cmd — Line-oriented Command Processors

Purpose: Create line-oriented command processors.

The cmd module contains one public class, Cmd, designed to be used as a base class for interactive shells and other command interpreters. By default it uses readline for interactive prompt handling, command line editing, and command completion.

Processing Commands

A command interpreter created with cmd uses a loop to read all lines from its input, parse them, and then dispatch the command to an appropriate *command handler*. Input lines are parsed into two parts: the command, and any other text on the line. If the user enters `foo bar`, and the interpreter class includes a method named `do_foo()`, it is called with `"bar"` as the only argument.

The end-of-file marker is dispatched to `do_EOF()`. If a command handler returns a true value, the program will exit cleanly. So to give a clean way to exit the interpreter, make sure to implement `do_EOF()` and have it return True.

This simple example program supports the “greet” command:

```
# cmd_simple.py

import cmd


class HelloWorld(cmd.Cmd):

    def do_greet(self, line):
        print("hello")

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

Running it interactively demonstrates how commands are dispatched and shows of some of the features included in Cmd.

```
$ python3 cmd_simple.py

(Cmd)
```

The first thing to notice is the command prompt, `(Cmd)`. The prompt can be configured through the attribute `prompt`. The prompt value is dynamic, and if a command handler changes the prompt attribute the new value is used to query for the next command.

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

Undocumented commands:
======================
EOF  greet
```

The `help` command is built into Cmd. With no arguments, `help` shows the list of commands available. If the input includes a command name, the output is more verbose and restricted to details of that command, when available.

If the command is `greet`, `do_greet()` is invoked to handle it:

```
(Cmd) greet
hello
```

If the class does not include a specific handler for a command, the method `default()` is called with the entire input line as an

If the class does not include a specific handler for a command, the method `default()` is called with the entire input line as an argument. The built-in implementation of `default()` reports an error.

```
(Cmd) foo
*** Unknown syntax: foo
```

Since `do_EOF()` returns True, typing Ctrl-D causes the interpreter to exit.

```
(Cmd) ^D$
```

No newline is printed on exit, so the results are a little messy.

Command Arguments

This example includes a few enhancements to eliminate some of the annoyances and add help for the `greet` command.

```
# cmd_arguments.py

import cmd


class HelloWorld(cmd.Cmd):

    def do_greet(self, person):
        """greet [person]
        Greet the named person"""
        if person:
            print("hi,", person)
        else:
            print('hi')

    def do_EOF(self, line):
        return True

    def postloop(self):
        print()


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

The docstring added to `do_greet()` becomes the help text for the command:

```
$ python3 cmd_arguments.py

(Cmd) help

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

Undocumented commands:
======================
EOF

(Cmd) help greet
greet [person]
        Greet the named person
```

The output shows one optional argument to `greet`, `person`. Although the argument is optional to the command, there is a distinction between the command and the callback method. The method always takes the argument, but sometimes the value is an empty string. It is left up to the command handler to determine if an empty argument is valid, or do any further parsing and processing of the command. In this example, if a person's name is provided then the greeting is personalized.

```
(Cmd) greet Alice
hi, Alice
(Cmd) greet
hi
```

Whether an argument is given by the user or not, the value passed to the command handler does not include the command itself. That simplifies parsing in the command handler, especially if multiple arguments are needed.

Live Help

In the previous example, the formatting of the help text leaves something to be desired. Since it comes from the docstring, it retains the indentation from the source file. The source could be changed to remove the extra white-space, but that would leave the application code looking poorly formatted. A better solution is to implement a help handler for the `greet` command, named `help_greet()`. The help handler is called to produce help text for the named command.

