6.7. `readline` — GNU readline interface

The `readline` module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the `rlcompleter` module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the built-in `input()` function.

> **Note:** The underlying Readline library API may be implemented by the `libedit` library instead of GNU readline. On MacOS X the `readline` module detects which library is being used at run time.
>
> The configuration file for `libedit` is different from that of GNU readline. If you programmatically load configuration strings you can check for the text "libedit" in `readline.__doc__` to differentiate between GNU readline and libedit.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

6.7.1. Init file

The following functions relate to the init file and user configuration:

`readline.`**`parse_and_bind`**(*string*)

Execute the init line provided in the *string* argument. This calls `rl_parse_and_bind()` in the underlying library.

`readline.`**`read_init_file`**([*filename*])

Execute a readline initialization file. The default filename is the last filename used. This calls `rl_read_init_file()` in the underlying library.

6.7.2. Line buffer

The following functions operate on the line buffer:

`readline.`**`get_line_buffer`**()

Return the current contents of the line buffer (`rl_line_buffer` in the underlying library).

`readline.`**`insert_text`**(*string*)

Insert text into the line buffer at the cursor position. This calls `rl_insert_text()` in the underlying library, but ignores the return value.

`readline.`**`redisplay`**()

Change what's displayed on the screen to reflect the current contents of the line buffer. This calls `rl_redisplay()` in the underlying library.

6.7.3. History file

The following functions operate on a history file:

`readline.`**`read_history_file`**([*filename*])

Load a readline history file, and append it to the history list. The default filename is `~/.history`. This calls `read_history()` in the underlying library.

`readline.`**`write_history_file`**([*filename*])

Save the history list to a readline history file, overwriting any existing file. The default filename is `~/.history`. This calls `write_history()` in the underlying library.

`readline.`**`append_history_file`**(*nelements*[, *filename*])

Append the last *nelements* items of history to a file. The default filename is `~/.history`. The file must already exist. This calls `append_history()` in the underlying library. This function only exists if Python was compiled for a version of the library that supports it.

New in version 3.5.

`readline.`**`get_history_length`**()
`readline.`**`set_history_length`**(*length*)

Set or return the desired number of lines to save in the history file. The `write_history_file()` function uses this value to truncate the history file, by calling `history_truncate_file()` in the underlying library. Negative values imply unlimited history file size.

6.7.4. History list

The following functions operate on a global history list:

`readline.`**`clear_history`**()

Clear the current history. This calls `clear_history()` in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

`readline.`**`get_current_history_length`**`()`

Return the number of items currently in the history. (This is different from `get_history_length()`, which returns the maximum number of lines that will be written to a history file.)

`readline.`**`get_history_item`**`(`*index*`)`

Return the current contents of history item at *index*. The item index is one-based. This calls `history_get()` in the underlying library.

`readline.`**`remove_history_item`**`(`*pos*`)`

Remove history item specified by its position from the history. The position is zero-based. This calls `remove_history()` in the underlying library.

`readline.`**`replace_history_item`**`(`*pos*, *line*`)`

Replace history item specified by its position with *line*. The position is zero-based. This calls `replace_history_entry()` in the underlying library.

`readline.`**`add_history`**`(`*line*`)`

Append *line* to the history buffer, as if it was the last line typed. This calls `add_history()` in the underlying library.

`readline.`**`set_auto_history`**`(`*enabled*`)`

Enable or disable automatic calls to `add_history()` when reading input via readline. The *enabled* argument should be a Boolean value that when true, enables auto history, and that when false, disables auto history.

New in version 3.6.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5. Startup hooks

`readline.`**`set_startup_hook`**`([`*function*`])`

Set or remove the function invoked by the `rl_startup_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

`readline.`**`set_pre_input_hook`**`([`*function*`])`

Set or remove the function invoked by the `rl_pre_input_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

6.7.6. Completion

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by `rlcompleter` to complete Python identifiers for the interactive interpreter. If the `readline` module is to be used with a custom completer, a different set of word delimiters should be set.

`readline.`**`set_completer`**([*function*])

Set or remove the completer function. If *function* is specified, it will be used as the new completer function; if omitted or `None`, any completer function already installed is removed. The completer function is called as `function(text, state)`, for *state* in `0`, `1`, `2`, ..., until it returns a non-string value. It should return the next possible completion starting with *text*.

The installed completer function is invoked by the *entry_func* callback passed to `rl_completion_matches()` in the underlying library. The *text* string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

`readline.`**`get_completer`**()

Get the completer function, or `None` if no completer function has been set.

`readline.`**`get_completion_type`**()

Get the type of completion being attempted. This returns the `rl_completion_type` variable in the underlying library as an integer.

`readline.`**`get_begidx`**()
`readline.`**`get_endidx`**()

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library.

`readline.`**`set_completer_delims`**(*string*)
`readline.`**`get_completer_delims`**()

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the `rl_completer_word_break_characters` variable in the underlying library.

`readline.`**`set_completion_display_matches_hook`**([*function*])

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. This sets or clears the `rl_completion_display_matches_hook` callback in the underlying library. The completion display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

6.7.7. Example

The following example demonstrates how to use the `readline` module's history reading and writing functions to automatically load and save a history file named `.python_history` from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's `PYTHONSTARTUP` file.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

This code is actually automatically run when Python is run in interactive mode (see Readline configuration).

The following example achieves the same goal but supports concurrent interactive sessions, by only appending the new history.

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")
```

```
try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

The following example extends the `code.InteractiveConsole` class to support history save/restore.

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```