cmd – Create line-oriented command processors

Purpose: Create line-oriented command processors.

Available In: 1.4 and later, with some additions in 2.3

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

Processing Commands

The interpreter 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 a command `foo bar`, and your 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 your interpreter, make sure to implement **`do_EOF()`** and have it return True.

This simple example program supports the "greet" command:

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

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

    def do_EOF(self, line):
        return True

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

By running it interactively, we can demonstrate how commands are dispatched as well as show of some of the features included in **`Cmd`** for free.

```
$ python cmd_simple.py
(Cmd)
```

The first thing to notice is the command prompt, `(Cmd)`. The prompt can be configured through the attribute prompt. If the prompt changes as the result of a command processor, the new value is used to query for the next command.

```
(Cmd) help

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

The `help` command is built into **`Cmd`**. With no arguments, it shows the list of commands available. If you include a command you want help on, the output is more verbose and restricted to details of that command, when available.

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

```
(Cmd) greet
hello
```

If your class does not include a specific command processor 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 will drop us out of the interpreter.

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

Notice that no newline is printed, so the results are a little messy.

Command Arguments

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

Navigation

Table of Contents
Previous: Program Frameworks
Next: shlex – Lexical analysis of shell-style syntaxes.

This Page

Show Source

Examples

The output from all the example programs from PyMOTW has been generated with Python 2.7, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    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()
```

First, let's look at the help. The docstring added to **do_greet()** becomes the help text for the command:

```
$ python cmd_arguments.py
(Cmd) help

Documented commands (type help ):
========================================
greet

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

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

The output shows one optional argument to the greet command, *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 processor 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 processor does not include the command itself. That simplifies parsing in the command processor, 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 our source. We could edit the source to remove the extra white-space, but that would leave our application looking poorly formatted. An alternative solution is to implement a help handler for the greet command, named **help_greet()**. When present, the help handler is called on to produce help text for the named command.

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    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 simple 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.

```
$ python 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 processor 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.

```
$ python 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 you to assemble a list of possible completions using your own criteria (query a database, look at at a file or directory on the filesystem, etc.). 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 either read it once and cache the contents to be scanned as needed.

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

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

    def do_greet(self, person):
        "Greet the person"
        if person and person in self.FRIENDS:
            greeting = 'hi, %s!' % person
        elif person:
            greeting = "hello, " + 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.

```
$ python 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.

```
(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.

```
import cmd

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

    def cmdloop(self, intro=None):
        print 'cmdloop(%s)' % intro
        return cmd.Cmd.cmdloop(self, intro)

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

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

    def parseline(self, line):
        print 'parseline(%s) =>' % line,
        ret = cmd.Cmd.parseline(self, line)
        print ret
        return ret

    def onecmd(self, s):
        print 'onecmd(%s)' % s
        return cmd.Cmd.onecmd(self, s)

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

    def default(self, line):
        print 'default(%s)' % line
        return cmd.Cmd.default(self, line)

    def precmd(self, line):
        print 'precmd(%s)' % line
        return cmd.Cmd.precmd(self, line)

    def postcmd(self, stop, line):
        print 'postcmd(%s, %s)' % (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__':
    Illustrate().cmdloop('Illustrating the methods of cmd.Cmd')
```

cmdloop() is the main processing loop of the interpreter. You can override it, but 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 processor. 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 processor is looked up and invoked. If none is found, **default()** is called instead. Finally postcmd() is called.

Here's an example session with `print` statements added:

```
$ python 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 above, 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 you can set it on the class directly.

When printing help, the `doc_header`, `misc_header`, `undoc_header`, and `ruler` attributes are used to format the output.

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

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    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()
```

```
$ python cmd_attributes.py
Simple command processor example.
prompt: prompt hello
hello: help

doc_header
----------
prompt

undoc_header
------------
EOF  help

hello:
```

Shelling Out

To supplement the standard command processing, **`Cmd`** includes 2 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.

```
import cmd
import os

class ShellEnabled(cmd.Cmd):

    last_output = ''

    def do_shell(self, line):
        "Run a shell command"
        print "running shell command:", line
        output = os.popen(line).read()
        print output
        self.last_output = output

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

    def do_EOF(self, line):
        return True

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

```
$ python cmd_do_shell.py
(Cmd) ?

Documented commands (type help ):
========================================
echo  shell

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

(Cmd) ? shell
Run a shell command
(Cmd) ? echo
Print the input, replacing '$out' with the output of the last shell com
(Cmd) shell pwd
running shell command: pwd
/Users/dhellmann/Documents/PyMOTW/in_progress/cmd

(Cmd) ! pwd
running shell command: pwd
/Users/dhellmann/Documents/PyMOTW/in_progress/cmd

(Cmd) echo $out
/Users/dhellmann/Documents/PyMOTW/in_progress/cmd

(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 | python cmd_do_help.py
(Cmd)
Documented commands (type help ):
========================================
greet

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

(Cmd)
```

If you would rather have your program read the script file directly, a few other changes may be needed. Since **`readline`** interacts with the terminal/tty device, rather than the standard input stream, you should disable it if you know your script is going to be reading from a file. Also, to avoid printing superfluous prompts, you can set the prompt 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.

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    # 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
    input = open(sys.argv[1], 'rt')
    try:
        HelloWorld(stdin=input).cmdloop()
    finally:
        input.close()
```

With *use_rawinput* set to False and *prompt* set to an empty string, we can call the script on this input file:

```
greet
greet Alice and Bob
```

to produce output like:

```
$ python cmd_file.py cmd_file.txt
hello,
hello, Alice and Bob
```

Commands from sys.argv

You can also process command line arguments to the program as a command for your interpreter class, instead of reading commands from stdin or a file. To use the command line arguments, you can call **onecmd()** directly, as in this example.

```
import cmd

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

    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.

```
$ python cmd_argv.py greet Command Line User
hello, Command Line User
$ python cmd_argv.py
(Cmd) greet Interactive User
hello, Interactive User
(Cmd)
```

See also:

cmd

The standard library documentation for this module.

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.