```
# cmd_do_help.py
```

```
# Set up gnureadline as readline if installed.
try:
    import gnureadline
    import sys
    sys.modules['readline'] = gnureadline
except ImportError:
    pass

import cmd


class HelloWorld(cmd.Cmd):

    def do_greet(self, person):
        if person:
            print("hi,", person)
        else:
            print('hi')

    def help_greet(self):
        print('\n'.join([
            'greet [person]',
            'Greet the named person',
        ]))

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

In this example, the text is static but formatted more nicely. It would also be possible to use previous command state to tailor the contents of the help text to the current context.

```
$ python3 cmd_do_help.py

(Cmd) help greet
greet [person]
Greet the named person
```

It is up to the help handler to actually output the help message, and not simply return the help text for handling elsewhere.

Auto-Completion

`Cmd` includes support for command completion based on the names of the commands with handler methods. The user triggers completion by hitting the tab key at an input prompt. When multiple completions are possible, pressing tab twice prints a list of the options.

Note

The GNU libraries needed for `readline` are not available on all platforms by default. In those cases, tab completion may not work. See readline for tips on installing the necessary libraries if your Python installation does not have them.

```
$ python3 cmd_do_help.py
```

```
$ python3 cmd_do_help.py

(Cmd) <tab><tab>
EOF    greet  help
(Cmd) h<tab>
(Cmd) help
```

Once the command is known, argument completion is handled by methods with the prefix `complete_`. This allows new completion handlers to assemble a list of possible completions using arbitrary criteria (i.e., querying a database or looking at a file or directory on the file system). In this case, the program has a hard-coded set of “friends” who receive a less formal greeting than named or anonymous strangers. A real program would probably save the list somewhere, and read it once then cache the contents to be scanned as needed.

```
# cmd_arg_completion.py

# Set up gnureadline as readline if installed.
try:
    import gnureadline
    import sys
    sys.modules['readline'] = gnureadline
except ImportError:
    pass

import cmd


class HelloWorld(cmd.Cmd):

    FRIENDS = ['Alice', 'Adam', 'Barbara', 'Bob']

    def do_greet(self, person):
        "Greet the person"
        if person and person in self.FRIENDS:
            greeting = 'hi, {}!'.format(person)
        elif person:
            greeting = 'hello, {}'.format(person)
        else:
            greeting = 'hello'
        print(greeting)

    def complete_greet(self, text, line, begidx, endidx):
        if not text:
            completions = self.FRIENDS[:]
        else:
            completions = [
                f
                for f in self.FRIENDS
                if f.startswith(text)
            ]
        return completions

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

When there is input text, `complete_greet()` returns a list of friends that match. Otherwise, the full list of friends is returned.

```
$ python3 cmd_arg_completion.py

(Cmd) greet <tab><tab>
Adam     Alice    Barbara  Bob
(Cmd) greet A<tab><tab>
Adam   Alice
(Cmd) greet Ad<tab>
(Cmd) greet Adam
hi, Adam!
```

If the name given is not in the list of friends, the formal greeting is given.

if the name given is not in the list of friends, the formal greeting is given.

```
(Cmd) greet Joe
hello, Joe
```

Overriding Base Class Methods

`Cmd` includes several methods that can be overridden as hooks for taking actions or altering the base class behavior. This example is not exhaustive, but contains many of the methods commonly useful.

```
# cmd_illustrate_methods.py

# Set up gnureadline as readline if installed.
try:
    import gnureadline
    import sys
    sys.modules['readline'] = gnureadline
except ImportError:
    pass

import cmd


class Illustrate(cmd.Cmd):
    "Illustrate the base class method use."

    def cmdloop(self, intro=None):
        print('cmdloop({})'.format(intro))
        return cmd.Cmd.cmdloop(self, intro)

    def preloop(self):
        print('preloop()')

    def postloop(self):
        print('postloop()')

    def parseline(self, line):
        print('parseline({!r}) =>'.format(line), end='')
        ret = cmd.Cmd.parseline(self, line)
        print(ret)
        return ret

    def onecmd(self, s):
        print('onecmd({})'.format(s))
        return cmd.Cmd.onecmd(self, s)

    def emptyline(self):
        print('emptyline()')
        return cmd.Cmd.emptyline(self)

