cmd2 Documentation

Release 0.9

Catherine Devlin and Todd Leonhardt

Jan 21, 2020

Contents

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

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

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

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

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

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

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.settable['maxrepeats'] = '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. Then we create an attribute to hold our setting, and then add a description of our setting to the `settable` dictionary. If our attribute name isn't in `settable`, then it won't be treated as a setting. 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 `@cmd2.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.settable['maxrepeats'] = '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. 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 is stored in the `__statement__` attribute of the `Namespace`.
4. Adds the usage message from the argument parser to your command.
5. Checks if the `-h/--help` option is present, and if so, display the help message for the command

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

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

Decorators provided by cmd2 for argument processing

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

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

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

Parameters

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

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

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

Parameters

- **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 member called __statement__ is added to the Namespace to provide command functions access to the Statement object. This can be useful if the command function needs to know the command line.

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

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

Parameters

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

Returns function that gets passed a list of argument strings

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.

Using the argument parser decorator

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

Here's what it looks like:

```
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

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

the `help tag` command displays:

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

create a html tag

positional arguments:
  tag         tag
  content     content to surround with tag

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

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

```
import argparse
from cmd2 import with_argparser

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

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

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

create an html tag

positional arguments:
  tag         tag
  content     content to surround with tag

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

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

```
import argparse
from cmd2 import with_argparser

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

Which yields:

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

```

(continues on next page)

(continued from previous page)

```
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.

Receiving an argument list

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

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

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

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

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

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

```
from cmd2 import with_argument_list

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

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

(continues on next page)

(continued from previous page)

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

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

    ...
```

Using custom argparse.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 and tab_autocompletion 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 `Cmd2ArgumentParser` class which inherits from `argparse.ArgumentParser` and improves error and help output:

class cmd2.argparse_custom.**Cmd2ArgumentParser**(**args*, ***kwargs*)
Custom ArgumentParser class that improves error and help output

add_subparsers(***kwargs*)
Custom override. Sets a default title if one was not given.

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

3.1.2 Builtin Commands

Applications which subclass *cmd2.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.

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.

set

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

```
(Cmd) set --long
allow_style: Terminal          # Allow ANSI text style sequences in output (valid␣
↪values: Terminal, Always, Never)
continuation_prompt: >         # On 2nd+ line of input
debug: False                   # Show full error stack on error
echo: False                    # Echo command issued into output
editor: vim                    # Program used by ``edit``
feedback_to_output: False      # include nonessentials in `|`, `>` results
locals_in_py: False            # Allow access to your application in py via self
max_completion_items: 50       # Maximum number of CompletionItems to display during␣
↪tab completion
prompt: (Cmd)                  # The prompt issued to solicit input
quiet: False                   # Don't print nonessential feedback
timing: False                  # Report execution times
```

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

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

shell

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

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

Remove Builtin Commands

Developers may not want to offer the commands builtin to *cmd2.cmd2.Cmd* to users of their application. To remove a command you must delete the method implementing that command from the *cmd2.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_finis(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.

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

3.1.5 Completion

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

- `edit`
- `run_pyscript`
- `run_script`
- `shell`

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

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

```
complete_foo = self.path_complete
```

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

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

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

Tab Completion Using Argparse Decorators

When using one the Argparse-based *Decorators provided by cmd2 for argument processing*, `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` / `choices_method`
- `completer_function` / `completer_method`

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

When tab-completing flags and/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 `CompletionItem` class which can be returned from any of the 4 completion functions: `choices_function`, `choices_method`, `completion_function`, or `completion_method`.

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

See the tab_autocompletion example or the implementation of the built-in **set** command for demonstration of how this is used.

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

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

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

More Python examples:

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

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

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

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

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

The `py` command also allows you to run Python scripts via `py run('myscript.py')`. This provides a more complicated and more powerful scripting capability than that provided by the simple text file scripts discussed in *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.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.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.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

use the categorize() function to create help categories

Use `help_method()` to custom roll your own help messages.

See *Help Messages*

Categorizing Commands

By default, the `help` command displays:

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

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, _):
```

(continues on next page)

(continued from previous page)

```
    """Which command"""
    self.poutput('Which')
```

Using the `categorize()` function:

You can call with a single function:

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

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

Or with an Iterable container of functions:

