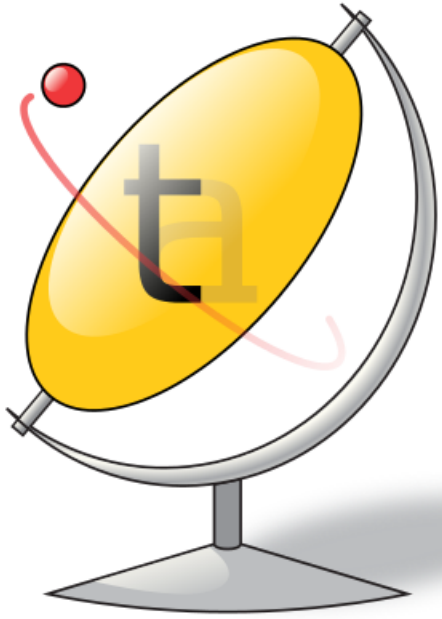
Textadept

A fast, minimalist, and remarkably extensible text editor



Quick Reference

Mitchell

Textadept Quick Reference

Textadept is a fast, minimalist, and remarkably extensible cross-platform text editor for programmers. This quick reference contains a wealth of knowledge on how to script and configure Textadept using the Lua programming language. It groups the editor's rich API into a series of tasks in a convenient and easy-to-use manner.

This book covers how to:

- Leverage Textadept's important files and folders
- Adeptly navigate and manipulate text
- Mark lines and text visually
- Show interactive lists and call tips
- Prompt for user input in various ways
- Configure colors, themes, and other settings
- Define lexers for highlighting source code
- And much more

Mitchell is the author and principal developer of Textadept and commands over 7 years of experience with Lua.

Textadept

Quick Reference

Mitchell

Textadept Quick Reference
by Mitchell

Contact the author at mitchell.att.foicica.com.

Editor: Ana Balan
Cover Designer: Mitchell
Interior Designer: Mitchell
Indexer: Mitchell

Printing history:
December 2013: First Edition

ISBN: 978-0-9912379-0-6

Contents

To Jeff Elkner

Introduction

Textadept is a fast, minimalist, and remarkably extensible cross-platform text editor for programmers. Written in a combination of C and Lua[1] and relentlessly optimized for speed and minimalism for over six years, Textadept is an ideal editor for programmers who want endless extensibility without sacrificing speed or succumbing to code bloat and featuritis.

Textadept runs in both graphical and text-based user interface environments. The text-based version of the editor is referred to as the "terminal version", since it executes within a terminal emulator. Textadept also supports the standard Lua and LuaJIT[2] environments. The version that utilizes LuaJIT is referred to as the "LuaJIT version".

Textadept Quick Reference is designed to help the user "get things done" when it comes to scripting and configuring Textadept. Its pragmatic approach assumes the user has a working knowledge of both Lua and Textadept. This book is broken up into a number of descriptive sections with conveniently grouped tasks that cover nearly every aspect of Textadept's Application Programming Interface (API). For the most part, the contents of each task are not listed in conceptual order. They are listed in procedural order, an order the user would likely follow when writing Lua scripts. This quick reference serves as a complement to Textadept's comprehensive Manual and extensive API documentation.

While this book aims to be a complete reference, it does omit some of the less useful features of Textadept's API. For example, although many of Textadept's table fields are both readable and writable, this reference sometimes chooses to cover only one of those operations. (Unless a field is marked "Read-only" or "Write-only", it is readable and writable.) This book also does not cover Lua's standard libraries.

Finally, the facilities in this book are designed to be used primarily in user-written Lua scripts and in the occasional "one-shot" Lua command. If the user keeps this in mind, he or she can realize Textadept's full potential.

1 *http://www.lua.org*
2 *http://luajit.org*

Download

Textadept binary packages for Windows, Mac OSX, and Linux platforms are available from *http://foicica.com/textadept*. Each package is self-contained and need not be installed. The user may also download a source package and compile Textadept manually by following the instructions in the editor's Manual.

Conventions

This book uses the following conventions.

Italic
: Used for filenames and for introducing new terms.

`Constant width`
: Used for environment variables, command line options, and Lua code, including functions, tables, and variables.

`Constant width italic`
: Used for user-specified parameters.

`[]`
: Used for optional function arguments, except in code examples that index Lua tables. Unless otherwise specified, optional arguments default to `nil`.

Terminology

This book uses the following terminology.

Buffer
: An object that contains editable text.

View
: An object that contains a single buffer.

Caret
: Either the visual that represents the text insertion point or the end point of a text selection.

Anchor
: The start point of a text selection or search.

Virtual Space
: The space past the ends of lines.

Lexer
: A Lua module that highlights the syntax of source code written in a particular programming language. Textadept refers to a programming language by its lexer's name.

Style
: A collection of display settings specific to source code comments, strings, keywords, and other ranges of text.

Language Module
: A Lua module automatically loaded by Textadept when editing source code in a particular programming language. The module's name matches the language's lexer name. Not all languages have language modules.

Environment Variables

Textadept utilizes the following environment variables.

`HOME` or `USERHOME`
: The user's home directory. Textadept's user data and preferences exist in a *.textadept/* sub-directory, denoted as *~/.textadept/* throughout this book.

`LANG`
: The user's default locale. Textadept will display localized text and messages in it if possible.

`TEXTADEPTJIT` (Mac OSX only)
: When set, *Textadept.app* runs its LuaJIT version.

`TA_LUA_PATH`
`TA_LUA_CPATH`
: The Textadept equivalent of `LUA_PATH` and `LUA_CPATH`. Used by Lua's `require()` function for finding modules.

Important Files and Directories

Textadept allows the user to configure and customize the editor using several important files and directories contained within his or her *~/.textadept/* directory.

~/.textadept/init.lua

The file executed on startup that allows the user to customize what Textadept does when it starts. Examples include changing the settings of existing modules, loading new modules, and running arbitrary Lua code. Example 1 shows a sample *~/.textadept/init.lua* file.

Example 1. Sample ~/.textadept/init.lua

```
-- Disable character autopairing with typeover.
textadept.editing.AUTOPAIR = false
textadept.editing.TYPEOVER_CHARS = false

-- Load a user module from ~/.textadept/modules/.
foo = require('foo')

-- Remap the new buffer command from Ctrl+N to
-- Ctrl+Shift+N.
keys.cN, keys.cn = buffer.new, nil

-- Define some global snippets.
snippets['foo'] = 'bar'
snippets['file'] = '%<buffer.filename>'

-- Recognize .luadoc files as Lua code.
textadept.file_types.extensions.luadoc = 'lua'

-- Change .html files to be recognized as XML files.
textadept.file_types.extensions.html = 'xml'

-- Recognize a shebang line like "#!/usr/bin/zsh" as
-- shell code.
textadept.file_types.shebangs.zsh = 'bash'

-- Adjust the default theme's font name and size.
if not CURSES then
  local font = 'DejaVu Sans Mono'
  ui.set_theme('light', {font = font, fontsize = 12})
end

-- Change the color of Java functions from orange to
-- black.
events.connect(events.LEXER_LOADED, function(lang)
  if lang ~= 'java' then return end
  local black = 'fore:%(color.light_black)'
  buffer.property['style.function'] = black
end)
```

~/.textadept/properties.lua

The file executed every time Textadept creates a new buffer or split view. Allows the user to set per-buffer properties (like indentation size) and view-related properties (like scrolling and autocompletion behavior). Any settings within *~/.textadept/properties.lua* override Textadept's default *properties.lua* settings. (The latter file is a good reference for configurable properties.) Example 2 shows a sample *~/.textadept/properties.lua* file.

Example 2. Sample ~/.textadept/properties.lua

```
-- Use a block-style caret.
buffer.caret_style = buffer.CARETSTYLE_BLOCK
buffer.caret_period = 0

-- Always use tabs of width 4 for indentation.
buffer.use_tabs, buffer.tab_width = true, 4

-- Disable code folding.
buffer.property['fold'] = '0'

-- Wrap long lines into view.
buffer.wrap_mode = buffer.WRAP_WORD
```

~/.textadept/locale.conf

Defines Textadept's localized messages. The user may override or manually set Textadept's locale by copying a locale file from the editor's *core/locales/* directory to *~/.textadept/locale.conf*.

~/.textadept/modules/

Contains user modules. When Textadept looks for modules to load via Lua's `require()` function, it looks in this directory first. The user can override one of Textadept's own modules by creating a new module of the same name in *~/.textadept/modules/*. For example, the user may create a *~/.textadept/modules/textadept/menu.lua* file with a completely different menu structure. Textadept will load that file on startup instead of its own.

~/.textadept/modules/`lexer`/post_init.lua

The file executed after Textadept loads the language module for editing source code in language *`lexer`*. A *post_init.lua* file allows the user to extend (not override) the functionality of its corresponding language module.

Textadept will not execute a *post_init.lua* file without its corresponding language module. Example 3 shows a sample *~/.textadept/modules/lua/post_init.lua* file.

Example 3. Sample ~/.textadept/modules/lua/post_init.lua

```
-- Always use tabs in Lua files.
events.connect(events.LEXER_LOADED, function(lang)
  if lang == 'lua' then buffer.use_tabs = true end
end)

-- Change Lua's run command, load more Adeptsense tags,
-- and add an additional key binding and snippet.
textadept.run.run_commands.lua = 'lua5.2'
_M.lua.sense:load_ctags('path_to_my_projects_tags')
keys.lua['c\n'] = function()
  buffer:line_end()
  buffer:add_text('end')
  buffer:new_line()
end
snippets.lua['ver'] = '%<_VERSION>'
```

~/.textadept/themes/
: Contains user themes. When Textadept looks for a color theme to load, it looks in this directory first. The user can override one of Textadept's own themes by creating a new theme of the same name in *~/.textadept/themes/*. Example 4 shows a sample *~/.textadept/themes/light.lua* file. The section "Create or Modify a Color Theme" on page 86 describes themes in more detail.

Example 4. Sample ~/.textadept/themes/light.lua

```
dofile(_HOME..'/themes/light.lua')
buffer.property['font'] = 'DejaVu Sans Mono'
buffer.property['fontsize'] = 12
```

~/.textadept/lexers/
: Contains user lexers. When Textadept looks for lexers to highlight source code with, it looks in this directory first. The user can override one of Textadept's own lexers by creating a new lexer of the same name in *~/.textadept/lexers/*. Example 5 shows a simple lexer for a non-existent language. The section "Define a Lexer" on page 109 describes lexers in more detail.

Example 5. Sample ~/.textadept/lexers/foo.lua

```
local l = lexer
local token, word_match = l.token, l.word_match
local P, R, S = lpeg.P, lpeg.R, lpeg.S

local M = {_NAME = 'foo'}

local ws = token(l.WHITESPACE, l.space^1)

local comment = token(l.COMMENT, '#' * l.nonnewline^0)

local string = token(l.STRING,
                     l.delimited_range('"', true))

local number = token(l.NUMBER, l.float + l.integer)

local keyword = token(l.KEYWORD, word_match{
  'else', 'for', 'function', 'if', 'return', 'while'
})

local special = token('special', P('foo'))

local identifier = token(l.IDENTIFIER, l.word)

local operator = token(l.OPERATOR, S('+-/*^<>=()[]{}'))

M._rules = {
  {'whitespace', ws},
  {'keyword', keyword},
  {'special', special},
  {'identifier', identifier},
  {'string', string},
  {'comment', comment},
  {'number', number},
  {'operator', operator},
}

M._tokenstyles = {special = l.STYLE_CONSTANT}

M._foldsymbols = {
  _patterns = {'[{}]', '#'},
  [l.COMMENT] = {['#'] = l.fold_line_comments('#')},
  [l.OPERATOR] = {['{'] = 1, ['}'] = -1}
}

return M
```

Command Line Options

Textadept processes command line options sequentially, so order matters. The editor has the following command line options.

filename
: Opens file *filename* for editing.

-h
--help
: Shows a list of all command line options.

-f
--force
: Forces Textadept to open in a new instance. On Mac OSX and Linux, Textadept is a single-instance application.

-u *dir*
--userhome *dir*
: Designates directory *dir* as the user's Textadept data directory in place of *~/.textadept/*.

-e *command*
--execute *command*
: Executes Lua command *command*. The section "Issue Lua Commands" on page 75 describes Lua commands in more detail.

-n
--nosession
: Does not restore the previous session on startup and does not save the current session before quitting.

-s *session*
--session *session*
: Loads session *session* on startup. *session* may be a file path or the name of a session in the user's *~/.textadept/* directory.

NOTE

Textadept will not save its state to *session* before quitting, but rather to its default session file. The section "Configure Session Settings" on page 17 describes how to change the default session file.

Define Custom Options

The user may register his or her own command line options within *~/.textadept/init.lua*.

`args.register(`*`short`*`, `*`long`*`, `*`narg`*`, `*`f`*`, `*`description`*`)`
: Registers a command line switch with short and long versions *`short`* and *`long`*, respectively. *`narg`* is the number of arguments the switch accepts, *`f`* is the function called when the switch is tripped, and *`description`* is the switch's description when displaying help.

Global Variables

Textadept defines the following global variables. If they are hidden by variables of the same name in local scopes, the user can access them using the prefix "`_G.`" (e.g. `_G.buffer`).

`buffer`
`view`
: The current buffer or view.

`_BUFFERS[`*`n`*`]`
`_BUFFERS[`*`buffer`*`]`
: Buffer number *`n`* or the number of buffer *`buffer`*.

`_VIEWS[`*`n`*`]`
`_VIEWS[`*`view`*`]`
: View number *`n`* or the number of view *`view`*.

`_L[`*`message`*`]`
: The localized form of string *`message`*.

`_M.`*`lexer`*
: The loaded language module for language *`lexer`*.

`arg[`*`n`*`]`
: Argument number *`n`* passed to Textadept on startup.

Platform Variables

Textadept defines a number of platform variables that vary depending on the environment the editor runs in.

`WIN32`
: Whether or not Textadept is running on Windows.

`OSX`
: Whether or not Textadept is running on Mac OSX.

`CURSES`
: Whether or not Textadept is running in the terminal.

`_HOME`
: The path to Textadept's home, or installation, directory.

`_USERHOME`
: The path to the user's Textadept data directory (typically *~/.textadept/*).

`_CHARSET`
: The filesystem's character encoding.

`_RELEASE`
: The Textadept release version string.

Handle Events

Textadept *emits events* when the user performs an action of interest, such as creating a buffer, typing a character, or selecting an item in an autocompletion list. The user can *connect* to these events in order to perform additional actions, such as displaying a call tip with information based on the selection from an autocompletion list. Events themselves are just arbitrary string names with associated sets of parameters. The user need not declare an event before emitting it. Textadept has its own set of event names that it uses, so the user should be mindful of this when connecting to and emitting custom events.

NOTE

Textadept's events are too numerous to list here. Instead, they are listed in the indexes and throughout this book.

`events.connect(`*`event`*`, `*`f`*`[, `*`index`*`])`
: Adds function *`f`* to the set of event handlers for event *`event`* at position *`index`*. If *`index`* is not given, appends *`f`* to the set of handlers.

`events.emit(`*`event`*`[, ...])`
: Sequentially calls all handler functions for event *`event`* with the given arguments. If any handler explicitly returns `true` or `false`, `events.emit()` returns that value and stops calling subsequent handlers.

`events.disconnect(`*`event`*`, `*`f`*`)`
: Removes function *`f`* from the set of handlers for event *`event`*.

Create Buffers and Views

Textadept introduces buffers and views as its principal objects. Since scripting and configuring the editor mostly involves manipulating buffers, their text, and their properties, much of this book is devoted to describing these facilities in detail. While the user can work with individual instances of buffers and views, it is really only useful to work with their global instances. Therefore, this book chooses to use the `buffer` and `view` notation. It also uses the terms "the buffer" and "the view" to refer to the current instances of the buffer and view that the user is working with, which are not necessarily global.

TIP

Unlike Lua indices (which start at 1) all buffer structure indices start at 0. Thus, buffer positions, line numbers, margin numbers, marker numbers, indicator numbers, and style numbers all start at 0. All other non-buffer structures start at 1, as expected.

`buffer.new()`
: Creates and returns a new buffer.

`view:split([`*`vertical`*`])`
: Splits the view horizontally into top and bottom views (unless *`vertical`* is `true`), focuses the new view, and returns both the old and new views.

  The terminal version does not support multiple views.

`view:unsplit()`
: Unsplits the view if possible, returning `true` on success.

Query View Information

The user can request the state and size of views.

`view.buffer` (Read-only)
: The buffer the view contains.

`view.size`
: The split resizer's pixel position if the view is a split one.

`ui.get_split_table()`
: Returns a split table that contains Textadept's current split view structure. The first two entries in a split table are the contents of each side of the split: either a view object or another split table. Each split table also has a `vertical` flag that indicates whether or not the split is a vertical one and a `size` field that indicates the split resizer's pixel position.

Handle Buffer and View Events

The user can connect to the following buffer and view events.