    def default(self, line):
        print('default({})'.format(line))
        return cmd.Cmd.default(self, line)

    def precmd(self, line):
        print('precmd({})'.format(line))
        return cmd.Cmd.precmd(self, line)

    def postcmd(self, stop, line):
        print('postcmd({}, {})'.format(stop, line))
        return cmd.Cmd.postcmd(self, stop, line)

    def do_greet(self, line):
        print('hello,', line)

    def do_EOF(self, line):
        "Exit"
        return True


if __name__ == '__main__':
```

```
if __name__ == '__main__':
    Illustrate().cmdloop('Illustrating the methods of cmd.Cmd')
```

`cmdloop()` is the main processing loop of the interpreter. Overriding it is usually not necessary, since the `preloop()` and `postloop()` hooks are available.

Each iteration through `cmdloop()` calls `onecmd()` to dispatch the command to its handler. The actual input line is parsed with `parseline()` to create a tuple containing the command, and the remaining portion of the line.

If the line is empty, `emptyline()` is called. The default implementation runs the previous command again. If the line contains a command, first `precmd()` is called then the handler is looked up and invoked. If none is found, `default()` is called instead. Finally `postcmd()` is called.

Here is an example session with `print` statements added:

```
$ python3 cmd_illustrate_methods.py

cmdloop(Illustrating the methods of cmd.Cmd)
preloop()
Illustrating the methods of cmd.Cmd
(Cmd) greet Bob
precmd(greet Bob)
onecmd(greet Bob)
parseline(greet Bob) => ('greet', 'Bob', 'greet Bob')
hello, Bob
postcmd(None, greet Bob)
(Cmd) ^Dprecmd(EOF)
onecmd(EOF)
parseline(EOF) => ('EOF', '', 'EOF')
postcmd(True, EOF)
postloop()
```

Configuring Cmd Through Attributes

In addition to the methods described earlier, there are several attributes for controlling command interpreters. `prompt` can be set to a string to be printed each time the user is asked for a new command. `intro` is the “welcome” message printed at the start of the program. `cmdloop()` takes an argument for this value, or it can be set on the class directly. When printing help, the `doc_header`, `misc_header`, `undoc_header`, and `ruler` attributes are used to format the output.

```
# cmd_attributes.py

import cmd


class HelloWorld(cmd.Cmd):

    prompt = 'prompt: '
    intro = "Simple command processor example."

    doc_header = 'doc_header'
    misc_header = 'misc_header'
    undoc_header = 'undoc_header'

    ruler = '-'

    def do_prompt(self, line):
        "Change the interactive prompt"
        self.prompt = line + ': '

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

This example class shows a command handler to let the user control the prompt for the interactive session.

```
$ python3 cmd_attributes.py
```

```
Simple command processor example.
prompt: prompt hello
hello: help

doc_header
----------
help  prompt

undoc_header
------------
EOF

hello:
```

Running Shell Commands

To supplement the standard command processing, `Cmd` includes two special command prefixes. A question mark (`?`) is equivalent to the built-in `help` command, and can be used in the same way. An exclamation point (`!`) maps to `do_shell()`, and is intended for “shelling out” to run other commands, as in this example.

```
# cmd_do_shell.py

import cmd
import subprocess


class ShellEnabled(cmd.Cmd):

    last_output = ''

    def do_shell(self, line):
        "Run a shell command"
        print("running shell command:", line)
        sub_cmd = subprocess.Popen(line,
                                   shell=True,
                                   stdout=subprocess.PIPE)
        output = sub_cmd.communicate()[0].decode('utf-8')
        print(output)
        self.last_output = output

    def do_echo(self, line):
        """Print the input, replacing '$out' with
        the output of the last shell command.
        """
        # Obviously not robust
        print(line.replace('$out', self.last_output))

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    ShellEnabled().cmdloop()
```