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

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

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

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

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

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

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

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

Server Information
```

(continues on next page)

(continued from previous page)

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

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

3.1.10 History

For Developers

- Describe how cmd2 tracks history
- how persistent history works
- differences in history and bash shell history (we only store valid commands in history)
- reference the public code structures we use to store history

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

`Cmd.__init__`(*completekey: str = 'tab', stdin=None, stdout=None, *, persistent_history_file: str = '', persistent_history_length: int = 1000, startup_script: 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*) → 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 will process command line arguments as either commands to be run or, if -t is specified, transcript files to run. This should be set to False if your application parses its own arguments.
- **transcript_files** – allow running transcript tests when allow_cli_args is False
- **allow_redirection** – should output redirection and pipes be allowed. this is only a security setting and does not alter parsing behavior.
- **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, then defaults to 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.

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

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

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

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

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

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

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

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

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

Application Lifecycle Attributes

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

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

Command Processing Loop

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

1. Output the prompt
2. Accept user input
3. Parse user input into *Statement* object

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

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

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

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

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

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

Postparsing Hooks

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

Precommand Hooks

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

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

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

(continues on next page)

(continued from previous page)

```
        self.register_precmd_hook(self.myhookmethod)

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

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

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

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

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

Postcommand Hooks

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

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

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

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

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

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

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

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

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

Command Finalization Hooks

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

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

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

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

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

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

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

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 appliction 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
"""
import cmd2
from cmd2 import style


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

    def __init__(self):
        super().__init__(multiline_commands=['echo'], persistent_history_file='cmd2_
↪history.dat',
```

(continues on next page)

(continued from previous page)

```
                        startup_script='scripts/startup.txt', use_ipython=True)

        self.intro = style('Welcome to cmd2!', fg='red', bg='white', bold=True)
        self.prompt = 'myapp> '

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

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

    @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(arg)


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

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`.

Shell Command

`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
```

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.

Exit code

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())`.

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) only cancels the current line, not the entire command loop. By default, a `cmd2` application will quit on receiving this signal. However, if `quit_on_sigint` is set to `False`, then the current line will simply be cancelled.

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

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

3.1.14 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.

3.1.15 Integrating with the OS

- how to redirect output
- executing OS commands from within `cmd2`
- editors
- paging
- exit codes
- Automation and calling cmd2 from other CLI/CLU tools via commands at invocation and quit

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 (type help <topic>):
========================================
alias  help     macro   orate  quit          run_script  set    shortcuts
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.

3.1.16 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

3.1.17 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.

Adding functionality

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"""
    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.settable.update({'mysetting': 'short help message for mysetting'})
```

You can also hide settings from the user by removing them from `self.settable`.

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 hooks, 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 cmd2_plugin_template for more info.

3.1.18 Prompt

`cmd2` can issue a prompt before soliciting user input.

Asynchronous Feedback

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

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

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

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

set_window_title() Sets the terminal window title

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

3.1.19 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

3.1.20 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`. See *History* for more details.

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 `@` shortcut. 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 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. 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 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 `py_bridge_name` attribute of your `cmd2.Cmd` class instance
- `command` and `args` are entered exactly like they would be entered on the command line of your `cmd2` application

3.1.21 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.cmd2.Cmd* and must also appear in the *cmd2.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

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 *prompt* setting. The prompt for subsequent lines of input is defined by this setting.

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 a running 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()*.

locals_in_py

Allow access to your application in one of the *Embedded Python Shells* via `self`.

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.

prompt

This setting contains the string which should be printed as a prompt for user input.

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. First create a class attribute with the default value. Then update the `settable` dictionary with your setting name and a short description before you initialize the superclass. Here's an example, from `examples/environment.py`:

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


class EnvironmentApp(cmd2.Cmd):
```

(continues on next page)

(continued from previous page)

```
    """ Example cmd2 application. """

    degrees_c = 22
    sunny = False

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

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

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


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

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

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

Hide Builtin Settings

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

Let's say your program does not have any *Multiline Commands*. You might want to hide the *continuation_prompt* setting from your users since it is only applicable to multiline commands. To do so, remove it from the *cmd2.cmd2.Cmd.settable* dictionary after you initialize your object:

```
class MyApp(cmd2.Cmd):

  def __init__(self):
      super().__init__()
      self.settable.pop('continuation_prompt')
```

3.1.22 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

3.1.23 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.

