cmd2 Documentation

Release 0.7.0

Catherine Devlin and Todd Leonhardt

February 23, 2017

Contents

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

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

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

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

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

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

Note: The tab-completion feature provided by cmd relies on underlying capability provided by GNU readline or an equivalent library. Linux distros will almost always come with the required library installed. For Mac OS X, we recommend installing the gnureadline Python module. For Windows, we recommend installing the pyreadline Python module.

CHAPTER 1

Resources

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

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

Contents:

Installation Instructions

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

Installing

First you need to make sure you have Python 2.7 or Python 3.3+, pip, and setuptools. Then you can just use pip to install from PyPI.

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

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

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

Requirements for Installing

- If you have Python 2 >=2.7.9 or Python 3 >=3.4 installed from python.org, you will already have pip and setuptools, but may need to upgrade to the latest versions:

On Linux or OS X:

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

On Windows:

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

Use pip for Installing

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

```
pip install cmd2
```

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

Install from GitHub using pip

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

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

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

Install from Debian or Ubuntu repos

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

For Python 2:

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

For Python 3:

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

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

Deploy cmd2.py with your project

`cmd2` is contained in only one Python file (**cmd2.py**), so it can be easily copied into your project. *The copyright and license notice must be retained.*

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

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

- six
- pyparsing

Upgrading cmd2

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

```
pip install -U cmd2
```

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

Uninstalling cmd2

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

```
pip uninstall cmd2
```

Overview

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

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

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

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

Features requiring no modifications

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

Script files

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

Cmd.**do_load**(*file_path=None*)

Runs commands in script at file or URL.

Usage: load [file_path]

Parameters **file_path** – str - a file path or URL pointing to a script (default: value stored in default_file_name)

Returns bool - True implies application should stop, False to continue like normal

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

Cmd.**do_save**(*arg*)
Saves command(s) from history to file.

Usage: save [N] [file_path]

Parameters **arg** – str - [N] [filepath]

- **N** - Number of command (from history), or * for all commands in history (default: most recent command)
- **file_path** - location to save script of command(s) to (default: value stored in `default_file_name`)

Cmd.**do_edit**(*arg*)
Edit a file or command in a text editor.

Usage: edit [N]|[file_path]

Parameters **arg** – str - [N]|[file_path]

- **N** - Number of command (from history), or * for all commands in history (default: most recent command)
- **file_path** - path to a file to open in editor

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

The optional arguments are mutually exclusive. Either a command number OR a file name can be supplied. If neither is supplied, the most recent command in the history is edited.

Edited commands are always run after the editor is closed.

Edited files are run on close if the `autorun_on_edit` settable parameter is True.

Comments

Comments are omitted from the argument list before it is passed to a `do_` method. By default, both Python-style and C-style comments are recognized; you may change this by overriding `app.commentGrammars` with a different pyparsing grammar.

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

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

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

Commands at invocation

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

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

Output redirection

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

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

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

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

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

Python

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

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

    py <command>: Executes a Python command.
    py: Enters interactive Python mode.
    End with `Ctrl-D` (Unix) / `Ctrl-Z` (Windows), `quit()`, 'exit()`.
    Non-python commands can be issued with `cmd("your command")`.

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

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

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

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`. However, the `ipy` shell cannot call "back" to your application with `cmd("")` and any changes made will not persist between sessions or back in the main application.

IPython provides many advantages, including:

- Comprehensive object introspection
- Input history, persistent across sessions
- Caching of output results during a session with automatically generated references
- Extensible tab completion, with support by default for completion of python variables and keywords

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

Searchable command history

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

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

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

`Cmd.`**`do_history`**(*instance*, *arg*)

history [arg]: lists past commands issued

no arg: list all
arg is integer: list one history item, by index
arg is string: string search
arg is /enclosed in forward-slashes/: regular expression search

Usage: history [options] (limit on which commands to include)

Options:

-h, --help	show this help message and exit
-s, --script	Script format; no separation lines

`Cmd.`**`do_list`**(*arg*)

list [arg]: lists command(s) from history in a flexible/searchable way.

