd2 application."""

    delattr(cmd2.Cmd, 'do_shell')
```

3.1.3 Clipboard Integration

Nearly every operating system has some notion of a short-term storage area which can be accessed by any program. Usually this is called the clipboard, but sometimes people refer to it as the paste buffer.

`cmd2` integrates with the operating system clipboard using the pyperclip module. Command output can be sent to the clipboard by ending the command with a greater than symbol:

```
mycommand args >
```

Think of it as though you are redirecting output to an unnamed, ephemeral place, you know, like the clipboard. You can also append output to the current contents of the clipboard by ending the command with two greater than symbols:

```
mycommand arg1 arg2 >>
```

Developers

If you would like your cmd2 based application to be able to use the clipboard in additional or alternative ways, you can use the following methods (which work uniformly on Windows, macOS, and Linux).

This module provides basic ability to copy from and paste to the clipboard/pastebuffer.

cmd2.clipboard.**get_paste_buffer**() → str
Get the contents of the clipboard / paste buffer.

Returns contents of the clipboard

cmd2.clipboard.**write_to_paste_buffer**(*txt: str*) → None
Copy text to the clipboard / paste buffer.

Parameters **txt** – text to copy to the clipboard

3.1.4 Commands

cmd2 is designed to make it easy for you to create new commands. These commmands form the backbone of your application. If you started writing your application using cmd, all the commands you have built will work when you move to cmd2. However, there are many more capabilities available in cmd2 which you can take advantage of to add more robust features to your commands, and which makes your commands easier to write. Before we get to all the good stuff, let's briefly discuss how to create a new command in your application.

Basic Commands

The simplest cmd2 application looks like this:

```
#!/usr/bin/env python
"""A simple cmd2 application."""
import cmd2


class App(cmd2.Cmd):
    """A simple cmd2 application."""


if __name__ == '__main__':
    import sys
    c = App()
    sys.exit(c.cmdloop())
```

This application subclasses cmd2.Cmd but has no code of it's own, so all functionality (and there's quite a bit) is inherited. Lets create a simple command in this application called echo which outputs any arguments given to it. Add this method to the class:

```
def do_echo(self, line):
    self.poutput(line)
```

When you type input into the cmd2 prompt, the first space delimited word is treated as the command name. cmd2 looks for a method called do_commandname. If it exists, it calls the method, passing the rest of the user input as the first argument. If it doesn't exist cmd2 prints an error message. As a result of this behavior, the only thing you have

to do to create a new command is to define a new method in the class with the appropriate name. This is exactly how you would create a command using the cmd module which is part of the python standard library.

Note: See *Generating Output* if you are unfamiliar with the `poutput()` method.

Statements

A command is passed one argument: a string which contains all the rest of the user input. However, in `cmd2` this string is actually a `Statement` object, which is a subclass of `str` to retain backwards compatibility.

`cmd2` has a much more sophsticated parsing engine than what's included in the cmd module. This parsing handles:

- quoted arguments
- output redirection and piping
- multi-line commands
- shortcut, macro, and alias expansion

In addition to parsing all of these elements from the user input, `cmd2` also has code to make all of these items work; it's almost transparent to you and to the commands you write in your own application. However, by passing your command the `Statement` object instead of just a plain string, you can get visibility into what `cmd2` has done with the user input before your command got it. You can also avoid writing a bunch of parsing code, because `cmd2` gives you access to what it has already parsed.

A `Statement` object is a subclass of `str` that contains the following attributes:

command Name of the command called. You already know this because of the method `cmd2` called, but it can sometimes be nice to have it in a string, i.e. if you want your error messages to contain the command name.

args A string containing the arguments to the command with output redirection or piping to shell commands removed. It turns out that the "string" value of the `Statement` object has all the output redirection and piping clauses removed as well. Quotes remain in the string.

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 by the user.

terminator Character used to end a multiline command. You can configure multiple termination characters, and this attribute will tell you which one the user typed.

For many simple commands, like the `echo` command above, you can ignore the `Statement` object and all of it's attributes and just use the passed value as a string. You might choose to use the `argv` attribute to do more sophisticated argument processing. Before you go too far down that path, you should check out the *Argument Processing* functionality included with `cmd2`.

Return Values

Most commands should return nothing (either by omitting a `return` statement, or by `return None`. This indicates that your command is finished (with or without errors), and that `cmd2` should prompt the user for more input.

If you return `True` from a command method, that indicates to `cmd2` that it should stop prompting for user input and cleanly exit. `cmd2` already includes a `quit` command, but if you wanted to make another one called `finis` you could:

```
def do_finish(self, line):
    """Exit the application"""
    return True
```

Exit Codes

`cmd2` has basic infrastructure to support sh/ksh/csh/bash type exit codes. The `cmd2.Cmd` object sets an `exit_code` attribute to zero when it is instantiated. The value of this attribute is returned from the `cmdloop()` call. Therefore, if you don't do anything with this attribute in your code, `cmdloop()` will (almost) always return zero. There are a few built-in `cmd2` commands which set `exit_code` to `-1` if an error occurs.

You can use this capability to easily return your own values to the operating system shell:

```
#!/usr/bin/env python
"""A simple cmd2 application."""
import cmd2


class App(cmd2.Cmd):
    """A simple cmd2 application."""

def do_bail(self, line):
    """Exit the application""
    self.perror("fatal error, exiting")
    self.exit_code = 2
    return true

if __name__ == '__main__':
    import sys
    c = App()
    sys.exit(c.cmdloop())
```

If the app was run from the *bash* operating system shell, then you would see the following interaction:

```
(Cmd) bail
fatal error, exiting
$ echo $?
2
```

Exception Handling

You may choose to catch and handle any exceptions which occur in a command method. If the command method raises an exception, `cmd2` will catch it and display it for you. The *debug setting* controls how the exception is displayed. If *debug* is *false*, which is the default, `cmd2` will display the exception name and message. If *debug* is *true*, `cmd2` will display a traceback, and then display the exception name and message.

There are a few exceptions which commands can raise that do not print as described above:

- *`cmd2.exceptions.SkipPostcommandHooks`* - all postcommand hooks are skipped and no exception prints
- *`cmd2.exceptions.Cmd2ArgparseError`* - behaves like `SkipPostcommandHooks`

- `SystemExit` - `stop` will be set to `True` in an attempt to stop the command loop
- `KeyboardInterrupt` - raised if running in a text script and `stop` isn't already True to stop the script

All other `BaseExceptions` are not caught by `cmd2` and will be raised

Disabling or Hiding Commands

See *Disabling Commands* for details of how to:

- remove commands included in `cmd2`
- hide commands from the help menu
- disable and re-enable commands at runtime

Modular Commands and Loading/Unloading Commands

See *Modular Commands* for details of how to:

- Define commands in separate CommandSet modules
- Load or unload commands at runtime

3.1.5 Completion

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

- *edit*
- *run_pyscript*
- *run_script*
- *shell*

cmd2.Cmd also adds tab completion of shell commands to the *shell* command.

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

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

Included Tab Completion Functions

cmd2 provides the following tab completion functions

- *cmd2.utils.basic_complete* - helper method for tab completion against a list
- *cmd2.Cmd.path_complete* - helper method provides flexible tab completion of file system paths
  - See the paged_output example for a simple use case
  - See the python_scripting example for a more full-featured use case
- *cmd2.Cmd.delimiter_complete* - helper method for tab completion against a list but each match is split on a delimiter
  - See the basic_completion example for a demonstration of how to use this feature
- *cmd2.Cmd.flag_based_complete* - helper method for tab completion based on a particular flag preceding the token being completed
- *cmd2.Cmd.index_based_complete* - helper method for tab completion based on a fixed position in the input string
  - See the basic_completion example for a demonstration of how to use these features
  - `flag_based_complete()` and `index_based_complete()` are basic methods and should only be used if you are not familiar with argparse. The recommended approach for tab completing positional tokens and flags is to use *argparse-based* completion.

Raising Exceptions During Completion

There are times when an error occurs while tab completing and a message needs to be reported to the user. These include the following example cases:

- Reading a database to retrieve a tab completion data set failed
- A previous command line argument that determines the data set being completed is invalid
- Tab completion hints

cmd2 provides the *cmd2.utils.CompletionError* exception class for this capability. If an error occurs in which it is more desirable to display a message than a stack trace, then raise a `CompletionError`. By default, the message displays in red like an error. However, `CompletionError` has a member called `apply_style`. Set this False if the error style should not be applied. For instance, `ArgparseCompleter` sets it to False when displaying completion hints.

Tab Completion Using argparse Decorators

When using one the argparse-based *cmd2.decorators*, cmd2 provides automatic tab completion of flag names.

Tab completion of argument values can be configured by using one of five parameters to `argparse.ArgumentParser.add_argument()`

- `choices`
- `choices_function` or `choices_method`
- `completer_function` or `completer_method`

See the arg_decorators or colors example for a demonstration of how to use the `choices` parameter. See the argparse_completion example for a demonstration of how to use the `choices_function` and `choices_method`

parameters. See the arg_decorators or argparse_completion example for a demonstration of how to use the `completer_method` parameter.

When tab completing flags or argument values for a `cmd2` command using one of these decorators, `cmd2` keeps track of state so that once a flag has already previously been provided, it won't attempt to tab complete it again. When no completion results exists, a hint for the current argument will be displayed to help the user.

CompletionItem For Providing Extra Context

When tab completing things like a unique ID from a database, it can often be beneficial to provide the user with some extra context about the item being completed, such as a description. To facilitate this, `cmd2` defines the *cmd2.argparse_custom.CompletionItem* class which can be returned from any of the 4 completion functions: `choices_function`, `choices_method`, `completion_function`, or `completion_method`.

See the argparse_completion example or the implementation of the built-in *do_set()* command for demonstration of how this is used.

For More Information

See *cmd2.argparse_custom* for a more detailed discussion of argparse completion.

3.1.6 Disabling Commands

`cmd2` allows a developer to:

- remove commands included in `cmd2`
- prevent commands from appearing in the help menu (hide commands)
- disable and re-enable commands at runtime

Remove A Command

When a command has been removed, the command method has been deleted from the object. The command doesn't show up in help, and it can't be executed. This approach is appropriate if you never want a built-in command to be part of your application. Delete the command method in your initialization code:

```
class RemoveBuiltinCommand(cmd2.Cmd):
    """An app which removes a built-in command from cmd2"""

    def __init__(self):
        super().__init__()
        # To remove built-in commands entirely, delete
        # the "do_*" function from the cmd2.Cmd class
        del cmd2.Cmd.do_edit
```

Hide A Command

When a command is hidden, it won't show up in the help menu, but if the user knows it's there and types the command, it will be executed. You hide a command by adding it to the `hidden_commands` list:

```
class HiddenCommands(cmd2.Cmd):
    ""An app which demonstrates how to hide a command"""
    def __init__(self):
        super().__init__()
        self.hidden_commands.append('py')
```

As shown above, you would typically do this as part of initializing your application. If you decide you want to unhide a command later in the execution of your application, you can by doing:

```
self.hidden_commands = [cmd for cmd in self.hidden_commands if cmd != 'py']
```

You might be thinking that the list comprehension is overkill and you'd rather do something like:

```
self.hidden_commands.remove('py')
```

You may be right, but `remove()` will raise a `ValueError` if `py` isn't in the list, and it will only remove the first one if it's in the list multiple times.

Disable A Command

One way to disable a command is to add code to the command method which determines whether the command should be executed or not. If the command should not be executed, your code can print an appropriate error message and return.

`cmd2` also provides another way to accomplish the same thing. Here's a simple app which disables the `open` command if the door is locked:

```
class DisabledCommands(cmd2.Cmd):
    """An application which disables and enables commands"""

    def do_lock(self, line):
        self.disable_command('open', "you can't open the door because it is locked")
        self.poutput('the door is locked')

    def do_unlock(self, line):
        self.enable_command('open')
        self.poutput('the door is unlocked')

    def do_open(self, line):
        """open the door"""
        self.poutput('opening the door')
```

This method has the added benefit of removing disabled commands from the help menu. But, this method only works if you know in advance that the command should be disabled, and if the conditions for re-enabling it are likewise known in advance.

Disable A Category of Commands

You can group or categorize commands as shown in *Categorizing Commands*. If you do so, you can disable and enable all the commands in a category with a single method call. Say you have created a category of commands called "Server Information". You can disable all commands in that category:

```
not_connected_msg = 'You must be connected to use this command'
self.disable_category('Server Information', not_connected_msg)
```

Similarly, you can re-enable all the commands in a category:

```
self.enable_category('Server Information')
```

3.1.7 Embedded Python Shells

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 `self_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.

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

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

More Python examples:

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

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

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

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

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

The `py` command also allows you to run Python scripts via `py run('myscript.py')`. This provides a more complicated and more powerful scripting capability than that provided by the simple text file scripts discussed in *Scripting*. 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 `run_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 `run_pyscript` as opposed to `py run()`. A simple example of using `run_pyscript` is shown below along with the arg_printer script:

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

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

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

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

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
- Good built-in ipdb debugger

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

3.1.8 Generating Output

A standard `cmd` application can produce output by using either of these methods:

```
print("Greetings, Professor Falken.", file=self.stdout)
self.stdout.write("Shall we play a game?\n")
```

While you could send output directly to `sys.stdout`, *cmd2.Cmd* can be initialized with a `stdin` and `stdout` variables, which it stores as `self.stdin` and `self.stdout`. By using these variables every time you produce output, you can trivially change where all the output goes by changing how you initialize your class.

cmd2.Cmd extends this approach in a number of convenient ways. See *Output Redirection and Pipes* for information on how users can change where the output of a command is sent. In order for those features to work, the output you generate must be sent to `self.stdout`. You can use the methods described above, and everything will work fine. *cmd2.Cmd* also includes a number of output related methods which you may use to enhance the output your application produces.

Ordinary Output

The *poutput()* method is similar to the Python built-in print function. *poutput()* adds two conveniences:

1. Since users can pipe output to a shell command, it catches `BrokenPipeError` and outputs the contents of `self.broken_pipe_warning` to `stderr`. `self.broken_pipe_warning` defaults to an empty string so this method will just swallow the exception. If you want to show an error message, put it in `self.broken_pipe_warning` when you initialize *Cmd*.
2. It examines and honors the *allow_style* setting. See *Colored Output* below for more details.

Here's a simple command that shows this method in action:

```
def do_echo(self, args):
    """A simple command showing how poutput() works"""
    self.poutput(args)
```

Error Messages

When an error occurs in your program, you can display it on `sys.stderr` by calling the *perror()* method. By default this method applies *cmd2.ansi.style_error()* to the output.

Warning Messages

pwarning() is just like *perror()* but applies *cmd2.ansi.style_warning()* to the output.

Feedback

You may have the need to display information to the user which is not intended to be part of the generated output. This could be debugging information or status information about the progress of long running commands. It's not output, it's not error messages, it's feedback. If you use the *timing* setting, the output of how long it took the command to run will be output as feedback. You can use the *pfeedback()* method to produce this type of output, and several *Settings* control how it is handled.

If the *quiet* setting is `True`, then calling *pfeedback()* produces no output. If *quiet* is `False`, the *feedback_to_output* setting is consulted to determine whether to send the output to `stdout` or `stderr`.

Exceptions

If your app catches an exception and you would like to display the exception to the user, the *pexcept()* method can help. The default behavior is to just display the message contained within the exception. However, if the *debug* setting is `True`, then the entire stack trace will be displayed.

Paging Output

If you know you are going to generate a lot of output, you may want to display it in a way that the user can scroll forwards and backwards through it. If you pass all of the output to be displayed in a single call to *ppaged()*, it will be piped to an operating system appropriate shell command to page the output. On Windows, the output is piped to `more`; on Unix-like operating systems like MacOS and Linux, it is piped to `less`.

Colored Output

You can add your own ANSI escape sequences to your output which tell the terminal to change the foreground and background colors. If you want to give yourself a headache, you can generate these by hand. You could also use a Python color library like plumbum.colors, colored, or colorama. Colorama is unique because when it's running on Windows, it wraps `stdout`, looks for ANSI escape sequences, and converts them into the appropriate `win32` calls to modify the state of the terminal.

`cmd2` imports and uses Colorama and provides a number of convenience methods for generating colorized output, measuring the screen width of colorized output, setting the window title in the terminal, and removing ANSI text style escape codes from a string. These functions are all documentated in *cmd2.ansi*.

After adding the desired escape sequences to your output, you should use one of these methods to present the output to the user:

- *cmd2.Cmd.poutput()*
- *cmd2.Cmd.perror()*
- *cmd2.Cmd.pwarning()*
- *cmd2.Cmd.pexcept()*
- *cmd2.Cmd.pfeedback()*
- *cmd2.Cmd.ppaged()*

These methods all honor the *allow_style* setting, which users can modify to control whether these escape codes are passed through to the terminal or not.

Aligning Text

If you would like to generate output which is left, center, or right aligned within a specified width or the terminal width, the following functions can help:

- *cmd2.utils.align_left()*
- *cmd2.utils.align_center()*
- *cmd2.utils.align_right()*

These functions differ from Python's string justifying functions in that they support characters with display widths greater than 1. Additionally, ANSI style sequences are safely ignored and do not count toward the display width. This means colored text is supported. If text has line breaks, then each line is aligned independently.

Columnar Output

When generating output in multiple columns, you often need to calculate the width of each item so you can pad it appropriately with spaces. However, there are categories of Unicode characters that occupy 2 cells, and other that occupy 0. To further complicate matters, you might have included ANSI escape sequences in the output to generate colors on the terminal.

The *cmd2.ansi.style_aware_wcswidth()* function solves both of these problems. Pass it a string, and regardless of which Unicode characters and ANSI text style escape sequences it contains, it will tell you how many characters on the screen that string will consume when printed.

3.1.9 Help

From our experience, end users rarely read documentation no matter how high- quality or useful that documentation might be. So it is important that you provide good built-in help within your application. Fortunately, cmd2 makes this easy.

Getting Help

cmd2 makes it easy for end users of cmd2 applications to get help via the built-in help command. The help command by itself displays a list of the commands available:

```
(Cmd) help

Documented commands (use 'help -v' for verbose/'help <topic>' for details):
============================================================================
alias  help     ipy    py    run_pyscript  set    shortcuts
edit   history  macro  quit  run_script    shell
```

The help command can also be used to provide detailed help for a specific command:

```
(Cmd) help quit
Usage: quit [-h]

Exit this application

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

Providing Help

cmd2 makes it easy for developers of cmd2 applications to provide this help. By default, the help for a command is the docstring for the do_* method defining the command - e.g. for a command **foo**, that command is implementd by defining the do_foo method and the docstring for that method is the help.

For commands which use one of the argparse decorators to parse arguments, help is provided by argparse. See *Help Messages* for more information.

Occasionally there might be an unusual circumstance where providing static help text isn't good enough and you want to provide dynamic information in the help text for a command. To meet this need, if a help_foo method is defined to match the do_foo method, then that method will be used to provide the help for command **foo**. This dynamic help is only supported for commands which do not use an argparse decorator because didn't want different output for help cmd than for cmd -h.

Categorizing Commands

By default, the `help` command displays:

```
Documented commands (use 'help -v' for verbose/'help <topic>' for details):
================================================================================
alias  help     ipy    py    run_pyscript  set    shortcuts
edit   history  macro  quit  run_script    shell
```

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 (use 'help -v' for verbose/'help <topic>' for details):

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

Command Management
==================
disable_commands  enable_commands

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

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

Other
=====
alias   edit  history  py    run_pyscript  set    shortcuts
config  help  macro    quit  run_script    shell  version
```

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

Using the `@with_category` decorator:

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

Using the `categorize()` function:

You can call with a single function:

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

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

Or with an Iterable container of functions:

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

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

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

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

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

```
Documented commands (use 'help -v' for verbose/'help <topic>' for details):

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
================================================================================
```

(continues on next page)

(continued from previous page)

```
alias                Manage aliases
config               Config command
edit                 Run a text editor and optionally open a file with it
help                 List available commands or provide detailed help for a specific␣
↪command
history              View, run, edit, save, or clear previously entered commands
macro                Manage macros
py                   Invoke Python command or shell
quit                 Exits this application
run_pyscript         Runs a python script file inside the console
run_script           Runs commands in script file that is encoded as either ASCII or␣
↪UTF-8 text
set                  Set a settable parameter or show current settings of parameters
shell                Execute a command as if at the OS prompt
shortcuts            List available shortcuts
version              Version command
```

When called with the `-v` flag for verbose help, the one-line description for each command is provided by the first line of the docstring for that command's associated `do_*` method.

3.1.10 History

For Developers

The `cmd` module from the Python standard library includes `readline` history.

cmd2.Cmd offers the same `readline` capabilities, but also maintains it's own data structures for the history of all commands entered by the user. When the class is initialized, it creates an instance of the *cmd2.history.History* class (which is a subclass of `list`) as *cmd2.Cmd.history*.

Each time a command is executed (this gets complex, see *Command Processing Loop* for exactly when) the parsed *cmd2.Statement* is appended to *cmd2.Cmd.history*.

`cmd2` adds the option of making this history persistent via optional arguments to *cmd2.Cmd.__init__()*. If you pass a filename in the `persistent_history_file` argument, the contents of *cmd2.Cmd.history* will be pickled into that history file. We chose to use pickle instead of plain text so that we can save the results of parsing all the commands.

Note: `readline` saves everything you type, whether it is a valid command or not. `cmd2` only saves input to internal history if the command parses successfully and is a valid command. This design choice was intentional, because the contents of history can be saved to a file as a script, or can be re-run. Not saving invalid input reduces unintentional errors when doing so.

However, this design choice causes an inconsistency between the `readline` history and the `cmd2` history when you enter an invalid command: it is saved to the `readline` history, but not to the `cmd2` history.

The *cmd2.Cmd.history* attribute, the *cmd2.history.History* class, and the *cmd2.history.HistoryItem* class are all part of the public API for *cmd2.Cmd*. You could use these classes to implement write your own `history` command (see below for documentation on how the included `history` command works). If you don't like pickled history, you could implement your own mechanism for saving and loading history from a plain text file.

For Users

You can use the up and down arrow keys to move through the history of previously entered commands.

If the `readline` module is installed, you can press `Control-p` to move to the previously entered command, and `Control-n` to move to the next command. You can also search through the command history using `Control-r`.

Eric Johnson hosts a nice readline cheat sheet, or you can dig into the GNU Readline User Manual for all the details, including instructions for customizing the key bindings.

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

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

If you want to display the last three commands entered:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Say you want to re-run some previously entered commands, but you would really like to make a few changes to them before doing so. When you use the `-e` or `--edit` option, `history` will write the selected commands out to a text

file, and open that file with a text editor. You make whatever changes, additions, or deletions, you want. When you leave the text editor, all the commands in the file are executed. To edit and then re-run commands 2-4 you would:

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

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

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

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

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

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

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

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

In addition to these five actions, the `history` command also has some options to control how the output is formatted. With no arguments, the `history` command displays the command number before each command. This is great when displaying history to the screen because it gives you an easy reference to identify previously entered commands. However, when creating a script or a transcript, the command numbers would prevent the script from loading properly. The `-s` or `--script` option instructs the `history` command to suppress the line numbers. This option is automatically set by the `--output_file`, `--transcript`, and `--edit` options. If you want to output the history commands with line numbers to a file, you can do it with output redirection:

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

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

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

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

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

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

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

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

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

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

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

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

3.1.11 Hooks

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

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

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

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

Application Lifecycle Hooks

You can run a script on initialization by passing the script filename in the `startup_script` parameter of *cmd2.Cmd.__init__()*.

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

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

    def myhookmethod(self) -> None:
        self.poutput("before the loop begins")
```

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

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

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

(continues on next page)

(continued from previous page)

```
    def myhookmethod(self) -> None:
        self.poutput("after the loop ends")
```

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

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

The approach of registering hooks instead of overriding methods allows multiple hooks to be called before the command loop begins or ends. Plugin authors should review *Hooks* for best practices writing hooks.