`events.BUFFER_NEW`
`events.BUFFER_DELETED`
: Emitted after creating a new buffer or deleting one.

`events.VIEW_NEW`
: Emitted after creating a new view.

Work with Files

Textadept provides many facilities for reading and writing to files. It can also work with directories and a wide range of different file encodings.

`io.open_file([`*`filename`*`])`
`io.open_file([`*`filenames`*`])`
: Opens string *`filename`* or the filenames in list *`filenames`*. If neither is given, opens the user-selected filenames.

`io.SNAPOPEN_MAX = `*`max`*
: Limits the number of files listed in the snapopen dialog to *`max`*. The default value is `1000`.

`io.snapopen(`*`path`*`[, `*`filter`*`[, `*`exclude_FILTER`*`[, `*`options`*`]]])`
`io.snapopen(`*`paths`*`[, `*`filter`*`[, `*`exclude_FILTER`*`[, `*`options`*`]]])`
: Prompts the user to select files to be opened from string directory *`path`* or the directories in list *`paths`* using a filtered list dialog. Files shown in the dialog do not match any pattern in either file filter *`filter`* or, unless *`exclude_FILTER`* is `true`, in `lfs.FILTER`. The following section, "File Filters", describes file filters. *`options`* is a table of additional options for `ui.dialogs.filteredlist()`.

`io.open_recent_file()`
: Prompts the user to select a recently opened file to be reopened.

`io.reload_file()`
: Reloads the current buffer's file contents, discarding any changes.

`textadept.editing.STRIP_TRAILING_SPACES = `*`bool`*
: Strip trailing whitespace before saving files. The default value is `false`.

`io.save_file()`
: Saves the current buffer to its file.

`io.save_file_as([`*`filename`*`])`
: Saves the current buffer to file *`filename`* or the user-specified filename.

`io.save_all_files()`
: Saves all unsaved buffers to their respective files.

`io.close_buffer()`
: Closes the current buffer, prompting the user to continue if there are unsaved changes, and returns `true` if the buffer was closed.

`io.close_all_buffers()`
: Closes all open buffers, prompting the user to continue if there are unsaved buffers, and returns `true` if the user did not cancel.

`lfs.dir_foreach(`*`dir, f`*`[, `*`filter`*`[, `*`exclude_FILTER`*`]])`
: Iterates over all files and sub-directories in directory *`dir`*, calling function *`f`* with each file found. Files passed to *`f`* do not match any pattern in either file filter *`filter`* or, unless *`exclude_FILTER`* is `true`, in `lfs.FILTER`. The follow-

ing section, “File Filters”, describes file filters.

File Filters

The user can specify *file filters* to exclude files and sub-directories from directory searches. A file filter is either a single Lua pattern that matches filenames to exclude, or a table of such patterns. File filter tables can also contain a `folders` subtable of directory-excluding patterns and an `extensions` subtable of raw file extensions to exclude. Table 3 on page 37 lists Lua pattern syntax. Patterns that start with ‘!’ exclude files and directories that do not match the remaining pattern. Example 6 shows a sample file filter for a C project under Mercurial (hg) version control.

Example 6. Sample file filter

```
local filter = {
  extensions = {'a', 'o', 'so'}, -- binary extensions
  folders = {'%.hg$', 'build'},  -- binary directories
  'COPYING', 'LICENSE',          -- unimportant files
}
```

Detect or Change File Encodings

The user can configure the *encoding* of individual files. File encodings specify how to display text characters. Textadept is capable of decoding files encoded in any of the encodings listed in Table 1, but by default it only attempts to decode the subset of encodings in the `io.encodings` list.

Table 1. Supported file encodings

Category	Encodings
European	ASCII, ISO-8859-{1,2,3,4,5,7,9,10,13,14,15,16}, KOI8-R, KOI8-U, KOI8-RU, CP{1250,1251,1252,1253,1254,1257}, CP{850,866,1131}, Mac{Roman,CentralEurope,Iceland,Croatian,Romania}, Mac{Cyrillic,Ukraine,Greek,Turkish}, Macintosh
Semitic	ISO-8859-{6,8}, CP{1255,1256}, CP862, Mac{Hebrew,Arabic}
Japanese	EUC-JP, SHIFT_JIS, CP932, ISO-2022-JP, ISO-2022-JP-2, ISO-2022-JP-1

Category	Encodings
Chinese	EUC-CN, HZ, GBK, CP936, GB18030, EUC-TW, BIG5, CP950, BIG5-HKSCS, BIG5-HKSCS:2004, BIG5-HKSCS:2001, BIG5-HKSCS:1999, ISO-2022-CN, ISO-2022-CN-EXT
Korean	EUC-KR, CP949, ISO-2022-KR, JOHAB
Armenian	ARMSCII-8
Georgian	Georgian-Academy, Georgian-PS
Tajik	KOI8-T
Kazakh	PT154, RK1048
Thai	ISO-8859-11, TIS-620, CP874, MacThai
Laotian	MuleLao-1, CP1133
Vietnamese	VISCII, TCVN, CP1258
Unicode	UTF-8, UCS-2, UCS-2BE, UCS-2LE, UCS-4, UCS-4BE, UCS-4LE, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, UTF-7, C99, JAVA

`io.encodings[#io.encodings + 1] = encoding`
: Adds encoding *`encoding`* to the list of encodings Textadept attempts to decode files in. The default encodings are `'UTF-8'`, `'ASCII'`, `'ISO-8859-1'`, and `'MacRoman'`, in that order.

`table.insert(io.encodings, i, encoding)`
: Identifies encoding *`encoding`* before the encoding at index *`i`*.

`io.boms[encoding] = bom`
: Associates byte-order mark *`bom`* with Unicode file encoding *`encoding`*.

`io.set_buffer_encoding(encoding)`
: Converts the current buffer's contents to encoding *`encoding`*.

Query File Information

The user can retrieve file information from buffers, as well as access recent filenames.

`buffer.filename`
: The absolute file path associated with the buffer.

`buffer.modify`
: Whether or not the buffer has unsaved changes.

`buffer.encoding`
: The string encoding of the file associated with the buffer, or `nil` for binary files.

`buffer.encoding_bom`
: The byte-order mark (if any) of the file associated with the buffer.

`io.recent_files[`*`n`*`]`
: The *n*th most recently opened file.

Handle Input and Output Events

The user can connect to the following file input and output events.

`events.URI_DROPPED` *`uri`*
: Emitted after dragging and dropping a file into a view. *`uri`* is the dropped file's UTF-8-encoded Uniform Resource Identifier. Example 7 on page 27 demonstrates how to convert filenames from UTF-8 to the filesystem's encoding.

`events.APPLEEVENT_ODOC` *`uri`*
: Emitted when Mac OSX tells Textadept to open a file. *`uri`* is the file's UTF-8-encoded Uniform Resource Identifier. Example 7 on page 27 demonstrates how to convert filenames from UTF-8 to the filesystem's encoding.

`events.FILE_OPENED` *`filename`*
: Emitted after opening a file in a new buffer. *`filename`* is the opened file's filename.

`events.FILE_BEFORE_SAVE` *`filename`*
`events.FILE_AFTER_SAVE` *`filename`*
: Emitted before or after saving a file to disk. *`filename`* is the filename of the file being saved.

`events.FILE_SAVED_AS` *`filename`*
: Emitted after saving a file under a different filename. *`filename`* is the new filename.

`events.FILE_CHANGED` *`filename`*
: Emitted when Textadept detects that an open file was modified externally. *`filename`* is that file's filename.

Work with Sessions

Textadept employs sessions to work with groups of files and to save and restore the editor's state.

`textadept.session.load([`*`filename`*`])`
: Loads session file *`filename`* or the user-selected session, returning `true` if a session file was opened and read.

`textadept.session.save([`*`filename`*`])`
: Saves the session to file *`filename`* or the user-selected file.

Configure Session Settings

The user's *~/.textadept/init.lua* file may change the default session file and the default session settings.

`textadept.session.DEFAULT_SESSION = `*`session_file`*
: Changes the default session file path to *`session_file`*. The default value is `_USERHOME`*/session*, or `_USERHOME`*/session_term* for the terminal version.

`textadept.session.MAX_RECENT_FILES = `*`max`*
: Limits the number of recent files saved in session files to *`max`*. The default value is `10`.

`textadept.session.SAVE_ON_QUIT = `*`bool`*
: Save the session when quitting. The default value is `true` unless the user passed the command line switch `-n` or `--nosession` to Textadept.

Move Around

Textadept lets the user navigate within buffers with a high degree of granularity. It also lets the user move between buffers and views. Movements within buffers scroll the caret into view if it is not already visible, while movements between buffers and views do not. These movements are broken down into categories and listed in the following sections.

Move Within Lines

The user can navigate within lines by character, word, and line boundary.

`buffer:char_left()`
`buffer:char_right()`
: Moves the caret left or right one character.

`buffer:word_part_left()`
`buffer:word_part_right()`
: Moves the caret to the previous or next part of the current word. Word parts are delimited by underscore characters or changes in capitalization.

`buffer:word_left_end()`
`buffer:word_right_end()`
: Moves the caret left or right one word, positioning it at the end of the previous or current word, respectively.

`buffer:word_left()`
`buffer:word_right()`
: Moves the caret left or right one word.

`buffer:home()`
`buffer:line_end()`
`buffer:home_display()`
`buffer:line_end_display()`
: Moves the caret to the beginning or end of the current line or current wrapped line, respectively.

`buffer:home_wrap()`
`buffer:line_end_wrap()`
: Moves the caret to the beginning or end of the current wrapped line or, if already there, to the beginning or end of the actual line, respectively.

`buffer:vc_home()`
`buffer:vc_home_display()`
`buffer:vc_home_wrap()`
: Moves the caret to the first visible character on the current line or current wrapped line or, if already there, to the beginning of the current line, current wrapped line, or actual line, respectively.

Move Between Lines

The user can jump to positions and move between lines.

`buffer:goto_pos(`*`pos`*`)`
: Moves the caret to position *`pos`*.

`buffer:goto_line(`*`line`*`)`
: Moves the caret to the beginning of line number *`line`*.

`textadept.editing.goto_line([`*`line`*`])`
: Moves the caret to the beginning of line number *`line`* or the user-specified line, ensuring *`line`* is visible.

`buffer:line_up()`
`buffer:line_down()`
: Moves the caret up or down one line.

Move Between Pages

The user can maneuver between parts of pages and full pages.

`buffer:stuttered_page_up()`
`buffer:stuttered_page_down()`
: Moves the caret to the top or bottom of the page or, if already there, up or down one page, respectively.

`buffer:page_up()`
`buffer:page_down()`
: Moves the caret up or down one page.

Move Between Buffers and Views

The user can cycle through buffers in a single view and also move between split views.

`view:goto_buffer(`*`n`*`[, `*`relative`*`])`
: Switches to buffer number *`n`* in the view. *`relative`* indicates whether or not *`n`* is an index relative to the current buffer's index in `_BUFFERS` instead of an absolute index.

`ui.switch_buffer()`
: Prompts the user to select a buffer to switch to.

`ui.goto_view(`*`n`*`[, `*`relative`*`])`
: Shifts to view number *`n`*. *`relative`* indicates whether or not *`n`* is an index relative to the current view's index in `_VIEWS` instead of an absolute index.

`ui.goto_file(`*`filename`*`[, `*`split`*`[, `*`preferred_view`*`[, `*`sloppy`*`]]])`
: Switches to the existing view whose buffer's filename is *`filename`*. If no view was found and *`split`* is `true`, splits the current view in order to show the requested file. If *`split`* is `false`, shifts to the next or *`preferred_view`* view in order to show the requested file. If *`sloppy`* is `true`, requires only the last part of *`filename`* to match a buffer's filename.

Other Movements

The user can move between matching braces, whole paragraphs, bookmarks, and more.

`textadept.editing.match_brace()`
: Moves the caret to the current brace character's matching brace.

`buffer:para_up()`
`buffer:para_down()`
: Moves the caret up or down one paragraph.

`textadept.bookmarks.goto_mark([`*`next`*`])`
: Moves the caret to the beginning of the next or previously bookmarked line, based on boolean *`next`*. If *`next`* is not given, prompts the user to select a bookmarked line to move the caret to the beginning of.

`buffer:move_caret_inside_view()`
: Moves the caret into view if it is not already, removing any selections.

`buffer:document_start()`
`buffer:document_end()`
: Moves the caret to the beginning or end of the buffer.

Handle Movement Events

The user can connect to the following buffer and view switch events.

`events.BUFFER_BEFORE_SWITCH`
`events.BUFFER_AFTER_SWITCH`
: Emitted right before or after switching to another buffer.

`events.VIEW_BEFORE_SWITCH`
`events.VIEW_AFTER_SWITCH`
: Emitted right before or after switching to another view.

Manipulate Text

Textadept possesses a wide array of powerful text manipulation capabilities, which are broken down into categories and listed in the following sections.

Retrieve Text

The user can request arbitrary ranges of text, the contents of lines, and even individual characters.

`buffer:get_text()`
: Returns the buffer's text.

`buffer:get_sel_text()`
: Returns the selected text. Multiple selections are included in order with no delimiters. Rectangular selections are included from top to bottom with end of line characters. Virtual space is not included.

`buffer:text_range(`*`start_pos`*`, `*`end_pos`*`)`
: Returns the range of text between positions *`start_pos`* and *`end_pos`*.

`buffer:get_line(`*`line`*`)`
: Returns the text on line number *`line`*, including end of line characters.

`buffer:get_cur_line()`
: Returns the current line's text and the caret's position on that line.

`buffer.char_at[`*`pos`*`]` (Read-only)
: The character byte at position *`pos`*.

Set Text

The user can add text at arbitrary buffer positions, insert special characters, and print messages to specific buffers.

`buffer:set_text(`*`text`*`)`
: Replaces the buffer's text with string *`text`*.

`buffer:add_text(`*`text`*`)`
: Adds string *`text`* to the buffer at the caret position and moves the caret to the end of the added text without scrolling it into view.

`buffer:insert_text(`*`pos`*`, `*`text`*`)`
: Inserts string *`text`* at position *`pos`*, removing any selections. If *`pos`* is `-1`, inserts *`text`* at the caret position.

`buffer:append_text(`*`text`*`)`
: Appends string *`text`* to the end of the buffer without modifying any existing selections or scrolling *`text`* into view.

`buffer:line_duplicate()`
: Duplicates the current line on a new line below.

`buffer:selection_duplicate()`
: Duplicates the selected text to its right. If no text is selected, duplicates the current line on a new line below.

`buffer:new_line()`
: Types a new line at the caret position according to `buffer.eol_mode`.

`ui.print(...)`
: Prints the given string messages to the message buffer.

`ui._print(`*`buffer_type`*`, ...)`
: Prints the given string messages to the buffer of type *`buffer_type`*.

Insert Snippets

The user can insert text *snippets* into source code and plain text. The section "Configure Snippets" on page 107 describes snippets in more detail.

`textadept.snippets._insert([text])`
: Inserts snippet text *`text`* or the snippet assigned to the trigger word behind the caret. Otherwise, if a snippet is active, goes to the active snippet's next placeholder. Returns `false` if no action was taken. The section "Configure Snippets" on page 107 describes snippet text.

`textadept.snippets._previous()`
: Jumps back to the previous snippet placeholder, reverting any changes from the current one. Returns `false` if no snippet is active.

`textadept.snippets._cancel_current()`
: Cancels the active snippet, removing all inserted text.

`textadept.snippets._select()`
: Prompts the user to select a snippet to be inserted from a list of global and language-specific snippets.

Replace Text

The user can replace the selected text, as well as replace an arbitrary *target range* of text. A target range is a user-defined region of text that some buffer functions operate on.

`buffer:replace_sel(`*`text`*`)`
: Replaces the selected text with string *`text`*, scrolling the caret into view.

`buffer.target_start = `*`start_pos`*
`buffer.target_end = `*`end_pos`*
: Defines the target range's beginning and end positions as *`start_pos`* and *`end_pos`*, respectively.

`buffer:target_from_selection()`
: Defines the target range's beginning and end positions as the main selection's beginning and end positions, respectively.

`buffer:replace_target(`*`text`*`)`
: Replaces the text in the target range with string *`text`* sans modifying any selections or scrolling the view.

Typing Text

Textadept automatically inserts the complement character of

any user-typed opening brace or quote character, and allows the user to subsequently type over that complement. The user's *~/.textadept/init.lua* file may configure or disable this behavior.

`textadept.editing.AUTOPAIR` = *`bool`*
: Automatically close opening brace and quote characters with their complements. The default value is `true`.

`textadept.editing.char_matches[`*`byte`*`]` = *`char`*
`textadept.editing.char_matches.`*`lexer`*`[`*`byte`*`]` = *`char`*
: Auto-pairs character *`char`* with the character represented by byte value *`byte`* globally or in language *`lexer`*. The default mappings are `34` ('"') with `'"'`, `39` (''') with `"'"`, `40` ('(') with `')'`, `91` ('[') with `']'`, and `123` ('{') with `'}'`.