3.1.24 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
```

(continues on next page)

(continued from previous page)

```
...
(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
```

(continues on next page)

(continued from previous page)

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

Some terminal emulators strip trailing space when you copy text from them. This could make the actual data generated by your app different than the text you pasted into the transcript, and it might not be readily obvious why the transcript is not passing. Consider using *Output Redirection 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
.
----------------------------------------------------------------------
Ran 1 test in 0.013s

OK
```

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

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

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

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

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 is a convenience method to run multiple commands via **onecmd_plus_hooks()**.

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

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

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

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

Here is a little more info on `runcmds_plus_hooks`:

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

Returns True if running of commands should stop

CHAPTER 5

API Reference

5.1 API Reference

5.1.1 cmd2.Cmd

class cmd2.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*)

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.

help_error
The error message displayed to the user when they request help for a command with no help defined.

default_error
The error message displayed when a non-existent command is run.

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.

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

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.

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.

If a continuation prompt is currently being displayed 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

cmd_func(*command: str*) → Optional[Callable]
Get the function for a command :param command: the name of the command

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

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 :param command: the command being disabled :param 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*) → 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

Accepts abbreviated parameter names so long as there is no ambiguity. Call without arguments for a list of settable parameters with their values.

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 :param category: the category to enable

enable_command(*command: str*) → None
Enable a command by restoring its functions :param command: the command being enabled

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, 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.
- **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*. :param msg: message to print (anything convertible to a str with '{}'.format() is OK) :param end: string appended after the end of the message, default a newline

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 commands's output has been piped to another process and that process terminates before the cmd2 command is finished executing.

Parameters

- **msg** – message to print (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 processed by `onecmd()` and after adding it to the history.

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

Returns a potentially modified version of the input Statement object

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. :param prompt: prompt to display to user :param 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. :return: 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_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.

runcmds_plus_hooks(*cmds: List[Union[cmd2.history.HistoryItem, str]], *, add_to_history: 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.

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.

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

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

5.1.2 Decorators

cmd2.decorators.**with_category**(*category: str*) → Callable
A decorator to apply a category to a command function.

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

Parameters

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

Returns function that gets passed a list of argument strings

cmd2.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]]

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 member called __statement__ is added to the Namespace to provide command functions access to the Statement object. This can be useful if the command function needs to know the command line.

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

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

Parameters

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

5.1.3 Exceptions

exception cmd2.cmd2.**EmbeddedConsoleExit**

Custom exception class for use with the py command.

exception cmd2.cmd2.**EmptyStatement**

Custom exception class for handling behavior when the user just presses <Enter>.

5.1.4 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.**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.

cmd2.ansi.**bg_lookup**(*bg_name: str*) → str

Look up ANSI escape codes based on background color name.

Parameters **bg_name** – background color name 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.**fg_lookup**(*fg_name: str*) → str

Look up ANSI escape codes based on foreground color name.

Parameters **fg_name** – foreground color name 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: str = ''*, *bg: str = ''*, *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. Defaults to no color.
- **bg** – background color. Relies on *bg_lookup()* to retrieve ANSI escape based on name. 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: str = 'bright_red'*, *bg: str = ''*, *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: str = 'green'*, *bg: str = ''*, *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: str = 'bright_yellow'*, *bg: str = ''*, *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

5.1.5 Utility Classes

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.

class cmd2.utils.**ByteBuf**(*std_sim_instance: cmd2.utils.StdSim*)
Used by StdSim to write binary data and stores the actual bytes written

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.

class cmd2.utils.**ContextFlag**
A context manager which is also used as a boolean flag value within the default sigint handler.

Its main use is as a flag to prevent the SIGINT handler in cmd2 from raising a KeyboardInterrupt while a critical code section has set the flag to True. Because signal handling is always done on the main thread, this class is not thread-safe since there is no need.

class cmd2.utils.**RedirectionSavedState**(*self_stdout: Union[cmd2.utils.StdSim, TextIO]*, *sys_stdout: Union[cmd2.utils.StdSim, TextIO]*, *pipe_proc_reader: Optional[cmd2.utils.ProcReader]*)
Created by each command to store information about their redirection.

5.1.6 Utility Functions

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_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

cmd2.decorators.**categorize**(*func: Union[Callable, Iterable[Callable]], category: str*) → None
Categorize a function.

