readline — The GNU readline Library

Purpose: Provides an interface to the GNU readline library for interacting with the user at a command prompt.

The `readline` module can be used to enhance interactive command line programs to make them easier to use. It is primarily used to provide command line text completion, or "tab completion".

Note

Because `readline` interacts with the console content, printing debug messages makes it difficult to see what is happening in the sample code versus what readline is doing for free. The following examples use the `logging` module to write debug information to a separate file. The log output is shown with each example.

Note

The GNU libraries needed for `readline` are not available on all platforms by default. If your system does not include them, you may need to recompile the Python interpreter to enable the module, after installing the dependencies. A stand-alone version of the library is also distributed from the Python Package Index under the name gnureadline. The examples in this section first try to import gnureadline, and then fall back to readline.

Special thanks to Jim Baker for pointing out this package.

Configuring

There are two ways to configure the underlying readline library, using a configuration file or the `parse_and_bind()` function. Configuration options include the key-binding to invoke completion, editing modes (`vi` or `emacs`), and many other values. Refer to the documentation for the GNU readline library for details.

The easiest way to enable tab-completion is through a call to `parse_and_bind()`. Other options can be set at the same time. This example changes the editing controls to use "vi" mode instead of the default of "emacs". To edit the current input line, press ESC then use normal `vi` navigation keys such as `j`, `k`, `l`, and `h`.

```
# readline_parse_and_bind.py

try:
    import gnureadline as readline
except ImportError:
    import readline

readline.parse_and_bind('tab: complete')
readline.parse_and_bind('set editing-mode vi')

while True:
    line = input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print('ENTERED: {!r}'.format(line))
```

The same configuration can be stored as instructions in a file read by the library with a single call. If `myreadline.rc` contains

```
# myreadline.rc

# Turn on tab completion
tab: complete

# Use vi editing mode instead of emacs
set editing-mode vi
```

the file can be read with `read_init_file()`

```
# readline_read_init_file.py

try:
    import gnureadline as readline
except ImportError:
    import readline

readline.read_init_file('myreadline.rc')

while True:
    line = input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print('ENTERED: {!r}'.format(line))
```

Completing Text

This program has a built-in set of possible commands and uses tab-completion when the user is entering instructions.

```
# readline_completer.py

try:
    import gnureadline as readline
except ImportError:
    import readline
import logging

LOG_FILENAME = '/tmp/completer.log'
logging.basicConfig(
    format='%(message)s',
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)


class SimpleCompleter:

    def __init__(self, options):
        self.options = sorted(options)

    def complete(self, text, state):
        response = None
        if state == 0:
            # This is the first time for this text,
            # so build a match list.
            if text:
                self.matches = [
                    s
                    for s in self.options
                    if s and s.startswith(text)
                ]
                logging.debug('%s matches: %s',
                              repr(text), self.matches)
            else:
                self.matches = self.options[:]
                logging.debug('(empty input) matches: %s',
                              self.matches)

        # Return the state'th item from the match list,
        # if we have that many.
        try:
            response = self.matches[state]
        except IndexError:
            response = None
        logging.debug('complete(%s, %s) => %s',
                      repr(text), state, repr(response))
        return response


def input_loop():
```

```
def input_loop():
    line = ''
    while line != 'stop':
        line = input('Prompt ("stop" to quit): ')
        print('Dispatch {}'.format(line))


# Register the completer function
OPTIONS = ['start', 'stop', 'list', 'print']
readline.set_completer(SimpleCompleter(OPTIONS).complete)

# Use the tab key for completion
readline.parse_and_bind('tab: complete')

# Prompt the user for text
input_loop()
```

The `input_loop()` function reads one line after another until the input value is `"stop"`. A more sophisticated program could actually parse the input line and run the command.

