cmd2 Documentation

Release 0.6.9a

Catherine Devlin

January 29, 2017

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

Resources

- cmd
- project bug tracker
- cmd2 project page
- 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:

1.1 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.

1.2 Features requiring no modifications

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

1.2.1 Script files

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

1.2.2 Comments

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

Comments can be useful in scripts. Used in an interactive session, they may indicate mental imbalance.

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

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

1.2.3 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$
```

1.2.4 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
```

1.2.5 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. (If that thought terrifies you, you can set the `locals_in_py` parameter to `False`. See see 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 >
```

1.2.6 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:

1.2.7 Quitting the application

`cmd2` pre-defines a `quit` command for you (with synonyms `exit` and simply `q`). It's trivial, but it's one less thing for you to remember.

1.2.8 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 parameters)

1.2.9 Misc. pre-defined commands

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

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

1.2.10 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.

1.3 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 parameters)

1.3.1 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` will probably do something evil if you do.

1.3.2 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'})
```

1.3.3 Default to shell

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

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

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

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

1.3.4 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 parameters)

1.3.5 Echo

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

1.3.6 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 parameters)

1.3.7 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
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: gedit                    # Program used by ``edit``
feedback_to_output: False        # include nonessentials in `|`, `>` results
prompt: (Cmd)                    #
quiet: False                     # Don't print nonessential feedback
timing: False                    # Report execution times
```

1.4 Features requiring application changes

1.4.1 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).

1.4.2 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`.)

ParsedString was developed to support sqlpython and reflects its needs. The parsing grammar and process are painfully complex and should not be considered stable; 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`.)

1.4.3 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?
```

1.4.4 Commands with flags

All `do_` methods are responsible for interpreting the arguments passed to them. However, `cmd2` lets a `do_` methods accept Unix-style *flags*. It uses *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
```

1.4.5 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.

1.4.6 color

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

1.4.7 quiet

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

1.4.8 `select`

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

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

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

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

1.5 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 optparse (for parsing UNIX-style options and flags).

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 packages in PyPI enable 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.

- CmdLoop
- cly
- CmDO (As of Feb. 2010, webpage is missing.)
- pycopia-CLI

cmdln, another package in PyPI, is an extension to cmd and, though it doesn't retain full cmd compatibility, shares its basic structure with cmd.

I've found several alternatives to cmd in the Cheese Shop - CmdLoop, cly, CMdO, and pycopia. cly looks wonderful, but I haven't been able to get it working under Windows, and that's a show-stopper for many potential sqlpython users. In any case, none of the alternatives are based on cmd - they're written from scratch, which means that a cmd-based app would need complete rewriting to use them. I like sticking close to the Standard Library whenever possible. cmd2 lets you do that.

CHAPTER 2

Compatibility

Tested and working with Python 2.5, 2.6, 2.7, 3.1; Jython 2.5

CHAPTER 3

Indices and tables

- genindex
- modindex
- search