Application Lifecycle Attributes

There are numerous attributes on *`cmd2.Cmd`* which affect application behavior upon entering or during the command loop:

- *`intro`* - if provided this serves as the intro banner printed once at start of application, after *`preloop()`* is called.
- *`prompt`* - see *Prompt* for more information.
- *`continuation_prompt`* - The prompt issued to solicit input for the 2nd and subsequent lines of a *multiline command*
- *`echo`* - if `True` write the prompt and the command into the output stream.

In addition, several arguments to *`cmd2.Cmd.__init__()`* also affect the command loop behavior:

- `allow_cli_args` - allows commands to be specified on the operating system command line which are executed before the command processing loop begins.
- `transcript_files` - see *Transcripts* for more information
- `startup_script` - run a script on initialization. See *Scripting* for more information.

Command Processing Loop

When you call *`cmd2.Cmd.cmdloop()`*, the following sequence of events are repeated until the application exits:

1. Output the prompt
2. Accept user input
3. Parse user input into a *`Statement`* object
4. Call methods registered with *`register_postparsing_hook()`*
5. Redirect output, if user asked for it and it's allowed
6. Start timer
7. Call methods registered with *`register_precmd_hook()`*
8. Call *`precmd()`* - for backwards compatibility with `cmd.Cmd`
9. Add statement to *History*
10. Call *do_command* method
11. Call methods registered with *`register_postcmd_hook()`*
12. Call *`postcmd()`* - for backwards compatibility with `cmd.Cmd`

13. Stop timer and display the elapsed time
14. Stop redirecting output if it was redirected
15. Call methods registered with *register_cmdfinalization_hook()*

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

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

If a hook raises an exception:

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

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

Postparsing Hooks

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

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

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

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

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

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

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

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

The hook method must return a *cmd2.plugin.PostparsingData* object, and it is very convenient to just return the object passed into the hook method. The hook method may modify the attributes of the object to influece the behavior of the application. If `params.stop` is set to true, a fatal failure is triggered prior to execution of the command, and the application exits.

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

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

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

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

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

Precommand Hooks

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

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

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

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

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

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

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

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

Postcommand Hooks

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

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

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

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

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

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

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

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

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

To purposefully and silently skip postcommand hooks, commands can raise any of of the following exceptions.

- *cmd2.exceptions.SkipPostcommandHooks*
- *cmd2.exceptions.Cmd2ArgparseError*

Command Finalization Hooks

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

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

(continues on next page)

(continued from previous page)

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

Command Finalization hooks must check whether the *statement* attribute of the passed *CommandFinalizationData* object contains a value. There are certain circumstances where these hooks may be called before the user input has been parsed, so you can't always rely on having a *statement*.

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

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

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

3.1.12 Initialization

Here is a basic example `cmd2` application which demonstrates many capabilities which you may wish to utilize while initializing the app:

```
#!/usr/bin/env python3
# coding=utf-8
"""A simple example cmd2 application demonstrating the following:
     1) Colorizing/stylizing output
     2) Using multiline commands
     3) Persistent history
     4) How to run an initialization script at startup
     5) How to group and categorize commands when displaying them in help
     6) Opting-in to using the ipy command to run an IPython shell
     7) Allowing access to your application in py and ipy
     8) Displaying an intro banner upon starting your application
     9) Using a custom prompt
    10) How to make custom attributes settable at runtime
"""
import cmd2
from cmd2 import style, fg, bg

class BasicApp(cmd2.Cmd):
    CUSTOM_CATEGORY = 'My Custom Commands'

    def __init__(self):
        super().__init__(multiline_commands=['echo'], persistent_history_file='cmd2_
↪history.dat',
                         startup_script='scripts/startup.txt', use_ipython=True)

        # Prints an intro banner once upon application startup
        self.intro = style('Welcome to cmd2!', fg=fg.red, bg=bg.white, bold=True)

        # Show this as the prompt when asking for input
        self.prompt = 'myapp> '
```

(continues on next page)

(continued from previous page)

```
        # Used as prompt for multiline commands after the first line
        self.continuation_prompt = '... '

        # Allow access to your application in py and ipy via self
        self.self_in_py = True

        # Set the default category name
        self.default_category = 'cmd2 Built-in Commands'

        # Color to output text in with echo command
        self.foreground_color = 'cyan'

        # Make echo_fg settable at runtime
        self.add_settable(cmd2.Settable('foreground_color',
                                        str,
                                        'Foreground color to use with echo command',
                                        choices=fg.colors()))

    @cmd2.with_category(CUSTOM_CATEGORY)
    def do_intro(self, _):
        """Display the intro banner"""
        self.poutput(self.intro)

    @cmd2.with_category(CUSTOM_CATEGORY)
    def do_echo(self, arg):
        """Example of a multiline command"""
        self.poutput(style(arg, fg=self.foreground_color))


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

Cmd class initializer

A `cmd2.Cmd` instance or subclass instance is an interactive CLI application framework. There is no good reason to instantiate `Cmd` itself; rather, it's useful as a superclass of a class you define yourself in order to inherit `Cmd`'s methods and encapsulate action methods.

Certain things must be initialized within the `__init__()` method of your class derived from `cmd2.Cmd``(all arguments to ``__init__()` are optional):

`Cmd.__init__`(*completekey: str = 'tab'*, *stdin=None*, *stdout=None*, ***, *persistent_history_file: str = ''*, *persistent_history_length: int = 1000*, *startup_script: str = ''*, *use_ipython: bool = False*, *allow_cli_args: bool = True*, *transcript_files: Optional[List[str]] = None*, *allow_redirection: bool = True*, *multiline_commands: Optional[List[str]] = None*, *terminators: Optional[List[str]] = None*, *shortcuts: Optional[Dict[str, str]] = None*, *command_sets: Optional[Iterable[cmd2.command_definition.CommandSet]] = None*, *auto_load_commands: bool = True*) → None

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

Parameters

- **completekey** – readline name of a completion key, default to Tab
- **stdin** – alternate input file object, if not specified, sys.stdin is used

- **stdout** – alternate output file object, if not specified, sys.stdout is used
- **persistent_history_file** – file path to load a persistent cmd2 command history from
- **persistent_history_length** – max number of history items to write to the persistent history file
- **startup_script** – file path to a script to execute at startup
- **use_ipython** – should the "ipy" command be included for an embedded IPython shell
- **allow_cli_args** – if `True`, then *cmd2.Cmd.__init__()* will process command line arguments as either commands to be run or, if `-t` or `--test` are given, transcript files to run. This should be set to `False` if your application parses its own command line arguments.
- **transcript_files** – pass a list of transcript files to be run on initialization. This allows running transcript tests when `allow_cli_args` is `False`. If `allow_cli_args` is `True` this parameter is ignored.
- **allow_redirection** – If `False`, prevent output redirection and piping to shell commands. This parameter prevents redirection and piping, but does not alter parsing behavior. A user can still type redirection and piping tokens, and they will be parsed as such but they won't do anything.
- **multiline_commands** – list of commands allowed to accept multi-line input
- **terminators** – list of characters that terminate a command. These are mainly intended for terminating multiline commands, but will also terminate single-line commands. If not supplied, the default is a semicolon. If your app only contains single-line commands and you want terminators to be treated as literals by the parser, then set this to an empty list.
- **shortcuts** – dictionary containing shortcuts for commands. If not supplied, then defaults to constants.DEFAULT_SHORTCUTS. If you do not want any shortcuts, pass an empty dictionary.
- **command_sets** – Provide CommandSet instances to load during cmd2 initialization. This allows CommandSets with custom constructor parameters to be loaded. This also allows the a set of CommandSets to be provided when *auto_load_commands* is set to False
- **auto_load_commands** – If True, cmd2 will check for all subclasses of *CommandSet* that are currently loaded by Python and automatically instantiate and register all commands. If False, CommandSets must be manually installed with *register_command_set*.

Cmd instance attributes

The `cmd2.Cmd` class provides a large number of public instance attributes which allow developers to customize a `cmd2` application further beyond the options provided by the `__init__()` method.

Public instance attributes

Here are instance attributes of `cmd2.Cmd` which developers might wish override:

- **broken_pipe_warning**: if non-empty, this string will be displayed if a broken pipe error occurs
- **continuation_prompt**: used for multiline commands on 2nd+ line of input
- **debug**: if `True` show full stack trace on error (Default: `False`)

- **default_category**: if any command has been categorized, then all other commands that haven't been categorized will display under this section in the help output.
- **default_error**: the error that prints when a non-existent command is run
- **default_sort_key**: the default key for sorting string results. Its default value performs a case-insensitive alphabetical sort.
- **default_to_shell**: if `True` attempt to run unrecognized commands as shell commands (Default: `False`)
- **disabled_commands**: commands that have been disabled from use. This is to support commands that are only available during specific states of the application. This dictionary's keys are the command names and its values are DisabledCommand objects.
- **doc_header**: Set the header used for the help function's listing of documented functions
- **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. This behavior does not occur when running a command at the prompt. (Default: `False`)
- **editor**: text editor program to use with *edit* command (e.g. `vim`)
- **exclude_from_history**: commands to exclude from the *history* command
- **exit_code**: this determines the value returned by `cmdloop()` when exiting the application
- **feedback_to_output**: if `True` send nonessential output to stdout, if `False` send them to stderr (Default: `False`)
- **help_error**: the error that prints when no help information can be found
- **hidden_commands**: commands to exclude from the help menu and tab completion
- **last_result**: stores results from the last command run to enable usage of results in a Python script or interactive console. Built-in commands don't make use of this. It is purely there for user-defined commands and convenience.
- **self_in_py**: if `True` allow access to your application in *py* command via `self` (Default: `False`)
- **macros**: dictionary of macro names and their values
- **max_completion_items**: max number of CompletionItems to display during tab completion (Default: 50)
- **pager**: sets the pager command used by the `Cmd.ppaged()` method for displaying wrapped output using a pager
- **pager_chop**: sets the pager command used by the `Cmd.ppaged()` method for displaying chopped/truncated output using a pager
- **py_bridge_name**: name by which embedded Python environments and scripts refer to the `cmd2` application by in order to call commands (Default: `app`)
- **py_locals**: dictionary that defines specific variables/functions available in Python shells and scripts (provides more fine-grained control than making everything available with **self_in_py**)
- **quiet**: if `True` then completely suppress nonessential output (Default: `False`)
- **quit_on_sigint**: if `True` Ctrl-C at the prompt will quit the program instead of just resetting prompt
- **settable**: dictionary that controls which of these instance attributes are settable at runtime using the *set* command
- **timing**: if `True` display execution time for each command (Default: `False`)

3.1.13 Miscellaneous Features

Timer

Turn the timer setting on, and `cmd2` will show the wall time it takes for each command to execute.

Exiting

Mention quit, and EOF handling built into `cmd2`.

select

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

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

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

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

Argument `opts` can be:

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

a list of strings -> will be offered as options

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

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

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

Disabling Commands

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

enable_command() Enable an individual command

enable_category() Enable an entire category of commands

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

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

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

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

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
```

Quit on SIGINT

On many shells, SIGINT (most often triggered by the user pressing Ctrl+C) while at the prompt only cancels the current line, not the entire command loop. By default, a `cmd2` application matches this behavior. However, if `quit_on_sigint` is set to `True`, the command loop will quit instead.

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

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

3.1.14 Modular Commands

Overview

Cmd2 also enables developers to modularize their command definitions into `CommandSet` objects. CommandSets represent a logical grouping of commands within an cmd2 application. By default, all CommandSets will be discovered and loaded automatically when the cmd2.Cmd class is instantiated with this mixin. This also enables the developer to dynamically add/remove commands from the cmd2 application. This could be useful for loadable plugins that add additional capabilities. Additionally, it allows for object-oriented encapsulation and garbage collection of state that is specific to a CommandSet.

Features

- Modular Command Sets - Commands can be broken into separate modules rather than in one god class holding all commands.

- Automatic Command Discovery - In your application, merely defining and importing a CommandSet is sufficient for cmd2 to discover and load your command. No manual registration is necessary.
- Dynamically Loadable/Unloadable Commands - Command functions and CommandSets can both be loaded and unloaded dynamically during application execution. This can enable features such as dynamically loaded modules that add additional commands.
- Events handlers - Four event handlers are provided in `CommandSet` class for custom initialization and cleanup steps. See *Event Handlers*.
- Subcommand Injection - Subcommands can be defined separately from the base command. This allows for a more action-centric instead of object-centric command system while still organizing your code and handlers around the objects being managed.

See API documentation for *cmd2.command_definition.CommandSet*

See the examples for more details: https://github.com/python-cmd2/cmd2/tree/master/plugins/command_sets/examples

Defining Commands

Command Sets

CommandSets group multiple commands together. The plugin will inspect functions within a `CommandSet` using the same rules as when they're defined in `cmd2.Cmd`. Commands must be prefixed with `do_`, help functions with `help_`, and completer functions with `complete_`.

A new decorator `with_default_category` is provided to categorize all commands within a CommandSet in the same command category. Individual commands in a CommandSet may be override the default category by specifying a specific category with `cmd.with_category`.

CommandSet command methods will always expect the same parameters as when defined in a `cmd2.Cmd` sub-class, except that `self` will now refer to the `CommandSet` instead of the cmd2 instance. The cmd2 instance can be accessed through `self._cmd` that is populated when the `CommandSet` is registered.

CommandSets will only be auto-loaded if the constructor takes no arguments. If you need to provide constructor arguments, see *Manual CommandSet Construction*

```
import cmd2
from cmd2 import CommandSet, with_default_category

@with_default_category('My Category')
class AutoLoadCommandSet(CommandSet):
    def __init__(self):
        super().__init__()

    def do_hello(self, _: cmd2.Statement):
        self._cmd.poutput('Hello')

    def do_world(self, _: cmd2.Statement):
        self._cmd.poutput('World')

class ExampleApp(cmd2.Cmd):
    """
    CommandSets are automatically loaded. Nothing needs to be done.
    """
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
    def do_something(self, arg):
        self.poutput('this is the something command')
```

Manual CommandSet Construction

If a CommandSet class requires parameters to be provided to the constructor, you man manually construct CommandSets and pass in the constructor to Cmd2.

```
import cmd2
from cmd2 import CommandSet, with_default_category

@with_default_category('My Category')
class CustomInitCommandSet(CommandSet):
    def __init__(self, arg1, arg2):
        super().__init__()

        self._arg1 = arg1
        self._arg2 = arg2

    def do_show_arg1(self, _: cmd2.Statement):
        self._cmd.poutput('Arg1: ' + self._arg1)

    def do_show_arg2(self, _: cmd2.Statement):
        self._cmd.poutput('Arg2: ' + self._arg2)

class ExampleApp(cmd2.Cmd):
    """
    CommandSets with constructor parameters are provided in the constructor
    """
    def __init__(self, *args, **kwargs):
        # gotta have this or neither the plugin or cmd2 will initialize
        super().__init__(*args, **kwargs)

    def do_something(self, arg):
        self.last_result = 5
        self.poutput('this is the something command')


def main():
    my_commands = CustomInitCommandSet(1, 2)
    app = ExampleApp(command_sets=[my_commands])
    app.cmdloop()
```

Dynamic Commands

You man also dynamically load and unload commands by installing and removing CommandSets at runtime. For example, if you could support runtime loadable plugins or add/remove commands based on your state.

You may need to disable command auto-loading if you need dynamically load commands at runtime.

```
import argparse
import cmd2
```

(continues on next page)

(continued from previous page)

```
from cmd2 import CommandSet, with_argparser, with_category, with_default_category


@with_default_category('Fruits')
class LoadableFruits(CommandSet):
    def __init__(self):
        super().__init__()

    def do_apple(self, _: cmd2.Statement):
        self._cmd.poutput('Apple')

    def do_banana(self, _: cmd2.Statement):
        self._cmd.poutput('Banana')


@with_default_category('Vegetables')
class LoadableVegetables(CommandSet):
    def __init__(self):
        super().__init__()

    def do_arugula(self, _: cmd2.Statement):
        self._cmd.poutput('Arugula')

    def do_bokchoy(self, _: cmd2.Statement):
        self._cmd.poutput('Bok Choy')


class ExampleApp(cmd2.Cmd):
    """
    CommandSets are loaded via the `load` and `unload` commands
    """

    def __init__(self, *args, **kwargs):
        # gotta have this or neither the plugin or cmd2 will initialize
        super().__init__(*args, auto_load_commands=False, **kwargs)

        self._fruits = LoadableFruits()
        self._vegetables = LoadableVegetables()

    load_parser = cmd2.Cmd2ArgumentParser()
    load_parser.add_argument('cmds', choices=['fruits', 'vegetables'])

    @with_argparser(load_parser)
    @with_category('Command Loading')
    def do_load(self, ns: argparse.Namespace):
        if ns.cmds == 'fruits':
            try:
                self.register_command_set(self._fruits)
                self.poutput('Fruits loaded')
            except ValueError:
                self.poutput('Fruits already loaded')

        if ns.cmds == 'vegetables':
            try:
                self.register_command_set(self._vegetables)
                self.poutput('Vegetables loaded')
            except ValueError:
```

(continues on next page)

(continued from previous page)

```
                self.poutput('Vegetables already loaded')

    @with_argparser(load_parser)
    def do_unload(self, ns: argparse.Namespace):
        if ns.cmds == 'fruits':
            self.unregister_command_set(self._fruits)
            self.poutput('Fruits unloaded')

        if ns.cmds == 'vegetables':
            self.unregister_command_set(self._vegetables)
            self.poutput('Vegetables unloaded')


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

Event Handlers

The following functions are called at different points in the `CommandSet` life cycle.

`on_register(self, cmd) -> None` - Called by cmd2.Cmd as the first step to registering a CommandSet. The commands defined in this class have not be added to the CLI object at this point. Subclasses can override this to perform any initialization requiring access to the Cmd object (e.g. configure commands and their parsers based on CLI state data).

`on_registered(self) -> None` - Called by cmd2.Cmd after a CommandSet is registered and all its commands have been added to the CLI. Subclasses can override this to perform custom steps related to the newly added commands (e.g. setting them to a disabled state).

`on_unregister(self) -> None` - Called by `cmd2.Cmd` as the first step to unregistering a CommandSet. Subclasses can override this to perform any cleanup steps which require their commands being registered in the CLI.

`on_unregistered(self) -> None` - Called by `cmd2.Cmd` after a CommandSet has been unregistered and all its commands removed from the CLI. Subclasses can override this to perform remaining cleanup steps.

Injecting Subcommands

Description

Using the *with_argparse* decorator, it is possible to define subcommands for your command. This has a tendency to either drive your interface into an object-centric interface. For example, imagine you have a tool that manages your media collection and you want to manage movies or shows. An object-centric approach would push you to have base commands such as *movies* and *shows* which each have subcommands *add*, *edit*, *list*, *delete*. If you wanted to present an action-centric command set, so that *add*, *edit*, *list*, and *delete* are the base commands, you'd have to organize your code around these similar actions rather than organizing your code around similar objects being managed.

Subcommand injection allows you to inject subcommands into a base command to present an interface that is sensible to a user while still organizing your code in whatever structure make more logical sense to the developer.

Example

This example is a variation on the Dynamic Commands example above. A *cut* command is introduced as a base command and each CommandSet

```
import argparse
import cmd2
from cmd2 import CommandSet, with_argparser, with_category, with_default_category


@with_default_category('Fruits')
class LoadableFruits(CommandSet):
    def __init__(self):
        super().__init__()

    def do_apple(self, _: cmd2.Statement):
        self._cmd.poutput('Apple')

    banana_parser = cmd2.Cmd2ArgumentParser()
    banana_parser.add_argument('direction', choices=['discs', 'lengthwise'])

    @cmd2.as_subcommand_to('cut', 'banana', banana_parser)
    def cut_banana(self, ns: argparse.Namespace):
        """Cut banana"""
        self._cmd.poutput('cutting banana: ' + ns.direction)


@with_default_category('Vegetables')
class LoadableVegetables(CommandSet):
    def __init__(self):
        super().__init__()

    def do_arugula(self, _: cmd2.Statement):
        self._cmd.poutput('Arugula')

    bokchoy_parser = cmd2.Cmd2ArgumentParser()
    bokchoy_parser.add_argument('style', choices=['quartered', 'diced'])

    @cmd2.as_subcommand_to('cut', 'bokchoy', bokchoy_parser)
    def cut_bokchoy(self, _: argparse.Namespace):
        self._cmd.poutput('Bok Choy')


class ExampleApp(cmd2.Cmd):
    """
    CommandSets are automatically loaded. Nothing needs to be done.
    """

    def __init__(self, *args, **kwargs):
        # gotta have this or neither the plugin or cmd2 will initialize
        super().__init__(*args, auto_load_commands=False, **kwargs)

        self._fruits = LoadableFruits()
        self._vegetables = LoadableVegetables()

    load_parser = cmd2.Cmd2ArgumentParser()
    load_parser.add_argument('cmds', choices=['fruits', 'vegetables'])
```

(continues on next page)

(continued from previous page)

```
    @with_argparser(load_parser)
    @with_category('Command Loading')
    def do_load(self, ns: argparse.Namespace):
        if ns.cmds == 'fruits':
            try:
                self.register_command_set(self._fruits)
                self.poutput('Fruits loaded')
            except ValueError:
                self.poutput('Fruits already loaded')

        if ns.cmds == 'vegetables':
            try:
                self.register_command_set(self._vegetables)
                self.poutput('Vegetables loaded')
            except ValueError:
                self.poutput('Vegetables already loaded')

    @with_argparser(load_parser)
    def do_unload(self, ns: argparse.Namespace):
        if ns.cmds == 'fruits':
            self.unregister_command_set(self._fruits)
            self.poutput('Fruits unloaded')

        if ns.cmds == 'vegetables':
            self.unregister_command_set(self._vegetables)
            self.poutput('Vegetables unloaded')

    cut_parser = cmd2.Cmd2ArgumentParser()
    cut_subparsers = cut_parser.add_subparsers(title='item', help='item to cut')

    @with_argparser(cut_parser)
    def do_cut(self, ns: argparse.Namespace):
        handler = ns.cmd2_handler.get()
        if handler is not None:
            # Call whatever subcommand function was selected
            handler(ns)
        else:
            # No subcommand was provided, so call help
            self.poutput('This command does nothing without sub-parsers registered')
            self.do_help('cut')


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

3.1.15 Multiline Commands

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

In multiline commands, output redirection characters like > and | are part of the command arguments unless they

appear after the terminator.