The `SimpleCompleter` class keeps a list of "options" that are candidates for auto-completion. The `complete()` method for an instance is designed to be registered with `readline` as the source of completions. The arguments are a `text` string to complete and a `state` value, indicating how many times the function has been called with the same text. The function is called repeatedly with the state incremented each time. It should return a string if there is a candidate for that state value or `None` if there are no more candidates. The implementation of `complete()` here looks for a set of matches when state is `0`, and then returns all of the candidate matches one at a time on subsequent calls.

When run, the initial output is:

```
$ python3 readline_completer.py

Prompt ("stop" to quit):
```

Pressing TAB twice causes a list of options to be printed.

```
$ python3 readline_completer.py

Prompt ("stop" to quit):
list   print  start  stop
Prompt ("stop" to quit):
```

The log file shows that `complete()` was called with two separate sequences of state values.

```
$ tail -f /tmp/completer.log

(empty input) matches: ['list', 'print', 'start', 'stop']
complete('', 0) => 'list'
complete('', 1) => 'print'
complete('', 2) => 'start'
complete('', 3) => 'stop'
complete('', 4) => None
(empty input) matches: ['list', 'print', 'start', 'stop']
complete('', 0) => 'list'
complete('', 1) => 'print'
complete('', 2) => 'start'
complete('', 3) => 'stop'
complete('', 4) => None
```

The first sequence is from the first TAB key-press. The completion algorithm asks for all candidates but does not expand the empty input line. Then on the second TAB, the list of candidates is recalculated so it can be printed for the user.

If the next input is "`l`" followed by another TAB, the screen shows:

```
Prompt ("stop" to quit): list
```

and the log reflects the different arguments to `complete()`:

```
'l' matches: ['list']
complete('l', 0) => 'list'
complete('l', 1) => None
```

Pressing RETURN now causes `input()` to return the value, and the `while` loop cycles.

```
Dispatch list
Prompt ("stop" to quit):
```

There are two possible completions for a command beginning with “`s`”. Typing “`s`”, then pressing TAB finds that “`start`” and “`stop`” are candidates, but only partially completes the text on the screen by adding a “`t`”.

The log file shows:

```
's' matches: ['start', 'stop']
complete('s', 0) => 'start'
complete('s', 1) => 'stop'
complete('s', 2) => None
```

and the screen:

```
Prompt ("stop" to quit): st
```

Note

If a completer function raises an exception, it is ignored silently and `readline` assumes there are no matching completions.

Accessing the Completion Buffer

The completion algorithm in `SimpleCompleter` only looks at the text argument passed to the function, but does not use any more of readline’s internal state. It is also possible to use `readline` functions to manipulate the text of the input buffer.

```
# readline_buffer.py

try:
    import gnureadline as readline
except ImportError:
    import readline
import logging

LOG_FILENAME = '/tmp/completer.log'
logging.basicConfig(
    format='%(message)s',
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)


class BufferAwareCompleter:

    def __init__(self, options):
        self.options = options
        self.current_candidates = []

    def complete(self, text, state):
        response = None
        if state == 0:
            # This is the first time for this text,
            # so build a match list.

            origline = readline.get_line_buffer()
            begin = readline.get_begidx()
            end = readline.get_endidx()
            being_completed = origline[begin:end]
            words = origline.split()

            logging.debug('origline=%s', repr(origline))
            logging.debug('begin=%s', begin)
            logging.debug('end=%s', end)
```

```
            logging.debug('being_completed=%s', being_completed)
            logging.debug('words=%s', words)

            if not words:
                self.current_candidates = sorted(
                    self.options.keys()
                )
            else:
                try:
                    if begin == 0:
                        # first word
                        candidates = self.options.keys()
                    else:
                        # later word
                        first = words[0]
                        candidates = self.options[first]

                    if being_completed:
                        # match options with portion of input
                        # being completed
                        self.current_candidates = [
                            w for w in candidates
                            if w.startswith(being_completed)
                        ]
                    else:
                        # matching empty string,
                        # use all candidates
                        self.current_candidates = candidates

                    logging.debug('candidates=%s',
                                  self.current_candidates)

                except (KeyError, IndexError) as err:
                    logging.error('completion error: %s', err)
                    self.current_candidates = []

        try:
            response = self.current_candidates[state]
        except IndexError:
            response = None
        logging.debug('complete(%s, %s) => %s',
                      repr(text), state, response)
        return response


def input_loop():
    line = ''
    while line != 'stop':
        line = input('Prompt ("stop" to quit): ')
        print('Dispatch {}'.format(line))


# Register our completer function
completer = BufferAwareCompleter({
    'list': ['files', 'directories'],
    'print': ['byname', 'bysize'],
    'stop': [],
})
readline.set_completer(completer.complete)

# Use the tab key for completion
readline.parse_and_bind('tab: complete')

# Prompt the user for text
input_loop()
```