`textadept.editing.TYPEOVER_CHARS` = *`bool`*
: Move over closing brace and quote characters under the caret when typing them. The default value is `true`.

`textadept.editing.typeover_chars[`*`byte`*`]` = `true`
`textadept.editing.typeover_chars.`*`lexer`*`[`*`byte`*`]` = `true`
: Moves over the character represented by byte value *`byte`* when it is typed globally or in language *`lexer`*. The default byte values are `34` ('"'), `39` (''') `41` (')'), `93` (']'), and `125` ('}').

Delete Text

The user can delete arbitrary ranges of text, as well as delete individual characters, words, and lines.

`buffer:delete_range(`*`pos`*`, `*`length`*`)`
: Deletes the range of text from position *`pos`* to *`pos`* + *`length`*.

`buffer:delete_back()`
: Deletes the character behind the caret if no text is selected. Otherwise, deletes the selected text.

`buffer:delete_back_not_line()`
: Deletes the character behind the caret unless either the caret is at the beginning of a line or text is selected. If text is selected, deletes it.

`buffer:del_word_left()`
`buffer:del_word_right()`
: Deletes the word to the left or right of the caret, including any leading or trailing non-word characters, respectively.

`buffer:del_word_right_end()`
: Deletes the word to the right of the caret, excluding any trailing non-word characters.

`buffer:del_line_left()`
`buffer:del_line_right()`
: Deletes the range of text from the caret to the beginning or end of the current line.

`buffer:line_delete()`
: Deletes the current line.

`buffer:clear_all()`
: Deletes the buffer's text.

Transform Text

The user can change line indentation, transpose characters and lines, and perform useful transformations on selected text.

`buffer:tab()`
`buffer:back_tab()`
: Indents or un-indents the text on the selected lines.

`buffer.line_indentation[`*`line`*`] = `*`width`*
: Changes the amount of indentation on line number *`line`* to *`width`* character columns.

`textadept.editing.transpose_chars()`
: Swaps the current character with the previous one or, if the caret is at the end of a line, switches the two characters before the caret.

`buffer:line_transpose()`
: Swaps the current line with the previous one.

`buffer:upper_case()`
`buffer:lower_case()`
: Converts the selected text to upper case or lower case letters.

`textadept.editing.enclose(`*`left`*`, `*`right`*`)`
: Encloses the selected text or the current word within strings *`left`* and *`right`*.

`textadept.editing.filter_through(`*`command`*`)`
: Passes the selected text or all buffer text to string shell command *`command`* as standard input and replaces the input text with the command's standard output. If the selected text spans multiple lines, all text on the lines that have text selected is passed as standard input.

`buffer:move_selected_lines_up()`
`buffer:move_selected_lines_down()`
: Shifts the selected lines up or down one line.

Split and Join Lines

The user can split and join lines using a target range (a user-defined region of text that some buffer functions operate on).

`buffer.target_start = `*`start_pos`*
`buffer.target_end = `*`end_pos`*
: Defines the target range's beginning and end positions as *`start_pos`* and *`end_pos`*, respectively.

`buffer:target_from_selection()`
: Defines the target range's beginning and end positions as the main selection's beginning and end positions, respectively.

`buffer:lines_split(`*`width`*`)`
: Splits the lines in the target range into lines *`width`* pixels wide. If *`width`* is `0`, splits the lines in the target range into lines as wide as the view.

  The terminal version's unit of measure is a character instead of a pixel.

`buffer:lines_join()`
: Joins the lines in the target range, inserting spaces between the words joined at line boundaries.

`textadept.editing.join_lines()`
: Similar to `buffer:lines_join()`, but ignores the target range and joins the currently selected lines or the current line with the line below it.

Block Comment Code

The user can comment and uncomment source code.

`textadept.editing.block_comment()`
: Comments or uncomments the selected lines based on the current language.

`textadept.editing.comment_string.`*`lexer`* `=` *`comment`*
: Defines the line comment prefix or block comment delimiters for language *`lexer`* as string *`comment`*. Block comment delimiters are separated by a '|' character.

Convert Text Between Encodings

The user can convert arbitrary text between encodings. Textadept's user interface uses the UTF-8 character encoding. While the editor takes care of most conversions automatically, the user may need to perform manual conversions when displaying or retrieving user interface text. Example 7 demonstrates some sample UTF-8 text conversions.

Example 7. Converting to and from UTF-8

```
-- Display a non-UTF-8 filename.
local filename = buffer.filename
local utf8_filename = filename:iconv('UTF-8', _CHARSET)
ui.statusbar_text = 'Opened file '..utf8_filename

-- Open a user-specified filename.
local utf8_name = ui.dialogs.inputbox{
  informative_text = 'Input file:'
}
local f = io.open(utf8_name:iconv(_CHARSET, 'UTF-8'))
if f then ... end
```

`string.iconv(`*`text`*`,` *`new`*`,` *`old`*`)`
: Converts string *`text`* from encoding *`old`* to encoding *`new`* using iconv, returning the string result. Table 1 on page 14 lists all available encodings.

Undo and Redo

The user can undo and redo buffer edits, as well as define a set of actions as a single action that is undone or redone.

`buffer:can_undo()`
`buffer:can_redo()`
: Returns whether or not there is an action to be undone or redone.

`buffer:undo()`
`buffer:redo()`
: Undoes the most recent action or redoes the next undone action.

`buffer:begin_undo_action()`
`buffer:end_undo_action()`
: Starts or ends a sequence of actions to be undone or redone as a single action.

`buffer:empty_undo_buffer()`
: Deletes the undo and redo history.

Employ the Clipboard

The user can cut, copy, and paste text using the system clipboard. Additionally, he or she can write arbitrary text to it.

NOTE

The terminal version cannot access the system clipboard. Instead, it uses its own internal one.

`buffer:cut()`
`buffer:copy()`
: Cuts or copies the selected text to the clipboard. Multiple selections are copied in order with no delimiters. Rectangular selections are copied from top to bottom with end of line characters. Virtual space is not copied.

`buffer:copy_allow_line()`
: Copies the selected text or the current line to the clipboard.

`buffer:line_cut()`
`buffer:line_copy()`
: Cuts or copies the current line to the clipboard.

`buffer:copy_range(`*`start_pos`*`, `*`end_pos`*`)`
: Copies the range of text between positions *`start_pos`* and *`end_pos`* to the clipboard.

`buffer:copy_text(`*`text`*`)`
: Copies string *`text`* to the clipboard.

`buffer.multi_paste = `*`mode`*
: Changes the multiple selection paste mode to *`mode`*. The default value is `buffer.MULTIPASTE_ONCE`, which inserts pasted text into the main selection only. `buffer.MULTIPASTE_EACH` pastes text into all selections.

`buffer:paste()`
: Pastes the clipboard's contents into the buffer, replacing any selected text according to `buffer.multi_paste`.

Query the Clipboard

The user can fetch the current clipboard text.

`ui.clipboard_text`
: The text on the clipboard.

Handle Text Events

The user can connect to the following text typed events.

`events.CHAR_ADDED `*`byte`*
: Emitted after the user types a text character into the buffer. *`byte`* is the text character's byte.

`events.SAVE_POINT_REACHED`
`events.SAVE_POINT_LEFT`
: Emitted after reaching or leaving a save point.

Select Text

Textadept lets the user create and modify single, multiple, and rectangular selections. Textadept mirrors any typed text at each additional selection. Additional selections on multiple lines allow the user to type on multiple lines. Similarly, a rectangular selection spanning multiple lines enables typing on each line. The following sections list many of Textadept's facilities for creating, modifying, and querying selections.

Make Simple Selections

The user can select arbitrary ranges of text, select the current word, line, paragraph, or block, and select text between various entities.

`buffer:set_sel(`*`start_pos`*`, `*`end_pos`*`)`
: Selects the range of text between positions *`start_pos`* and *`end_pos`*, scrolling the selected text into view.

`buffer.anchor = `*`start_pos`*
`buffer.current_pos = `*`end_pos`*
`buffer.selection_start = `*`start_pos`*
`buffer.selection_end = `*`end_pos`*
: Similar to `buffer:set_sel()`, but does not scroll the selected text into view.

`buffer:swap_main_anchor_caret()`
: Swaps the main selection's beginning and end positions.

`textadept.editing.select_word()`
`textadept.editing.select_line()`
`textadept.editing.select_paragraph()`
: Selects the current word, line, or paragraph.

`textadept.editing.select_indented_block()`
: Selects the surrounding block of text whose lines' indentation levels are greater than or equal to the current line's level. If a text block is selected and the lines immediately above and below it are one indentation level lower, adds those lines to the selection.

`textadept.editing.match_brace(true)`
: Selects the range of text between the current brace character and its matching brace.

`textadept.editing.select_enclosed(`*`left`*`, `*`right`*`)`
: Selects the range of text between strings *`left`* and *`right`* that enclose the caret. If that range is already selected, adds *`left`* and *`right`* to the selection.

`buffer:select_all()`
: Selects all of the buffer's text without scrolling the view.

`buffer:set_empty_selection(`*`pos`*`)`
: Moves the caret to position *`pos`* without scrolling the view and removes any selections.

`buffer:clear_selections()`
: Removes all selections and moves the caret to the beginning of the buffer.

Make Movement Selections

The user can create or extend selections while moving the caret.

`buffer:char_left_extend()`
`buffer:char_right_extend()`
: Moves the caret left or right one character, extending the selected text to the new position.

`buffer:word_part_left_extend()`
`buffer:word_part_right_extend()`
: Moves the caret to the previous or next part of the current word, extending the selected text to the new position. Word parts are delimited by underscore characters or changes in capitalization.

`buffer:word_left_extend()`
`buffer:word_right_extend()`
: Moves the caret left or right one word, extending the selected text to the new position.

`buffer:word_left_end_extend()`
`buffer:word_right_end_extend()`
: Moves the caret left or right one word, positioning it at the end of the previous or current word, respectively, and extends the selected text to the new position.

`buffer:home_extend()`
`buffer:line_end_extend()`
`buffer:home_display_extend()`
`buffer:line_end_display_extend()`
: Moves the caret to the beginning or end of the current line or to the beginning or end of the current wrapped line, respectively, and extends the selected text to the new position.

`buffer:home_wrap_extend()`
`buffer:line_end_wrap_extend()`
: Moves the caret to the beginning or end of the current wrapped line or, if already there, to the beginning or end of the actual line, respectively, and extends the selected text to the new position.

`buffer:vc_home_extend()`
`buffer:vc_home_display_extend()`
`buffer:vc_home_wrap_extend()`
: Moves the caret to the first visible character on the current line or current wrapped line or, if already there, to the beginning of the current line, current wrapped line, or actual line, respectively, and extends the selected text to the new position.

`buffer:line_up_extend()`
`buffer:line_down_extend()`
: Moves the caret up or down one line, extending the selected text to the new position.

`buffer:para_up_extend()`
`buffer:para_down_extend()`
: Moves the caret up or down one paragraph, extending the selected text to the new position.

`buffer:stuttered_page_up_extend()`
`buffer:stuttered_page_down_extend()`
: Moves the caret to the top or bottom of the page or, if already there, up or down one page, respectively, and extends the selected text to the new position.

`buffer:page_up_extend()`
`buffer:page_down_extend()`
: Moves the caret up or down one page, extending the selected text to the new position.

`buffer:document_start_extend()`
`buffer:document_end_extend()`
: Moves the caret to the beginning or end of the buffer, extending the selected text to the new position.

`textadept.editing.match_brace(true)`
: Moves the caret to the current brace character's matching brace, extending the selected text to the new position.

Modal Selection

The user can create or extend selections with caret movements by enabling modal selection.

`buffer.selection_mode = mode`
: Changes the selection mode to *mode*. When set, caret movement alters the selected text until this field is set

again to the same value or until `buffer:cancel()` is called. When *`mode`* is `buffer.SEL_STREAM`, caret movement selects characters. `buffer.SEL_RECTANGLE` allows rectangular selections, `buffer.SEL_LINES` allows selections by line, and `buffer.SEL_THIN` allows thin rectangular selections.

Make Multiple Selections

The user can construct multiple selections and cycle between them.

`buffer:set_selection(`*`end_pos`*`, `*`start_pos`*`)`
: Selects the range of text between positions *`start_pos`* and *`end_pos`*, removing all other selections.

`buffer:add_selection(`*`end_pos`*`, `*`start_pos`*`)`
: Selects the range of text between positions *`start_pos`* and *`end_pos`* as the main selection, retaining all other selections as additional selections. Since an empty selection still counts as a selection, use `buffer:set_selection()` first when setting a list of selections.

`textadept.editing.select_word()`
: Selects the selected word's next occurrence as a multiple selection.

`buffer.main_selection = `*`n`*
: Indicates selection number *`n`* is the main selection.

`buffer:rotate_selection()`
: Designates the next additional selection to be the main selection.

Modify Multiple Selections

The user can alter multiple selections by manipulating their boundaries.

`buffer.anchor = `*`start_pos`*
`buffer.current_pos = `*`end_pos`*
`buffer.selection_start = `*`start_pos`*
`buffer.selection_end = `*`end_pos`*
: Changes the beginning and end positions of text selected by the main selection to *`start_pos`* and *`end_pos`*, respectively.

`buffer.selection_n_anchor[n] = start_pos`
`buffer.selection_n_caret[n] = end_pos`
`buffer.selection_n_start[n] = start_pos`
`buffer.selection_n_end[n] = end_pos`

Changes the beginning and end positions of text selected by selection number *n* to *start_pos* and *end_pos*, respectively.

`buffer.selection_n_anchor_virtual_space[n] = start_pos`
`buffer.selection_n_caret_virtual_space[n] = end_pos`

Changes the beginning and end positions of virtual space selected by selection number *n* to *start_pos* and *end_pos*, respectively.

WARNING

The `buffer.selection_n_*` fields may only be used on existing selections created by `buffer:set_selection()` or `buffer:add_selection()`. Using those fields in an attempt to create selections will crash Textadept.

Make Rectangular Selections

The user can create and modify rectangular selections through boundary manipulation or caret movement.

`buffer.rectangular_selection_modifier = modifier`

Changes the modifier key used in combination with a mouse drag in order to create a rectangular selection to *modifier*. The default value is `buffer.MOD_ALT` on non-Linux platforms and `buffer.MOD_SUPER` (usually the left Windows or Command key) on Linux platforms. `buffer.MOD_CTRL` is used to create multiple selections.

The terminal version does not support the mouse.

`buffer.rectangular_selection_anchor = start_pos`
`buffer.rectangular_selection_caret = end_pos`

Defines the rectangular selection's beginning and end positions as *start_pos* and *end_pos*, respectively.

`buffer.rectangular_selection_anchor_virtual_space = s`
`buffer.rectangular_selection_caret_virtual_space = e`

Defines the beginning and end positions of virtual space in the rectangular selection as *s* and *e*, respectively.

`buffer:char_left_rect_extend()`
`buffer:char_right_rect_extend()`
: Moves the caret left or right one character, extending the rectangular selection to the new position.

`buffer:home_rect_extend()`
`buffer:line_end_rect_extend()`
: Moves the caret to the beginning or end of the current line, extending the rectangular selection to the new position.

`buffer:vc_home_rect_extend()`
: Moves the caret to the first visible character on the current line or, if already there, to the beginning of the current line, and extends the rectangular selection to the new position.

`buffer:line_up_rect_extend()`
`buffer:line_down_rect_extend()`
: Moves the caret up or down one line, extending the rectangular selection to the new position.

`buffer:page_up_rect_extend()`
`buffer:page_down_rect_extend()`
: Moves the caret up or down one page, extending the rectangular selection to the new position.

Query Selection Information

The user can obtain information related to currently selected text. This information is broken down into categories and listed in the following sections.

Basic Selection Information

The user can acquire the currently selected text's beginning and end positions along with its properties.

`buffer.selection_start`
`buffer.selection_end`
: The selection's beginning and end positions.

`buffer:get_line_sel_start_position(`*`line`*`)`
`buffer:get_line_sel_end_position(`*`line`*`)`
: Returns the beginning or end position of the selected text on line number *`line`*, or `-1` if *`line`* has no selection.

`buffer.selection_is_rectangle` (Read-only)
Whether or not the selection is a rectangular selection.

`buffer.selection_empty` (Read-only)
Whether or not no text is selected.

Multiple Selection Information

The user can request the current number of multiple selections along with the beginning and end positions of each one.

`buffer.selections` (Read-only)
The number of active selections.

`buffer.main_selection`
The number of the main, or most recent, selection.

`buffer.selection_n_anchor[`*n*`]`
`buffer.selection_n_caret[`*n*`]`
`buffer.selection_n_start[`*n*`]`
`buffer.selection_n_end[`*n*`]`
The beginning and end positions of text selected by selection number *n*.

`buffer.selection_n_anchor_virtual_space[`*n*`]`
`buffer.selection_n_caret_virtual_space[`*n*`]`
The beginning and end positions of virtual space selected by selection number *n*.