The help command output will group this function under the specified category heading

Parameters

- **func** – function or list of functions to categorize
- **category** – category to put it in

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 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.

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 a 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 ValueError if text or fill_char contains an unprintable character 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 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.

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 a 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 ValueError if text or fill_char contains an unprintable character 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 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.

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 a 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 ValueError if text or fill_char contains an unprintable character 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 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.

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 a 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 ValueError if text or fill_char contains an unprintable character 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 are safely ignored and do not count toward the display width. This means colored text is supported.

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 new line ValueError if max_width is less than 1

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

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.**cast**(*current: Any*, *new: str*) → Any

Tries to force a new value into the same type as the current when trying to set the value for a parameter.

Parameters

- **current** – current value for the parameter, type varies
- **new** – new value

Returns new value with same type as current, or the current value if there was an error casting

cmd2.utils.**which**(*exe_name: str*) → Optional[str]
Find the full path of a given executable on a Linux or Mac machine

Parameters **exe_name** – name of the executable to check, ie 'vi' or 'ls'

Returns a full path or None if exe not found

cmd2.utils.**is_text_file**(*file_path: str*) → bool
Returns if a file contains only ASCII or UTF-8 encoded text.

Parameters **file_path** – path to the file being checked

Returns True if the file is a text file, False if it is binary.

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.**norm_fold**(*astr: str*) → 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

cmd2.utils.**try_int_or_force_to_lower_case**(*input_str: str*) → Union[int, str]
Tries to convert the passed-in string to an integer. If that fails, it converts it to lower case using norm_fold. :param input_str: string to convert :return: the string as an integer or a lower case version of the string

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.**unquote_specific_tokens**(*args: List[str], tokens_to_unquote: List[str]*) → None
Unquote a specific tokens in a list of command-line arguments This is used when certain tokens have to be passed to another command :param args: the command line args :param tokens_to_unquote: the tokens, which if present in args, to unquote

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

cmd2.utils.**natural_keys**(*input_str: str*) → 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

cmd2.utils.**expand_user_in_tokens**(*tokens: List[str]*) → None
Call expand_user() on all tokens in a list of strings :param tokens: tokens to expand

cmd2.utils.**expand_user**(*token: str*) → str
Wrap os.expanduser() to support expanding ~ in quoted strings :param token: the string to expand

cmd2.utils.**find_editor**() → str
Find a reasonable editor to use by default for the system that the cmd2 application is running on.

cmd2.utils.**get_exes_in_path**(*starts_with: str*) → List[str]
Returns names of executables in a user's path

Parameters **starts_with** – what the exes should start with. leave blank for all exes in path.

Returns a list of matching exe names

cmd2.utils.**files_from_glob_patterns**(*patterns: List[str], access=0*) → List[str]
Return a list of file paths based on a list of glob patterns.

Only files are returned, not directories, and optionally only files for which the user has a specified access to.

Parameters

- **patterns** – list of file names and/or glob patterns
- **access** – file access type to verify (os.* where * is F_OK, R_OK, W_OK, or X_OK)

Returns list of files matching the names and/or glob patterns

cmd2.utils.**files_from_glob_pattern**(*pattern: str, access=0*) → List[str]
Return a list of file paths based on a glob pattern.

Only files are returned, not directories, and optionally only files for which the user has a specified access to.

Parameters

- **pattern** – file name or glob pattern
- **access** – file access type to verify (os.* where * is F_OK, R_OK, W_OK, or X_OK)

Returns list of files matching the name or glob pattern

CHAPTER 6

Meta

Documentation Conventions

6.1 Documentation Conventions

6.1.1 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

6.1.2 Style Checker

Use doc8 to check the style of the documentation. This tool can be invoked using the proper options by typing:

```
$ invoke doc8
```

6.1.3 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'

6.1.4 Indenting

In reStructuredText all indenting is significant. Use 2 spaces per indenting level.

6.1.5 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.

6.1.6 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.

6.1.7 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`.

6.1.8 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.

6.1.9 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*

6.1.10 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.

6.1.11 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.cmd2.Cmd.poutput` method is similar to the Python built-in
print function.
```

Which renders as: The *cmd2.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.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.

3. Reference a portion of the 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.

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.

6.1.12 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.

6.2 Copyright

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W