This echo command implementation replaces the string `$out` in its argument with the output from the previous shell command.

```
$ python3 cmd_do_shell.py

(Cmd) ?

Documented commands (type help <topic>):
========================================
echo  help  shell

Undocumented commands:
======================
EOF

(Cmd) ? shell
```

```
(Cmd) ? shell
Run a shell command
(Cmd) ? echo
Print the input, replacing '$out' with
        the output of the last shell command
(Cmd) shell pwd
running shell command: pwd
.../pymotw-3/source/cmd

(Cmd) ! pwd
running shell command: pwd
.../pymotw-3/source/cmd

(Cmd) echo $out
.../pymotw-3/source/cmd
```

Alternative Inputs

While the default mode for `Cmd()` is to interact with the user through the `readline` library, it is also possible to pass a series of commands in to standard input using standard Unix shell redirection.

```
$ echo help | python3 cmd_do_help.py

(Cmd)
Documented commands (type help <topic>):
========================================
greet  help

Undocumented commands:
======================
EOF

(Cmd)
```

To have the program read a script file directly, a few other changes may be needed. Since `readline` interacts with the terminal/tty device, rather than the standard input stream, it should be disabled when the script is going to be reading from a file. Also, to avoid printing superfluous prompts, the prompt can be set to an empty string. This example shows how to open a file and pass it as input to a modified version of the `HelloWorld` example.

```
# cmd_file.py
```

```
import cmd


class HelloWorld(cmd.Cmd):

    # Disable rawinput module use
    use_rawinput = False

    # Do not show a prompt after each command read
    prompt = ''

    def do_greet(self, line):
        print("hello,", line)

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    import sys
    with open(sys.argv[1], 'rt') as input:
        HelloWorld(stdin=input).cmdloop()
```

With `use_rawinput` set to False and `prompt` set to an empty string, the script can be called on an input file with one command on each line.

```
# cmd_file.txt
```

```
greet
greet Alice and Bob
```

Running the example script with the example input produces the following output.

```
$ python3 cmd_file.py cmd_file.txt

hello,
hello, Alice and Bob
```

Commands from sys.argv

Command line arguments to the program can also be processed as commands for the interpreter class, instead of reading commands from the console or a file. To use the command line arguments, call `onecmd()` directly, as in this example.

```
# cmd_argv.py
```

```
import cmd


class InteractiveOrCommandLine(cmd.Cmd):
    """Accepts commands via the normal interactive
    prompt or on the command line.
    """

    def do_greet(self, line):
        print('hello,', line)

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    import sys
    if len(sys.argv) > 1:
        InteractiveOrCommandLine().onecmd(' '.join(sys.argv[1:]))
    else:
        InteractiveOrCommandLine().cmdloop()
```

Since `onecmd()` takes a single string as input, the arguments to the program need to be joined together before being passed in.

```
$ python3 cmd_argv.py greet Command-Line User

hello, Command-Line User

$ python3 cmd_argv.py

(Cmd) greet Interactive User
hello, Interactive User
(Cmd)
```

See also

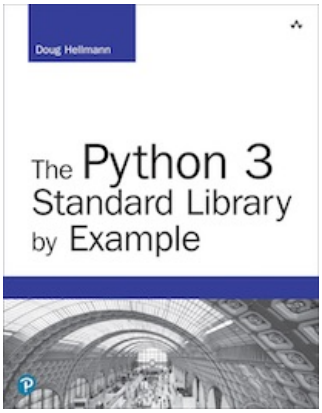
- Standard library documentation for cmd
- cmd2 – Drop-in replacement for cmd with additional features.
- GNU readline – The GNU Readline library provides functions that allow users to edit input lines as they are typed.
- `readline` – The Python standard library interface to readline.
- `subprocess` – Managing other processes and their output.

getpass — Secure Password Prompt

shlex — Parse Shell-style Syntaxes

This page was last updated 2016-12-30.

Navigation
getpass — Secure Password Prompt
shlex — Parse Shell-style Syntaxes



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example