Continuation prompt

When a user types a *Multiline Command* it may span more than one line of input. The prompt for the first line of input is specified by the *cmd2.Cmd.prompt* instance attribute - see *Customizing the Prompt*. The prompt for subsequent lines of input is defined by the *cmd2.Cmd.continuation_prompt* attribute.

Use cases

Multiline commands should probably be used sparingly in order to preserve a good user experience for your cmd2-based line-oriented command interpreter application.

However, some use cases benefit significantly from the ability to have commands that span more than one line. For example, you might want the ability for your user to type in a SQL command, which can often span lines and which are terminated with a semicolon.

We estimate that less than 5 percent of cmd2 applications use this feature. But it is here for those uses cases where it provides value.

3.1.16 Integrating with the OS

How to redirect output

See *Output Redirection and Pipes*

Executing OS commands from within cmd2

cmd2 includes a shell command which executes it's arguments in the operating system shell:

```
(Cmd) shell ls -al
```

If you use the default *Shortcuts* defined in cmd2 you'll get a ! shortcut for shell, which allows you to type:

```
(Cmd) !ls -al
```

NOTE: cmd2 provides user-friendly tab completion throughout the process of running a shell command - first for the shell command name itself, and then for file paths in the argument section.

Editors

cmd2 includes the built-in edit command which runs a text editor and optionally opens a file with it:

```
(Cmd) edit foo.txt
```

The editor used is determined by the editor settable parameter and can be either a text editor such as **vim** or a graphical editor such as **VSCode**. To set it:

```
set editor <program_name>
```

If you have the EDITOR environment variable set, then this will be the default value for editor. If not, then cmd2 will attempt to search for any in a list of common editors for your operating system.

Terminal pagers

Output of any command can be displayed one page at a time using the *ppaged()* method.

Alternatively, a terminal pager can be invoked directly using the ability to run shell commands with the `!` shortcut like so:

```
(Cmd) !less foo.txt
```

NOTE: Once you are in a terminal pager, that program temporarily has control of your terminal, **NOT** `cmd2`. Typically you can use either the arrow keys or <PageUp>/<PageDown> keys to scroll around or type `q` to quit the pager and return control to your `cmd2` application.

Exit codes

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

Invoking With Arguments

Typically you would invoke a `cmd2` program by typing:

```
$ python mycmd2program.py
```

or:

```
$ mycmd2program.py
```

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

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

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

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

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

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

```
$ python examples/example.py help

Documented commands (use 'help -v' for verbose/'help <topic>' for details):
================================================================================
alias  help     macro   orate  quit          run_script  set    shortcuts
```

(continues on next page)

(continued from previous page)

```
edit   history  mumble  py     run_pyscript  say         shell  speak

(Cmd)
```

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

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

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

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

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

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

Alternatively you can simply wrap the command plus arguments in quotes (either single or double quotes):

```
$ python example/example.py "speak -p hello there"
ellohay heretay
(Cmd)
```

Automating cmd2 apps from other CLI/CLU tools

While `cmd2` is designed to create **interactive** command-line applications which enter a Read-Evaluate-Print-Loop (REPL), there are a great many times when it would be useful to use a `cmd2` application as a run-and-done command-line utility for purposes of automation and scripting.

This is easily achieved by combining the following capabilities of `cmd2`:

1. Ability to invoke a `cmd2` application with arguments
2. Ability to set an exit code when leaving a `cmd2` application
3. Ability to exit a `cmd2` application with the `quit` command

Here is a simple example which doesn't require the quit command since the custom `exit` command quits while returning an exit code:

```
$ python examples/exit_code.py "exit 23"
'examples/exit_code.py' exiting with code: 23
$ echo $?
23
```

Here is another example using `quit`:

```
$ python example/example.py "speak -p hello there" quit
ellohay heretay
$
```

3.1.17 Packaging a cmd2 application for distribution

As a general-purpose tool for building interactive command-line applications, `cmd2` is designed to be used in many ways. How you distribute your `cmd2` application to customers or end users is up to you. See the Overview of Packaging for Python from the Python Packaging Authority for a thorough discussion of the extensive options within the Python ecosystem.

For developers wishing to package a `cmd2` application into a single binary image or compressed file, we can recommend all of the following based on personal and professional experience:

- Deploy your `cmd2` Python app using Docker * Powerful and flexible - allows you to control entire user space and setup other applications like databases * As long as it isn't problematic for your customers to have Docker installed, then this is probably the best option
- PyInstaller * Quick and easy - it "just works" and everything you need is installable via `pip` * Packages up all of the dependencies into a single directory which you can then zip up
- Nuitka * Converts your Python to C and compiles it to a native binary file * This can be particularly convenient if you wish to obfuscate the Python source code behind your application * Recommend invoking with `--follow-imports` flag like: `python3 -m nuitka --follow-imports your_app.py`
- Conda Constructor * Allows you to create a custom Python distro based on Miniconda

3.1.18 Plugins

`cmd2` has a built-in plugin framework which allows developers to create a a `cmd2` plugin which can extend basic `cmd2` functionality and can be used by multiple applications.

There are many ways to add functionality to `cmd2` using a plugin. Most plugins will be implemented as a mixin. A mixin is a class that encapsulates and injects code into another class. Developers who use a plugin in their `cmd2` project will inject the plugin's code into their subclass of *cmd2.Cmd*.

Mixin and Initialization

The following short example shows how to mix in a plugin and how the plugin gets initialized.

Here's the plugin:

```
class MyPlugin:
    def __init__(self, *args, **kwargs):
        # code placed here runs before cmd2.Cmd initializes
        super().__init__(*args, **kwargs)
        # code placed here runs after cmd2.Cmd initializes
```

and an example app which uses the plugin:

```
import cmd2
import cmd2_myplugin

class Example(cmd2_myplugin.MyPlugin, cmd2.Cmd):
    """An class to show how to use a plugin"""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, *args, **kwargs):
        # code placed here runs before cmd2.Cmd or
        # any plugins initialize
        super().__init__(*args, **kwargs)
        # code placed here runs after cmd2.Cmd and
        # all plugins have initialized
```

Note how the plugin must be inherited (or mixed in) before *cmd2.Cmd*. This is required for two reasons:

- The `cmd.Cmd.__init__` method in the python standard library does not call `super().__init__()`. Because of this oversight, if you don't inherit from `MyPlugin` first, the `MyPlugin.__init__()` method will never be called.
- You may want your plugin to be able to override methods from *cmd2.Cmd*. If you mixin the plugin after `cmd2.Cmd`, the python method resolution order will call *cmd2.Cmd* methods before it calls those in your plugin.

Add commands

Your plugin can add user visible commands. You do it the same way in a plugin that you would in a *cmd2.Cmd* app:

```
class MyPlugin:
    def do_say(self, statement):
        """Simple say command"""
        self.poutput(statement)
```

You have all the same capabilities within the plugin that you do inside a *cmd2.Cmd* app, including argument parsing via decorators and custom help methods.

Add (or hide) settings

A plugin may add user controllable settings to the application. Here's an example:

```
class MyPlugin:
    def __init__(self, *args, **kwargs):
        # code placed here runs before cmd2.Cmd initializes
        super().__init__(*args, **kwargs)
        # code placed here runs after cmd2.Cmd initializes
        self.mysetting = 'somevalue'
        self.add_settable(cmd2.Settable('mysetting', str, 'short help message for␣
↪mysetting'))
```

You can hide settings from the user by calling *remove_settable()*. See *Settings* for more information.

Decorators

Your plugin can provide a decorator which users of your plugin can use to wrap functionality around their own commands.

Override methods

Your plugin can override core *cmd2.Cmd* methods, changing their behavior. This approach should be used sparingly, because it is very brittle. If a developer chooses to use multiple plugins in their application, and several of the plugins

override the same method, only the first plugin to be mixed in will have the overridden method called.

Hooks are a much better approach.

Hooks

Plugins can register hook methods, which are called by *cmd2.Cmd* during various points in the application and command processing lifecycle. Plugins should not override any of the deprecated hook methods, instead they should register their hooks as described in the *Hooks* section.

You should name your hooks so that they begin with the name of your plugin. Hook methods get mixed into the cmd2 application and this naming convention helps avoid unintentional method overriding.

Here's a simple example:

```
class MyPlugin:
    def __init__(self, *args, **kwargs):
        # code placed here runs before cmd2 initializes
        super().__init__(*args, **kwargs)
        # code placed here runs after cmd2 initializes
        # this is where you register any hook functions
        self.register_postparsing_hook(self.cmd2_myplugin_postparsing_hook)

    def cmd2_myplugin_postparsing_hook(self, data: cmd2.plugin.PostparsingData) ->␣
↪cmd2.plugin.PostparsingData:
        """Method to be called after parsing user input, but before running the␣
↪command"""
        self.poutput('in postparsing_hook')
        return data
```

Registration allows multiple plugins (or even the application itself) to each inject code to be called during the application or command processing lifecycle.

See the *Hooks* documentation for full details of the application and command lifecycle, including all available hooks and the ways hooks can influence the lifecycle.

Classes and Functions

Your plugin can also provide classes and functions which can be used by developers of cmd2 based applications. Describe these classes and functions in your documentation so users of your plugin will know what's available.

Examples

See https://github.com/python-cmd2/cmd2-plugin-template for more info.

3.1.19 Prompt

cmd2 issues a configurable prompt before soliciting user input.

Customizing the Prompt

This prompt can be configured by setting the *cmd2.Cmd.prompt* instance attribute. This contains the string which should be printed as a prompt for user input. See the Pirate example for the simple use case of statically setting the prompt.

Continuation Prompt

When a user types a *Multiline Command* it may span more than one line of input. The prompt for the first line of input is specified by the `cmd2.Cmd.prompt` instance attribute. The prompt for subsequent lines of input is defined by the `cmd2.Cmd.continuation_prompt` attribute.See the Initialization example for a demonstration of customizing the continuation prompt.

Updating the prompt

If you wish to update the prompt between commands, you can do so using one of the *Application Lifecycle Hooks* such as a *Postcommand hook*. See PythonScripting for an example of dynamically updating the prompt.

Asynchronous Feedback

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

`Cmd.`**`async_alert`**(*alert_msg: str*, *new_prompt: Optional[str] = None*) → None

Display an important message to the user while they are at a command line prompt. To the user it appears as if an alert message is printed above the prompt and their current input text and cursor location is left alone.

Raises a *RuntimeError* if called while another thread holds *terminal_lock*.

IMPORTANT: This function will not print an alert unless it can acquire self.terminal_lock to ensure a prompt is onscreen. Therefore it is best to acquire the lock before calling this function to guarantee the alert prints and to avoid raising a RuntimeError.

Parameters

- **`alert_msg`** – the message to display to the user
- **`new_prompt`** – if you also want to change the prompt that is displayed, then include it here see async_update_prompt() docstring for guidance on updating a prompt

`Cmd.`**`async_update_prompt`**(*new_prompt: str*) → None

Update the command line prompt while the user is still typing at it. This is good for alerting the user to system changes dynamically in between commands. For instance you could alter the color of the prompt to indicate a system status or increase a counter to report an event. If you do alter the actual text of the prompt, it is best to keep the prompt the same width as what's on screen. Otherwise the user's input text will be shifted and the update will not be seamless.

Raises a *RuntimeError* if called while another thread holds *terminal_lock*.

IMPORTANT: This function will not update the prompt unless it can acquire self.terminal_lock to ensure a prompt is onscreen. Therefore it is best to acquire the lock before calling this function to guarantee the prompt changes and to avoid raising a RuntimeError.

If user is at a continuation prompt while entering a multiline command, the onscreen prompt will not change. However self.prompt will still be updated and display immediately after the multiline line command completes.

Parameters **`new_prompt`** – what to change the prompt to

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

Cmd.**set_window_title**(*title: str*) → None
Set the terminal window title.

Raises a *RuntimeError* if called while another thread holds *terminal_lock*.

IMPORTANT: This function will not set the title unless it can acquire self.terminal_lock to avoid writing to stderr while a command is running. Therefore it is best to acquire the lock before calling this function to guarantee the title changes and to avoid raising a RuntimeError.

Parameters **title** – the new window title

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

3.1.20 Output Redirection and Pipes

As in POSIX shells, output of a command can be redirected and/or piped. This feature is fully cross-platform and works identically on Windows, macOS, and Linux.

Output Redirection

Redirect to a file

Redirecting the output of a cmd2 command to a file works just like in POSIX shells:

- send to a file with >, as in `mycommand args > filename.txt`
- append to a file with >>, as in `mycommand args >> filename.txt`

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'`.

Redirect to the clipboard

cmd2 output redirection supports an additional feature not found in most shells - if the file name following the > or >> is left blank, then the output is redirected to the operating system clipboard so that it can then be pasted into another program.

- overwrite the clipboard with `mycommand args >`
- append to the clipboard with `mycommand args >>`

Pipes

Piping the output of a cmd2 command to a shell command works just like in POSIX shells:

- pipe as input to a shell command with |, as in `mycommand args | wc`

Multiple Pipes and Redirection

Multiple pipes, optionally followed by a redirect, are supported. Thus, it is possible to do something like the following:

```
(Cmd) help | grep py | wc > output.txt
```

The above runs the **help** command, pipes its output to **grep** searching for any lines containing *py*, then pipes the output of grep to the **wc** "word count" command, and finally writes redirects the output of that to a file called *output.txt*.

Disabling Redirection

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.

Limitations of Redirection

Some limitations apply to redirection and piping within `cmd2` applications:

- Can only pipe to shell commands, not other `cmd2` application commands
- **stdout** gets redirected/piped, **stderr** does not

3.1.21 Scripting

Operating system shells have long had the ability to execute a sequence of commands saved in a text file. These script files make long sequences of commands easier to repeatedly execute. `cmd2` supports two similar mechanisms: command scripts and python scripts.

Command Scripts

A command script contains a sequence of commands typed at the the prompt of a `cmd2` based application. Unlike operating system shell scripts, command scripts can't contain logic or loops.

Creating Command Scripts

Command scripts can be created in several ways:

- creating a text file using any method of your choice
- using the built-in *edit* command to create or edit an existing text file

- saving previously entered commands to a script file using *history -s*

If you create create a text file from scratch, just include one command per line, exactly as you would type it inside a `cmd2` application.

Running Command Scripts

Command script files can be executed using the built-in *run_script* command or the `@` shortcut (if your application is using the default shortcuts). Both ASCII and UTF-8 encoded unicode text files are supported. The *run_script* command supports tab completion of file system paths. There is a variant *_relative_run_script* command or `@@` shortcut (if using the default shortcuts) for use within a script which uses paths relative to the first script.

Comments

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

Comments are useful in scripts, but would be pointless within an interactive session.

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

Python Scripts

If you require logic flow, loops, branching, or other advanced features, you can write a python script which executes in the context of your `cmd2` app. This script is run using the *run_pyscript* command. Here's a simple example that uses the arg_printer script:

```
(Cmd) run_pyscript examples/scripts/arg_printer.py foo bar 'baz 23'
Running Python script 'arg_printer.py' which was called with 3 arguments
arg 1: 'foo'
arg 2: 'bar'
arg 3: 'baz 23'
```

run_pyscript supports tab completion of file system paths, and as shown above it has the ability to pass command-line arguments to the scripts invoked.

Python scripts executed with *run_pyscript* can run `cmd2` application commands by using the syntax:

```
app('command args')
```

where:

- `app` is a configurable name which can be changed by setting the *cmd2.Cmd.py_bridge_name* attribute
- `command` and `args` are entered exactly like they would be entered by a user of your application.

See python_scripting example and associated conditional script for more information.

Advanced Support

When implementing a command, setting `self.last_result` allows for application-specific data to be returned to a python script from the command. This can allow python scripts to make decisions based on the result of previous

application commands.

The application command (default: `app`) returns a `cmd2.CommandResult` for each command. The `cmd2.CommandResult` object provides the captured output to `stdout` and `stderr` while a command is executing. Additionally, it provides the value that command stored in `self.last_result`.

Additionally, an external test Mixin plugin has been provided to allow for python based external testing of the application. For example, for system integration tests scenarios where the python application is a component of a larger suite of tools and components. This interface allows python based tests to call commands and validate results as part of a larger test suite. See *External Test Plugin*

3.1.22 Settings

Settings provide a mechanism for a user to control the behavior of a `cmd2` based application. A setting is stored in an instance attribute on your subclass of *cmd2.Cmd* and must also appear in the *cmd2.Cmd.settable* dictionary. Developers may set default values for these settings and users can modify them at runtime using the *set* command. Developers can *Create New Settings* and can also *Hide Builtin Settings* from the user.

Builtin Settings

`cmd2` has a number of builtin settings. These settings control the behavior of certain application features and *Builtin Commands*. Users can use the *set* command to show all settings and to modify the value of any setting.

allow_style

Output generated by `cmd2` programs may contain ANSI escape seqences which instruct the terminal to apply colors or text styling (i.e. bold) to the output. The `allow_style` setting controls the behavior of these escape sequences in output generated with any of the following methods:

- *cmd2.Cmd.poutput()*
- *cmd2.Cmd.perror()*
- *cmd2.Cmd.pwarning()*
- *cmd2.Cmd.pexcept()*
- *cmd2.Cmd.pfeedback()*
- *cmd2.Cmd.ppaged()*

This setting can be one of three values:

- `Never` - all ANSI escape sequences which instruct the terminal to style output are stripped from the output.
- `Terminal` - (the default value) pass through ANSI escape sequences when the output is being sent to the terminal, but if the output is redirected to a pipe or a file the escape sequences are stripped.
- `Always` - ANSI escape sequences are always passed through to the output

debug

The default value of this setting is `False`, which causes the *pexcept()* method to only display the message from an exception. However, if the debug setting is `True`, then the entire stack trace will be printed.

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. This behavior does not occur when running a command at the prompt.

editor

Similar to the `EDITOR` shell variable, this setting contains the name of the program which should be run by the *edit* command.

feedback_to_output

Controls whether feedback generated with the *pfeedback()* method is sent to `sys.stdout` or `sys.stderr`. If `False` the output will be sent to `sys.stderr`

If `True` the output is sent to `stdout` (which is often the screen but may be *redirected*). The feedback output will be mixed in with and indistinguishable from output generated with *poutput()*.

max_completion_items

Maximum number of CompletionItems to display during tab completion. A CompletionItem is a special kind of tab completion hint which displays both a value and description and uses one line for each hint. Tab complete the `set` command for an example.

If the number of tab completion hints exceeds `max_completion_items`, then they will be displayed in the typical columnized format and will not include the description text of the CompletionItem.

quiet

If `True`, output generated by calling *pfeedback()* is suppressed. If `False`, the *feedback_to_output* setting controls where the output is sent.

timing

If `True`, the elapsed time is reported for each command executed.

Create New Settings

Your application can define user-settable parameters which your code can reference. In your initialization code:

1. Create an instance attribute with a default value.
2. Create a *Settable* object which describes your setting.
3. Pass the *Settable* object to *cmd2.Cmd.add_settable()*.

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. """

    def __init__(self):
        super().__init__()
        self.degrees_c = 22
        self.sunny = False
        self.add_settable(cmd2.Settable('degrees_c',
                                        int,
                                        'Temperature in Celsius',
                                        onchange_cb=self._onchange_degrees_c
                                        ))
        self.add_settable(cmd2.Settable('sunny', bool, 'Is it sunny outside?'))

    def do_sunbathe(self, arg):
        """Attempt to sunbathe."""
        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, param_name, old, new):
        # if it's over 40C, it's gotta be sunny, right?
        if new > 40:
            self.sunny = True


if __name__ == '__main__':
    import sys
    c = EnvironmentApp()
    sys.exit(c.cmdloop())
```

If you want to be notified when a setting changes (as we do above), then be sure to supply a method to the `onchange_cb` parameter of the *.cmd2.utils.Settable*. This method will be called after the user changes a setting, and will receive both the old value and the new value.

```
(Cmd) set --long | grep sunny
sunny: False                # Is it sunny outside?
(Cmd) set --long | grep degrees
degrees_c: 22               # Temperature in Celsius
(Cmd) sunbathe
Too dim.
(Cmd) set degrees_c 41
degrees_c - was: 22
now: 41
(Cmd) set sunny
```

(continues on next page)

(continued from previous page)

```
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?
```

Hide Builtin Settings

You may want to prevent a user from modifying a builtin setting. A setting must appear in the *cmd2.Cmd.settable* dictionary in order for it to be available to the *set* command.

Let's say that you never want end users of your program to be able to enable full debug tracebacks to print out if an error occurs. You might want to hide the *debug* setting. To do so, remove it from the *cmd2.Cmd.settable* dictionary after you initialize your object. The *cmd2.Cmd.remove_settable()* convenience method makes this easy:

```
class MyApp(cmd2.Cmd):

  def __init__(self):
      super().__init__()
      self.remove_settable('debug')
```

3.1.23 Shortcuts, Aliases, and Macros

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

@ run script file

@@ run 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):
      shortcuts = dict(cmd2.DEFAULT_SHORTCUTS)
      shortcuts.update({'*': 'sneeze', '~': 'squirm'})
      cmd2.Cmd.__init__(self, shortcuts=shortcuts)
```

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

Note: Command, alias, and macro names cannot start with a shortcut

Aliases

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

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

Ex: alias create ls !ls -lF

Redirectors and pipes should be quoted in alias definition to prevent the alias create command from being redirected:

```
alias create save_results print_results ">" out.txt
```

Tab completion recognizes an alias, and completes as if its actual value was on the command line.

For more details run: help alias create

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

Ex: alias list

For more details run: help alias list

Use alias delete to remove aliases

For more details run: help alias delete

Note: Aliases cannot have the same name as a command or macro

Macros

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

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

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

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

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

Similar to aliases, pipes and redirectors need to be quoted in the definition of a macro:

```
macro create lc !cat "{1}" "|" less
```

To use the literal string {1} in your command, escape it this way: {{1}}. Because macros do not resolve until after hitting <Enter>, tab completion will only complete paths while typing a macro.

For more details run: help macro create

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

For more details on listing macros run: help macro list

For more details on deleting macros run: help macro delete

Note: Macros cannot have the same name as a command or alias

3.1.24 Startup Commands

cmd2 provides a couple different ways for running commands immediately after your application starts up:

1. Commands at Invocation
2. Startup Script

Commands run as part of a startup script are always run immediately after the application finishes initializing so they are guaranteed to run before any *Commands At Invocation*.