Rectangular Selection Information

The user can retrieve the rectangular selection's beginning and end positions.

`buffer.rectangular_selection_anchor`
`buffer.rectangular_selection_caret`
The rectangular selection's beginning and end positions.

`buffer.rectangular_selection_anchor_virtual_space`
`buffer.rectangular_selection_caret_virtual_space`
The virtual space's beginning and end positions in the rectangular selection.

Search for Text

Textadept supplies a variety of tools to search for text: a simple search API, a more complex API for search and replace, a Find & Replace Pane for interactive search and replace, and an incremental find entry. The first two tools make use of the search flags defined in Table 2 and the regular expression syntax in Table 3. The last two tools make use of the search flags defined in the section "Interact with the Find & Replace Pane" on page 40 and the Lua pattern syntax in Table 3.

NOTE

Due to size concerns, Textadept's regular expressions are a smaller, limited subset of general regular expressions. Regex searches are only available through the API, while Lua patterns are available in the Find & Replace Pane.

Table 2. Buffer search flags

Bit Flag	Description
`buffer.FIND_MATCHCASE`	Match search text case sensitively.
`buffer.FIND_WHOLEWORD`	Match search text only when it is surrounded by non-word characters.
`buffer.FIND_WORDSTART`	Match search text only when the previous character is a non-word character.
`buffer.FIND_REGEXP`	Interpret search text as a regular expression. (See Table 3.)

Table 3. Regular expression and Lua pattern special characters

Regex	Lua	Meaning
.	.	Matches any character.
	%a	Matches any letter.
	%c	Matches any control character.
	%d	Matches any digit.

Regex	Lua	Meaning
	%g	Matches any printable character except space.
	%l	Matches any lower case character.
	%p	Matches any punctuation character.
	%s	Matches any space character.
	%u	Matches any upper case character.
	%w	Matches any alphanumeric character.
	%x	Matches any hexadecimal digit.
[*set*]	[*set*]	Matches any character in *set*, including ranges (e.g. `[A-Za-z]`).
[^*set*]	[^*set*]	Matches the complement of *set*.
*	*	Matches the previous *character class* zero or more times. The previous expressions are character classes.
+	+	Matches the previous class one or more times.
	-	Matches the previous class zero or more times, but as few times as possible.
	?	Matches the previous class once, or not at all.
	%b*xy*	Matches a balanced string that starts with *x* and ends with *y*.
	%f[*set*]	Matches a position where the next character belongs to *set*, but the previous character does not.
\<		Matches the beginning of a word.
\>		Matches the end of a word.
^	^	Matches the beginning of a line unless inside a set.
$	$	Matches the end of a line unless inside a set.
(	(	The beginning of a captured matching region.
)	)	The end of a captured matching region.
n	%*n*	Represents the *n*th captured matching region's text. In replacement text, "\0" and "%0" represent all matched text.
x	%*x*	Represents non-alphanumeric character *x*, ignoring any special meaning it may have by itself.

Simple Search

The user can perform simple searches anchored at the caret position.

`buffer:search_anchor()`
: Anchors the position that `buffer:search_next()` and `buffer:search_prev()` start at to the caret position.

`buffer:search_next(`*`flags`*`, `*`text`*`)`
`buffer:search_prev(`*`flags`*`, `*`text`*`)`
: Searches for and selects the next or previous occurrence of string *`text`* using search flags bit-mask *`flags`*, returning that occurrence's position or `-1` if *`text`* was not found. Selected text is not scrolled into view. *`flags`* is an additive combination of the flags listed in Table 2.

Search and Replace

The user can execute search and replace within a target range of text (a user-defined region of text that some buffer functions operate on).

`buffer.search_flags = `*`flags`*
: Specifies search flags bit-mask *`flags`* as the search flags used by `buffer:search_in_target()`. *`flags`* is an additive combination of the flags listed in Table 2.

`buffer.target_start = `*`start_pos`*
`buffer.target_end = `*`end_pos`*
: Defines the target range's beginning and end positions as *`start_pos`* and *`end_pos`*, respectively. These fields are also set by a successful `buffer:search_in_target()`.

`buffer:target_from_selection()`
: Defines the target range's beginning and end positions as the main selection's beginning and end positions, respectively.

`buffer:search_in_target(`*`text`*`)`
: Searches for the first occurrence of string *`text`* in the target range using search flags bit-mask `buffer.search_flags` and, if found, sets the new target range to that occurrence, returning its position or `-1` if *`text`* was not found.

`buffer:replace_target(`*`text`*`)`
: Replaces the text in the target range with string *`text`* sans modifying any selections or scrolling the view.

`buffer:replace_target_re(`*`text`*`)`
: Similar to `buffer:replace_target()`, but first replaces any "*n*" sequences in string *`text`* with the text of capture number *`n`* from the regular expression (or the entire match for *`n`* `= 0`), and then returns the replacement text's length.

Query Regex Search Captures

The user can fetch the captures from a regular expression search.

`buffer.tag[`*`n`*`]` (Read-only)
: The text of capture number *`n`* from a regular expression search.

Interact with the Find & Replace Pane

The user can summon the Find & Replace Pane, specify search and replacement text, select search flags, and perform search and replace operations.

`ui.find.focus()`
: Displays and focuses the Find & Replace Pane.

`ui.find.find_entry_text = `*`text`*
`ui.find.replace_entry_text = `*`text`*
: Places string *`text`* in the "Find" or "Replace" entry. Replacement text only recognizes Lua captures from a Lua pattern search, but always allows embedded Lua code enclosed within "`%()`".

`ui.find.match_case = `*`bool`*
: Match search text case sensitively. The default value is `false`.

`ui.find.whole_word = `*`bool`*
: Match search text only when it is surrounded by non-word characters. The default value is `false`.

`ui.find.lua = `*`bool`*
: Interpret search text as a Lua pattern. The default value

is false. Table 3 lists Lua pattern syntax.

`ui.find.in_files = bool`
: Find search text in a list of files. The default value is false.

`ui.find.find_next()`
`ui.find.find_prev()`
`ui.find.replace()`
`ui.find.replace_all()`
: Mimics pressing the "Find Next", "Find Prev", "Replace", or "Replace All" button. Calling `ui.find.replace_all()` with text selected performs "Replace All" only within the selection.

`ui.find.FILTER = filter`
: Specifies file filter *filter* as the filter for `ui.find.find_in_files()`. Files searched do not match any pattern in *filter*. The section "File Filters" on page 14 describes file filters.

`ui.find.find_in_files([dir])`
: Searches directory *dir* or the user-specified directory for files that match search text and search options, and prints the results to a buffer titled "Files Found".

WARNING

While the `ui.find.FILTER` file filter excludes many common binary files and version control directories from searches, `ui.find.find_in_files()` could still scan unrecognized binary files or large, unwanted sub-directories. Searches also block Textadept from receiving additional input, making its interface temporarily unresponsive.

`ui.find.goto_file_found(line[, next])`
: Jumps to the source of the find in files search result on line number *line* in the buffer titled "Files Found" or, if *line* is nil, jumps to the next or previous search result, depending on boolean *next*.

Incremental Search

The user can execute an incremental search, which searches the buffer while he or she types.

`ui.find.find_incremental(`*`text`*`, `*`next`*`[, `*`anchor`*`])`
: Begins an incremental search using the command entry if *`text`* is `nil`. Otherwise, continues an incremental search by searching for the next or previous instance of string *`text`*, depending on boolean *`next`*. *`anchor`* indicates whether or not to search for *`text`* starting from the caret position instead of the position where the incremental search began. Only the `ui.find.match_case` find option is recognized.

`ui.find.find_incremental_next()`
`ui.find.find_incremental_prev()`
: Continues an incremental search by searching for the next or previous match starting from the caret position.

Handle Find & Replace Events

The user can connect to the following find and replace events.

`events.FIND` *`text`*`, `*`next`*
: Emitted to find text via the Find & Replace Pane. *`text`* is the text to search for and *`next`* indicates whether or not to search forward.

`events.FIND_WRAPPED`
: Emitted when a text search wraps, either from bottom to top (when searching for a next occurrence), or from top to bottom (when searching for a previous occurrence).

`events.REPLACE` *`text`*
: Emitted to replace selected (found) text. *`text`* is the replacement text.

`events.REPLACE_ALL` *`find_text`*`, `*`repl_text`*
: Emitted to replace all occurrences of found text. *`find_text`* is the text to search for and *`repl_text`* is the replacement text.

Query Buffer Information

Textadept provides a wide variety of generic buffer information, which is broken down into categories and listed in the following sections.

Query Position Information

The user can obtain positional information from several sources.

`buffer.current_pos`
`buffer.anchor`
: The caret and anchor positions.

`buffer:position_before(`*`pos`*`)`
: Returns the position of the character before position *`pos`* (taking multi-byte characters into account), or `0` if there is no character before *`pos`*.

`buffer:position_after(`*`pos`*`)`
: Returns the position of the character after position *`pos`* (taking multi-byte characters into account), or `buffer.length` if there is no character after *`pos`*.

`buffer:word_start_position(`*`pos`*`, `*`only_word_chars`*`)`
: Returns the position of the beginning of the word at position *`pos`*. If *`pos`* has a non-word character to its left and *`only_word_chars`* is `false`, returns the last word character's position.

`buffer:word_end_position(`*`pos`*`, `*`only_word_chars`*`)`
: Returns the position of the end of the word at position *`pos`*. If *`pos`* has a non-word character to its right and *`only_word_chars`* is `false`, returns the first word character's position.

`buffer:position_from_line(`*`line`*`)`
: Returns the position at the beginning of line number *`line`*.

`buffer.line_indent_position[`*`line`*`]` (Read-only)
: The position at the end of indentation on line number *`line`*.

`buffer.line_end_position[`*`line`*`]` (Read-only)
: The position at the end of line number *`line`*, but before any end of line characters.

`buffer:find_column(`*`line`*`, `*`column`*`)`
: Returns the position of column number *`column`* on line number *`line`* (taking tab and multi-byte characters into account).

Query Line and Line Number Information

The user can fetch line information and line numbers from a number of sources.

`buffer.line_count` (Read-only)
: The number of lines in the buffer.

`buffer.lines_on_screen` (Read-only)
: The number of completely visible lines in the view.

`buffer.first_visible_line`
: The line number of the line at the top of the view.

`buffer:line_from_position(`*`pos`*`)`
: Returns the line number of the line that contains position *`pos`*.

`buffer.line_indentation[`*`line`*`]`
: The number of columns of indentation on line number *`line`*.

`buffer:line_length(`*`line`*`)`
: Returns the number of bytes on line number *`line`*, including end of line characters.

`buffer:wrap_count(`*`line`*`)`
: Returns the number of wrapped lines needed to fully display line number *`line`*.

`buffer:visible_from_doc_line(`*`line`*`)`
: Returns the displayed line number of actual line number *`line`*, taking hidden lines into account.

`buffer:doc_line_from_visible(`*`display_line`*`)`
: Returns the actual line number of displayed line number *`display_line`*, taking hidden lines into account.

Query Measurement Information

The user can acquire various measurements like text length, width, and height.

`buffer.length` (Read-only)
`buffer.text_length` (Read-only)
: The number of bytes in the buffer.

`buffer.column[pos]` (Read-only)
: The column number (taking tab widths into account) for position *pos*. Multi-byte characters count as single characters.

`buffer:count_characters(start_pos, end_pos)`
: Returns the number of whole characters (taking multi-byte characters into account) between positions *start_pos* and *end_pos*.

`buffer:text_width(style, text)`
: Returns the pixel width string *text* would have when styled with style number *style*.

  The terminal version's unit of measure is a character instead of a pixel.

`buffer:text_height(line)`
: Returns the pixel height of line number *line*.

  The terminal version's unit of measure is a character instead of a pixel.

Configure Line Margins

Textadept displays up to 5 different left-hand margins, numbered from 0 to 4. Each margin displays either line numbers, marker symbols, or text. The user's *~/.textadept/properties.lua* file may configure how Textadept displays line margins.

`buffer.margin_type_n[n] = type`
: Assigns margin type *type* to margin number *n*. The margin type `buffer.MARGIN_NUMBER` displays line numbers. `buffer.MARGIN_SYMBOL`, `buffer.MARGIN_BACK`, and `buffer.MARGIN_FORE` all display marker symbols, but the latter two have background and foreground colors that match the default text background and foreground colors, respectively. `buffer.MARGIN_TEXT` and `buffer.MARGIN_RTEXT` display left-justified and right-justified text, respectively. The default value for *n* = `0` is `buffer.MARGIN_NUMBER`, while for all other *n* it is `buffer.MARGIN_SYMBOL`.

`buffer.margin_width_n[n] = width`
: Fixes the pixel margin width of margin number *n* at *width*. The default value for *n* = `0` depends on the current

font size. The default value for *n* = 1 is 4. For *n* = 2, it is 12. For all other *n*, the default value is 0.

The terminal version's unit of measure is a character instead of a pixel. The default width values for *n* = 0, *n* = 1, and *n* = 2 are 3, 1, and 1, respectively.

`buffer.margin_mask_n[`*n*`] = `*mask*
Specifies marker bit-mask *mask* as the set of *markers* whose symbols margin number *n* can display. The section "Mark Lines with Markers" on page 47 describes markers and their symbols in more detail. *mask* is a 32-bit value whose bits correspond to Textadept's 32 markers. Margin *n* must be able to display marker symbols. The default value for *n* = 1 is `bit32.bnot(buffer.MASK_FOLDERS)`, which only displays non-folding markers. For *n* = 2, the default value is `buffer.MASK_FOLDERS`, which only displays folding markers. For all other *n*, the default value is 0.

`buffer.margin_sensitive_n[`*n*`] = `*bool*
Whether or not mouse clicks in margin number *n* emit `events.MARGIN_CLICK` events. The default value for *n* = 2 (the number of the fold margin) is `true`, while for all other *n* it is `false`.

`buffer.margin_cursor_n[`*n*`] = `*type*
Shows mouse cursor type *type* over margin number *n*. The default value for all *n* is `buffer.CURSORREVERSEARROW`. Table 12 on page 97 lists all available cursor types.

The terminal version cannot change the mouse cursor.

`buffer.margin_text[`*line*`] = `*text*
Displays string *text* in text margins on line number *line*.

`buffer.margin_style[`*line*`] = `*style*
Assigns style number *style* to the text in text margins on line number *line*.

`buffer:margin_text_clear_all()`
Clears all text in text margins.

`buffer.margin_options = buffer.MARGINOPTION_SUBLINESELECT`
Selects only a wrapped line's sub-line (rather than the entire line) when the line number margin is clicked.

Query Margin Information

The user can request margin widths and any text in text margins.

`buffer.margin_width_n[`*`n`*`]`
: The pixel width of margin number *`n`*.

  The terminal version's unit of measure is a character instead of a pixel.

`buffer.margin_text[`*`line`*`]`
: The text displayed in text margins on line number *`line`*.

Handle Margin Events

The user can connect to the following margin event.

`events.MARGIN_CLICK` *`margin, position, modifiers`*
: Emitted when clicking the mouse inside a sensitive margin. *`margin`* is the margin number clicked, *`position`* is the beginning position of the clicked margin's line, and *`modifiers`* is a bit-mask of any modifier keys used (`buffer.MOD_CTRL` for Control, `buffer.MOD_SHIFT` for Shift, `buffer.MOD_ALT` for Alt, and `buffer.MOD_META` for Command).

  The terminal version cannot detect mouse clicks.

Mark Lines with Markers

Textadept offers 32 *markers*, numbered from 0 to 31, to mark lines with. Each marker has an assigned symbol that is displayed in properly configured margins. For lines with multiple markers, only the symbol for the marker that has the highest marker number is shown. Tables 4 and 6 list all available marker symbols. The section "Configure Line Margins" on page 45 describes how to set up margins to display marker symbols. Markers move in sync with the lines they were added to as text is inserted and deleted. When a line that has a marker on it is deleted, that marker moves to the previous line. Textadept uses marker numbers 25 to 31 internally for the fold markers listed in Table 5, leaving marker numbers 0 through 24 at the user's disposal.

Table 4. Marker symbols

Marker Symbol	Visual or Description
`buffer.MARK_CIRCLE`	●
`buffer.MARK_SMALLRECT`	■
`buffer.MARK_ROUNDRECT`	A rounded rectangle.
`buffer.MARK_LEFTRECT`	▌
`buffer.MARK_FULLRECT`	▉
`buffer.MARK_SHORTARROW`	A small, right-facing arrow.
`buffer.MARK_ARROW`	►
`buffer.MARK_ARROWS`	›››
`buffer.MARK_DOTDOTDOT`	…
`buffer.MARK_PIXMAP`	An XPM image.
`buffer.MARK_RGBAIMAGE`	An RGBA image.
`buffer.MARK_CHARACTER + i`	The character whose ASCII value is *i*.
`buffer.MARK_EMPTY`	
`buffer.MARK_BACKGROUND`	Changes a line's background color.
`buffer.MARK_UNDERLINE`	Underlines an entire line.