In this example, commands with sub-options are being completed. The `complete()` method needs to look at the position of the completion within the input buffer to determine whether it is part of the first word or a later word. If the target is the first word, the keys of the options dictionary are used as candidates. If it is not the first word, then the first word is used to find candidates from the options dictionary.

There are three top-level commands, two of which have sub-commands.

There are three top-level commands, two of which have sub-commands.

- list
  - files
  - directories
- print
  - byname
  - bysize
- stop

Following the same sequence of actions as before, pressing TAB twice gives the three top-level commands:

```
$ python3 readline_buffer.py

Prompt ("stop" to quit):
list   print  stop
Prompt ("stop" to quit):
```

and in the log:

```
origline=''
begin=0
end=0
being_completed=
words=[]
complete('', 0) => list
complete('', 1) => print
complete('', 2) => stop
complete('', 3) => None
origline=''
begin=0
end=0
being_completed=
words=[]
complete('', 0) => list
complete('', 1) => print
complete('', 2) => stop
complete('', 3) => None
```

If the first word is "`list `" (with a space after the word), the candidates for completion are different.

```
Prompt ("stop" to quit): list
directories  files
```

The log shows that the text being completed is *not* the full line, but the portion after `list`.

```
origline='list '
begin=5
end=5
being_completed=
words=['list']
candidates=['files', 'directories']
complete('', 0) => files
complete('', 1) => directories
complete('', 2) => None
origline='list '
begin=5
end=5
being_completed=
words=['list']
candidates=['files', 'directories']
complete('', 0) => files
complete('', 1) => directories
complete('', 2) => None
```

Input History

readline tracks the input history automatically. There are two different sets of functions for working with the history. The history for the current session can be accessed with `get_current_history_length()` and `get_history_item()`. That same history can be saved to a file to be reloaded later using `write_history_file()` and `read_history_file()`. By default the entire history is saved but the maximum length of the file can be set with `set_history_length()`. A length of -1 means no limit.

```
# readline_history.py
```

```
try:
    import gnureadline as readline
except ImportError:
    import readline
import logging
import os

LOG_FILENAME = '/tmp/completer.log'
HISTORY_FILENAME = '/tmp/completer.hist'

logging.basicConfig(
    format='%(message)s',
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)


def get_history_items():
    num_items = readline.get_current_history_length() + 1
    return [
        readline.get_history_item(i)
        for i in range(1, num_items)
    ]


class HistoryCompleter:

    def __init__(self):
        self.matches = []

    def complete(self, text, state):
        response = None
        if state == 0:
            history_values = get_history_items()
            logging.debug('history: %s', history_values)
            if text:
                self.matches = sorted(
                    h
                    for h in history_values
                    if h and h.startswith(text)
                )
            else:
                self.matches = []
            logging.debug('matches: %s', self.matches)
        try:
            response = self.matches[state]
        except IndexError:
            response = None
        logging.debug('complete(%s, %s) => %s',
                      repr(text), state, repr(response))
        return response


def input_loop():
    if os.path.exists(HISTORY_FILENAME):
        readline.read_history_file(HISTORY_FILENAME)
    print('Max history file length:',
          readline.get_history_length())
    print('Startup history:', get_history_items())
    try:
        while True:
            line = input('Prompt ("stop" to quit): ')
```

```
            if line == 'stop':
                break
            if line:
                print('Adding {!r} to the history'.format(line))
    finally:
        print('Final history:', get_history_items())
        readline.write_history_file(HISTORY_FILENAME)


# Register our completer function
readline.set_completer(HistoryCompleter().complete)

# Use the tab key for completion
readline.parse_and_bind('tab: complete')

# Prompt the user for text
input_loop()
```