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. You can use either single or double quotes for this purpose.

```
$ python examples/example.py "say hello" "say Gracie" quit
hello
Gracie
```

You can end your commands with a **quit** command so that your cmd2 application runs like a non-interactive command-line utility (CLU). This means that it can then be scripted from an external application and easily used in automation.

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

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

Startup Script

You can execute commands from an 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')
```

This text file should contain a *Command Script*. See the AliasStartup example for a demonstration.

3.1.25 Table Creation

cmd2 provides a table creation class called *cmd2.table_creator.TableCreator*. This class handles ANSI style sequences and characters with display widths greater than 1 when performing width calculations. It was designed with the ability to build tables one row at a time. This helps when you have large data sets that you don't want to hold in memory or when you receive portions of the data set incrementally.

`TableCreator` has one public method: *cmd2.table_creator.TableCreator.generate_row()*

This function and the *cmd2.table_creator.Column* class provide all features needed to build tables with headers, borders, colors, horizontal and vertical alignment, and wrapped text. However, it's generally easier to inherit from this class and implement a more granular API rather than use `TableCreator` directly.

The following table classes build upon `TableCreator` and are provided in the *cmd2.table_creator* module. They can be used as is or as examples for how to build your own table classes.

cmd2.table_creator.SimpleTable - Implementation of TableCreator which generates a borderless table with an optional divider row after the header. This class can be used to create the whole table at once or one row at a time.

cmd2.table_creator.BorderedTable - Implementation of TableCreator which generates a table with borders around the table and between rows. Borders between columns can also be toggled. This class can be used to create the whole table at once or one row at a time.

cmd2.table_creator.AlternatingTable - Implementation of BorderedTable which uses background colors to distinguish between rows instead of row border lines. This class can be used to create the whole table at once or one row at a time.

See the table_creation example to see these classes in use

3.1.26 Transcripts

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.

Creating From History

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

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

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

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

Creating From A Script File

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

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

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

Creating Manually

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

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

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

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

Following each command is the output generated by that command.

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

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

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

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

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

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

will be ignored.

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 editor
editor: vim/ /
```

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 and Pipes* 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.

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
.
----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
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 passed an `allow_cli_args` parameter containing *False* to *cmd2.Cmd.__init__()* 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()
```

CHAPTER 4

Examples

4.1 Examples

4.1.1 Alternate Event Loops

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

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

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

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

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

```
import cmd2

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

    # ... your class code here ...

if __name__ == '__main__':
```

(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 runs multiple commands via *onecmd_plus_hooks()*.

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

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

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

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

CHAPTER 5

Plugins

5.1 Plugins

5.1.1 External Test Plugin

Overview

The External Test Plugin supports testing of a cmd2 application by exposing access cmd2 commands with the same context as from within a cmd2 *Python Scripts*. This interface captures `stdout`, `stderr`, as well as any application-specific data returned by the command. This also allows for verification of an application's support for *Python Scripts* and enables the cmd2 application to be tested as part of a larger system integration test.

Example cmd2 Application

The following short example shows how to mix in the external test plugin to create a fixture for testing your cmd2 application.

Define your cmd2 application

```
import cmd2
class ExampleApp(cmd2.Cmd):
    """An class to show how to use a plugin"""
    def __init__(self, *args, **kwargs):
        # gotta have this or neither the plugin or cmd2 will initialize
        super().__init__(*args, **kwargs)

    def do_something(self, arg):
        self.last_result = 5
        self.poutput('this is the something command')
```

Defining the test fixture

In your test, define a fixture for your cmd2 application

```
import cmd2_ext_test
import pytest

class ExampleAppTester(cmd2_ext_test.ExternalTestMixin, ExampleApp):
    def __init__(self, *args, **kwargs):
        # gotta have this or neither the plugin or cmd2 will initialize
        super().__init__(*args, **kwargs)

@pytest.fixture
def example_app():
    app = ExampleAppTester()
    app.fixture_setup()
    yield app
    app.fixture_teardown()
```

Writing Tests

Now write your tests that validate your application using the `app_cmd()` function to access the cmd2 application's commands. This allows invocation of the application's commands in the same format as a user would type. The results from calling a command matches what is returned from running an python script with cmd2's *run_pyscript* command, which provides `stdout`, `stderr`, and the command's result data.

```
from cmd2 import CommandResult

def test_something(example_app):
    # execute a command
    out = example_app.app_cmd("something")

    # validate the command output and result data
    assert isinstance(out, CommandResult)
    assert str(out.stdout).strip() == 'this is the something command'
    assert out.data == 5
```

CHAPTER 6

Testing

6.1 Testing

6.1.1 Overview

This covers special considerations when writing unit tests for a cmd2 application.

6.1.2 Testing Commands

The *External Test Plugin* provides a mixin class with an :meth:‘‘ function that allows external calls to application commands. The `app_cmd()` function captures and returns stdout, stderr, and the command-specific result data.

6.1.3 Mocking

If you need to mock anything in your cmd2 application, and most specifically in sub-classes of *Cmd* or *CommandSet*, you must use Autospeccing, spec=True, or whatever equivalant is provided in the mocking library you're using.

In order to automatically load functions as commands cmd2 performs a number of reflection calls to look up attributes of classes defined in your cmd2 application. Many mocking libraries will automatically create mock objects to match any attribute being requested, regardless of whether they're present in the object being mocked. This behavior can incorrectly instruct cmd2 to treat a function or attribute as something it needs to recognize and process. To prevent this, you should always mock with Autospeccing or spec=True enabled.

6.1.4 Example of spec=True

```
def test_mocked_methods():
    with mock.patch.object(MockMethodApp, 'foo', spec=True):
        cli = MockMethodApp()
```

CHAPTER 7

API Reference

7.1 API Reference

These pages document the public API for `cmd2`. If a method, class, function, attribute, or constant is not documented here, consider it private and subject to change. There are many classes, methods, functions, and constants in the source code which do not begin with an underscore but are not documented here. When looking at the source code for this library, you can not safely assume that because something doesn't start with an underscore, it is a public API.

If a release of this library changes any of the items documented here, the version number will be incremented according to the Semantic Version Specification.

This documentation is for `cmd2` version 1.3.10.

7.1.1 cmd2.Cmd

class cmd2.**Cmd**(*completekey: str = 'tab', stdin=None, stdout=None, *, persistent_history_file: str = '', persistent_history_length: int = 1000, startup_script: str = '', use_ipython: bool = False, allow_cli_args: bool = True, transcript_files: Optional[List[str]] = None, allow_redirection: bool = True, multiline_commands: Optional[List[str]] = None, terminators: Optional[List[str]] = None, shortcuts: Optional[Dict[str, str]] = None, command_sets: Optional[Iterable[cmd2.command_definition.CommandSet]] = None, auto_load_commands: bool = True*)

An easy but powerful framework for writing line-oriented command interpreters.

Extends the Python Standard Library's cmd package by adding a lot of useful features to the out of the box configuration.

Line-oriented command interpreters are often useful for test harnesses, internal tools, and rapid prototypes.

__init__(*completekey: str = 'tab'*, *stdin=None*, *stdout=None*, ***, *persistent_history_file: str = ''*, *persistent_history_length: int = 1000*, *startup_script: str = ''*, *use_ipython: bool = False*, *allow_cli_args: bool = True*, *transcript_files: Optional[List[str]] = None*, *allow_redirection: bool = True*, *multiline_commands: Optional[List[str]] = None*, *terminators: Optional[List[str]] = None*, *shortcuts: Optional[Dict[str, str]] = None*, *command_sets: Optional[Iterable[cmd2.command_definition.CommandSet]] = None*, *auto_load_commands: bool = True*) → None

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

Parameters

- **completekey** – readline name of a completion key, default to Tab
- **stdin** – alternate input file object, if not specified, sys.stdin is used
- **stdout** – alternate output file object, if not specified, sys.stdout is used
- **persistent_history_file** – file path to load a persistent cmd2 command history from
- **persistent_history_length** – max number of history items to write to the persistent history file
- **startup_script** – file path to a script to execute at startup
- **use_ipython** – should the "ipy" command be included for an embedded IPython shell
- **allow_cli_args** – if `True`, then *cmd2.Cmd.__init__()* will process command line arguments as either commands to be run or, if `-t` or `--test` are given, transcript files to run. This should be set to `False` if your application parses its own command line arguments.
- **transcript_files** – pass a list of transcript files to be run on initialization. This allows running transcript tests when `allow_cli_args` is `False`. If `allow_cli_args` is `True` this parameter is ignored.
- **allow_redirection** – If `False`, prevent output redirection and piping to shell commands. This parameter prevents redirection and piping, but does not alter parsing behavior. A user can still type redirection and piping tokens, and they will be parsed as such but they won't do anything.
- **multiline_commands** – list of commands allowed to accept multi-line input
- **terminators** – list of characters that terminate a command. These are mainly intended for terminating multiline commands, but will also terminate single-line commands. If not supplied, the default is a semicolon. If your app only contains single-line commands and you want terminators to be treated as literals by the parser, then set this to an empty list.
- **shortcuts** – dictionary containing shortcuts for commands. If not supplied, then defaults to constants.DEFAULT_SHORTCUTS. If you do not want any shortcuts, pass an empty dictionary.
- **command_sets** – Provide CommandSet instances to load during cmd2 initialization. This allows CommandSets with custom constructor parameters to be loaded. This also allows the a set of CommandSets to be provided when *auto_load_commands* is set to False
- **auto_load_commands** – If True, cmd2 will check for all subclasses of *CommandSet* that are currently loaded by Python and automatically instantiate and register all commands. If False, CommandSets must be manually installed with *register_command_set*.

default_error
The error message displayed when a non-existent command is run. Default: `{} is not a recognized command, alias, or macro`

help_error
The error message displayed to the user when they request help for a command with no help defined. Default: `No help on {}`

prompt
The prompt issued to solicit input. The default value is `(Cmd)`. See *Prompt* for more information.

continuation_prompt
The prompt issued to solicit input for the 2nd and subsequent lines of a *multiline command* Default: `>`.

echo
If `True`, output the prompt and user input before executing the command. When redirecting a series of commands to an output file, this allows you to see the command in the output.

settable
This dictionary contains the name and description of all settings available to users.

Users use the *set* command to view and modify settings. Settings are stored in instance attributes with the same name as the setting.

history
A record of previously entered commands.

This attribute is an instance of *cmd2.history.History*, and each command is an instance of *cmd2.Statement*.

statement_parser
An instance of *cmd2.parsing.StatementParser* initialized and configured appropriately for parsing user input.

intro
Set an introduction message which is displayed to the user before the *Command Processing Loop* begins.

py_bridge_name
The symbol name which *Python Scripts* run using the *run_pyscript* command can use to reference the parent `cmd2` application.

ALPHABETICAL_SORT_KEY() → str
Normalize and casefold Unicode strings for saner comparisons.

Parameters **astr** – input unicode string

Returns a normalized and case-folded version of the input string

NATURAL_SORT_KEY() → List[Union[int, str]]
Converts a string into a list of integers and strings to support natural sorting (see natural_sort).

For example: natural_keys('abc123def') -> ['abc', '123', 'def'] :param input_str: string to convert :return: list of strings and integers

add_settable(*settable: cmd2.utils.Settable*) → None
Convenience method to add a settable parameter to `self.settables`

Parameters **settable** – Settable object being added

aliases
Read-only property to access the aliases stored in the StatementParser

allow_style
Read-only property needed to support do_set when it reads allow_style

async_alert(*alert_msg: str*, *new_prompt: Optional[str] = None*) → None
Display an important message to the user while they are at a command line prompt. To the user it appears as if an alert message is printed above the prompt and their current input text and cursor location is left alone.

Raises a *RuntimeError* if called while another thread holds *terminal_lock*.

IMPORTANT: This function will not print an alert unless it can acquire self.terminal_lock to ensure a prompt is onscreen. Therefore it is best to acquire the lock before calling this function to guarantee the alert prints and to avoid raising a RuntimeError.

Parameters

- **alert_msg** – the message to display to the user
- **new_prompt** – if you also want to change the prompt that is displayed, then include it here see async_update_prompt() docstring for guidance on updating a prompt

async_update_prompt(*new_prompt: str*) → None
Update the command line prompt while the user is still typing at it. This is good for alerting the user to system changes dynamically in between commands. For instance you could alter the color of the prompt to indicate a system status or increase a counter to report an event. If you do alter the actual text of the prompt, it is best to keep the prompt the same width as what's on screen. Otherwise the user's input text will be shifted and the update will not be seamless.

Raises a *RuntimeError* if called while another thread holds *terminal_lock*.

IMPORTANT: This function will not update the prompt unless it can acquire self.terminal_lock to ensure a prompt is onscreen. Therefore it is best to acquire the lock before calling this function to guarantee the prompt changes and to avoid raising a RuntimeError.

If user is at a continuation prompt while entering a multiline command, the onscreen prompt will not change. However self.prompt will still be updated and display immediately after the multiline line command completes.

Parameters **new_prompt** – what to change the prompt to

build_settables()
Create the dictionary of user-settable parameters

cmd_func(*command: str*) → Optional[Callable]
Get the function for a command

Parameters **command** – the name of the command

Example