Table 5. Fold marker numbers

Marker Number	Description
`buffer.MARKNUM_FOLDEROPEN`	The first line of an expanded fold.
`buffer.MARKNUM_FOLDERSUB`	A line within an expanded fold.
`buffer.MARKNUM_FOLDERTAIL`	The last line of an expanded fold.
`buffer.MARKNUM_FOLDER`	The first line of a collapsed fold.
`buffer.MARKNUM_FOLDEROPENMID`	The first line of an expanded fold within an expanded fold.
`buffer.MARKNUM_FOLDERMIDTAIL`	The last line of an expanded fold within an expanded fold.
`buffer.MARKNUM_FOLDEREND`	The first line of a collapsed fold within an expanded fold.

Table 6. Fold marker symbols

Fold Marker Symbol	Visual or Description
buffer.MARK_ARROW	►
buffer.MARK_ARROWDOWN	▼
buffer.MARK_MINUS	–
buffer.MARK_BOXMINUS	⊟
buffer.MARK_BOXMINUSCONNECTED	A boxed minus sign connected to a vertical line.
buffer.MARK_CIRCLEMINUS	⊖
buffer.MARK_CIRCLEMINUSCONNECTED	A circled minus sign connected to a vertical line.
buffer.MARK_PLUS	+
buffer.MARK_BOXPLUS	⊞
buffer.MARK_BOXPLUSCONNECTED	A boxed plus sign connected to a vertical line.
buffer.MARK_CIRCLEPLUS	⊕
buffer.MARK_CIRCLEPLUSCONNECTED	A circled plus sign connected to a vertical line.
buffer.MARK_VLINE	|
buffer.MARK_TCORNER	├
buffer.MARK_LCORNER	└
buffer.MARK_TCORNERCURVE	A curved, T-shaped corner.
buffer.MARK_LCORNERCURVE	A curved, L-shaped corner.

`_SCINTILLA.next_marker_number()`
: Returns a unique marker number.

`buffer:marker_define(`*`marker`*`, `*`symbol`*`)`
: Assigns marker symbol *`symbol`* to marker number *`marker`*. *`symbol`* is shown in marker symbol margins next to lines marked with *`marker`*. Tables 4 and 6 list the available marker symbols. The section "Assign Marker Colors" on page 90 describes how to change the color and alpha values of *`marker`*.

The terminal version requires *symbol* to be buffer.MARK_CHARACTER + *i*.

TIP

The user should define markers in either his or her *~/.textadept/properties.lua* file or within an events.VIEW_NEW handler, so subsequent views can recognize them.

buffer:marker_define_pixmap(*marker*, *pixmap*)
: Associates marker number *marker* with XPM image *pixmap*. The buffer.MARK_PIXMAP marker symbol must be assigned to *marker*. The Appendix on page 123 describes the XPM image format.

buffer.rgba_image_width = *width*
buffer.rgba_image_height = *height*
: Indicates that the pixel width and height of the RGBA image to be defined using buffer:marker_define_rgba_image() are *width* and *height*, respectively.

buffer.rgba_image_scale = *factor*
: Indicates that the scale factor percentage of the RGBA image to be defined using buffer:marker_define_rgba_image() is *factor*.

buffer:marker_define_rgba_image(*marker*, *pixels*)
: Associates marker number *marker* with RGBA image *pixels*. The dimensions for *pixels* (buffer.rgba_image_width and buffer.rgba_image_height) must have already been defined. The buffer.MARK_RGBAIMAGE symbol must be assigned to *marker*. The Appendix on page 123 describes the RGBA image format.

buffer:marker_add(*line*, *marker*)
: Adds marker number *marker* to line number *line*, returning the added marker's handle or 0 if *line* is invalid.

buffer:marker_add_set(*line*, *mask*)
: Adds the markers specified in marker bit-mask *mask* to line number *line*. *mask* is a 32-bit value whose bits correspond to Textadept's 32 markers,

buffer:marker_delete_handle(*handle*)
: Deletes the marker with handle *handle*.

`buffer:marker_delete(`*`line`*`, `*`marker`*`)`
: Deletes marker number *`marker`* from line number *`line`*. If *`marker`* is `-1`, deletes all markers from *`line`*.

`buffer:marker_delete_all(`*`marker`*`)`
: Deletes marker number *`marker`* from any line that has it. If *`marker`* is `-1`, deletes all markers from all lines.

Bookmark Lines

The user can toggle bookmarks on individual lines.

`textadept.bookmarks.toggle([`*`on`*`])`
: Toggles the bookmark on the current line unless *`on`* is given. If *`on`* is `true` or `false`, adds or removes the bookmark, respectively.

`textadept.bookmarks.clear()`
: Clears all bookmarks in the current buffer.

Query Marker Information

The user can acquire marker locations, fetch the set of markers on a particular line, and learn how markers were defined.

`buffer:marker_line_from_handle(`*`handle`*`)`
: Returns the line number that marker handle *`handle`* was added to, or `-1` if that line was not found.

`buffer:marker_get(`*`line`*`)`
: Returns a bit-mask that represents the markers that were added to line number *`line`*. The mask is a 32-bit value whose bits correspond to Textadept's 32 markers.

`buffer:marker_next(`*`line`*`, `*`mask`*`)`
`buffer:marker_previous(`*`line`*`, `*`mask`*`)`
: Returns the next or previous line number, starting from line number *`line`*, that has had all of the markers specified by marker bit-mask *`mask`* added to it. Returns `-1` if no line was found. *`mask`* is a 32-bit value whose bits correspond to Textadept's 32 markers.

`buffer:marker_symbol_defined(`*`marker`*`)`
: Returns the symbol assigned to marker number *`marker`*.

Annotate Lines

Textadept allows the user to annotate lines with styled, read-only text displayed underneath them. This may be useful for displaying compiler errors, runtime errors, or other diagnostic information.

`buffer.annotation_text[`*`line`*`] = `*`text`*
: Displays string *`text`* as the annotation text for line number *`line`*.

`buffer.annotation_style[`*`line`*`] = `*`style`*
: Assigns style number *`style`* to the annotation text for line number *`line`*.

`buffer:annotation_clear_all()`
: Clears annotations from all lines.

`buffer.annotation_visible = `*`mode`*
: Changes the annotation visibility mode to *`mode`*. The default value is `buffer.ANNOTATION_BOXED`, which indents annotations to match the annotated text, and outlines them with a box. `buffer.ANNOTATION_STANDARD` draws annotations left-justified with no decoration, while `buffer.ANNOTATION_HIDDEN` hides annotations.

Query Annotated Lines

The user can request a line's annotation text along with the number of lines needed to display that text.

`buffer.annotation_text[`*`line`*`]`
: The annotation text for line number *`line`*.

`buffer.annotation_lines[`*`line`*`]` (Read-only)
: The number of annotation text lines for line number *`line`*.

Mark Text with Indicators

Textadept supplies 32 *indicators*, numbered from 0 to 31, to mark text with. Each indicator has an assigned indicator style from the list in Table 7. The editor displays indicators along with any existing styles text may have.

Table 7. Indicator styles

Indicator Style	Visual or Description
`buffer.INDIC_SQUIGGLEPIXMAP`	A squiggly underline.
`buffer.INDIC_PLAIN`	An underline.
`buffer.INDIC_DASH`	A dashed underline.
`buffer.INDIC_DOTS`	A dotted underline.
`buffer.INDIC_COMPOSITIONTHICK`	A thick underline.
`buffer.INDIC_STRIKEOUT`	~~A strikeout line.~~
`buffer.INDIC_BOX`	A bounding box.
`buffer.INDIC_DOTBOX`	A dotted bounding box.
`buffer.INDIC_STRAIGHTBOX`	A translucent box.
`buffer.INDIC_ROUNDBOX`	A translucent box with rounded corners.
`buffer.INDIC_TT`	An underline of small 'T' shapes.
`buffer.INDIC_DIAGONAL`	An underline of diagonal hatches.
`buffer.INDIC_SQUIGGLELOW`	A squiggly underline two pixels high.
`buffer.INDIC_HIDDEN`	Plain text with no decorations.

`_SCINTILLA.next_indic_number()`
: Returns a unique indicator number.

`buffer.indic_style[`*`indicator`*`] = `*`style`*
: Assigns indicator style *`style`* to indicator number *`indicator`*. Table 7 lists all available indicator styles. The section "Assign Indicator Colors" on page 91 describes how to change the color and alpha values of *`indicator`*.

  The terminal version requires *`style`* to be `buffer.INDIC_STRAIGHTBOX`, but cannot draw it translucently.

TIP

The user should either assign indicator styles in his or her *~/.textadept/properties.lua* file or within an `events.VIEW_NEW` handler, so subsequent views can recognize them.

`buffer.indic_under[`*`indicator`*`] = `*`bool`*
: Draw indicator number *`indicator`* behind text instead of

in front of it. The default value is `false`.

`buffer.indicator_current = indicator`
: Designates indicator number *`indicator`* as the indicator used by `buffer:indicator_fill_range()` and `buffer:indicator_clear_range()`.

`buffer:indicator_fill_range(pos, length)`
`buffer:indicator_clear_range(pos, length)`
: Fills or clears the range of text from position *`pos`* to *`pos`* + *`length`* with indicator number `buffer.indicator_current`.

Highlight Words

The user can highlight all instances of a word (for example, all instances of a variable name). Subsequent calls to `textadept.editing.select_word()` select each occurrence of that word, providing a simple renaming tool.

`textadept.editing.highlight_word()`
: Highlights all occurrences of the selected text or the current word.

Query Indicator Information

The user can fetch indicator locations and retrieve the indicators present at particular positions.

`buffer:indicator_start(indicator, pos)`
`buffer:indicator_end(indicator, pos)`
: Returns the previous or next boundary position, starting from position *`pos`*, of indicator number *`indicator`*. Returns `0` or `buffer.length`, respectively, if *`indicator`* was not found.

`buffer:indicator_all_on_for(pos)`
: Returns a bit-mask that represents the indicators present at position *`pos`*. The mask is a 32-bit value whose bits correspond to Textadept's 32 indicators.

Handle Indicator Events

The user can connect to the following indicator click events.

`events.INDICATOR_CLICK` *`position, modifiers`*
: Emitted when clicking the mouse on text that has an indicator present. *`position`* is the clicked text's position and *`modifiers`* is a bit-mask of any modifier keys used (`buffer.MOD_CTRL` for Control, `buffer.MOD_SHIFT` for Shift, `buffer.MOD_ALT` for Alt, and `buffer.MOD_META` for Command).

  The terminal version cannot detect mouse clicks.

`events.INDICATOR_RELEASE` *`position`*
: Emitted when releasing the mouse after clicking on text that has an indicator present. *`position`* is the clicked text's position.

  The terminal version cannot detect mouse releases.

Show an Interactive List

Textadept has the ability to display two types of interactive lists that update as the user types: an *autocompletion list* and a *user list*. An autocompletion list is a list of completions shown for the current word. A user list is a more general list of options presented to the user. Both types of lists have similar behavior and may display images alongside text. All of the above is described in the following sections.

Display an Autocompletion List

The user must define an autocompletion list's separator character and sorted order before showing the list itself.

`buffer.auto_c_separator =` *`byte`*
: Defines byte value *`byte`* as the character that separates autocompletion list items in the list to be passed to `buffer:auto_c_show()`. The default value is `32` (' ').

`buffer.auto_c_order =` *`mode`*
: Specifies order mode *`mode`* as the order the list to be passed to `buffer:auto_c_show()` is sorted in. The default value is `buffer.ORDER_PRESORTED`, which indicates the list to be passed is already in sorted, alphabetical order. `buffer.ORDER_PERFORMSORT` indicates the list should be sorted in place, while `buffer.ORDER_CUSTOM` indicates the list is already in a custom order.

`buffer:auto_c_show(len_entered, items)`
: Displays an autocompletion list constructed from string *`items`* (whose items are separated by `buffer.auto_c_separator` characters) using *`len_entered`* number of characters behind the caret as the prefix of the word to be autocompleted. The sorted order of *`items`* (`buffer.auto_c_order`) must have already been defined.

`buffer:auto_c_select(prefix)`
: Selects the first item that starts with string *`prefix`* in an autocompletion list, using the case sensitivity setting `buffer.auto_c_ignore_case`.

`buffer:auto_c_complete()`
: Completes the current word with the one selected in an autocompletion list.

`buffer:auto_c_cancel()`
: Cancels an autocompletion list.

`textadept.editing.autocomplete_word([words])`
: Displays an autocompletion list for the word behind the caret, returning `true` if completions were found. The displayed list is built from existing words in the buffer and the set of words in string *`words`*.

Display a User List

The user must define a user list's separator character, sorted order, and identifier number before presenting the list itself. A user list is slightly different than an autocompletion list in that it emits an event instead of inserting the selected item.

`buffer.auto_c_separator = byte`
: Defines byte value *`byte`* as the character that separates user list items in the list to be passed to `buffer:user_list_show()`. The default value is `32` (' ').

`buffer.auto_c_order = mode`
: Specifies list order mode *`mode`* as the order the list to be passed to `buffer:user_list_show()` is sorted in. The default value is `buffer.ORDER_PRESORTED`, which indicates the list to be passed is already in sorted, alphabetical order. `buffer.ORDER_PERFORMSORT` indicates the list should be sorted in place, while `buffer.ORDER_CUSTOM` indicates the

list is already in a custom order.

`_SCINTILLA.next_user_list_type()`
: Returns a unique user list identifier number.

`buffer:user_list_show(`*`id`*`, `*`items`*`)`
: Displays a user list identified by list identifier number *`id`* and constructed from string *`items`* (whose items are separated by `buffer.auto_c_separator` characters). The sorted order of *`items`* (`buffer.auto_c_order`) must have already been defined.

Configure List Behavior and Display

The user's *~/.textadept/properties.lua* file may configure how autocompletion and user lists behave.

`buffer.auto_c_choose_single = `*`bool`*
: Automatically choose the item in a single-item autocompletion list. The default value is `true`. This option has no effect for a user list.

`buffer.auto_c_fill_ups = `*`chars`* (Write-only)
: Allows the user to type any character in string set *`chars`* in order to choose the currently selected item in an autocompletion or user list. The default value is `''`.

`buffer:auto_c_stops(`*`chars`*`)`
: Allows the user to type any character in string set *`chars`* in order to cancel an autocompletion or user list. The default set is empty.

`buffer.auto_c_auto_hide = `*`bool`*
: Automatically cancel an autocompletion or user list when no entries match typed text. The default value is `true`.

`buffer.auto_c_cancel_at_start = `*`bool`*
: Cancel an autocompletion list when backspacing to a position before where autocompletion started (instead of before the word being completed). The default value is `true`. This option has no effect for a user list.

`buffer.auto_c_ignore_case = `*`bool`*
: Ignore case when searching an autocompletion or user list for matches. The default value is `false`.

`buffer.auto_c_case_insensitive_behaviour = mode`
: Switches to list behavior mode *mode* when `buffer.auto_c_ignore_case` is `true`. The default value is `buffer.CASEINSENSITIVEBEHAVIOUR_RESPECTCASE`, which prefers to select case-sensitive matches. `buffer.CASEINSENSITIVEBEHAVIOUR_IGNORECASE` has no preference.

`buffer.auto_c_max_width = chars`
`buffer.auto_c_max_height = items`
: Limits an autocompletion or user list's displayed number of characters per item to *chars*, and limits the number of items per page to *items*. The default values are `0` and `5`, respectively. When *chars* is `0`, the width is automatically sized to fit the longest list item.

`buffer.auto_c_drop_rest_of_word = bool`
: Delete any word characters immediately to the right of autocompleted text. The default value is `false`.

Display Images in Lists

The user can render custom images next to items in autocompletion and user lists by first registering each image with a unique type number. He or she can then append to each list item the type separator character specific to lists followed by an image's type number. Example 8 demonstrates how to display registered images in lists.

Example 8. Show registered images in lists

```
buffer:register_image(1, "...")
buffer:register_image(2, "...")
buffer:auto_c_show('foo?1 bar?2 baz?1')
```

`buffer:register_image(type, pixmap)`
: Registers XPM image *pixmap* to type number *type*. The Appendix on page 123 describes the XPM image format.

  The terminal version displays the first character in string *pixmap* as a list image.

`buffer.rgba_image_width = width`
`buffer.rgba_image_height = height`
: Indicates that the pixel width and height of the RGBA image to be defined using `buffer:register_rgba_image()`

are *width* and *height*, respectively.

`buffer.rgba_image_scale = factor`
: Indicates that the scale factor percentage of the RGBA image to be defined using `buffer:register_rgba_image()` is *factor*.

`buffer:register_rgba_image(type, pixels)`
: Registers RGBA image *pixels* to type number *type*. The dimensions for *pixels* (`buffer.rgba_image_width` and `buffer.rgba_image_height`) must have already been defined. The Appendix on page 123 describes the RGBA image format.

`buffer.auto_c_type_separator = byte`
: Defines byte value *byte* as the character that separates list items from their image types in the list to be passed to `buffer:auto_c_show()` or `buffer:user_list_show()`. The default value is `63` ('?').