The `HistoryCompleter` remembers everything typed, and uses those values when completing subsequent inputs.

```
$ python3 readline_history.py

Max history file length: -1
Startup history: []
Prompt ("stop" to quit): foo
Adding 'foo' to the history
Prompt ("stop" to quit): bar
Adding 'bar' to the history
Prompt ("stop" to quit): blah
Adding 'blah' to the history
Prompt ("stop" to quit): b
bar   blah
Prompt ("stop" to quit): b
Prompt ("stop" to quit): stop
Final history: ['foo', 'bar', 'blah', 'stop']
```

The log shows this output when the “b” is followed by two TABs.

```
history: ['foo', 'bar', 'blah']
matches: ['bar', 'blah']
complete('b', 0) => 'bar'
complete('b', 1) => 'blah'
complete('b', 2) => None
history: ['foo', 'bar', 'blah']
matches: ['bar', 'blah']
complete('b', 0) => 'bar'
complete('b', 1) => 'blah'
complete('b', 2) => None
```

When the script is run the second time, all of the history is read from the file.

```
$ python3 readline_history.py

Max history file length: -1
Startup history: ['foo', 'bar', 'blah', 'stop']
Prompt ("stop" to quit):
```

There are functions for removing individual history items and clearing the entire history, as well.

Hooks

There are several hooks available for triggering actions as part of the interaction sequence. The *startup* hook is invoked immediately before printing the prompt, and the *pre-input* hook is run after the prompt, but before reading text from the user.

```
# readline_hooks.py


try:
    import gnureadline as readline
except ImportError:
```

```
except ImportError:
    import readline


def startup_hook():
    readline.insert_text('from startup_hook')


def pre_input_hook():
    readline.insert_text(' from pre_input_hook')
    readline.redisplay()


readline.set_startup_hook(startup_hook)
readline.set_pre_input_hook(pre_input_hook)
readline.parse_and_bind('tab: complete')

while True:
    line = input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print('ENTERED: {!r}'.format(line))
```

Either hook is a potentially good place to use `insert_text()` to modify the input buffer.

```
$ python3 readline_hooks.py

Prompt ("stop" to quit): from startup_hook from pre_input_hook
```

If the buffer is modified inside the pre-input hook, `redisplay()` must be called to update the screen.

See also

- Standard library documentation for readline
- GNU readline – Documentation for the GNU readline library.
- readline init file format – The initialization and configuration file format.
- effbot: The readline module – Effbot's guide to the readline module.
- gnureadline – A statically linked version of `readline` available for many platforms and installable via `pip`.
- pyreadline – pyreadline, developed as a Python-based replacement for readline to be used on Windows.
- `cmd` – The `cmd` module uses `readline` extensively to implement tab-completion in the command interface. Some of the examples here were adapted from the code in `cmd`.
- `rlcompleter` – `rlcompleter` uses `readline` to add tab-completion to the interactive Python interpreter.

getopt — Command Line Option Parsing

getpass — Secure Password Prompt

Quick Links

Configuring
Completing Text
Accessing the Completion Buffer
Input History
Hooks

This page was last updated 2017-11-14.

Navigation

getopt — Command Line Option Parsing
getpass — Secure Password Prompt

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