```
>>> helpfunc = self.cmd_func('help')
```

helpfunc now contains a reference to the `do_help` method

cmdloop(*intro: Optional[str] = None*) → int
This is an outer wrapper around _cmdloop() which deals with extra features provided by cmd2.

_cmdloop() provides the main loop equivalent to cmd.cmdloop(). This is a wrapper around that which deals with the following extra features provided by cmd2: - transcript testing - intro banner - exit code

Parameters **intro** – if provided this overrides self.intro and serves as the intro banner printed once at start

complete(*text: str*, *state: int*) → Optional[str]
Override of cmd2's complete method which returns the next possible completion for 'text'

This completer function is called by readline as complete(text, state), for state in 0, 1, 2, ..., until it returns a non-string value. It should return the next possible completion starting with text.

Since readline suppresses any exception raised in completer functions, they can be difficult to debug. Therefore this function wraps the actual tab completion logic and prints to stderr any exception that occurs before returning control to readline.

Parameters

- **text** – the current word that user is typing
- **state** – non-negative integer

Returns the next possible completion for text or None

complete_help_command(*text: str*, *line: str*, *begidx: int*, *endidx: int*) → List[str]
Completes the command argument of help

complete_help_subcommands(*text: str, line: str, begidx: int, endidx: int, arg_tokens: Dict[str, List[str]]*) → List[str]
Completes the subcommands argument of help

complete_set_value(*text: str, line: str, begidx: int, endidx: int, arg_tokens: Dict[str, List[str]]*) → List[str]
Completes the value argument of set

default(*statement: cmd2.parsing.Statement*) → Optional[bool]
Executed when the command given isn't a recognized command implemented by a do_* method.

Parameters **statement** – Statement object with parsed input

delimiter_complete(*text: str, line: str, begidx: int, endidx: int, match_against: Iterable[T_co], delimiter: str*) → List[str]
Performs tab completion against a list but each match is split on a delimiter and only the portion of the match being tab completed is shown as the completion suggestions. This is useful if you match against strings that are hierarchical in nature and have a common delimiter.

An easy way to illustrate this concept is path completion since paths are just directories/files delimited by a slash. If you are tab completing items in /home/user you don't get the following as suggestions:

/home/user/file.txt /home/user/program.c /home/user/maps/ /home/user/cmd2.py

Instead you are shown:

file.txt program.c maps/ cmd2.py

For a large set of data, this can be visually more pleasing and easier to search.

Another example would be strings formatted with the following syntax: company::department::name In this case the delimiter would be :: and the user could easily narrow down what they are looking for if they were only shown suggestions in the category they are at in the string.

Parameters

- **text** – the string prefix we are attempting to match (all matches must begin with it)
- **line** – the current input line with leading whitespace removed
- **begidx** – the beginning index of the prefix text
- **endidx** – the ending index of the prefix text
- **match_against** – the list being matched against

- **delimiter** – what delimits each portion of the matches (ex: paths are delimited by a slash)

Returns a list of possible tab completions

disable_category(*category: str*, *message_to_print: str*) → None
Disable an entire category of commands.

Parameters

- **category** – the category to disable
- **message_to_print** – what to print when anything in this category is run or help is called on it while disabled. The variable COMMAND_NAME can be used as a placeholder for the name of the command being disabled. ex: message_to_print = "{} is currently disabled".format(COMMAND_NAME)

disable_command(*command: str*, *message_to_print: str*) → None
Disable a command and overwrite its functions

Parameters

- **command** – the command being disabled
- **message_to_print** – what to print when this command is run or help is called on it while disabled

  The variable COMMAND_NAME can be used as a placeholder for the name of the command being disabled. ex: message_to_print = "{} is currently disabled".format(COMMAND_NAME)

do__relative_run_script(*args: argparse.Namespace*) → Optional[bool]
Run commands in script file that is encoded as either ASCII or UTF-8 text

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

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

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

do_alias(*args: argparse.Namespace*) → None
Manage aliases

An alias is a command that enables replacement of a word by another string.

do_edit(*args: argparse.Namespace*) → None
Run a text editor and optionally open a file with it

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

set editor (program-name)

do_eof(*_: argparse.Namespace*) → bool
Called when <Ctrl>-D is pressed

do_help(*args: argparse.Namespace*) → None
List available commands or provide detailed help for a specific command

do_history(*args: argparse.Namespace*) → Optional[bool]
View, run, edit, save, or clear previously entered commands

do_macro(*args: argparse.Namespace*) → None
Manage macros

A macro is similar to an alias, but it can contain argument placeholders.

do_py(*args: argparse.Namespace*, *, *pyscript: Optional[str] = None*) → Optional[bool]
Invoke Python command or shell

Note that, when invoking a command directly from the command line, this shell has limited ability to parse Python statements into tokens. In particular, there may be problems with whitespace and quotes depending on their placement.

If you see strange parsing behavior, it's best to just open the Python shell by providing no arguments to py and run more complex statements there.

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

do_run_pyscript(*args: argparse.Namespace*) → Optional[bool]
Run a Python script file inside the console

do_run_script(*args: argparse.Namespace*) → Optional[bool]
Run commands in script file that is encoded as either ASCII or UTF-8 text

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

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

do_set(*args: argparse.Namespace*) → None
Set a settable parameter or show current settings of parameters

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

do_shortcuts(*_: argparse.Namespace*) → None
List available shortcuts

enable_category(*category: str*) → None
Enable an entire category of commands

Parameters **category** – the category to enable

enable_command(*command: str*) → None
Enable a command by restoring its functions

Parameters **command** – the command being enabled

find_commandset_for_command(*command_name: str*) → Optional[cmd2.command_definition.CommandSet]
Finds the CommandSet that registered the command name :param command_name: command name to search :return: CommandSet that provided the command

find_commandsets(*commandset_type: Type[cmd2.command_definition.CommandSet], *, subclass_match: bool = False*) → List[cmd2.command_definition.CommandSet]
Find all CommandSets that match the provided CommandSet type. By default, locates a CommandSet that is an exact type match but may optionally return all CommandSets that are sub-classes of the provided type :param commandset_type: CommandSet sub-class type to search for :param subclass_match: If True, return all sub-classes of provided type, otherwise only search for exact match :return: Matching CommandSets

flag_based_complete(*text: str, line: str, begidx: int, endidx: int, flag_dict: Dict[str, Union[Iterable[T_co], Callable]], *, all_else: Union[None, Iterable[T_co], Callable] = None*) → List[str]
Tab completes based on a particular flag preceding the token being completed.

Parameters

- **text** – the string prefix we are attempting to match (all matches must begin with it)
- **line** – the current input line with leading whitespace removed
- **begidx** – the beginning index of the prefix text
- **endidx** – the ending index of the prefix text
- **flag_dict** – dictionary whose structure is the following: *keys* - flags (ex: -c, –create) that result in tab completion for the next argument in the command line *values* - there are two types of values: 1. iterable list of strings to match against (dictionaries, lists, etc.) 2. function that performs tab completion (ex: path_complete)
- **all_else** – an optional parameter for tab completing any token that isn't preceded by a flag in flag_dict

Returns a list of possible tab completions

get_all_commands() → List[str]
Return a list of all commands

get_help_topics() → List[str]
Return a list of help topics

get_names()
Return an alphabetized list of names comprising the attributes of the cmd2 class instance.

get_visible_commands() → List[str]
Return a list of commands that have not been hidden or disabled

in_pyscript() → bool
Return whether a pyscript is running

in_script() → bool
Return whether a text script is running

index_based_complete(*text: str, line: str, begidx: int, endidx: int, index_dict: Mapping[int, Union[Iterable[T_co], Callable]], *, all_else: Union[None, Iterable[T_co], Callable] = None*) → List[str]
Tab completes based on a fixed position in the input string.

Parameters

- **text** – the string prefix we are attempting to match (all matches must begin with it)
- **line** – the current input line with leading whitespace removed
- **begidx** – the beginning index of the prefix text
- **endidx** – the ending index of the prefix text
- **index_dict** – dictionary whose structure is the following: *keys* - 0-based token indexes into command line that determine which tokens perform tab completion *values* - there are two types of values: 1. iterable list of strings to match against (dictionaries, lists, etc.) 2. function that performs tab completion (ex: path_complete)
- **all_else** – an optional parameter for tab completing any token that isn't at an index in index_dict

Returns a list of possible tab completions

onecmd(*statement: Union[cmd2.parsing.Statement, str], *, add_to_history: bool = True*) → bool
This executes the actual do_* method for a command.

If the command provided doesn't exist, then it executes default() instead.

Parameters

- **statement** – intended to be a Statement instance parsed command from the input stream, alternative acceptance of a str is present only for backward compatibility with cmd
- **add_to_history** – If True, then add this command to history. Defaults to True.

Returns a flag indicating whether the interpretation of commands should stop

onecmd_plus_hooks(*line: str*, ***, *add_to_history: bool = True*, *raise_keyboard_interrupt: bool = False*, *py_bridge_call: bool = False*) → bool

Top-level function called by cmdloop() to handle parsing a line and running the command and all of its hooks.

Parameters

- **line** – command line to run
- **add_to_history** – If True, then add this command to history. Defaults to True.
- **raise_keyboard_interrupt** – if True, then KeyboardInterrupt exceptions will be raised if stop isn't already True. This is used when running commands in a loop to be able to stop the whole loop and not just the current command. Defaults to False.
- **py_bridge_call** – This should only ever be set to True by PyBridge to signify the beginning of an app() call from Python. It is used to enable/disable the storage of the command's stdout.

Returns True if running of commands should stop

parseline(*line: str*) → Tuple[str, str, str]

Parse the line into a command name and a string containing the arguments.

NOTE: This is an override of a parent class method. It is only used by other parent class methods.

Different from the parent class method, this ignores self.identchars.

Parameters **line** – line read by readline

Returns tuple containing (command, args, line)

path_complete(*text: str*, *line: str*, *begidx: int*, *endidx: int*, ***, *path_filter: Optional[Callable[[str], bool]] = None*) → List[str]

Performs completion of local file system paths

Parameters

- **text** – the string prefix we are attempting to match (all matches must begin with it)
- **line** – the current input line with leading whitespace removed
- **begidx** – the beginning index of the prefix text
- **endidx** – the ending index of the prefix text
- **path_filter** – optional filter function that determines if a path belongs in the results this function takes a path as its argument and returns True if the path should be kept in the results

Returns a list of possible tab completions

perror(*msg: Any = ''*, ***, *end: str = '\n'*, *apply_style: bool = True*) → None

Print message to sys.stderr

Parameters

- **msg** – message to print (anything convertible to a str with '{}'.format() is OK)
- **end** – string appended after the end of the message, default a newline
- **apply_style** – If True, then ansi.style_error will be applied to the message text. Set to False in cases where the message text already has the desired style. Defaults to True.

pexcept(*msg: Any*, ***, *end: str = '\n'*, *apply_style: bool = True*) → None
Print Exception message to sys.stderr. If debug is true, print exception traceback if one exists.

Parameters

- **msg** – message or Exception to print
- **end** – string appended after the end of the message, default a newline
- **apply_style** – If True, then ansi.style_error will be applied to the message text. Set to False in cases where the message text already has the desired style. Defaults to True.

pfeedback(*msg: Any*, ***, *end: str = '\n'*) → None
For printing nonessential feedback. Can be silenced with *quiet*. Inclusion in redirected output is controlled by *feedback_to_output*.

Parameters

- **msg** – message to print (anything convertible to a str with '{}'.format() is OK)
- **end** – string appended after the end of the message, default a newline

postcmd(*stop: bool*, *statement: cmd2.parsing.Statement*) → bool
Hook method executed just after a command is executed by *onecmd()*.

Parameters

- **stop** – return *True* to request the command loop terminate
- **statement** – subclass of str which also contains the parsed input

See *register_postcmd_hook()* and *register_cmdfinalization_hook()* for more robust ways to run hooks after the command is executed. See *Postcommand Hooks* and *Command Finalization Hooks* for more information.

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

See *register_postloop_hook()* for a more robust way to run hooks after the command loop completes. See *Application Lifecycle Hooks* for more information.

poutput(*msg: Any = ''*, ***, *end: str = '\n'*) → None
Print message to self.stdout and appends a newline by default

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

Parameters

- **msg** – message to print (anything convertible to a str with '{}'.format() is OK)
- **end** – string appended after the end of the message, default a newline

ppaged(*msg: Any*, ***, *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, default a newline
- **chop** –

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

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

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

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

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

precmd(*statement: cmd2.parsing.Statement*) → cmd2.parsing.Statement

Hook method executed just before the command is executed by *onecmd()* and after adding it to history.

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

Returns a potentially modified version of the input Statement object

See *register_postparsing_hook()* and *register_precmd_hook()* for more robust ways to run hooks before the command is executed. See *Postparsing Hooks* and *Precommand Hooks* for more information.

preloop()

Hook method executed once when the *cmdloop()* method is called.

See *register_preloop_hook()* for a more robust way to run hooks before the command loop begins. See *Application Lifecycle Hooks* for more information.

pwarning(*msg: Any = ''*, *, *end: str = '\n'*, *apply_style: bool = True*) → None

Wraps perror, but applies ansi.style_warning by default

Parameters

- **msg** – message to print (anything convertible to a str with '{}'.format() is OK)
- **end** – string appended after the end of the message, default a newline
- **apply_style** – If True, then ansi.style_warning will be applied to the message text. Set to False in cases where the message text already has the desired style. Defaults to True.

read_input(*prompt: str*, *, *allow_completion: bool = False*) → str

Read input from appropriate stdin value. Also allows you to disable tab completion while input is being read.

Parameters

- **prompt** – prompt to display to user
- **allow_completion** – if True, then tab completion of commands is enabled. This generally should be set to False unless reading the command line. Defaults to False.

Returns the line read from stdin with all trailing new lines removed

Raises any exceptions raised by input() and stdin.readline()

register_cmdfinalization_hook(*func: Callable[[cmd2.plugin.CommandFinalizationData], cmd2.plugin.CommandFinalizationData]*) → None
Register a hook to be called after a command is completed, whether it completes successfully or not.

register_command_set(*cmdset: cmd2.command_definition.CommandSet*) → None
Installs a CommandSet, loading all commands defined in the CommandSet

Parameters **cmdset** – CommandSet to load

register_postcmd_hook(*func: Callable[[cmd2.plugin.PostcommandData], cmd2.plugin.PostcommandData]*) → None
Register a hook to be called after the command function.

register_postloop_hook(*func: Callable[[None], None]*) → None
Register a function to be called at the end of the command loop.

register_postparsing_hook(*func: Callable[[cmd2.plugin.PostparsingData], cmd2.plugin.PostparsingData]*) → None
Register a function to be called after parsing user input but before running the command

register_precmd_hook(*func: Callable[[cmd2.plugin.PrecommandData], cmd2.plugin.PrecommandData]*) → None
Register a hook to be called before the command function.

register_preloop_hook(*func: Callable[[None], None]*) → None
Register a function to be called at the beginning of the command loop.

remove_settable(*name: str*) → None
Convenience method for removing a settable parameter from `self.settables`

Parameters **name** – name of the settable being removed

Raises KeyError if the Settable matches this name

runcmds_plus_hooks(*cmds: List[Union[cmd2.history.HistoryItem, str]], *, add_to_history: bool = True, stop_on_keyboard_interrupt: bool = True*) → bool
Used when commands are being run in an automated fashion like text scripts or history replays. The prompt and command line for each command will be printed if echo is True.

Parameters

- **cmds** – commands to run
- **add_to_history** – If True, then add these commands to history. Defaults to True.
- **stop_on_keyboard_interrupt** – stop command loop if Ctrl-C is pressed instead of just moving to the next command. Defaults to True.

Returns True if running of commands should stop

select(*opts: Union[str, List[str], List[Tuple[Any, Optional[str]]]], prompt: str = 'Your choice? '*) → str
Presents a numbered menu to the user. Modeled after the bash shell's SELECT. Returns the item chosen.

Argument `opts` can be:

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

a list of strings -> will be offered as options

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

set_window_title(*title: str*) → None
Set the terminal window title.

Raises a *RuntimeError* if called while another thread holds *terminal_lock*.

IMPORTANT: This function will not set the title unless it can acquire self.terminal_lock to avoid writing to stderr while a command is running. Therefore it is best to acquire the lock before calling this function to guarantee the title changes and to avoid raising a RuntimeError.

Parameters **title** – the new window title

shell_cmd_complete(*text: str*, *line: str*, *begidx: int*, *endidx: int*, *, *complete_blank: bool = False*) → List[str]

Performs completion of executables either in a user's path or a given path

Parameters

- **text** – the string prefix we are attempting to match (all matches must begin with it)
- **line** – the current input line with leading whitespace removed
- **begidx** – the beginning index of the prefix text
- **endidx** – the ending index of the prefix text
- **complete_blank** – If True, then a blank will complete all shell commands in a user's path. If False, then no completion is performed. Defaults to False to match Bash shell behavior.

Returns a list of possible tab completions

sigint_handler(*signum: int*, *frame*) → None

Signal handler for SIGINTs which typically come from Ctrl-C events.

If you need custom SIGINT behavior, then override this function.

Parameters

- **signum** – signal number
- **frame** – required param for signal handlers

tokens_for_completion(*line: str*, *begidx: int*, *endidx: int*) → Tuple[List[str], List[str]]

Used by tab completion functions to get all tokens through the one being completed.

Parameters

- **line** – the current input line with leading whitespace removed
- **begidx** – the beginning index of the prefix text
- **endidx** – the ending index of the prefix text

Returns A 2 item tuple where the items are **On Success** - tokens: list of unquoted tokens - this is generally the list needed for tab completion functions - raw_tokens: list of tokens with any quotes preserved = this can be used to know if a token was quoted or is missing a closing quote Both lists are guaranteed to have at least 1 item. The last item in both lists is the token being tab completed **On Failure** - Two empty lists

unregister_command_set(*cmdset: cmd2.command_definition.CommandSet*)

Uninstalls a CommandSet and unloads all associated commands :param cmdset: CommandSet to uninstall

visible_prompt

Read-only property to get the visible prompt with any ANSI style escape codes stripped.

Used by transcript testing to make it easier and more reliable when users are doing things like coloring the prompt using ANSI color codes.

Returns prompt stripped of any ANSI escape codes

7.1.2 cmd2.ansi

Support for ANSI escape sequences which are used for things like applying style to text, setting the window title, and asynchronous alerts.

cmd2.ansi.**BG_RESET = '\x1b[49m'**
ANSI sequence to reset the terminal background attributes

class cmd2.ansi.**ColorBase**
Base class used for defining color enums. See fg and bg classes for examples.

Child classes should define enums in the follow structure:

key: color name (e.g. black)

value: anything that when cast to a string returns an ANSI sequence

cmd2.ansi.**FG_RESET = '\x1b[39m'**
ANSI sequence to reset the foreground attributes

cmd2.ansi.**INTENSITY_BRIGHT = '\x1b[1m'**
ANSI sequence to make the text bright

cmd2.ansi.**INTENSITY_DIM = '\x1b[2m'**
ANSI sequence to make the text dim

cmd2.ansi.**INTENSITY_NORMAL = '\x1b[22m'**
ANSI sequence to make the text normal

cmd2.ansi.**RESET_ALL = '\x1b[0m'**
ANSI sequence to reset all terminal attributes

cmd2.ansi.**STYLE_ALWAYS = 'Always'**
Constant for `cmd2.ansi.allow_style` to indicate ANSI style sequences should always be output.

cmd2.ansi.**STYLE_NEVER = 'Never'**
Constant for `cmd2.ansi.allow_style` to indicate ANSI style sequences should be removed from all output.

cmd2.ansi.**STYLE_TERMINAL = 'Terminal'**
Constant for `cmd2.ansi.allow_style` to indicate ANSI style sequences should be removed if the output is not going to the terminal.

cmd2.ansi.**UNDERLINE_DISABLE = '\x1b[24m'**
ANSI sequence to turn off underline

cmd2.ansi.**UNDERLINE_ENABLE = '\x1b[4m'**
ANSI sequence to turn on underline

cmd2.ansi.**allow_style = 'Terminal'**
When using outside of a cmd2 app, set this variable to one of:

- `STYLE_NEVER` - remove ANSI style sequences from all output
- `STYLE_TERMINAL` - remove ANSI style sequences if the output is not going to the terminal
- `STYLE_ALWAYS` - always output ANSI style sequences

to control the output of ANSI style sequences by methods in this module.

The default is `STYLE_TERMINAL`.

cmd2.ansi.**async_alert_str**(*, *terminal_columns: int*, *prompt: str*, *line: str*, *cursor_offset: int*, *alert_msg: str*) → str
Calculate the desired string, including ANSI escape codes, for displaying an asynchronous alert message.

Parameters

- **terminal_columns** – terminal width (number of columns)
- **prompt** – prompt that is displayed on the current line
- **line** – current contents of the Readline line buffer
- **cursor_offset** – the offset of the current cursor position within line
- **alert_msg** – the message to display to the user

Returns the correct string so that the alert message appears to the user to be printed above the current line.

class cmd2.ansi.**bg**
Enum class for background colors

cmd2.ansi.**bg_lookup**(*bg_name: Union[str, cmd2.ansi.bg]*) → str
Look up ANSI escape codes based on background color name.

Parameters **bg_name** – background color name or enum to look up ANSI escape code(s) for

Returns ANSI escape code(s) associated with this color

Raises ValueError: if the color cannot be found

class cmd2.ansi.**fg**
Enum class for foreground colors

cmd2.ansi.**fg_lookup**(*fg_name: Union[str, cmd2.ansi.fg]*) → str
Look up ANSI escape codes based on foreground color name.

Parameters **fg_name** – foreground color name or enum to look up ANSI escape code(s) for

Returns ANSI escape code(s) associated with this color

Raises ValueError: if the color cannot be found

cmd2.ansi.**set_title_str**(*title: str*) → str
Get the required string, including ANSI escape codes, for setting window title for the terminal.

Parameters **title** – new title for the window

Returns string to write to sys.stderr in order to set the window title to the desired test

cmd2.ansi.**strip_style**(*text: str*) → str
Strip ANSI style sequences from a string.

Parameters **text** – string which may contain ANSI style sequences

Returns the same string with any ANSI style sequences removed

cmd2.ansi.**style**(*text: Any*, **, fg: Union[str, cmd2.ansi.fg] = ''*, *bg: Union[str, cmd2.ansi.bg] = ''*, *bold: bool = False*, *dim: bool = False*, *underline: bool = False*) → str
Apply ANSI colors and/or styles to a string and return it. The styling is self contained which means that at the end of the string reset code(s) are issued to undo whatever styling was done at the beginning.

Parameters

- **text** – Any object compatible with str.format()
- **fg** – foreground color. Relies on *fg_lookup()* to retrieve ANSI escape based on name or enum. Defaults to no color.
- **bg** – background color. Relies on *bg_lookup()* to retrieve ANSI escape based on name or enum. Defaults to no color.

- **bold** – apply the bold style if True. Can be combined with dim. Defaults to False.
- **dim** – apply the dim style if True. Can be combined with bold. Defaults to False.
- **underline** – apply the underline style if True. Defaults to False.

Returns the stylized string

cmd2.ansi.**style_aware_wcswidth**(*text: str*) → int
Wrap wcswidth to make it compatible with strings that contains ANSI style sequences

Parameters **text** – the string being measured

Returns the width of the string when printed to the terminal

cmd2.ansi.**style_aware_write**(*fileobj: IO*, *msg: str*) → None
Write a string to a fileobject and strip its ANSI style sequences if required by allow_style setting

Parameters

- **fileobj** – the file object being written to
- **msg** – the string being written

cmd2.ansi.**style_error**(*text: Any*, ***, *fg: Union[str, cmd2.ansi.fg] = <fg.bright_red: '\x1b[91m'>*, *bg: Union[str, cmd2.ansi.bg] = ''*, *bold: bool = False*, *dim: bool = False*, *underline: bool = False*) → str
Partial function supplying arguments to *cmd2.ansi.style()* which colors text to signify an error

cmd2.ansi.**style_success**(*text: Any*, ***, *fg: Union[str, cmd2.ansi.fg] = <fg.green: '\x1b[32m'>*, *bg: Union[str, cmd2.ansi.bg] = ''*, *bold: bool = False*, *dim: bool = False*, *underline: bool = False*) → str
Partial function supplying arguments to *cmd2.ansi.style()* which colors text to signify success

cmd2.ansi.**style_warning**(*text: Any*, ***, *fg: Union[str, cmd2.ansi.fg] = <fg.bright_yellow: '\x1b[93m'>*, *bg: Union[str, cmd2.ansi.bg] = ''*, *bold: bool = False*, *dim: bool = False*, *underline: bool = False*) → str
Partial function supplying arguments to *cmd2.ansi.style()* which colors text to signify a warning

7.1.3 cmd2.argparse_completer

This module defines the ArgparseCompleter class which provides argparse-based tab completion to cmd2 apps. See the header of argparse_custom.py for instructions on how to use these features.

class cmd2.argparse_completer.**ArgparseCompleter**(*parser: argparse.ArgumentParser*, *cmd2_app: cmd2.cmd2.Cmd*, ***, *parent_tokens: Optional[Dict[str, List[str]]] = None*)
Automatic command line tab completion based on argparse parameters

complete_command(*tokens: List[str]*, *text: str*, *line: str*, *begidx: int*, *endidx: int*, ***, *cmd_set: Optional[cmd2.command_definition.CommandSet] = None*) → List[str]
Complete the command using the argparse metadata and provided argument dictionary :raises: CompletionError for various types of tab completion errors

complete_subcommand_help(*tokens: List[str]*, *text: str*, *line: str*, *begidx: int*, *endidx: int*) → List[str]
Supports cmd2's help command in the completion of subcommand names :param tokens: command line tokens :param text: the string prefix we are attempting to match (all matches must begin with it) :param line: the current input line with leading whitespace removed :param begidx: the beginning index of the prefix text :param endidx: the ending index of the prefix text :return: List of subcommand completions

format_help(*tokens: List[str]*) → str
Supports cmd2's help command in the retrieval of help text :param tokens: command line tokens :return: help text of the command being queried

7.1.4 cmd2.argparse_custom

This module adds capabilities to argparse by patching a few of its functions. It also defines a parser class called Cmd2ArgumentParser which improves error and help output over normal argparse. All cmd2 code uses this parser and it is recommended that developers of cmd2-based apps either use it or write their own parser that inherits from it. This will give a consistent look-and-feel between the help/error output of built-in cmd2 commands and the app-specific commands. If you wish to override the parser used by cmd2's built-in commands, see override_parser.py example.

Since the new capabilities are added by patching at the argparse API level, they are available whether or not Cmd2ArgumentParser is used. However, the help and error output of Cmd2ArgumentParser is customized to notate nargs ranges whereas any other parser class won't be as explicit in their output.

Added capabilities

Extends argparse nargs functionality by allowing tuples which specify a range (min, max). To specify a max value with no upper bound, use a 1-item tuple (min,)

Example:

```
# -f argument expects at least 3 values
parser.add_argument('-f', nargs=(3,))

# -f argument expects 3 to 5 values
parser.add_argument('-f', nargs=(3, 5))
```

Tab Completion

cmd2 uses its ArgparseCompleter class to enable argparse-based tab completion on all commands that use the @with_argparse wrappers. Out of the box you get tab completion of commands, subcommands, and flag names, as well as instructive hints about the current argument that print when tab is pressed. In addition, you can add tab completion for each argument's values using parameters passed to add_argument().

Below are the 5 add_argument() parameters for enabling tab completion of an argument's value. Only one can be used at a time.

`choices` - pass a list of values to the choices parameter.

Example:

```
parser.add_argument('-o', '--options', choices=['An Option', 'SomeOtherOption
↪'])
parser.add_argument('-o', '--options', choices=my_list)
```

`choices_function` - pass a function that returns choices. This is good in cases where the choice list is dynamically generated when the user hits tab.

Example:

```
def my_choices_function():
    ...
    return my_generated_list

parser.add_argument('-o', '--options', choices_function=my_choices_function)
```

`choices_method` - this is equivalent to choices_function, but the function needs to be an instance method of a cmd2.Cmd or cmd2.CommandSet subclass. When ArgparseCompleter calls the method, it well detect whether is is bound to a CommandSet or Cmd subclass. If bound to a cmd2.Cmd subclass, it will pass the app instance as the *self* argument. This is good in cases where the list of choices being generated relies on state data of the cmd2-based app. If bound to a cmd2.CommandSet subclass, it will pass the CommandSet instance as the *self* argument.

Example:

```
def my_choices_method(self):
    ...
    return my_generated_list

parser.add_argument("arg", choices_method=my_choices_method)
```

`completer_function` - pass a tab completion function that does custom completion. Since custom tab completion operations commonly need to modify cmd2's instance variables related to tab completion, it will be rare to need a completer function. completer_method should be used in those cases.

Example:

```
def my_completer_function(text, line, begidx, endidx):
    ...
    return completions
parser.add_argument('-o', '--options', completer_function=my_completer_
↪function)
```

`completer_method` - this is equivalent to completer_function, but the function needs to be an instance method of a cmd2.Cmd or cmd2.CommandSet subclass. When ArgparseCompleter calls the method, it well detect whether is is bound to a CommandSet or Cmd subclass. If bound to a cmd2.Cmd subclass, it will pass the app instance as the *self* argument. This is good in cases where the list of choices being generated relies on state data of the cmd2-based app. If bound to a cmd2.CommandSet subclass, it will pass the CommandSet instance as the *self* argument, and the app instance as the positional argument. cmd2 provides a few completer methods for convenience (e.g., path_complete, delimiter_complete)

Example:

```
# This adds file-path completion to an argument
parser.add_argument('-o', '--options', completer_method=cmd2.Cmd.path_
↪complete)
```

You can use functools.partial() to prepopulate values of the underlying choices and completer functions/methods.

Example:

```
# This says to call path_complete with a preset value for its path_filter␣
↪argument
completer_method = functools.partial(path_complete,
                                     path_filter=lambda path: os.path.
↪isdir(path))
parser.add_argument('-o', '--options', choices_method=completer_method)
```

Of the 5 tab completion parameters, choices is the only one where argparse validates user input against items in the choices list. This is because the other 4 parameters are meant to tab complete data sets that are viewed as dynamic. Therefore it is up to the developer to validate if the user has typed an acceptable value for these arguments.

The following functions exist in cases where you may want to manually add a choice-providing function/method to an existing argparse action. For instance, in __init__() of a custom action class.

- set_choices_function(action, func)
- set_choices_method(action, method)
- set_completer_function(action, func)
- set_completer_method(action, method)

There are times when what's being tab completed is determined by a previous argument on the command line. In theses cases, Autocompleter can pass a dictionary that maps the command line tokens up through the one being completed to their argparse argument name. To receive this dictionary, your choices/completer function should have an argument called arg_tokens.

Example:

```
def my_choices_method(self, arg_tokens)
def my_completer_method(self, text, line, begidx, endidx, arg_tokens)
```

All values of the arg_tokens dictionary are lists, even if a particular argument expects only 1 token. Since ArgparseCompleter is for tab completion, it does not convert the tokens to their actual argument types or validate their values. All tokens are stored in the dictionary as the raw strings provided on the command line. It is up to the developer to determine if the user entered the correct argument type (e.g. int) and validate their values.

CompletionItem Class - This class was added to help in cases where uninformative data is being tab completed. For instance, tab completing ID numbers isn't very helpful to a user without context. Returning a list of CompletionItems instead of a regular string for completion results will signal the ArgparseCompleter to output the completion results in a table of completion tokens with descriptions instead of just a table of tokens:

```
Instead of this:
    1     2     3

The user sees this:
    ITEM_ID     Item Name
    1           My item
    2           Another item
    3           Yet another item
```

The left-most column is the actual value being tab completed and its header is that value's name. The right column header is defined using the descriptive_header parameter of add_argument(). The right column values come from the CompletionItem.description value.

Example:

```
token = 1
token_description = "My Item"
completion_item = CompletionItem(token, token_description)
```

Since descriptive_header and CompletionItem.description are just strings, you can format them in such a way to have multiple columns:

```
ITEM_ID     Item Name            Checked Out    Due Date
1           My item              True           02/02/2022
2           Another item         False
3           Yet another item     False
```

To use CompletionItems, just return them from your choices or completer functions.

To avoid printing a ton of information to the screen at once when a user presses tab, there is a maximum threshold for the number of CompletionItems that will be shown. Its value is defined in cmd2.Cmd.max_completion_items.

It defaults to 50, but can be changed. If the number of completion suggestions exceeds this number, they will be displayed in the typical columnized format and will not include the description value of the CompletionItems.

Patched argparse functions

`argparse._ActionsContainer.add_argument` - adds arguments related to tab completion and enables nargs range parsing. See _add_argument_wrapper for more details on these arguments.

`argparse.ArgumentParser._get_nargs_pattern` - adds support for nargs ranges. See _get_nargs_pattern_wrapper for more details.

`argparse.ArgumentParser._match_argument` - adds support for nargs ranges. See _match_argument_wrapper for more details.

`argparse._SubParsersAction.remove_parser` - new function which removes a sub-parser from a sub-parsers group. See _SubParsersAction_remove_parser for more details.

class cmd2.argparse_custom.**ChoicesCallable**(*is_method: bool*, *is_completer: bool*, *to_call: Callable*)

Enables using a callable as the choices provider for an argparse argument. While argparse has the built-in choices attribute, it is limited to an iterable.

class cmd2.argparse_custom.**Cmd2ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=None*, *formatter_class=<class 'cmd2.argparse_custom.Cmd2HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

Custom ArgumentParser class that improves error and help output

add_subparsers(***kwargs*)

Custom override. Sets a default title if one was not given.

Parameters **kwargs** – additional keyword arguments

Returns argparse Subparser Action

error(*message: str*) → None

Custom override that applies custom formatting to the error message

format_help() → str

Copy of format_help() from argparse.ArgumentParser with tweaks to separately display required parameters

class cmd2.argparse_custom.**Cmd2AttributeWrapper**(*attribute: Any*)

Wraps a cmd2-specific attribute added to an argparse Namespace. This makes it easy to know which attributes in a Namespace are arguments from a parser and which were added by cmd2.

get() → Any

Get the value of the attribute

set(*new_val: Any*) → None

Set the value of the attribute

class cmd2.argparse_custom.**Cmd2HelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Custom help formatter to configure ordering of help text

class cmd2.argparse_custom.**CompletionItem**(*value: object*, *desc: str = ''*, **args*, ***kwargs*)

Completion item with descriptive text attached

See header of this file for more information

cmd2.argparse_custom.**DEFAULT_ARGUMENT_PARSER**
alias of *cmd2.argparse_custom.Cmd2ArgumentParser*

cmd2.argparse_custom.**generate_range_error**(*range_min: int*, *range_max: Union[int, float]*) → str
Generate an error message when the the number of arguments provided is not within the expected range

cmd2.argparse_custom.**set_choices_function**(*action: argparse.Action*, *choices_function: Callable*) → None
Set choices_function on an argparse action

cmd2.argparse_custom.**set_choices_method**(*action: argparse.Action*, *choices_method: Callable*) → None
Set choices_method on an argparse action

cmd2.argparse_custom.**set_completer_function**(*action: argparse.Action*, *completer_function: Callable*) → None
Set completer_function on an argparse action

cmd2.argparse_custom.**set_completer_method**(*action: argparse.Action*, *completer_method: Callable*) → None
Set completer_method on an argparse action

cmd2.argparse_custom.**set_default_argument_parser**(*parser: Type[argparse.ArgumentParser]*) → None
Set the default ArgumentParser class for a cmd2 app

7.1.5 cmd2.constants

This module contains constants used throughout cmd2.

cmd2.constants.**DEFAULT_SHORTCUTS**
If you do not supply shortcuts to *cmd2.Cmd.__init__()*, the shortcuts defined here will be used instead.

cmd2.constants.**COMMAND_NAME**
Used by *cmd2.Cmd.disable_command()* and *cmd2.Cmd.disable_category()*. Those methods allow you to selectively disable single commands or an entire category of commands. Should you want to include the name of the command in the error message displayed to the user when they try and run a disabled command, you can include this constant in the message where you would like the name of the command to appear. cmd2 will replace this constant with the name of the command the user tried to run before displaying the error message.

This constant is imported into the package namespace; the preferred syntax to import and reference it is:

```
import cmd2
errmsg = "The {} command is currently disabled.".format(cmd2.COMMAND_NAME)
```

See src/examples/help_categories.py for an example.

7.1.6 cmd2.command_definition

Supports the definition of commands in separate classes to be composed into cmd2.Cmd

class cmd2.command_definition.**CommandSet**
Base class for defining sets of commands to load in cmd2.

with_default_category can be used to apply a default category to all commands in the CommandSet.

do_, help_, and complete_ functions differ only in that self is the CommandSet instead of the cmd2 app

on_register(*cmd*) → None
Called by cmd2.Cmd as the first step to registering a CommandSet. The commands defined in this class have not be added to the CLI object at this point. Subclasses can override this to perform any initialization requiring access to the Cmd object (e.g. configure commands and their parsers based on CLI state data).

Parameters **cmd** (cmd2.Cmd) – The cmd2 main application

on_registered() → None
Called by cmd2.Cmd after a CommandSet is registered and all its commands have been added to the CLI. Subclasses can override this to perform custom steps related to the newly added commands (e.g. setting them to a disabled state).

on_unregister() → None
Called by cmd2.Cmd as the first step to unregistering a CommandSet. Subclasses can override this to perform any cleanup steps which require their commands being registered in the CLI.

on_unregistered() → None
Called by cmd2.Cmd after a CommandSet has been unregistered and all its commands removed from the CLI. Subclasses can override this to perform remaining cleanup steps.

cmd2.command_definition.**with_default_category**(*category: str*, *, *heritable: bool = True*)
Decorator that applies a category to all do_* command methods in a class that do not already have a category specified.

CommandSets that are decorated by this with *heritable* set to True (default) will set a class attribute that is inherited by all subclasses unless overridden. All commands of this CommandSet and all subclasses of this CommandSet that do not declare an explicit category will be placed in this category. Subclasses may use this decorator to override the default category.

If *heritable* is set to False, then only the commands declared locally to this CommandSet will be placed in the specified category. Dynamically created commands, and commands declared in sub-classes will not receive this category.

Parameters

- **category** – category to put all uncategorized commands in
- **heritable** – Flag whether this default category should apply to sub-classes. Defaults to True

Returns decorator function

7.1.7 cmd2.decorators

Decorators for cmd2 commands

cmd2.decorators.**as_subcommand_to**(*command: str*, *subcommand: str*, *parser: argparse.ArgumentParser*, *, *help: Optional[str] = None*, *aliases: Iterable[str] = None*) → Callable[[argparse.Namespace], Optional[bool]]
Tag this method as a subcommand to an existing argparse decorated command.

Parameters

- **command** – Command Name. Space-delimited subcommands may optionally be specified
- **subcommand** – Subcommand name
- **parser** – argparse Parser for this subcommand

- **help** – Help message for this subcommand which displays in the list of subcommands of the command we are adding to. This is passed as the help argument to ArgumentParser.add_subparser().
- **aliases** – Alternative names for this subcommand. This is passed as the alias argument to ArgumentParser.add_subparser().

Returns Wrapper function that can receive an argparse.Namespace

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

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

Parameters

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

Returns function that gets passed argparse-parsed args in a `Namespace` A *cmd2.argparse_custom.Cmd2AttributeWrapper* called `cmd2_statement` is included in the `Namespace` to provide access to the *cmd2.Statement* object that was created when parsing the command line. This can be useful if the command function needs to know the command line.

Example

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-p', '--piglatin', action='store_true', help='atinLay')
>>> parser.add_argument('-s', '--shout', action='store_true', help='N00B␣
↪EMULATION MODE')
>>> parser.add_argument('-r', '--repeat', type=int, help='output [n] times')
>>> parser.add_argument('words', nargs='+', help='words to print')
>>>
>>> class MyApp(cmd2.Cmd):
>>>     @cmd2.with_argparser(parser, preserve_quotes=True)
>>>     def do_argprint(self, args):
>>>         "Print the options and argument list this options command was called␣
↪with."
>>>         self.poutput('args: {!r}'.format(args))
```

Example with unknown args

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-p', '--piglatin', action='store_true', help='atinLay')
>>> parser.add_argument('-s', '--shout', action='store_true', help='N00B␣
↪EMULATION MODE')
>>> parser.add_argument('-r', '--repeat', type=int, help='output [n] times')
>>>
```

(continues on next page)

(continued from previous page)

```
>>> class MyApp(cmd2.Cmd):
>>>     @cmd2.with_argparser(parser, with_unknown_args=True)
>>>     def do_argprint(self, args, unknown):
>>>         "Print the options and argument list this options command was called␣
↪with."
>>>         self.poutput('args: {!r}'.format(args))
>>>         self.poutput('unknowns: {}'.format(unknown))
```

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

Deprecated decorator. Use *with_argparser(parser, with_unknown_args=True)* instead.

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

Parameters

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

Returns function that gets passed argparse-parsed args in a `Namespace` and a list of unknown argument strings. A *cmd2.argparse_custom.Cmd2AttributeWrapper* called `cmd2_statement` is included in the `Namespace` to provide access to the *cmd2.Statement* object. that was created when parsing the command line. This can be useful if the command function needs to know the command line.

Example

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-p', '--piglatin', action='store_true', help='atinLay')
>>> parser.add_argument('-s', '--shout', action='store_true', help='N00B␣
↪EMULATION MODE')
>>> parser.add_argument('-r', '--repeat', type=int, help='output [n] times')
>>>
>>> class MyApp(cmd2.Cmd):
>>>     @cmd2.with_argparser(parser, with_unknown_args=True)
>>>     def do_argprint(self, args, unknown):
>>>         "Print the options and argument list this options command was called␣
↪with."
>>>         self.poutput('args: {!r}'.format(args))
>>>         self.poutput('unknowns: {}'.format(unknown))
```

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

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

Parameters

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

Returns function that gets passed a list of argument strings

Example

```
>>> class MyApp(cmd2.Cmd):
>>>     @cmd2.with_argument_list
>>>     def do_echo(self, arglist):
>>>         self.poutput(' '.join(arglist)
```

`cmd2.decorators.`**with_category**(*category: str*) → Callable

A decorator to apply a category to a `do_*` command method.

Parameters **category** – the name of the category in which this command should be grouped when displaying the list of commands.

Example

```
>>> class MyApp(cmd2.Cmd):
>>>   @cmd2.with_category('Text Functions')
>>>   def do_echo(self, args)
>>>     self.poutput(args)
```

For an alternative approach to categorizing commands using a function, see *categorize()*

7.1.8 cmd2.exceptions

Custom cmd2 exceptions

class `cmd2.exceptions.`**SkipPostcommandHooks**

Custom exception class for when a command has a failure bad enough to skip post command hooks, but not bad enough to print the exception to the user.

class `cmd2.exceptions.`**Cmd2ArgparseError**

A `SkipPostcommandHooks` exception for when a command fails to parse its arguments. Normally argparse raises a SystemExit exception in these cases. To avoid stopping the command loop, catch the SystemExit and raise this instead. If you still need to run post command hooks after parsing fails, just return instead of raising an exception.

class `cmd2.exceptions.`**CommandSetRegistrationError**

Exception that can be thrown when an error occurs while a CommandSet is being added or removed from a cmd2 application.

7.1.9 cmd2.history

Classes for storing the history of previously entered commands.

class `cmd2.history.`**History**(*seq=()*)

A list of *HistoryItem* objects with additional methods for searching and managing the list.

Cmd instantiates this class into the *history* attribute, and adds commands to it as a user enters them.

See *History* for information about the built-in command which allows users to view, search, run, and save previously entered commands.

Developers interested in accessing previously entered commands can use this class to gain access to the historical record.

append(*new: cmd2.parsing.Statement*) → None
Append a new statement to the end of the History list.

Parameters **new** – Statement object which will be composed into a HistoryItem and added to the end of the list

clear() → None
Remove all items from the History list.

get(*index: Union[int, str]*) → cmd2.history.HistoryItem
Get item from the History list using 1-based indexing.

Parameters **index** – optional item to get (index as either integer or string)

Returns a single *HistoryItem*

regex_search(*regex: str*, *include_persisted: bool = False*) → List[cmd2.history.HistoryItem]
Find history items which match a given regular expression

Parameters

- **regex** – the regular expression to search for.
- **include_persisted** – if True, then search full history including persisted history

Returns a list of history items, or an empty list if the string was not found

span(*span: str*, *include_persisted: bool = False*) → List[cmd2.history.HistoryItem]
Return an index or slice of the History list,

Parameters

- **span** – string containing an index or a slice
- **include_persisted** – if True, then retrieve full results including from persisted history

Returns a list of HistoryItems

This method can accommodate input in any of these forms:

a -a a..b or a:b a.. or a: ..a or :a -a.. or -a: ..-a or :-a

Different from native python indexing and slicing of arrays, this method uses 1-based array numbering. Users who are not programmers can't grok zero based numbering. Programmers can sometimes grok zero based numbering. Which reminds me, there are only two hard problems in programming:

- naming
- cache invalidation
- off by one errors

start_session() → None
Start a new session, thereby setting the next index as the first index in the new session.

str_search(*search: str*, *include_persisted: bool = False*) → List[cmd2.history.HistoryItem]
Find history items which contain a given string

Parameters

- **search** – the string to search for
- **include_persisted** – if True, then search full history including persisted history

Returns a list of history items, or an empty list if the string was not found

truncate(*max_length: int*) → None
Truncate the length of the history, dropping the oldest items if necessary

Parameters **max_length** – the maximum length of the history, if negative, all history items will be deleted

Returns nothing

class cmd2.history.**HistoryItem**(*statement=None*, *idx=None*)
Class used to represent one command in the history list

statement
The *Statement* object parsed from user input

idx
The 1-based index of this statement in the history list

expanded
Return the command as run which includes shortcuts and aliases resolved plus any changes made in hooks

Proxy property for `self.statement.expanded_command_line`

pr(*script=False*, *expanded=False*, *verbose=False*) → str
Represent this item in a pretty fashion suitable for printing.

If you pass verbose=True, script and expanded will be ignored

Returns pretty print string version of a HistoryItem

raw
The raw input from the user for this item.

Proxy property for `self.statement.raw`

7.1.10 cmd2.parsing

Classes for parsing and storing user input.

class cmd2.parsing.**StatementParser**(*terminators: Optional[Iterable[str]] = None*, *multiline_commands: Optional[Iterable[str]] = None*, *aliases: Optional[Dict[str, str]] = None*, *shortcuts: Optional[Dict[str, str]] = None*)
Parse user input as a string into discrete command components.

__init__(*terminators: Optional[Iterable[str]] = None*, *multiline_commands: Optional[Iterable[str]] = None*, *aliases: Optional[Dict[str, str]] = None*, *shortcuts: Optional[Dict[str, str]] = None*) → None
Initialize an instance of StatementParser.

The following will get converted to an immutable tuple before storing internally: terminators, multiline commands, and shortcuts.

Parameters

- **terminators** – iterable containing strings which should terminate commands
- **multiline_commands** – iterable containing the names of commands that accept multiline input
- **aliases** – dictionary containing aliases
- **shortcuts** – dictionary containing shortcuts

get_command_arg_list(*command_name: str, to_parse: Union[cmd2.parsing.Statement, str], preserve_quotes: bool*) → Tuple[cmd2.parsing.Statement, List[str]]

Convenience method used by the argument parsing decorators.

Retrieves just the arguments being passed to their `do_*` methods as a list.

Parameters

- **command_name** – name of the command being run
- **to_parse** – what is being passed to the `do_*` method. It can be one of two types:
  1. An already parsed *Statement*
  2. An argument string in cases where a `do_*` method is explicitly called. Calling `do_help('alias create')` would cause `to_parse` to be 'alias create'.

     In this case, the string will be converted to a *Statement* and returned along with the argument list.
- **preserve_quotes** – if `True`, then quotes will not be stripped from the arguments

Returns A tuple containing the *Statement* and a list of strings representing the arguments

is_valid_command(*word: str*, *, *is_subcommand: bool = False*) → Tuple[bool, str]

Determine whether a word is a valid name for a command.

Commands can not include redirection characters, whitespace, or termination characters. They also cannot start with a shortcut.

Parameters

- **word** – the word to check as a command
- **is_subcommand** – Flag whether this command name is a subcommand name

Returns a tuple of a boolean and an error string

If word is not a valid command, return `False` and an error string suitable for inclusion in an error message of your choice:

```
checkit = '>'
valid, errmsg = statement_parser.is_valid_command(checkit)
if not valid:
    errmsg = "alias: {}".format(errmsg)
```

parse(*line: str*) → cmd2.parsing.Statement

Tokenize the input and parse it into a *Statement* object, stripping comments, expanding aliases and shortcuts, and extracting output redirection directives.

Parameters **line** – the command line being parsed

Returns a new *Statement* object

Raises Cmd2ShlexError if a shlex error occurs (e.g. No closing quotation)

parse_command_only(*rawinput: str*) → cmd2.parsing.Statement

Partially parse input into a *Statement* object.

The command is identified, and shortcuts and aliases are expanded. Multiline commands are identified, but terminators and output redirection are not parsed.

This method is used by tab completion code and therefore must not generate an exception if there are unclosed quotes.

The *Statement* object returned by this method can at most contain values in the following attributes: *args*, *raw*, *command*, *multiline_command*

args will include all output redirection clauses and command terminators.

Different from *parse()* this method does not remove redundant whitespace within args. However, it does ensure args has no leading or trailing whitespace.

Parameters **rawinput** – the command line as entered by the user

Returns a new *Statement* object

split_on_punctuation(*tokens: List[str]*) → List[str]
Further splits tokens from a command line using punctuation characters.

Punctuation characters are treated as word breaks when they are in unquoted strings. Each run of punctuation characters is treated as a single token.

Parameters **tokens** – the tokens as parsed by shlex

Returns a new list of tokens, further split using punctuation

tokenize(*line: str*) → List[str]
Lex a string into a list of tokens. Shortcuts and aliases are expanded and comments are removed.

Parameters **line** – the command line being lexed

Returns A list of tokens

Raises Cmd2ShlexError if a shlex error occurs (e.g. No closing quotation)

class cmd2.**Statement**(*args=''*, *raw=''*, *command=''*, *arg_list=NOTHING*, *multiline_command=''*, *terminator=''*, *suffix=''*, *pipe_to=''*, *output=''*, *output_to=''*)
String subclass with additional attributes to store the results of parsing.

The cmd module in the standard library passes commands around as a string. To retain backwards compatibility, cmd2 does the same. However, we need a place to capture the additional output of the command parsing, so we add our own attributes to this subclass.

Instances of this class should not be created by anything other than the *cmd2.parsing.StatementParser.parse()* method, nor should any of the attributes be modified once the object is created.

The string portion of the class contains the arguments, but not the command, nor the output redirection clauses.

Tips:

1. argparse is your friend for anything complex. cmd2 has two decorators (*with_argparser()*, and *with_argparser_and_unknown_args()*) which you can use to make your command method receive a namespace of parsed arguments, whether positional or denoted with switches.
2. For commands with simple positional arguments, use *args* or *arg_list*
3. If you don't want to have to worry about quoted arguments, see *argv* for a trick which strips quotes off for you.

command
The name of the command after shortcuts and macros have been expanded

args
The arguments to the command as a string with spaces between the words, excluding output redirection and command terminators. If the user used quotes in their input, they remain here, and you will have to handle them on your own.

arg_list
The arguments to the command as a list, excluding output redirection and command terminators. Each argument is represented as an element in the list. Quoted arguments remain quoted. If you want to remove the quotes, use *cmd2.utils.strip_quotes()* or use `argv[1:]`

raw
If you want full access to exactly what the user typed at the input prompt you can get it, but you'll have to parse it on your own, including:

- shortcuts and aliases
- quoted commands and arguments
- output redirection
- multi-line command terminator handling

If you use multiline commands, all the input will be passed to you in this string, but there will be embedded newlines where the user hit return to continue the command on the next line.

multiline_command
If the command is a multi-line command, the name of the command will be in this attribute. Otherwise, it will be an empty string.

terminator
If the command is a multi-line command, this attribute contains the termination character entered by the user to signal the end of input

suffix
Any characters present between the input terminator and the output redirection tokens.

pipe_to
If the user piped the output to a shell command, this attribute contains the entire shell command as a string. Otherwise it is an empty string.

output
If output was redirected by the user, this contains the redirection token, i.e. >>.

output_to
If output was redirected by the user, this contains the requested destination with quotes preserved.

argv
a list of arguments a-la `sys.argv`.

The first element of the list is the command after shortcut and macro expansion. Subsequent elements of the list contain any additional arguments, with quotes removed, just like bash would. This is very useful if you are going to use `argparse.parse_args()`.

If you want to strip quotes from the input, you can use `argv[1:]`.

command_and_args
Combine command and args with a space separating them.

Quoted arguments remain quoted. Output redirection and piping are excluded, as are any command terminators.

expanded_command_line
Concatenate *command_and_args()* and *post_command()*

post_command
A string containing any ending terminator, suffix, and redirection chars

7.1.11 cmd2.plugin

class cmd2.plugin.**PostparsingData**(*stop*, *statement*)
Data class containing information passed to postparsing hook methods

stop
Request the command loop terminate by setting `True`

statement
The *Statement* object parsed from user input

class cmd2.plugin.**PrecommandData**(*statement*)
Data class containing information passed to precommand hook methods

statement
The *Statement* object parsed from user input

class cmd2.plugin.**PostcommandData**(*stop*, *statement*)
Data class containing information passed to postcommand hook methods

stop
Request the command loop terminate by setting `True`

statement
The *Statement* object parsed from user input

class cmd2.plugin.**CommandFinalizationData**(*stop*, *statement*)
Data class containing information passed to command finalization hook methods

stop
Request the command loop terminate by setting `True`

statement
The *Statement* object parsed from user input

7.1.12 cmd2.py_bridge

Bridges calls made inside of a Python environment to the Cmd2 host app while maintaining a reasonable degree of isolation between the two.

class cmd2.py_bridge.**CommandResult**
Encapsulates the results from a cmd2 app command

Stdout str - output captured from stdout while this command is executing

Stderr str - output captured from stderr while this command is executing None if no error captured.

Stop bool - return value of onecmd_plus_hooks after it runs the given command line.

Data possible data populated by the command.

Any combination of these fields can be used when developing a scripting API for a given command. By default stdout, stderr, and stop will be captured for you. If there is additional command specific data, then write that to cmd2's last_result member. That becomes the data member of this tuple.

In some cases, the data member may contain everything needed for a command and storing stdout and stderr might just be a duplication of data that wastes memory. In that case, the StdSim can be told not to store output with its pause_storage member. While this member is True, any output sent to StdSim won't be saved in its buffer.

The code would look like this:

```
if isinstance(self.stdout, StdSim):
    self.stdout.pause_storage = True

if isinstance(sys.stderr, StdSim):
    sys.stderr.pause_storage = True
```

See *StdSim* for more information.

Note: Named tuples are immutable. The contents are there for access, not for modification.

class cmd2.py_bridge.**PyBridge**(*cmd2_app*)
Provides a Python API wrapper for application commands.