`buffer:clear_registered_images()`
: Clears all registered images.

Query Interactive List Information

The user can determine whether or not an interactive list is active, as well as request some of its current properties.

`buffer.auto_c_separator`
: The byte value of the character that separates list items in the lists passed to `buffer:auto_c_show()` and `buffer:user_list_show()`.

`buffer:auto_c_active()`
: Returns whether or not an autocompletion or user list is visible.

`buffer:auto_c_pos_start()`
: Returns the position where autocompletion started or where a user list was shown.

`buffer.auto_c_current` (Read-only)
`buffer.auto_c_current_text` (Read-only)
: The index and text of the currently selected item in an autocompletion or user list.

Handle Interactive List Events

The user can connect to the following interactive list events.

`events.AUTO_C_CHAR_DELETED`
: Emitted after deleting a character while an autocompletion or user list is active.

`events.AUTO_C_CANCELLED`
: Emitted when canceling an autocompletion or user list.

`events.AUTO_C_SELECTION` *`text, position`*
: Emitted after selecting an item from an autocompletion list, but before inserting that item into the buffer. *`text`* is the selection's text and *`position`* is the autocompleted word's beginning position.

`events.USER_LIST_SELECTION` *`id, text, position`*
: Emitted after selecting an item in a user list. *`id`* is that list's identifier number, *`text`* is the selection's text, and *`position`* is the position the list was displayed at.

Show a Call Tip

Textadept allows the user to display a *call tip*. A call tip is a small pop-up window that conveys a piece of textual information, such as the arguments for a function. The user can also highlight a range of text within a call tip.

`buffer:call_tip_show(`*`pos, text`*`)`
: Displays a call tip at position *`pos`* with string *`text`* as the call tip's contents. Any "\001" or "\002" bytes in *`text`* are replaced by clickable up or down arrow visuals, respectively. Typically, these arrows are used to indicate that an identifier, such as a function name, has more than one call tip. The section "Define and Assign Styles" on page 87 describes how to change the style of a call tip, which uses the `'style.calltip'` style name.

  The terminal version does not support arrow visuals.

`buffer:call_tip_set_hlt(`*`start_pos, end_pos`*`)`
: Highlights a call tip's text between positions *`start_pos`* to *`end_pos`* with the color `buffer.call_tip_fore_hlt`.

`buffer:call_tip_cancel()`
: Removes a call tip from view.

Configure Call Tip Display

The user's *~/.textadept/properties.lua* file may configure how Textadept displays a call tip.

`buffer.call_tip_position = bool`
: Display a call tip above the current line instead of below it. The default value is `false`.

`buffer.call_tip_use_style = pixels`
: Fixes the pixel width of tab characters in a call tip at *`pixels`*. The default value is the pixel width of one tab character.

  The terminal version's unit of measure is a character instead of a pixel.

Query Call Tip Information

The user can ascertain whether or not a call tip is visible and, if it is, retrieve its display position.

`buffer:call_tip_active()`
: Returns whether or not a call tip is visible.

`buffer:call_tip_pos_start()`
: Returns a call tip's display position.

Handle Call Tip Events

The user can connect to the following call tip event.

`events.CALL_TIP_CLICK position`
: Emitted when clicking on a call tip. *`position`* is `1` if the up arrow was clicked, `2` if the down arrow was clicked, and `0` otherwise.

  The terminal version cannot detect mouse clicks.

Fold or Hide Lines

Textadept supports *code folding*. Code folding allows the user to temporarily hide blocks of source code, aiding him or her to focus on code he or she is currently interested in. The buffer's lexer determines code fold points that the editor denotes with fold margin markers. Textadept also allows the user to show or hide arbitrary lines.

`buffer:toggle_fold(`*`line`*`)`
: Toggles the fold point on line number *`line`* between expanded (where all of its child lines are displayed) and contracted (where all of its child lines are hidden).

`buffer:fold_line(`*`line, action`*`)`
: Contracts, expands, or toggles the fold point on line number *`line`*, depending on *`action`*, which may be `buffer.FOLDACTION_CONTRACT`, `buffer.FOLDACTION_EXPAND`, or `buffer.FOLDACTION_TOGGLE`, respectively.

`buffer:fold_children(`*`line, action`*`)`
: Contracts, expands, or toggles the fold point on line number *`line`*, as well as all of its child fold points, depending on *`action`*, which may be `buffer.FOLDACTION_CONTRACT`, `buffer.FOLDACTION_EXPAND`, or `buffer.FOLDACTION_TOGGLE`, respectively.

`buffer:fold_all(`*`action`*`)`
: Contracts, expands, or toggles all fold points, depending on *`action`*, which may be `buffer.FOLDACTION_CONTRACT`, `buffer.FOLDACTION_EXPAND`, or `buffer.FOLDACTION_TOGGLE`, respectively.

`buffer:hide_lines(`*`start_line, end_line`*`)`
`buffer:show_lines(`*`start_line, end_line`*`)`
: Hides or shows the range of lines between line numbers *`start_line`* and *`end_line`*.

`buffer:ensure_visible(`*`line`*`)`
: Ensures line number *`line`* is visible by expanding any fold points hiding it.

`buffer:ensure_visible_enforce_policy(`*`line`*`)`
: Ensures line number *`line`* is visible by expanding any fold points hiding it based on the vertical caret policy previously defined in `buffer:set_visible_policy()`.

Query Folded or Hidden Line Information

The user can retrieve a line's fold information and whether or not that line is visible.

`buffer.fold_level[`*`line`*`]`
: The fold level bit-mask for line number *`line`*. The mask consists of an level between 0 and `buffer.FOLDLEVELNUMBERMASK` combined with either of the following bit flags. `buffer.FOLDLEVELHEADERFLAG` indicates the line is a fold point, and `buffer.FOLDLEVELWHITEFLAG` indicates the line is blank.

`buffer.fold_parent[`*`line`*`]` (Read-only)
: The line number of the fold point that contains child line number *`line`*, or `-1` if no fold point was found.

`buffer:get_last_child(`*`line, level`*`)`
: Returns the line number of the last line after line number *`line`* whose fold level is greater than *`level`*. If *`level`* is `-1`, returns the level of *`line`*.

`buffer.fold_expanded[`*`line`*`]`
: Whether or not the fold point for line number *`line`* is expanded.

`buffer:contracted_fold_next(`*`line`*`)`
: Returns the line number of the next contracted fold point starting from line number *`line`*, or `-1` if none exists.

`buffer.line_visible[`*`line`*`]` (Read-only)
: Whether or not line number *`line`* is visible.

`buffer.all_lines_visible` (Read-only)
: Whether or not all lines are visible.

Scroll the View

Textadept provides the following facilities for scrolling the view.

`buffer.first_visible_line = `*`line`*
: Scrolls line number *`line`* to the top of the view.

`buffer.x_offset = `*`pixels`*
: Scrolls to the horizontal pixel position *`pixels`*.

The terminal version's unit of measure is a character instead of a pixel.

`buffer:line_scroll_up()`
`buffer:line_scroll_down()`
: Scrolls the buffer up or down one line, keeping the caret visible.

`buffer:line_scroll(`*`columns`*`, `*`lines`*`)`
: Scrolls the buffer right *`columns`* columns and down *`lines`* lines. Negative values are allowed.

`buffer:scroll_caret()`
: Scrolls the caret into view based on the policies previously defined in `buffer:set_x_caret_policy()` and `buffer:set_y_caret_policy()`.

`buffer:scroll_range(`*`secondary_pos`*`, `*`primary_pos`*`)`
: Scrolls into view the range of text between positions *`primary_pos`* and *`secondary_pos`*, with priority given to *`primary_pos`*.

`buffer:vertical_centre_caret()`
: Centers the current line in the view.

`buffer:scroll_to_start()`
`buffer:scroll_to_end()`
: Scrolls to the beginning or end of the buffer without moving the caret.

Prompt for Input with Dialogs

Textadept can prompt the user for input with a variety of dialogs. Each dialog type, along with its set of options, is listed in the sections below.

Prompt with Messagebox Dialogs

The user can utilize messagebox dialogs to display status messages, ask for confirmation before performing an action, or request input. Example 9 demonstrates how to request input.

Example 9. Request input with a messagebox dialog

```
local mode = ui.dialogs.msgbox{
  title = 'EOL Mode', text = 'Which EOL?',
  icon = 'gtk-dialog-question', button1 = 'CRLF',
  button2 = 'CR', button3 = 'LF'
}
if mode > 0 then
  buffer.eol_mode = mode
  buffer:convert_eols(mode)
end
```

`ui.dialogs.msgbox(`*`options`*`)`
`ui.dialogs.ok_msgbox(`*`options`*`)`
`ui.dialogs.yesno_msgbox(`*`options`*`)`

Prompts the user with a messagebox dialog defined by dialog options table *`options`*, returning the selected button's index. If *`options`*`.string_output` is `true`, returns the selected button's label. If the dialog timed out, returns `0` or `"timeout"`. If the user canceled the dialog, returns `-1` or `"delete"`. A `msgbox` dialog has a default "OK" button, an `ok_msgbox` dialog has default "OK" and "Cancel" buttons, and a `yesno_msgbox` dialog has default "Yes", "No", and "Cancel" buttons.

Messagebox Dialog Options

The user can specify the following messagebox dialog options.

`title = `*`text`*
`text = `*`text`*
`informative_text = `*`text`*

Specifies string *`text`* as the dialog's title text, main message text, or extra informative text.

`icon = `*`name`*
`icon_file = `*`path`*

Displays the icon represented by GTK stock icon *`name`* or filesystem path *`path`* next to the dialog's message text. Example stock icons are "gtk-dialog-error", "gtk-dialog-info", "gtk-dialog-question", and "gtk-dialog-warning". The dialog does not display an icon by default.

The terminal version cannot display icons.

`button1 = label`
`button2 = label`
`button3 = label`
: Changes the right-most, middle, or left-most button's label to string *`label`*.

`string_output = true`
: Return the selected button's label (instead of its index) or the dialog's exit status (instead of its exit code).

`width = width`
`height = height`
: Defines the dialog's pixel dimensions as *`width`* and *`height`*, if possible.

  The terminal version's unit of measure is a character instead of a pixel.

`float = true`
: Shows the dialog on top of all desktop windows.

  The terminal version cannot float dialogs.

`timeout = n`
: Wait *`n`* seconds for the user to select a button before timing out.

  The terminal version cannot timeout dialogs.

Prompt with Inputbox Dialogs

The user can employ inputbox dialogs to prompt for a single line of textual input. Secure inputboxes mask any entered text, much like password dialogs. Example 10 demonstrates how to prompt for standard input.

Example 10. Request text input with an inputbox dialog

```
local button, line = ui.dialogs.inputbox{
  title = 'Goto Line',
  informative_text = 'Line Number:', text = '1'
}
line = tonumber(line)
if button == 1 and line then
  buffer:ensure_visible_enforce_policy(line - 1)
  buffer:goto_line(line - 1)
end
```

`ui.dialogs.inputbox(`*`options`*`)`
`ui.dialogs.standard_inputbox(`*`options`*`)`
`ui.dialogs.secure_inputbox(`*`options`*`)`
`ui.dialogs.secure_standard_inputbox(`*`options`*`)`

Prompts the user with a one-line inputbox dialog defined by dialog options table *`options`*, returning the selected button's index along with the user's input text. If *`options`*`.string_output` is `true`, returns the selected button's label along with the user's input text. If the dialog timed out, returns `0` or `"timeout"`. If the user canceled the dialog, returns `-1` or `"delete"`. `inputbox` and `secure_inputbox` dialogs have a default "OK" button, while `standard_inputbox` and `secure_standard_inputbox` dialogs have default "OK" and "Cancel" buttons.

Inputbox Dialog Options

The user can supply the following inputbox dialog options.

`title = `*`text`*
`informative_text = `*`text`*
`text = `*`text`*

Specifies string *`text`* as the dialog's title text, main message text, or initial input text.

`button1 = `*`label`*
`button2 = `*`label`*
`button3 = `*`label`*

Changes the right-most, middle, or left-most button's label to string *`label`*.

`string_output = true`

Return the selected button's label (instead of its index) or the dialog's exit status (instead of its exit code).

`width = `*`width`*
`height = `*`height`*

Defines the dialog's pixel dimensions as *`width`* and *`height`*, if possible.

The terminal version's unit of measure is a character instead of a pixel.

`float = true`

Shows the dialog on top of all desktop windows.

The terminal version cannot float dialogs.

`timeout = n`
: Wait *n* seconds for the user to select a button before timing out.

  The terminal version cannot timeout dialogs.

Prompt with File Selection Dialogs

The user can utilize file selection dialogs to request files to be opened or saved to. Example 11 demonstrates how to use a fileselect dialog to open files.

Example 11. Open files from a fileselect dialog

```
local filenames = ui.dialogs.fileselect{
  title = 'Open C File', with_directory = _HOME,
  with_extension = {'c', 'h'}, select_multiple = true
}
if filenames then io.open_file(filenames) end
```

`ui.dialogs.fileselect(`*options*`)`
: Prompts the user with a file selection dialog defined by dialog options table *options*, returning the string file selected or, if *options*`.select_multiple` is `true`, the list of files selected. If the user canceled the dialog, returns `nil`.

`ui.dialogs.filesave(`*options*`)`
: Prompts the user with a file save dialog defined by dialog options table *options*, returning the string file chosen. If the user canceled the dialog, returns `nil`.

File Selection Dialog Options

The user can specify the following file selection dialog options.

`title = `*text*
: Specifies string *text* as the dialog's title text.

`with_directory = `*path*
`with_file = `*name*
: Displays filesystem directory *path* in the view and selects or provides filename *name*.

`with_extension = `*list*
: Requires displayed files to have an extension in list *list*.

`select_multiple = true`
: Allows the user to select multiple files in the fileselect dialog.

`select_only_directories = true`
: Requires the user to select a directory in the fileselect dialog.

`no_create_directories = true`
: Prevents the user from creating new directories in the filesave dialog.

Prompt with a Textbox Dialog

The user can employ textbox dialogs to display editable or read-only textual data. Example 12 demonstrates how to do so.

Example 12. Display a textbox dialog

```
ui.dialogs.textbox{
  title = 'License Agreement',
  informative_text = 'You agree to:',
  text_from_file = _HOME..'/LICENSE'
}
```

`ui.dialogs.textbox(`*`options`*`)`
: Prompts the user with a multiple-line textbox dialog defined by dialog options table *`options`*, returning the selected button's index. If *`options`*`.string_output` is `true`, returns the selected button's label. If *`options`*`.editable` is `true`, also returns the textbox's text. If the dialog timed out, returns `0` or `"timeout"`. If the user canceled the dialog, returns `-1` or `"delete"`. A `textbox` dialog has a default "OK" button.

Textbox Dialog Options

The user can provide the following textbox dialog options.

`title =` *`text`*
`informative_text =` *`text`*
: Specifies string *`text`* as the dialog's title text or main message text.

`text = text`
`text_from_file = filename`
: Fills the dialog's textbox with string *`text`* or the contents of file *`filename`*.

`button1 = label`
`button2 = label`
`button3 = label`
: Changes the right-most, middle, or left-most button's label to string *`label`*.

`editable = true`
: Allows the user to edit the textbox's text.

`focus_textbox = true`
: Focuses the textbox instead of the buttons.

`scroll_to = "bottom"`
: Scrolls to the bottom of the textbox's text.

`selected = true`
: Selects all of the textbox's text.

`monospaced_font = true`
: Displays the textbox's text in a monospaced font.

`string_output = true`
: Return the selected button's label (instead of its index) or the dialog's exit status (instead of its exit code).

`width = width`
`height = height`
: Defines the dialog's pixel dimensions as *`width`* and *`height`*, if possible.

  The terminal version's unit of measure is a character instead of a pixel.

`float = true`
: Shows the dialog on top of all desktop windows.

  The terminal version cannot float dialogs.

`timeout = n`
: Wait *`n`* seconds for the user to select a button before timing out.

  The terminal version cannot timeout dialogs.

Prompt with Dropdown Dialogs

The user can utilize dropdown dialogs to prompt for an item from a list. Example 13 demonstrates how to do this.

Example 13. Prompt for an item from a dropdown dialog

```
local button, encoding = ui.dialogs.dropdown{
  title = 'Select Encoding', items = io.encodings,
  string_output = true,
  width = not CURSES and 200 or 20
}
if button == _L['_OK'] then
  io.set_buffer_encoding(encoding)
end
```

`ui.dialogs.dropdown(`*`options`*`)`
`ui.dialogs.standard_dropdown(`*`options`*`)`
: Prompts the user with a dropdown item selection dialog defined by dialog options table *`options`*, returning the selected button's index along with the selected item's index. If *`options`*`.string_output` is `true`, returns the selected button's label along with the selected item's text. If the dialog closed due to *`options`*`.exit_onchange`, returns `4` along with either the selected item's index or its text. If the dialog timed out, returns `0` or `"timeout"`. If the user canceled the dialog, returns `-1` or `"delete"`. Dropdown dialogs have the same default buttons as inputboxes do.