Parameters **`arg`** – str - behavior varies as follows:

- no arg -> list most recent command
- arg is integer -> list one history item, by index
- a..b, a:b, a:, ..b -> list spans from a (or start) to b (or end)
- arg is string -> list all commands matching string search
- arg is /enclosed in forward-slashes/ -> regular expression search

Cmd.**do_run**(*arg*)
run [arg]: re-runs an earlier command

Parameters **arg** – str - determines which command is re-run, as follows:

- no arg -> run most recent command
- arg is integer -> run one history item, by index
- arg is string -> run most recent command by string search
- arg is /enclosed in forward-slashes/ -> run most recent by regex

Quitting the application

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

Abbreviated commands

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

This behavior can be turned off with app.abbrev (see *Other user-settable parameters*)

Misc. pre-defined commands

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

Cmd.**do_quit**(*arg*)
Exits this application.

Cmd.**do_pause**(*text*)
Displays the specified text then waits for the user to press <Enter>.

Usage: pause [text]

Parameters **text** – str - Text to display to the user (default: blank line)

Cmd.**do_shell**(*command*)
Execute a command as if at the OS prompt.

Usage: shell command

Parameters **command** – str - shell command to execute

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

Transcript-based testing

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

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

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

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

Features requiring only parameter changes

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

Case-insensitivity

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

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

Shortcuts

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

? help

! shell: run as OS-level command

@ load script file

@@ load script file; filename is relative to current script location

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

```
class App(Cmd2):
    Cmd2.shortcuts.update({'*': 'sneeze', '~': 'squirm'})
```

Default to shell

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

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

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

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

Timing

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

Echo

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

Debug

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

Other user-settable parameters

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

```
(Cmd) set --long
abbrev: True                    # Accept abbreviated commands
autorun_on_edit: True           # Automatically run files after editing
case_insensitive: True          # upper- and lower-case both OK
colors: True                    # Colorized output (*nix only)
continuation_prompt: >          # On 2nd+ line of input
debug: False                    # Show full error stack on error
default_file_name: command.txt # for ``save``, ``load``, etc.
echo: False                     # Echo command issued into output
editor: vim                     # Program used by ``edit``
feedback_to_output: False       # include nonessentials in `|`, `>` results
locals_in_py: True              # Allow access to your application in py via self
prompt: (Cmd)                   # The prompt issued to solicit input
quiet: False                    # Don't print nonessential feedback
timing: False                   # Report execution times
```

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

```
set abbrev False
```

Features requiring application changes

Multiline commands

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

Parsed statements

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

command Name of the command called

raw Full input exactly as typed.

terminator Character used to end a multiline command

suffix Remnant of input after terminator

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

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

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

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

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

Environment parameters

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

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

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

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

Commands with flags

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

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

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

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

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

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

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

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

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

Controlling how arguments are parsed for commands with flags

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

cmd2.**set_posix_shlex**(*val*)
Allows user of cmd2 to choose between POSIX and non-POSIX splitting of args for @options commands.

Parameters **val** – bool - True => POSIX, False => Non-POSIX

cmd2.**set_strip_quotes**(*val*)
Allows user of cmd2 to choose whether to automatically strip outer-quotes when POSIX_SHLEX is False.

Parameters **val** – bool - True => strip quotes on args and option args for @option commands if POSIX_SHLEX is False.

cmd2.**set_use_arg_list**(*val*)
Allows user of cmd2 to choose between passing @options commands an argument string or list of arg strings.

Parameters **val** – bool - True => arg is a list of strings, False => arg is a string (for @options commands)

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

---

poutput, pfeedback, perror

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

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

color

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

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

quiet

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

select

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

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

`Cmd.`**`select`**(*options*, *prompt='Your choice? '*)
Presents a numbered menu to the user. Modelled after the bash shell's SELECT. Returns the item chosen.

Argument `options` 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!
```

Alternatives to cmd and cmd2

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

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

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

- Python Prompt Toolkit
- Click

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

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

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

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

CHAPTER 2

Compatibility

Tested and working with Python 2.7 and 3.3+.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

C

D

S