7.1.13 cmd2.table_creator

class cmd2.table_creator.**HorizontalAlignment**
Horizontal alignment of text in a cell

CENTER = 2

LEFT = 1

RIGHT = 3

class cmd2.table_creator.**VerticalAlignment**
Vertical alignment of text in a cell

BOTTOM = 3

MIDDLE = 2

TOP = 1

class cmd2.table_creator.**Column**(*header: str*, ***, *width: Optional[int] = None*, *header_horiz_align: cmd2.table_creator.HorizontalAlignment = <HorizontalAlignment.LEFT: 1>*, *header_vert_align: cmd2.table_creator.VerticalAlignment = <VerticalAlignment.BOTTOM: 3>*, *data_horiz_align: cmd2.table_creator.HorizontalAlignment = <HorizontalAlignment.LEFT: 1>*, *data_vert_align: cmd2.table_creator.VerticalAlignment = <VerticalAlignment.TOP: 1>*, *max_data_lines: Union[int, float] = inf*)
Table column configuration

__init__(*header: str*, ***, *width: Optional[int] = None*, *header_horiz_align: cmd2.table_creator.HorizontalAlignment = <HorizontalAlignment.LEFT: 1>*, *header_vert_align: cmd2.table_creator.VerticalAlignment = <VerticalAlignment.BOTTOM: 3>*, *data_horiz_align: cmd2.table_creator.HorizontalAlignment = <HorizontalAlignment.LEFT: 1>*, *data_vert_align: cmd2.table_creator.VerticalAlignment = <VerticalAlignment.TOP: 1>*, *max_data_lines: Union[int, float] = inf*) → None
Column initializer

Parameters

- **header** – label for column header
- **width** – display width of column. This does not account for any borders or padding which may be added (e.g pre_line, inter_cell, and post_line). Header and data text

wrap within this width using word-based wrapping (defaults to width of header or 1 if header is blank)

- **header_horiz_align** – horizontal alignment of header cells (defaults to left)
- **header_vert_align** – vertical alignment of header cells (defaults to bottom)
- **data_horiz_align** – horizontal alignment of data cells (defaults to left)
- **data_vert_align** – vertical alignment of data cells (defaults to top)
- **max_data_lines** – maximum lines allowed in a data cell. If line count exceeds this, then the final line displayed will be truncated with an ellipsis. (defaults to INFINITY)

Raises ValueError if width is less than 1

Raises ValueError if max_data_lines is less than 1

class cmd2.table_creator.**TableCreator**(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4*)

Base table creation class. This class handles ANSI style sequences and characters with display widths greater than 1 when performing width calculations. It was designed with the ability to build tables one row at a time. This helps when you have large data sets that you don't want to hold in memory or when you receive portions of the data set incrementally.

TableCreator has one public method: generate_row()

This function and the Column class provide all features needed to build tables with headers, borders, colors, horizontal and vertical alignment, and wrapped text. However, it's generally easier to inherit from this class and implement a more granular API rather than use TableCreator directly. There are ready-to-use examples of this defined after this class.

__init__(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4*) → None

TableCreator initializer

Parameters

- **cols** – column definitions for this table
- **tab_width** – all tabs will be replaced with this many spaces. If a row's fill_char is a tab, then it will be converted to one space.

generate_row(*, *row_data: Optional[Sequence[Any]] = None*, *fill_char: str = ' '*, *pre_line: str = ''*, *inter_cell: str = ' '*, *post_line: str = ''*) → str

Generate a header or data table row

Parameters

- **row_data** – If this is None then a header row is generated. Otherwise data should have an entry for each column in the row. (Defaults to None)
- **fill_char** – character that fills remaining space in a cell. Defaults to space. If this is a tab, then it will be converted to one space. (Cannot be a line breaking character)
- **pre_line** – string to print before each line of a row. This can be used for a left row border and padding before the first cell's text. (Defaults to blank)
- **inter_cell** – string to print where two cells meet. This can be used for a border between cells and padding between it and the 2 cells' text. (Defaults to 2 spaces)
- **post_line** – string to print after each line of a row. This can be used for padding after the last cell's text and a right row border. (Defaults to blank)

Returns row string

Raises ValueError if data isn't the same length as self.cols

Raises TypeError if fill_char is more than one character (not including ANSI style sequences)

Raises ValueError if fill_char, pre_line, inter_cell, or post_line contains an unprintable character like a newline

class cmd2.table_creator.**SimpleTable**(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4, divider_char: Optional[str] = '-'*)

Implementation of TableCreator which generates a borderless table with an optional divider row after the header. This class can be used to create the whole table at once or one row at a time.

__init__(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4, divider_char: Optional[str] = '-'*) → None

SimpleTable initializer

Parameters

- **cols** – column definitions for this table
- **tab_width** – all tabs will be replaced with this many spaces. If a row's fill_char is a tab, then it will be converted to one space.
- **divider_char** – optional character used to build the header divider row. Set this to None if you don't want a divider row. Defaults to dash. (Cannot be a line breaking character)

Raises TypeError if fill_char is more than one character (not including ANSI style sequences)

Raises ValueError if text or fill_char contains an unprintable character

classmethod base_width(*num_cols: int*) → int

Utility method to calculate the display width required for a table before data is added to it. This is useful when determining how wide to make your columns to have a table be a specific width.

Parameters **num_cols** – how many columns the table will have

Returns base width

Raises ValueError if num_cols is less than 1

generate_data_row(*row_data: Sequence[Any]*) → str

Generate a data row

Parameters **row_data** – data with an entry for each column in the row

Returns data row string

generate_header() → str

Generate table header with an optional divider row

generate_table(*table_data: Sequence[Sequence[Any]], *, include_header: bool = True, row_spacing: int = 1*) → str

Generate a table from a data set

Parameters

- **table_data** – Data with an entry for each data row of the table. Each entry should have data for each column in the row.
- **include_header** – If True, then a header will be included at top of table. (Defaults to True)
- **row_spacing** – A number 0 or greater specifying how many blank lines to place between each row (Defaults to 1)

Raises ValueError if row_spacing is less than 0

total_width() → int
Calculate the total display width of this table

class cmd2.table_creator.**BorderedTable**(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4, column_borders: bool = True, padding: int = 1*)
Implementation of TableCreator which generates a table with borders around the table and between rows. Borders between columns can also be toggled. This class can be used to create the whole table at once or one row at a time.

__init__(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4, column_borders: bool = True, padding: int = 1*) → None
BorderedTable initializer

Parameters

- **cols** – column definitions for this table
- **tab_width** – all tabs will be replaced with this many spaces. If a row's fill_char is a tab, then it will be converted to one space.
- **column_borders** – if True, borders between columns will be included. This gives the table a grid-like appearance. Turning off column borders results in a unified appearance between a row's cells. (Defaults to True)
- **padding** – number of spaces between text and left/right borders of cell

Raises ValueError if padding is less than 0

classmethod base_width(*num_cols: int*, *, *column_borders: bool = True*, *padding: int = 1*) → int
Utility method to calculate the display width required for a table before data is added to it. This is useful when determining how wide to make your columns to have a table be a specific width.

Parameters

- **num_cols** – how many columns the table will have
- **column_borders** – if True, borders between columns will be included in the calculation (Defaults to True)
- **padding** – number of spaces between text and left/right borders of cell

Returns base width

Raises ValueError if num_cols is less than 1

generate_data_row(*row_data: Sequence[Any]*) → str
Generate a data row

Parameters **row_data** – data with an entry for each column in the row

Returns data row string

generate_header() → str
Generate table header

generate_header_bottom_border()
Generate a border which appears at the bottom of the header

generate_row_bottom_border()
Generate a border which appears at the bottom of rows

generate_table(*table_data: Sequence[Sequence[Any]], *, include_header: bool = True*) → str
Generate a table from a data set

Parameters

- **table_data** – Data with an entry for each data row of the table. Each entry should have data for each column in the row.
- **include_header** – If True, then a header will be included at top of table. (Defaults to True)

generate_table_bottom_border()
Generate a border which appears at the bottom of the table

generate_table_top_border()
Generate a border which appears at the top of the header and data section

total_width() → int
Calculate the total display width of this table

class cmd2.table_creator.**AlternatingTable**(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4, column_borders: bool = True, padding: int = 1, bg_odd: Optional[cmd2.ansi.bg] = None, bg_even: Optional[cmd2.ansi.bg] = <bg.bright_black: '\x1b[100m'>*)
Implementation of BorderedTable which uses background colors to distinguish between rows instead of row border lines. This class can be used to create the whole table at once or one row at a time.

__init__(*cols: Sequence[cmd2.table_creator.Column], *, tab_width: int = 4, column_borders: bool = True, padding: int = 1, bg_odd: Optional[cmd2.ansi.bg] = None, bg_even: Optional[cmd2.ansi.bg] = <bg.bright_black: '\x1b[100m'>*) → None
AlternatingTable initializer

Parameters

- **cols** – column definitions for this table
- **tab_width** – all tabs will be replaced with this many spaces. If a row's fill_char is a tab, then it will be converted to one space.
- **column_borders** – if True, borders between columns will be included. This gives the table a grid-like appearance. Turning off column borders results in a unified appearance between a row's cells. (Defaults to True)
- **padding** – number of spaces between text and left/right borders of cell
- **bg_odd** – optional background color for odd numbered rows (defaults to None)
- **bg_even** – optional background color for even numbered rows (defaults to gray)

Raises ValueError if padding is less than 0

generate_data_row(*row_data: Sequence[Any]*) → str
Generate a data row

Parameters **row_data** – data with an entry for each column in the row

Returns data row string

generate_table(*table_data: Sequence[Sequence[Any]], *, include_header: bool = True*) → str
Generate a table from a data set

Parameters

- **table_data** – Data with an entry for each data row of the table. Each entry should have data for each column in the row.
- **include_header** – If True, then a header will be included at top of table. (Defaults to True)

7.1.14 cmd2.utils

Settings

class cmd2.utils.**Settable**(*name: str, val_type: Callable, description: str, *, onchange_cb: Callable[[str, Any, Any], Any] = None, choices: Iterable[T_co] = None, choices_function: Optional[Callable] = None, choices_method: Optional[Callable] = None, completer_function: Optional[Callable] = None, completer_method: Optional[Callable] = None*)

Used to configure a cmd2 instance member to be settable via the set command in the CLI

__init__(*name: str, val_type: Callable, description: str, *, onchange_cb: Callable[[str, Any, Any], Any] = None, choices: Iterable[T_co] = None, choices_function: Optional[Callable] = None, choices_method: Optional[Callable] = None, completer_function: Optional[Callable] = None, completer_method: Optional[Callable] = None*)

Settable Initializer

Parameters

- **name** – name of the instance attribute being made settable
- **val_type** – callable used to cast the string value from the command line into its proper type and even validate its value. Setting this to bool provides tab completion for true/false and validation using str_to_bool(). The val_type function should raise an exception if it fails. This exception will be caught and printed by Cmd.do_set().
- **description** – string describing this setting
- **onchange_cb** – optional function or method to call when the value of this settable is altered by the set command. (e.g. onchange_cb=self.debug_changed)

  Cmd.do_set() passes the following 3 arguments to onchange_cb: param_name: str - name of the changed parameter old_value: Any - the value before being changed new_value: Any - the value after being changed

The following optional settings provide tab completion for a parameter's values. They correspond to the same settings in argparse-based tab completion. A maximum of one of these should be provided.

Parameters

- **choices** – iterable of accepted values
- **choices_function** – function that provides choices for this argument
- **choices_method** – cmd2-app method that provides choices for this argument (See note below)
- **completer_function** – tab completion function that provides choices for this argument
- **completer_method** – cmd2-app tab completion method that provides choices for this argument (See note below)

Note: For choices_method and completer_method, do not set them to a bound method. This is because ArgparseCompleter passes the self argument explicitly to these functions.

Therefore instead of passing something like self.path_complete, pass cmd2.Cmd.path_complete.

Quote Handling

cmd2.utils.**is_quoted**(*arg: str*) → bool
Checks if a string is quoted

Parameters **arg** – the string being checked for quotes

Returns True if a string is quoted

cmd2.utils.**quote_string**(*arg: str*) → str
Quote a string

cmd2.utils.**quote_string_if_needed**(*arg: str*) → str
Quote a string if it contains spaces and isn't already quoted

cmd2.utils.**strip_quotes**(*arg: str*) → str
Strip outer quotes from a string.

Applies to both single and double quotes.

Parameters **arg** – string to strip outer quotes from

Returns same string with potentially outer quotes stripped

IO Handling

class cmd2.utils.**StdSim**(*inner_stream*, *, *echo: bool = False*, *encoding: str = 'utf-8'*, *errors: str = 'replace'*)
Class to simulate behavior of sys.stdout or sys.stderr. Stores contents in internal buffer and optionally echos to the inner stream it is simulating.

clear() → None
Clear the internal contents

getbytes() → bytes
Get the internal contents as bytes

getvalue() → str
Get the internal contents as a str

isatty() → bool
StdSim only considered an interactive stream if *echo* is True and *inner_stream* is a tty.

line_buffering
Handle when the inner stream doesn't have a line_buffering attribute which is the case when running unit tests because pytest sets stdout to a pytest EncodedFile object.

read(*size: Optional[int] = -1*) → str
Read from the internal contents as a str and then clear them out

readbytes() → bytes
Read from the internal contents as bytes and then clear them out

write(*s: str*) → None
Add str to internal bytes buffer and if echo is True, echo contents to inner stream

class cmd2.utils.**ByteBuf**(*std_sim_instance: cmd2.utils.StdSim*)
Used by StdSim to write binary data and stores the actual bytes written

write(*b: bytes*) → None
Add bytes to internal bytes buffer and if echo is True, echo contents to inner stream.

class cmd2.utils.**ProcReader**(*proc: subprocess.Popen, stdout: Union[cmd2.utils.StdSim, TextIO], stderr: Union[cmd2.utils.StdSim, TextIO]*)
Used to capture stdout and stderr from a Popen process if any of those were set to subprocess.PIPE. If neither are pipes, then the process will run normally and no output will be captured.

send_sigint() → None
Send a SIGINT to the process similar to if <Ctrl>+C were pressed

terminate() → None
Terminate the process

wait() → None
Wait for the process to finish

Tab Completion

class cmd2.utils.**CompletionError**(**args*, *apply_style: bool = True*, ***kwargs*)
Raised during tab completion operations to report any sort of error you want printed. This can also be used just to display a message, even if it's not an error. For instance, ArgparseCompleter raises CompletionErrors to display tab completion hints and sets apply_style to False so hints aren't colored like error text.

Example use cases

- Reading a database to retrieve a tab completion data set failed
- A previous command line argument that determines the data set being completed is invalid
- Tab completion hints

cmd2.utils.**basic_complete**(*text: str, line: str, begidx: int, endidx: int, match_against: Iterable[T_co]*) → List[str]
Basic tab completion function that matches against a list of strings without considering line contents or cursor position. The args required by this function are defined in the header of Python's cmd.py.

Parameters

- **text** – the string prefix we are attempting to match (all matches must begin with it)
- **line** – the current input line with leading whitespace removed
- **begidx** – the beginning index of the prefix text
- **endidx** – the ending index of the prefix text
- **match_against** – the strings being matched against

Returns a list of possible tab completions

Text Alignment

class cmd2.utils.**TextAlignment**
Horizontal text alignment

CENTER = 2

LEFT = 1

RIGHT = 3

cmd2.utils.**align_text**(*text: str*, *alignment: cmd2.utils.TextAlignment*, ***, *fill_char: str = ' '*, *width: Optional[int] = None*, *tab_width: int = 4*, *truncate: bool = False*) → str

Align text for display within a given width. Supports characters with display widths greater than 1. ANSI style sequences do not count toward the display width. If text has line breaks, then each line is aligned independently.

There are convenience wrappers around this function: align_left(), align_center(), and align_right()

Parameters

- **text** – text to align (can contain multiple lines)
- **alignment** – how to align the text
- **fill_char** – character that fills the alignment gap. Defaults to space. (Cannot be a line breaking character)
- **width** – display width of the aligned text. Defaults to width of the terminal.
- **tab_width** – any tabs in the text will be replaced with this many spaces. if fill_char is a tab, then it will be converted to one space.
- **truncate** – if True, then each line will be shortened to fit within the display width. The truncated portions are replaced by a '…' character. Defaults to False.

Returns aligned text

Raises TypeError if fill_char is more than one character (not including ANSI style sequences)

Raises ValueError if text or fill_char contains an unprintable character

Raises ValueError if width is less than 1

cmd2.utils.**align_left**(*text: str*, ***, *fill_char: str = ' '*, *width: Optional[int] = None*, *tab_width: int = 4*, *truncate: bool = False*) → str

Left align text for display within a given width. Supports characters with display widths greater than 1. ANSI style sequences do not count toward the display width. If text has line breaks, then each line is aligned independently.

Parameters

- **text** – text to left align (can contain multiple lines)
- **fill_char** – character that fills the alignment gap. Defaults to space. (Cannot be a line breaking character)
- **width** – display width of the aligned text. Defaults to width of the terminal.
- **tab_width** – any tabs in the text will be replaced with this many spaces. if fill_char is a tab, then it will be converted to one space.
- **truncate** – if True, then text will be shortened to fit within the display width. The truncated portion is replaced by a '…' character. Defaults to False.

Returns left-aligned text

Raises TypeError if fill_char is more than one character (not including ANSI style sequences)

Raises ValueError if text or fill_char contains an unprintable character

Raises ValueError if width is less than 1

cmd2.utils.**align_right**(*text: str*, ***, *fill_char: str = ' '*, *width: Optional[int] = None*, *tab_width: int = 4*, *truncate: bool = False*) → str

Right align text for display within a given width. Supports characters with display widths greater than 1. ANSI style sequences do not count toward the display width. If text has line breaks, then each line is aligned independently.

Parameters

- **text** – text to right align (can contain multiple lines)
- **fill_char** – character that fills the alignment gap. Defaults to space. (Cannot be a line breaking character)
- **width** – display width of the aligned text. Defaults to width of the terminal.
- **tab_width** – any tabs in the text will be replaced with this many spaces. if fill_char is a tab, then it will be converted to one space.
- **truncate** – if True, then text will be shortened to fit within the display width. The truncated portion is replaced by a '…' character. Defaults to False.

Returns right-aligned text

Raises TypeError if fill_char is more than one character (not including ANSI style sequences)

Raises ValueError if text or fill_char contains an unprintable character

Raises ValueError if width is less than 1

cmd2.utils.**align_center**(*text: str*, **, fill_char: str = ' '*, *width: Optional[int] = None*, *tab_width: int = 4*, *truncate: bool = False*) → str

Center text for display within a given width. Supports characters with display widths greater than 1. ANSI style sequences do not count toward the display width. If text has line breaks, then each line is aligned independently.

Parameters

- **text** – text to center (can contain multiple lines)
- **fill_char** – character that fills the alignment gap. Defaults to space. (Cannot be a line breaking character)
- **width** – display width of the aligned text. Defaults to width of the terminal.
- **tab_width** – any tabs in the text will be replaced with this many spaces. if fill_char is a tab, then it will be converted to one space.
- **truncate** – if True, then text will be shortened to fit within the display width. The truncated portion is replaced by a '…' character. Defaults to False.

Returns centered text

Raises TypeError if fill_char is more than one character (not including ANSI style sequences)

Raises ValueError if text or fill_char contains an unprintable character

Raises ValueError if width is less than 1

cmd2.utils.**truncate_line**(*line: str*, *max_width: int*, ***, *tab_width: int = 4*) → str

Truncate a single line to fit within a given display width. Any portion of the string that is truncated is replaced by a '…' character. Supports characters with display widths greater than 1. ANSI style sequences do not count toward the display width.

If there are ANSI style sequences in the string after where truncation occurs, this function will append them to the returned string.

This is done to prevent issues caused in cases like: truncate_string(fg.blue + hello + fg.reset, 3) In this case, "hello" would be truncated before fg.reset resets the color from blue. Appending the remaining style sequences makes sure the style is in the same state had the entire string been printed. align_text() relies on this behavior when preserving style over multiple lines.

Parameters

- **line** – text to truncate

- **max_width** – the maximum display width the resulting string is allowed to have
- **tab_width** – any tabs in the text will be replaced with this many spaces

Returns line that has a display width less than or equal to width

Raises ValueError if text contains an unprintable character like a newline

Raises ValueError if max_width is less than 1

Miscellaneous

cmd2.utils.**str_to_bool**(*val: str*) → bool
Converts a string to a boolean based on its value.

Parameters **val** – string being converted

Returns boolean value expressed in the string

Raises ValueError if the string does not contain a value corresponding to a boolean value

cmd2.utils.**namedtuple_with_defaults**(*typename: str, field_names: Union[str, List[str]], default_values: collections.abc.Iterable = ()*)
Convenience function for defining a namedtuple with default values

From: https://stackoverflow.com/questions/11351032/namedtuple-and-default-values-for-optional-keyword-arguments

Examples:

```
>>> Node = namedtuple_with_defaults('Node', 'val left right')
>>> Node()
Node(val=None, left=None, right=None)
>>> Node = namedtuple_with_defaults('Node', 'val left right', [1, 2, 3])
>>> Node()
Node(val=1, left=2, right=3)
>>> Node = namedtuple_with_defaults('Node', 'val left right', {'right':7})
>>> Node()
Node(val=None, left=None, right=7)
>>> Node(4)
Node(val=4, left=None, right=7)
```

cmd2.utils.**categorize**(*func: Union[Callable, Iterable[Callable]], category: str*) → None
Categorize a function.

The help command output will group the passed function under the specified category heading

Parameters

- **func** – function or list of functions to categorize
- **category** – category to put it in

Example

```
>>> import cmd2
>>> class MyApp(cmd2.Cmd):
>>>   def do_echo(self, arglist):
>>>     self.poutput(' '.join(arglist)
>>>
>>>   cmd2.utils.categorize(do_echo, "Text Processing")
```

For an alternative approach to categorizing commands using a decorator, see *with_category()*

cmd2.utils.**remove_duplicates**(*list_to_prune: List[T]*) → List[T]
Removes duplicates from a list while preserving order of the items.

Parameters **list_to_prune** – the list being pruned of duplicates

Returns The pruned list

cmd2.utils.**alphabetical_sort**(*list_to_sort: Iterable[str]*) → List[str]
Sorts a list of strings alphabetically.

For example: ['a1', 'A11', 'A2', 'a22', 'a3']

To sort a list in place, don't call this method, which makes a copy. Instead, do this:

my_list.sort(key=norm_fold)

Parameters **list_to_sort** – the list being sorted

Returns the sorted list

cmd2.utils.**natural_sort**(*list_to_sort: Iterable[str]*) → List[str]
Sorts a list of strings case insensitively as well as numerically.