Dropdown Dialog Options

The user can supply the following dropdown dialog options.

`title = `*`text`*
`informative_text = `*`text`*
: Specifies string *`text`* as the dialog's title text or main message text.

`items = `*`list`*
: Shows string item list *`list`* in the dropdown.

`button1 = `*`label`*
`button2 = `*`label`*
`button3 = `*`label`*
: Changes the right-most, middle, or left-most button's label to string *`label`*.

`exit_onchange = true`
: Closes the dialog after selecting a new item.

`select = index`
: Initially selects the list item at index number *`index`*. The default value is `1`.

`string_output = true`
: Return the selected button's label (instead of its index) and the selected item's text (instead of its index). If no item was selected, return the dialog's exit status (instead of its exit code).

`width = width`
`height = height`
: Defines the dialog's pixel dimensions as *`width`* and *`height`*, if possible.

  The terminal version's unit of measure is a character instead of a pixel.

`float = true`
: Shows the dialog on top of all desktop windows.

  The terminal version cannot float dialogs.

`timeout = n`
: Wait *`n`* seconds for the user to select a button before timing out.

  The terminal version cannot timeout dialogs.

Prompt with a Filtered List Dialog

The user can employ filtered list dialogs to prompt for items from a *filtered list*. Filtered lists allow the user to filter down a list's contents to match his or her typed text. Space characters in typed text count as wildcards. Example 14 demonstrates how to prompt for items from a filtered list dialog.

Example 14. Prompt for items from a filtered list dialog

```
local button, i = ui.dialogs.filteredlist{
  title = 'Title', columns = {'Foo', 'Bar'},
  items = {'a', 'b', 'c', 'd'}
}
if button == 1 then ui.print('Selected row ' + i) end
```

`ui.dialogs.filteredlist(`*`options`*`)`
: Prompts the user with a filtered list item selection dialog defined by dialog options table *`options`*, returning the selected button's index along with the index or indices of the selected item or items (depending on whether or not *`options`*`.select_multiple` is `true`). If *`options`*`.string_output` is `true`, returns the selected button's label along with the text of the selected item or items. If the dialog timed out, returns `0` or `"timeout"`. If the user canceled the dialog, returns `-1` or `"delete"`. A `filteredlist` dialog has a default "OK" button.

Filtered List Dialog Options

The user can provide the following filtered list dialog options.

`title =` *`text`*
`informative_text =` *`text`*
: Specifies string *`text`* as the dialog's title text or main message text.

`columns =` *`names`*
`items =` *`list`*
: Shows string item list *`list`* in the filtered list, headed by column names list *`names`*.

`button1 =` *`label`*
`button2 =` *`label`*
`button3 =` *`label`*
: Changes the right-most, middle, or left-most button's label to string *`label`*.

`select_multiple = true`
: Allows the user to select multiple items.

`search_column =` *`index`*
: Specifies column number *`index`* as the column to filter the input text against. The default value is `1`.

`output_column =` *`index`*
: Specifies column number *`index`* as the column to use for `string_output`. The default value is `1`.

`string_output = true`
: Return the selected button's label (instead of its index) and the selected item's text (instead of its index). If no item was selected, return the dialog's exit status (instead

of its exit code).

`width = width`
`height = height`
: Defines the dialog's pixel dimensions as *`width`* and *`height`*, if possible.

  The terminal version's unit of measure is a character instead of a pixel.

`float = true`
: Shows the dialog on top of all desktop windows.

  The terminal version cannot float dialogs.

`timeout = n`
: Wait *`n`* seconds for the user to select a button before timing out.

  The terminal version cannot timeout dialogs.

Manipulate the Command Entry

Textadept provides a command entry that supports multiple key modes. Each key mode allows the entry to adopt a different role. The entry's default roles include Lua command input mode and incremental search mode. The section "Configure Key Bindings" on page 103 describes key modes. The user can also supply a list of completions for autocompleting the current command entry word.

`ui.command_entry.enter_mode(mode)`
: Opens the command entry in key mode *`mode`*.

`ui.command_entry.entry_text = text`
: Places string *`text`* in the command entry.

`ui.command_entry.show_completions(completions)`
: Shows completion list *`completions`* for the current word prefix. Word prefix characters are alphanumerics and underscores.

`ui.command_entry.finish_mode([f])`
: Exits the current key mode, closes the command entry, and calls function *`f`* (if given) with the command entry's text as an argument.

Issue Lua Commands

The user can execute Lua commands and code within Textadept's Lua State.

`ui.command_entry.focus()`
: Opens the Lua command entry.

`ui.command_entry.execute_lua(`*`code`*`)`
: Executes string *`code`* as Lua code that is subject to an "abbreviated" environment. In this environment, the contents of the `buffer`, `view`, and `ui` tables are also considered as global functions and fields.

`textadept.menu.select_command()`
: Prompts the user to select a menu command to run.

Handle Command Entry Events

The user can connect to the following command entry event.

`events.COMMAND_ENTRY_KEYPRESS` *`code, shift, ctrl, alt, meta`*
: Emitted when pressing a key in the command entry. *`code`* is the numeric key code and *`shift`*, *`ctrl`*, *`alt`*, and *`meta`* indicate whether or not those modifier keys (Shift, Control, Alt, and Command, respectively) are pressed.

Autocomplete Code with Adeptsense

Textadept utilizes *Adeptsenses* to autocomplete code and display API documentation. Language modules generally define, configure, and provide these "senses" with symbol completion lists that are either generated from an external tool like Ctags[3] or supplied manually. Some senses may require fine-tuning in order to provide more accurate results. All of these facilities are described in the following sections.

Define an Adeptsense

The user can create and configure a generic Adeptsense for a particular programming language. Examples 15 and 16 show

3 *http://ctags.sourceforge.net*

two sample Adeptsenses: one for a statically typed language and the other for a dynamically typed language.

Example 15. Sample ANSI C Adeptsense

```
local sense = textadept.adeptsense.new('ansi_c')
local as = textadept.adeptsense
sense.ctags_kinds = {
  e = as.FIELD, f = as.FUNCTION, g = as.CLASS,
  m = as.FIELD, s = as.CLASS, t = as.CLASS
}
sense.syntax.type_declarations = {
  -- Foo bar, Foo *bar, Foo* bar, Foo &bar, etc.
  '([%w_%.]+)[%s%*&]+%_[^%w_]'
}
sense:add_trigger('.')
sense:add_trigger('->')
sense:load_ctags(_USERHOME..'/modules/ansi_c/tags')
sense.api_files = {_USERHOME..'/modules/ansi_c/api'}
```

Example 16. Sample Python Adeptsense

```
local sense = textadept.adeptsense.new('python')
local as = textadept.adeptsense
sense.ctags_kinds = {
  c = as.CLASS, f = as.FUNCTION, m = as.FIELD
}
sense.syntax.class_definition = [[
^%s*class%s+([%w_]+)%s*%(?%s*([%w_.]*)]]
sense.syntax.type_declarations = {
  -- bar = Foo(args)
  '%_%s*=%s*([%u][%w_.]+)%s*%b()%s*$'
}
sense.syntax.type_assignments = {
  ['^[\'"]'] = 'str',
  ['^%('] = 'tuple',
  ['^%['] = 'list',
  ['^{'] = 'dict',
  ['^open%s*%b()%s*$'] = 'file'
}
sense:add_trigger('.')
sense:load_ctags(_USERHOME..'/modules/python/tags')
sense.api_files = {_USERHOME..'/modules/python/api'}
```

`textadept.adeptsense.new(`*`lexer`*`)`
: Creates and returns a new Adeptsense for language *`lexer`*.

sense.syntax.self = *keyword*
: Specifies string *keyword* as the keyword that is used as an object's reference to itself from within that object's class definition. The default value is `'self'`. *sense* is an Adeptsense.

sense.syntax.class_definition = *patt*
: Specifies pattern *patt* as the pattern that matches a class definition. *patt* captures that definition's class name and (optionally) its superclass name. The default value is `'class%s+([%w_]+)'`. *sense* is an Adeptsense.

sense.syntax.word_chars = *patt*
: Specifies pattern *patt* as the pattern that contains all of the characters allowed in an identifier. The default value is `'%w_'`. *sense* is an Adeptsense.

sense.syntax.symbol_chars = *patt*
: Specifies pattern *patt* as the pattern that contains all of the characters allowed in a symbol, including member operators. The default value is `'[%w_%.]'`. *sense* is an Adeptsense.

sense.syntax.type_declarations[
#sense.syntax.type_declarations + 1] = *patt*
: Specifies pattern *patt* as a pattern that matches the class type declaration for a symbol. *patt* represents a symbol with "%_" and captures that symbol's class type name. A default pattern is `'(%u[%w_%.]+)%s+%_'`. *sense* is an Adeptsense.

sense.syntax.type_declarations_exclude[*class*] = true
: Prevents Adeptsense *sense* from autocompleting class type *class*.

sense.syntax.type_assignments[*patt*] = *class*
: Specifies pattern *patt* as the pattern that matches a type declaration of class type *class*. *patt* matches text immediately after the '=' in a variable assignment. *sense* is an Adeptsense.

sense:add_trigger(*c*[, *only_fields*[, *only_functions*]])
: Allows the user to autocomplete the symbol behind the caret by typing character(s) *c*. *sense* is an Adeptsense. If either *only_fields* or *only_functions* is true, *sense* displays the appropriate subset of completions.

Provide Completions with Ctags

The user can supply Ctags files to generate symbol completion lists.

sense.ctags_kinds[*kind*] = *type*
: Identifies Ctags kind *kind* as Adeptsense type *type* for Adeptsense *sense*. The available types are `text` `adept.adeptsense.CLASS`[4], `textadept.adeptsense.FUNCTION`, and `textadept.adeptsense.FIELD`.

sense:load_ctags(*tag_file*[, *no_locations*])
: Generates a set of symbol completion lists from Ctags file *tag_file* and adds the set to Adeptsense *sense*. *no_locations* indicates whether or not to store the location part of tags. If `true`, *sense*:goto_ctag() cannot be used with this set of tags.

sense:handle_ctag(*tag_name*, *file_name*, *ex_cmd*, *ext_fields*)
: Handles unrecognized Ctag kinds in *sense*:load_ctags() for Adeptsense *sense*. The parameters are extracted from Ctags' tag format:

```
tag_name<TAB>file_name<TAB>ex_cmd;"<TAB>ext_fields
```

sense:handle_clear()
: Helps clear Adeptsense *sense* along with *sense*:clear().

Provide Manual Completions

The user can manually create and assign completion lists to individual symbols.

sense.completions[*class*] = *completions*
: Assigns completion list *completions* to class type *class* in Adeptsense *sense*. *completions* contains tables that hold function and field completion strings assigned to `functions` and `fields` keys, respectively.

sense.inherited_classes[*class*] = *superclasses*
: Indicates class type *class* inherits completions from all superclasses in list *superclasses* for Adeptsense *sense*.

sense:handle_clear()
: Helps clear Adeptsense *sense* along with *sense*:clear().

4 "Classes" are simply containers for functions and fields. A language does not need to be object-oriented in order to use the `CLASS` type.

Provide API Documentation

The user can supply documentation for symbols using API files. Example 17 lists a few sample API file entries.

Example 17. Sample API file entries

```
string The Lua string module.
lower string.lower(s)\nReturns a lower case copy of s.
max math.max(a, b)\nReturns the larger of a and b.
max math.max(...)\nReturns the largest argument.
pi math.pi\nThe value of π.
```

sense.api_files[#*sense*.api_files + 1] = *file*
: Adds API documentation file *file* to Adeptsense *sense*. Files list symbol names, not full symbols, along with their documentation.

Fine-Tune Adeptsense Behavior

The user can subclass Adeptsense functions in order to provide more accurate results for specific languages. When requested to perform autocompletion, Adeptsense retrieves the current symbol, determines its class type, and then obtains the completion list for it. Examples 18, 19, and 20 show some sample subclassing functions. The user can access an Adeptsense's original functions using the sense's `self.super` table.

sense:get_symbol()
: Returns the full symbol and the current symbol behind the caret for Adeptsense *sense*. For example, `buffer.cur` would return `"buffer"` and `"cur"`.

Example 18. Identify array symbols in Ruby

```
function sense:get_symbol()
  local line, p = buffer:get_cur_line()
  if line:sub(1, p):match('%[.-%]%s*%.$') then
    return 'Array', ''
  end
  -- ...
  -- (More symbol checking.)
  -- ...
  -- Otherwise, fall back on default functionality.
  return self.super.get_symbol(self)
end
```

sense:get_class(*symbol*)

Returns the class type of string *symbol* for Adeptsense *sense*. If *symbol* is *sense*.syntax.self and occurs inside a class definition that matches *sense*.syntax.class_definition, that class is returned. Otherwise, the buffer is searched backwards for either a type declaration of *symbol* according to the patterns in *sense*.syntax.type_declarations, or for a type assignment of *symbol* according to *sense*.syntax.type_assignments, whichever comes first.

Example 19. Recognize number class types in Ruby

```
function sense:get_class(symbol)
  local class = self.super.get_class(self, symbol)
  if class then return class end
  if tonumber(symbol:match('^%d+%.?%d*$')) then
    return symbol:find('%.') and 'Float' or 'Integer'
  end
end
```

sense:get_completions(*symbol*, *only_fields*, *only_functions*)

Returns the list of completions for string *symbol* in Adeptsense *sense*. If either *only_fields* or *only_functions* is true, returns the appropriate subset of completions.

Example 20. Autocomplete Java packages

```
function sense:get_completions(symbol, ...)
  if buffer:get_cur_line():find('^%s*import') then
    if symbol == 'import' then symbol = '' end
    -- "self.imports" must have been previously
    -- populated.
    local import = self.imports or {}
    for package in symbol:gmatch('[^%.]+') do
      if not import[package] then return nil end
      import = import[package]
    end
    local c = {}
    for k, v in pairs(import) do
      local parent = type(v) == 'table'
      c[#c + 1] = parent and k..'?1' or v..'?2'
    end
    table.sort(c)
    return c
  end
  return self.super.get_completions(self, symbol, ...)
end
```

sense:get_apidoc(*symbol*)
: Returns the list of API documentation strings for string *symbol* in Adeptsense *sense*. A `pos` key in that list holds the index of the documentation string that should be shown.

Call on Adeptsense

The user can call on the current language's Adeptsense.

`textadept.adeptsense.complete([nil, [`*only_fields*`[, `*only_functions*`]]])`
: Shows an autocompletion list for the symbol behind the caret using the current language's Adeptsense, returning `true` on success. If either *only_fields* or *only_functions* is `true`, displays the appropriate subset of completions.

`textadept.adeptsense.show_apidoc()`
: Shows a call tip with API documentation for the symbol behind the caret using the current language's Adeptsense. If a call tip is already shown, cycles to the next one if it exists.

`textadept.adeptsense.goto_ctag([nil[, `*kind*`[, `*title*`]]])`
: Prompts the user to select a known symbol of kind *kind* from the current language's Adeptsense to jump to. If *kind* is `nil`, displays all known symbols. *title* is the filtered list dialog prompt's title.

Configure Adeptsense Display

The user can change the Adeptsense images rendered next to completions and indicate whether or not global classes, functions, and fields are always shown.

`textadept.adeptsense.FUNCTION_IMAGE = `*pixmap*
`textadept.adeptsense.FIELD_IMAGE = `*pixmap*
: Defines the XPM image for functions or fields as *pixmap*. The Appendix on page 123 describes the XPM image format.

`textadept.adeptsense.always_show_globals = `*bool*
: Include globals in the list of completions offered. The default value is `true`.

Compile and Run Code

Textadept knows some of the shell commands that compile and run source code files. The message buffer shows a command's output, marking any recognized warnings and errors.

`textadept.run.compile()`
`textadept.run.run()`
: Compiles or runs the current file based on its extension or language using the shell command from the `textadept.run.compile_command` or `textadept.run.run_command` table, respectively.

`textadept.run.goto_error(`*`line`*`, `*`next`*`)`
: Jumps to the source of the recognized compile/run warning or error on line number *`line`* in the message buffer. If *`line`* is `nil`, jumps to the next or previous warning or error, depending on boolean *`next`*. Displays an annotation with the warning or error message if possible.

Configure Compile and Run Settings

The user's *~/.textadept/init.lua* file may supply new shell commands or modify existing ones for particular file extensions or programming languages. Shell commands may contain any of the macros listed in Table 8.

Table 8. Command line string macros

Macro	Description
%f or %(filename)	The file's name, including its extension.
%e or %(filename_noext)	The file's name, excluding its extension.
%d or %(filedir)	The current file's directory path.
%(filepath)	The current file's full path.