For example: ['a1', 'A2', 'a3', 'A11', 'a22']

To sort a list in place, don't call this method, which makes a copy. Instead, do this:

my_list.sort(key=natural_keys)

Parameters **list_to_sort** – the list being sorted

Returns the list sorted naturally

7.1.15 cmd2_ext_test

External Test Plugin

Modules

- *cmd2.Cmd* - functions and attributes of the main class in this library
- *cmd2.ansi* - convenience classes and functions for generating ANSI escape sequences to style text in the terminal
- *cmd2.argparse_completer* - classes for `argparse`-based tab completion
- *cmd2.argparse_custom* - classes and functions for extending `argparse`
- *cmd2.constants* - just like it says on the tin
- *cmd2.decorators* - decorators for `cmd2` commands
- *cmd2.exceptions* - custom `cmd2` exceptions
- *cmd2.history* - classes for storing the history of previously entered commands
- *cmd2.parsing* - classes for parsing and storing user input
- *cmd2.plugin* - data classes for hook methods
- *cmd2.py_bridge* - classes for bridging calls from the embedded python environment to the host app
- *cmd2.table_creator* - table creation module
- *cmd2.utils* - various utility classes and functions
- *cmd2_ext_test* - External test plugin

CHAPTER 8

Meta

Documentation Conventions

8.1 Documentation Conventions

8.1.1 Guiding Principles

Follow the Documentation Principles described by Write The Docs

In addition:

- We have gone to great lengths to retain compatibility with the standard library cmd, the documentation should make it easy for developers to understand how to move from cmd to cmd2, and what benefits that will provide
- We should provide both descriptive and reference documentation.
- API reference documentation should be generated from docstrings in the code
- Documentation should include rich hyperlinking to other areas of the documentation, and to the API reference

8.1.2 Style Checker

Use doc8 to check the style of the documentation. This tool can be invoked using the proper options by typing:

```
$ invoke doc8
```

8.1.3 Naming Files

All source files in the documentation must:

- have all lower case file names
- if the name has multiple words, separate them with an underscore

- end in '.rst'

8.1.4 Indenting

In reStructuredText all indenting is significant. Use 2 spaces per indenting level.

8.1.5 Wrapping

Hard wrap all text so that line lengths are no greater than 79 characters. It makes everything easier when editing documentation, and has no impact on reading documentation because we render to html.

8.1.6 Titles and Headings

reStructuredText allows flexibility in how headings are defined. You only have to worry about the heirarchy of headings within a single file. Sphinx magically handles the intra-file heirarchy on it's own. This magic means that no matter how you style titles and headings in the various files that comprise the documentation, Sphinx will render properly structured output. To ensure we have a similar consistency when viewing the source files, we use the following conventions for titles and headings:

1. When creating a heading for a section, do not use the overline and underline syntax. Use the underline syntax only:

```
Document Title
==============
```

2. The string of adornment characters on the line following the heading should be the same length as the title.

3. The title of a document should use the '=' adornment character on the next line and only one heading of this level should appear in each file.

4. Sections within a document should have their titles adorned with the '-' character:

```
Section Title
-------------
```

5. Subsections within a section should have their titles adorned with the '~' character:

```
Subsection Title
~~~~~~~~~~~~~~~~
```

6. Use two blank lines before every title unless it's the first heading in the file. Use one blank line after every heading.

7. If your document needs more than three levels of sections, break it into separate documents.

8.1.7 Inline Code

This documentation declares `python` as the default Sphinx domain. Python code or interactive Python sessions can be presented by either:

- finishing the preceding paragraph with a `::` and indenting the code
- use the `.. code-block::` directive

If you want to show non-Python code, like shell commands, then use `.. code-block:: shell`.

8.1.8 External Hyperlinks

If you want to use an external hyperlink target, define the target at the top of the page or the top of the section, not the bottom. The target definition should always appear before it is referenced.

8.1.9 Links To Other Documentation Pages and Sections

We use the Sphinx autosectionlabel extension. This allows you to reference any header in any document by:

```
See :ref:`features/argument_processing:Help Messages`
```

or:

```
See :ref:`custom title<features/argument_processing:Help Messages>`
```

Which render like

See *Help Messages*

and

See *custom title*

8.1.10 API Documentation

The API documentation is mostly pulled from docstrings in the source code using the Sphinx autodoc extension. However, Sphinx has issues generating documentation for instance attributes (see cmd2 issue 821 for the full discussion). We have chosen to not use code as the source of instance attribute documentation. Instead, it is added manually to the documentation files in `cmd2/docs/api`. See `cmd2/docs/api/cmd.rst` to see how to add documentation for an attribute.

For module data members and class attributes, the `autodoc` extension allows documentation in a comment with special formatting (using a #: to start the comment instead of just #), or in a docstring after the definition. This project has standardized on the docstring after the definition approach. Do not use the specially formatted comment approach.

When using the Sphix `autoclass` directive, it must be preceded by two blank lines like so:

```
Classes for storing the history of previously entered commands.


.. autoclass:: cmd2.history.History
    :members:


.. autoclass:: cmd2.history.HistoryItem
    :members:
```

8.1.11 Links to API Reference

To reference a method or function, use one of the following approaches:

1. Reference the full dotted path of the method:

   ```
   The :meth:`cmd2.Cmd.poutput` method is similar to the Python built-in
   print function.
   ```

Which renders as: The *cmd2.Cmd.poutput()* method is similar to the Python built-in print function.

2. Reference the full dotted path to the method, but only display the method name:

```
The :meth:`~cmd2.Cmd.poutput` method is similar to the Python built-in print function.
```

Which renders as: The *poutput()* method is similar to the Python built-in print function.

Avoid either of these approaches:

1. Reference just the class name without enough dotted path:

```
The :meth:`.Cmd.poutput` method is similar to the Python built-in print
function.
```

Because `cmd2.Cmd` subclasses `cmd.Cmd` from the standard library, this approach does not clarify which class it is referring to.

2. Reference just a method name:

```
The :meth:`poutput` method is similar to the Python built-in print
function.
```

While Sphinx may be smart enough to generate the correct output, the potential for multiple matching references is high, which causes Sphinx to generate warnings. The build pipeline that renders the documentation treats warnings as fatal errors. It's best to just be specific about what you are referencing.

See https://github.com/python-cmd2/cmd2/issues/821 for the discussion of how we determined this approach.

8.1.12 Referencing cmd2

Whenever you reference `cmd2` in the documentation, enclose it in double backticks. This indicates an inline literal in restructured text, and makes it stand out when rendered as html.

Python Module Index

C

cmd2.ansi, 102
cmd2.argparse_completer, 104
cmd2.argparse_custom, 105
cmd2.clipboard, 28
cmd2.command_definition, 109
cmd2.constants, 109
cmd2.decorators, 110
cmd2.py_bridge, 119

Index

Symbols

__init__() (*cmd2.Cmd method*), 89
__init__() (*cmd2.parsing.StatementParser method*), 115
__init__() (*cmd2.table_creator.AlternatingTable method*), 124
__init__() (*cmd2.table_creator.BorderedTable method*), 123
__init__() (*cmd2.table_creator.Column method*), 120
__init__() (*cmd2.table_creator.SimpleTable method*), 122
__init__() (*cmd2.table_creator.TableCreator method*), 121
__init__() (*cmd2.utils.Settable method*), 125

A

add_settable() (*cmd2.Cmd method*), 91
add_subparsers() (*cmd2.argparse_custom.Cmd2ArgumentParser method*), 108
aliases (*cmd2.Cmd attribute*), 91
align_center() (*in module cmd2.utils*), 129
align_left() (*in module cmd2.utils*), 128
align_right() (*in module cmd2.utils*), 128
align_text() (*in module cmd2.utils*), 127
allow_style (*cmd2.Cmd attribute*), 91
allow_style (*in module cmd2.ansi*), 102
alphabetical_sort() (*in module cmd2.utils*), 131
ALPHABETICAL_SORT_KEY() (*cmd2.Cmd method*), 91
AlternatingTable (*class in cmd2.table_creator*), 124
append() (*cmd2.history.History method*), 114
arg_list (*cmd2.Statement attribute*), 117
ArgparseCompleter (*class in cmd2.argparse_completer*), 104
args (*cmd2.Statement attribute*), 117
argv (*cmd2.Statement attribute*), 118
as_subcommand_to() (*in module cmd2.decorators*), 110
async_alert() (*cmd2.Cmd method*), 91
async_alert_str() (*in module cmd2.ansi*), 102
async_update_prompt() (*cmd2.Cmd method*), 92

B

base_width() (*cmd2.table_creator.BorderedTable class method*), 123
base_width() (*cmd2.table_creator.SimpleTable class method*), 122
basic_complete() (*in module cmd2.utils*), 127
bg (*class in cmd2.ansi*), 103
bg_lookup() (*in module cmd2.ansi*), 103
BG_RESET (*in module cmd2.ansi*), 102
BorderedTable (*class in cmd2.table_creator*), 123
BOTTOM (*cmd2.table_creator.VerticalAlignment attribute*), 120
build_settables() (*cmd2.Cmd method*), 92
ByteBuf (*class in cmd2.utils*), 126

C

categorize() (*in module cmd2.utils*), 130
CENTER (*cmd2.table_creator.HorizontalAlignment attribute*), 120
CENTER (*cmd2.utils.TextAlignment attribute*), 127
ChoicesCallable (*class in cmd2.argparse_custom*), 108
clear() (*cmd2.history.History method*), 114
clear() (*cmd2.utils.StdSim method*), 126
Cmd (*class in cmd2*), 89
cmd2.ansi (*module*), 102
cmd2.argparse_completer (*module*), 104
cmd2.argparse_custom (*module*), 105
cmd2.clipboard (*module*), 28
cmd2.command_definition (*module*), 109
cmd2.constants (*module*), 109
cmd2.decorators (*module*), 110
cmd2.py_bridge (*module*), 119
Cmd2ArgparseError (*class in cmd2.exceptions*), 113

Cmd2ArgumentParser (*class in cmd2.argparse_custom*), 108
Cmd2AttributeWrapper (*class in cmd2.argparse_custom*), 108
Cmd2HelpFormatter (*class in cmd2.argparse_custom*), 108
cmd_func() (*cmd2.Cmd method*), 92
cmdloop() (*cmd2.Cmd method*), 92
ColorBase (*class in cmd2.ansi*), 102
Column (*class in cmd2.table_creator*), 120
command (*cmd2.Statement attribute*), 117
command_and_args (*cmd2.Statement attribute*), 118
COMMAND_NAME (*in module cmd2.constants*), 109
CommandFinalizationData (*class in cmd2.plugin*), 119
CommandResult (*class in cmd2.py_bridge*), 119
CommandSet (*class in cmd2.command_definition*), 109
CommandSetRegistrationError (*class in cmd2.exceptions*), 113
complete() (*cmd2.Cmd method*), 92
complete_command() (*cmd2.argparse_completer.ArgparseCompleter method*), 104
complete_help_command() (*cmd2.Cmd method*), 93
complete_help_subcommands() (*cmd2.Cmd method*), 93
complete_set_value() (*cmd2.Cmd method*), 93
complete_subcommand_help() (*cmd2.argparse_completer.ArgparseCompleter method*), 104
CompletionError (*class in cmd2.utils*), 127
CompletionItem (*class in cmd2.argparse_custom*), 108
continuation_prompt (*cmd2.Cmd attribute*), 91

D

default() (*cmd2.Cmd method*), 93
DEFAULT_ARGUMENT_PARSER (*in module cmd2.argparse_custom*), 109
default_error (*cmd2.Cmd attribute*), 90
DEFAULT_SHORTCUTS (*in module cmd2.constants*), 109
delimiter_complete() (*cmd2.Cmd method*), 93
disable_category() (*cmd2.Cmd method*), 94
disable_command() (*cmd2.Cmd method*), 94
do__relative_run_script() (*cmd2.Cmd method*), 94
do_alias() (*cmd2.Cmd method*), 94
do_edit() (*cmd2.Cmd method*), 94
do_eof() (*cmd2.Cmd method*), 94
do_help() (*cmd2.Cmd method*), 94
do_history() (*cmd2.Cmd method*), 94
do_macro() (*cmd2.Cmd method*), 94
do_py() (*cmd2.Cmd method*), 95
do_quit() (*cmd2.Cmd method*), 95
do_run_pyscript() (*cmd2.Cmd method*), 95
do_run_script() (*cmd2.Cmd method*), 95
do_set() (*cmd2.Cmd method*), 95
do_shell() (*cmd2.Cmd method*), 95
do_shortcuts() (*cmd2.Cmd method*), 95

E

echo (*cmd2.Cmd attribute*), 91
enable_category() (*cmd2.Cmd method*), 95
enable_command() (*cmd2.Cmd method*), 95
error() (*cmd2.argparse_custom.Cmd2ArgumentParser method*), 108
expanded (*cmd2.history.HistoryItem attribute*), 115
expanded_command_line (*cmd2.Statement attribute*), 118

F

fg (*class in cmd2.ansi*), 103
fg_lookup() (*in module cmd2.ansi*), 103
FG_RESET (*in module cmd2.ansi*), 102
find_commandset_for_command() (*cmd2.Cmd method*), 95
find_commandsets() (*cmd2.Cmd method*), 95
flag_based_complete() (*cmd2.Cmd method*), 95
format_help() (*cmd2.argparse_completer.ArgparseCompleter method*), 104
format_help() (*cmd2.argparse_custom.Cmd2ArgumentParser method*), 108

G

generate_data_row() (*cmd2.table_creator.AlternatingTable method*), 124
generate_data_row() (*cmd2.table_creator.BorderedTable method*), 123
generate_data_row() (*cmd2.table_creator.SimpleTable method*), 122
generate_header() (*cmd2.table_creator.BorderedTable method*), 123
generate_header() (*cmd2.table_creator.SimpleTable method*), 122
generate_header_bottom_border() (*cmd2.table_creator.BorderedTable method*), 123
generate_range_error() (*in module cmd2.argparse_custom*), 109
generate_row() (*cmd2.table_creator.TableCreator method*), 121

generate_row_bottom_border() (cmd2.table_creator.BorderedTable method), 123
generate_table() (cmd2.table_creator.AlternatingTable method), 124
generate_table() (cmd2.table_creator.BorderedTable method), 123
generate_table() (cmd2.table_creator.SimpleTable method), 122
generate_table_bottom_border() (cmd2.table_creator.BorderedTable method), 124
generate_table_top_border() (cmd2.table_creator.BorderedTable method), 124
get() (cmd2.argparse_custom.Cmd2AttributeWrapper method), 108
get() (cmd2.history.History method), 114
get_all_commands() (cmd2.Cmd method), 96
get_command_arg_list() (cmd2.parsing.StatementParser method), 115
get_help_topics() (cmd2.Cmd method), 96
get_names() (cmd2.Cmd method), 96
get_paste_buffer() (in module cmd2.clipboard), 28
get_visible_commands() (cmd2.Cmd method), 96
getbytes() (cmd2.utils.StdSim method), 126
getvalue() (cmd2.utils.StdSim method), 126

H

help_error (cmd2.Cmd attribute), 91
History (class in cmd2.history), 113
history (cmd2.Cmd attribute), 91
HistoryItem (class in cmd2.history), 115
HorizontalAlignment (class in cmd2.table_creator), 120

I

idx (cmd2.history.HistoryItem attribute), 115
in_pyscript() (cmd2.Cmd method), 96
in_script() (cmd2.Cmd method), 96
index_based_complete() (cmd2.Cmd method), 96
INTENSITY_BRIGHT (in module cmd2.ansi), 102
INTENSITY_DIM (in module cmd2.ansi), 102
INTENSITY_NORMAL (in module cmd2.ansi), 102
intro (cmd2.Cmd attribute), 91
is_quoted() (in module cmd2.utils), 126
is_valid_command() (cmd2.parsing.StatementParser method), 116
isatty() (cmd2.utils.StdSim method), 126

L

LEFT (cmd2.table_creator.HorizontalAlignment attribute), 120
LEFT (cmd2.utils.TextAlignment attribute), 127
line_buffering (cmd2.utils.StdSim attribute), 126

M

MIDDLE (cmd2.table_creator.VerticalAlignment attribute), 120
multiline_command (cmd2.Statement attribute), 118

N

namedtuple_with_defaults() (in module cmd2.utils), 130
natural_sort() (in module cmd2.utils), 131
NATURAL_SORT_KEY() (cmd2.Cmd method), 91

O

on_register() (cmd2.command_definition.CommandSet method), 110
on_registered() (cmd2.command_definition.CommandSet method), 110
on_unregister() (cmd2.command_definition.CommandSet method), 110
on_unregistered() (cmd2.command_definition.CommandSet method), 110
onecmd() (cmd2.Cmd method), 96
onecmd_plus_hooks() (cmd2.Cmd method), 97
output (cmd2.Statement attribute), 118
output_to (cmd2.Statement attribute), 118

P

parse() (cmd2.parsing.StatementParser method), 116
parse_command_only() (cmd2.parsing.StatementParser method), 116
parseline() (cmd2.Cmd method), 97
path_complete() (cmd2.Cmd method), 97
perror() (cmd2.Cmd method), 97
pexcept() (cmd2.Cmd method), 98
pfeedback() (cmd2.Cmd method), 98
pipe_to (cmd2.Statement attribute), 118
post_command (cmd2.Statement attribute), 118
postcmd() (cmd2.Cmd method), 98
PostcommandData (class in cmd2.plugin), 119
postloop() (cmd2.Cmd method), 98
PostparsingData (class in cmd2.plugin), 119
poutput() (cmd2.Cmd method), 98
ppaged() (cmd2.Cmd method), 98
pr() (cmd2.history.HistoryItem method), 115
precmd() (cmd2.Cmd method), 99
PrecommandData (class in cmd2.plugin), 119

preloop() (*cmd2.Cmd method*), 99
ProcReader (*class in cmd2.utils*), 127
prompt (*cmd2.Cmd attribute*), 91
pwarning() (*cmd2.Cmd method*), 99
py_bridge_name (*cmd2.Cmd attribute*), 91
PyBridge (*class in cmd2.py_bridge*), 120

Q

quote_string() (*in module cmd2.utils*), 126
quote_string_if_needed() (*in module cmd2.utils*), 126

R

raw (*cmd2.history.HistoryItem attribute*), 115
raw (*cmd2.Statement attribute*), 118
read() (*cmd2.utils.StdSim method*), 126
read_input() (*cmd2.Cmd method*), 99
readbytes() (*cmd2.utils.StdSim method*), 126
regex_search() (*cmd2.history.History method*), 114
register_cmdfinalization_hook() (*cmd2.Cmd method*), 99
register_command_set() (*cmd2.Cmd method*), 100
register_postcmd_hook() (*cmd2.Cmd method*), 100
register_postloop_hook() (*cmd2.Cmd method*), 100
register_postparsing_hook() (*cmd2.Cmd method*), 100
register_precmd_hook() (*cmd2.Cmd method*), 100
register_preloop_hook() (*cmd2.Cmd method*), 100
remove_duplicates() (*in module cmd2.utils*), 130
remove_settable() (*cmd2.Cmd method*), 100
RESET_ALL (*in module cmd2.ansi*), 102
RIGHT (*cmd2.table_creator.HorizontalAlignment attribute*), 120
RIGHT (*cmd2.utils.TextAlignment attribute*), 127
runcmds_plus_hooks() (*cmd2.Cmd method*), 100

S

select() (*cmd2.Cmd method*), 100
send_sigint() (*cmd2.utils.ProcReader method*), 127
set() (*cmd2.argparse_custom.Cmd2AttributeWrapper method*), 108
set_choices_function() (*in module cmd2.argparse_custom*), 109
set_choices_method() (*in module cmd2.argparse_custom*), 109
set_completer_function() (*in module cmd2.argparse_custom*), 109
set_completer_method() (*in module cmd2.argparse_custom*), 109
set_default_argument_parser() (*in module cmd2.argparse_custom*), 109
set_title_str() (*in module cmd2.ansi*), 103
set_window_title() (*cmd2.Cmd method*), 100
Settable (*class in cmd2.utils*), 125
settable (*cmd2.Cmd attribute*), 91
shell_cmd_complete() (*cmd2.Cmd method*), 101
sigint_handler() (*cmd2.Cmd method*), 101
SimpleTable (*class in cmd2.table_creator*), 122
SkipPostcommandHooks (*class in cmd2.exceptions*), 113
span() (*cmd2.history.History method*), 114
split_on_punctuation() (*cmd2.parsing.StatementParser method*), 117
start_session() (*cmd2.history.History method*), 114
Statement (*class in cmd2*), 117
statement (*cmd2.history.HistoryItem attribute*), 115
statement (*cmd2.plugin.CommandFinalizationData attribute*), 119
statement (*cmd2.plugin.PostcommandData attribute*), 119
statement (*cmd2.plugin.PostparsingData attribute*), 119
statement (*cmd2.plugin.PrecommandData attribute*), 119
statement_parser (*cmd2.Cmd attribute*), 91
StatementParser (*class in cmd2.parsing*), 115
StdSim (*class in cmd2.utils*), 126
stop (*cmd2.plugin.CommandFinalizationData attribute*), 119
stop (*cmd2.plugin.PostcommandData attribute*), 119
stop (*cmd2.plugin.PostparsingData attribute*), 119
str_search() (*cmd2.history.History method*), 114
str_to_bool() (*in module cmd2.utils*), 130
strip_quotes() (*in module cmd2.utils*), 126
strip_style() (*in module cmd2.ansi*), 103
style() (*in module cmd2.ansi*), 103
STYLE_ALWAYS (*in module cmd2.ansi*), 102
style_aware_wcswidth() (*in module cmd2.ansi*), 104
style_aware_write() (*in module cmd2.ansi*), 104
style_error() (*in module cmd2.ansi*), 104
STYLE_NEVER (*in module cmd2.ansi*), 102
style_success() (*in module cmd2.ansi*), 104
STYLE_TERMINAL (*in module cmd2.ansi*), 102
style_warning() (*in module cmd2.ansi*), 104
suffix (*cmd2.Statement attribute*), 118

T

TableCreator (*class in cmd2.table_creator*), 121
terminate() (*cmd2.utils.ProcReader method*), 127
terminator (*cmd2.Statement attribute*), 118

TextAlignment (*class in cmd2.utils*), 127
tokenize() (*cmd2.parsing.StatementParser method*), 117
tokens_for_completion() (*cmd2.Cmd method*), 101
TOP (*cmd2.table_creator.VerticalAlignment attribute*), 120
total_width() (*cmd2.table_creator.BorderedTable method*), 124
total_width() (*cmd2.table_creator.SimpleTable method*), 123
truncate() (*cmd2.history.History method*), 115
truncate_line() (*in module cmd2.utils*), 129

U

UNDERLINE_DISABLE (*in module cmd2.ansi*), 102
UNDERLINE_ENABLE (*in module cmd2.ansi*), 102
unregister_command_set() (*cmd2.Cmd method*), 101

V

VerticalAlignment (*class in cmd2.table_creator*), 120
visible_prompt (*cmd2.Cmd attribute*), 101

W

wait() (*cmd2.utils.ProcReader method*), 127
with_argparser() (*in module cmd2.decorators*), 111
with_argparser_and_unknown_args() (*in module cmd2.decorators*), 112
with_argument_list() (*in module cmd2.decorators*), 112
with_category() (*in module cmd2.decorators*), 113
with_default_category() (*in module cmd2.command_definition*), 110
write() (*cmd2.utils.ByteBuf method*), 126
write() (*cmd2.utils.StdSim method*), 126
write_to_paste_buffer() (*in module cmd2.clipboard*), 28