`textadept.run.compile_command[`*`ext`*`] = `*`command`*
`textadept.run.compile_command[`*`lexer`*`] = `*`command`*
`textadept.run.run_command[`*`ext`*`] = `*`command`*
`textadept.run.run_command[`*`lexer`*`] = `*`command`*
: Assigns shell command *`command`* to compile or run files that have extension *`ext`* or language *`lexer`*. *`command`* is a command line string or function that returns such a

string and may contain any of the macros listed in Table 8.

`textadept.run.error_patterns[#textadept.run.error_patterns + 1] =` *`patt`*
: Adds pattern *`patt`* to the list of patterns that match compile or runtime warning or error messages. *`patt`* matches a single line, capturing the filename the warning or error occurred in, the line number the warning or error occurred on, and (optionally) the warning or error message's text, all in that order.

Query Compile and Run Information

The user can request the working directory of compile and run shell commands.

`textadept.run.cwd` (Read-only)
: The most recently executed compile or run shell command's working directory.

Handle Compile and Run Events

The user can connect to the following compile and run shell command output events.

`events.COMPILE_OUTPUT` *`lexer, output`*
`events.RUN_OUTPUT` *`lexer, output`*
: Emitted when executing a language's compile or run shell command. *`lexer`* is the language's lexer name and *`output`* is a line of string output from that command.

Configure Textadept

Textadept allows the user to configure nearly every aspect of it, including indentation and line endings, color themes, file types, key bindings, snippets, and more. The user can have many of these options apply only to certain programming languages and file types. Most of Textadept's configuration settings are described in the following sections. In addition, Example 3 on page 6 demonstrates one of the methods for setting some Lua-specific options.

Configure Indentation and Line Endings

The user can change an individual file's indentation settings and end of line characters. His or her *~/.textadept/properties.lua* file or the current language module configures a buffer's default indentation settings, while Textadept itself determines which default end of line characters to use.

`buffer.use_tabs = bool`
: Use tabs instead of spaces in indentation. The default value is `false`. Changing the current setting does not convert any of the buffer's existing indentation. Use `textadept.editing.convert_indentation()` to do so.

`buffer.tab_width = width`
: Specifies character count *`width`* as the number of space characters represented by a tab character. The default value is `2`.

`buffer.indent = width`
: Specifies character count *`width`* as the number of spaces in one level of indentation. The default value is `0`, which uses the value of `buffer.tab_width`.

`textadept.editing.convert_indentation()`
: Converts all line indentation between tabs and spaces according to `buffer.use_tabs`.

`buffer.tab_indents = bool`
: Indent text when tabbing within indentation. The default value is `true`.

`buffer.back_space_un_indents = bool`
: Un-indent text when backspacing within indentation. The default value is `true`.

`textadept.editing.AUTOINDENT = bool`
: Match the previous line's indentation level after inserting a new line. The default value is `true`.

`buffer.eol_mode = mode`
: Changes the current end of line mode to *`mode`*. The default value is `buffer.EOL_CRLF` (carriage return with line feed) on Windows platforms, and `buffer.EOL_LF` (line feed) on all others. `buffer.EOL_CR` (carriage return) is also an option. Changing the current mode does not convert any of the buffer's existing end of line characters. Use

`buffer:convert_eols()` to do so.

`buffer:convert_eols(`*`mode`*`)`
: Converts all end of line characters to those in end of line mode *`mode`*. When *`mode`* is `buffer.EOL_CRLF`, `buffer.EOL_CR`, or `buffer.EOL_LF`, converts all end of line characters to carriage returns with line feeds, carriage returns only, or line feeds only, respectively.

Query Indentation and Line Ending Settings

The user can retrieve the current indentation and end of line character settings.

`buffer.use_tabs`
: Whether or not tabs are used instead of spaces in indentation.

`buffer.tab_width`
: The number of space characters represented by a tab character.

`buffer.indent`
: The number of spaces in one level of indentation. When `buffer.indent` is `0`, `buffer.tab_width` is used instead.

`buffer.eol_mode`
: The current end of line mode, either `buffer.EOL_CRLF` (carriage return with line feed), `buffer.EOL_CR` (carriage return), or `buffer.EOL_LF` (line feed).

Configure Character Classifications

The user's *~/.textadept/properties.lua* file may alter the buffer's word, whitespace, and punctuation character classifications in order to modify the buffer's default behavior when moving between words or searching for whole words.

`buffer.word_chars = `*`chars`*
: Declares that all of the characters in string *`chars`* are word characters. The default value is a string that contains alphanumeric characters, an underscore, and all characters greater than ASCII value 127.

`buffer.whitespace_chars = `*`chars`*
: Declares that all of the characters in string *`chars`* are

whitespace characters. Declare this only after declaring `buffer.word_chars`. The default value is a string that contains all non-newline characters less than ASCII value 33.

`buffer.punctuation_chars = chars`
: Declares that all of the characters in string *`chars`* are punctuation characters. Declare this only after declaring `buffer.word_chars`. The default value is a string that contains all non-word and non-whitespace characters.

`buffer:set_chars_default()`
: Resets `buffer.word_chars`, `buffer.whitespace_chars`, and `buffer.punctuation_chars` to their respective defaults.

Query Character Classifications

The user can obtain character classification information.

`buffer.word_chars`
`buffer.whitespace_chars`
`buffer.punctuation_chars`
: The string set of characters recognized as word, whitespace, and punctuation characters, respectively.

Configure the Color Theme

The user's *~/.textadept/init.lua* file may change the default color theme name. The following section, "Create or Modify a Color Theme", describes how to create a theme.

`ui.set_theme(`*`name`*`[, `*`props`*`])`
: Switches the editor theme to string *`name`* and (optionally) assigns the properties contained in table *`props`*. If *`name`* contains slashes, it is assumed to be an absolute path to a theme instead of a theme name.

Create or Modify a Color Theme

The user can customize Textadept's look and feel with color themes. Color themes reside in his or her *~/.textadept/themes/* directory. Each theme is a single Lua file that defines Textadept's colors, specifies how the editor displays text, and assigns colors and alpha values to various view properties. Colors are integers that range from 0 to 0xFFFFFF. Alpha

transparency values are either integers that range from 0 to 255 (`buffer.ALPHA_TRANSPARENT to buffer.ALPHA_OPAQUE`), or `buffer.ALPHA_NOALPHA` values. All of Textadept's theme settings are described in the following sections. Example 4 on page 6 demonstrates how to modify an existing theme.

Define Colors

The user can define colors in "0xBBGGRR" hexadecimal format or its decimal equivalent.

NOTE

The terminal version only recognizes 16 colors, regardless of how many colors the user's terminal actually supports. Recognized colors are: black (`0x000000`), red (`0x800000`), green (`0x008000`), yellow (`0x808000`), blue (`0x000080`), magenta (`0x800080`), cyan (`0x008080`), white (`0xC0C0C0`), light black (`0x404040`), light red (`0xFF0000`), light green (`0x00FF00`), light yellow (`0xFFFF00`), light blue (`0x0000FF`), light magenta (`0xFF00FF`), light cyan (`0x00FFFF`), and light white (`0xFFFFFF`). Even if the user's terminal uses a different color map, he or she must use these color values. His or her terminal will remap them automatically.

`buffer.property['color.`*`name`*`'] = `*`color`*

Assigns color *`color`* to arbitrary color name *`name`*. By convention, color names have the "`color.`" prefix, but this is not enforced.

TIP

Colors may also be in "#RRGGBB" string notation, but only in the context of properties.

Define and Assign Styles

The user can assign custom styles to style names. Textadept uses styles to display text, from the default font to line numbers in the margin to source code comments, strings, and keywords. Each of these elements has a style name. Table 9 lists most of Textadept's default style names. Styles themselves are strings composed of comma-separated settings taken from Table 10. Any "`%(`*`prop`*`)`" sequences represent the value of `buf`

fer.property['*prop*'] at the time that property is read. Example 21 shows some sample styles. Internally, Textadept supports up to 256 different styles, numbered from 0 to 255.

Example 21. Sample styles

```
local red = 'fore:%(color.red)'
local bold_red = red..',bold'
local normal_red = bold_red..',notbold'
```

Table 9. Default style names

default	calltip	class	comment
constant	embedded	error	function
identifier	keyword	label	linenumber
number	operator	preprocessor	regex
string	type	variable	whitespace

Table 10. Style settings

Setting	Description
font:*name*	Use font name *name* for text.
size:*int*	Use font size *int* for text.
[not]bold	Whether or not text is bold.
[not]italics	Whether or not text is italic.
[not]underlined	Whether or not text is underlined.
fore:*color*	Use foreground color *color* for text.
back:*color*	Use background color *color* for text.
[not]eolfilled	Whether or not the background color of text extends to the view's right margin.
case:*char*	Use letter case *char* for text: `u` for upper case, `l` for lower case, or `m` for normal, mixed case.
[not]visible	Whether or not text is visible.
[not]changeable	Whether text is changeable or read-only.
[not]hotspot	Whether or not text is clickable.

`buffer.property['style.`*`name`*`'] = `*`style`*

Assigns style *`style`* to style name *`name`*. The "`style.`" prefix is required. Table 9 lists all default style names.

The terminal version may require styles to set the "bold" attribute in order to display the "light" colors.

TIP

All styles inherit from the default style `'style.default'` in order to minimize repetition.

Query Colors

The user can fetch the decimal format of defined colors. This format is required by the color settings listed in the following sections.

`buffer.property_int['color.`*`name`*`']`

The decimal value of the color assigned to color name *`name`*. By convention, color names have the "`color.`" prefix, but this is not enforced.

Assign Caret Colors

The user can specify a custom caret color and change the appearance of the line that contains the caret.

`buffer.caret_fore = `*`color`*

Specifies color *`color`* as the caret's foreground color.

`buffer.caret_line_back = `*`color`*

Specifies color *`color`* as the background color of the line that contains the caret.

`buffer.caret_line_back_alpha = `*`alpha`*

Specifies alpha value *`alpha`* as the caret line's background alpha. The default value is `buffer.ALPHA_NOALPHA`.

The terminal version cannot render alpha transparency.

`buffer.additional_caret_fore = `*`color`*

Specifies color *`color`* as the foreground color of additional carets.

Assign Selection Colors

The user can assign custom selection colors and alpha values.

`buffer:set_sel_fore(`*`use_setting`*`, `*`color`*`)`
`buffer:set_sel_back(`*`use_setting`*`, `*`color`*`)`
: Overrides the selection's default foreground or background color with color *`color`* if *`use_setting`* is `true`.

`buffer.sel_alpha = `*`alpha`*
: Specifies alpha value *`alpha`* as the selection's alpha. The default value is `buffer.ALPHA_NOALPHA`.

  The terminal version cannot render alpha transparency.

`buffer.additional_sel_fore = `*`color`* (Write-only)
`buffer.additional_sel_back = `*`color`* (Write-only)
: Specifies color *`color`* as the foreground or background color of additional selections.

`buffer.additional_sel_alpha = `*`alpha`*
: Specifies alpha value *`alpha`* as the alpha of additional selections. The default value is `buffer.ALPHA_NOALPHA`.

  The terminal version cannot render alpha transparency.

Assign Margin Colors

The user can designate custom fold margin colors.

`buffer:set_fold_margin_colour(`*`use_setting`*`, `*`color`*`)`
`buffer:set_fold_margin_hi_colour(`*`use_setting`*`, `*`color`*`)`
: Overrides the fold margin's default color or highlight color with color *`color`* if *`use_setting`* is `true`.

Assign Marker Colors

The user can apply custom colors and alpha values to existing markers. Textadept provides 32 markers, numbered from 0 to 31, to mark lines with. Textadept uses marker numbers 25 to 31 internally for the fold markers listed in Table 5 on page 48.

`buffer.marker_fore[`*`marker`*`] = `*`color`* (Write-only)
`buffer.marker_back[`*`marker`*`] = `*`color`* (Write-only)
: Specifies color *`color`* as the foreground or background color of marker number *`marker`*.

`buffer.marker_alpha[`*`marker`*`] = `*`alpha`* (Write-only)
Specifies alpha value *`alpha`* as the alpha of marker number *`marker`* when *`marker`* is shown in the text area (not the margin). A marker is only shown in the text area when the margin configured to display it has zero width. The default value is `buffer.ALPHA_NOALPHA`.

The terminal version cannot render alpha transparency.

`buffer:marker_enable_highlight(`*`enabled`*`)`
Highlights the fold markers for the current fold block if *`enabled`* is `true`.

`buffer.marker_back_selected[`*`marker`*`] = `*`color`* (Write-only)
Specifies color *`color`* as the background color of highlighted fold marker number *`marker`*.

`buffer.marker_back[textadept.bookmarks.MARK_BOOKMARK] =` *`color`*
Changes the bookmark marker's color to color *`color`*.

`buffer.marker_back[textadept.run.MARK_WARNING] = `*`color`*
`buffer.marker_back[textadept.run.MARK_ERROR] = `*`color`*
Changes the compile/run warning or error marker's color to color *`color`*.

TIP

The user can change a marker's symbol by calling `buffer:marker_define()`, `buffer:marker_define_pixmap()`, or `buffer:marker_define_rgba_image()`. The section "Mark Lines with Markers" on page 47 describes how to assign symbols to marker numbers.

Assign Indicator Colors

The user can apply custom colors and alpha values to existing indicators. Textadept provides 32 indicators, numbered from 0 to 31, to mark text with.

`buffer.indic_fore[`*`indicator`*`] = `*`color`*
Specifies color *`color`* as the foreground color of indicator number *`indicator`*.

`buffer.indic_alpha[`*`indicator`*`] = `*`alpha`*
`buffer.indic_outline_alpha[`*`indicator`*`] = `*`alpha`*

Specifies alpha value *`alpha`* as the fill alpha or outline color alpha of indicator number *`indicator`*, whose style is either `buffer.INDIC_ROUNDBOX`, `buffer.INDIC_STRAIGHTBOX`, or `buffer.INDIC_DOTBOX`. The default values are `30` and `50`, respectively.

The terminal version cannot render alpha transparency.

`buffer.indic_fore[textadept.editing.INDIC_BRACEMATCH] =`
`color`

Changes the matching brace indicator's color to color *`color`*.

The terminal version uses style name `'style.bracelight'` instead of `textadept.editing.INDIC_BRACEMATCH`.

`buffer.indic_fore[textadept.editing.INDIC_HIGHLIGHT] =`
`color`

Changes the highlighted word indicator's color to color *`color`*.

TIP

The user can change an indicator's symbol by setting `buffer.indic_style`. The section "Mark Text with Indicators" on page 52 describes how to assign styles to indicator numbers.

Assign and Query Hotspot Colors

The user can specify and retrieve custom colors for *hotspots*. Hotspots are regions of clickable text that behave like web browser links.

`buffer:set_hotspot_active_fore(`*`use_setting`*`, `*`color`*`)`
`buffer:set_hotspot_active_back(`*`use_setting`*`, `*`color`*`)`

Overrides the default foreground or background color of active hotspots with color *`color`* if *`use_setting`* is `true`.

`buffer:get_hotspot_active_fore()`
`buffer:get_hotspot_active_back()`

Returns the foreground or background color of active hotspots.

Assign Miscellaneous Colors

The user can choose other miscellaneous custom colors.

`buffer.call_tip_fore_hlt = color` (Write-only)
: Specifies color *color* as a call tip's highlighted text foreground color.

`buffer:set_whitespace_fore(use_setting, color)`
`buffer:set_whitespace_back(use_setting, color)`
: Overrides the foreground or background color of whitespace with color *color* if *use_setting* is `true`.

NOTE

These settings are only applicable when `buffer.view_ws` is not `buffer.WS_INVISIBLE`.

`buffer.edge_colour = color`
: Specifies color *color* as the edge or background color for long lines according to `buffer.edge_mode`.

Configure the Display Settings

The user can configure many different display settings for individual views using his or her *~/.textadept/properties.lua* file. These settings are broken down into categories and listed in the following sections.

Configure Caret Display

The user can change how Textadept displays the caret and whether or not the caret can move into virtual space.

`buffer.caret_style = kind`
: Changes the caret's visual representation to caret style *kind*. The default value is `buffer.CARETSTYLE_LINE`, which draws a typical line caret. `buffer.CARETSTYLE_BLOCK` draws a block caret while `buffer.CARETSTYLE_INVISIBLE` does not draw a caret at all.

  The terminal version cannot draw line carets.

`buffer.caret_width = width`
: Specifies pixel width *width* as the line caret's width in in-

sert mode. *width* can only be 0, 1, 2, or 3. The default value is 1.

buffer.caret_period = *period*
: Adjusts the time between caret blinks to *period* milliseconds. The default value is 500.

  The terminal version cannot blink its caret.

buffer.caret_line_visible = *bool*
: Color the background of the line that contains the caret a different color. The default value is true, except for the terminal version, where it is false.

buffer.caret_line_visible_always = *bool*
: Always show the caret line, even when the window is not in focus. The default value is false, showing the line only when the window is in focus.

  The terminal version does not support this option.

buffer.additional_carets_visible = *bool*
: Display additional carets. The default value is true.

buffer.additional_carets_blink = *bool*
: Allow additional carets to blink. The default value is true.

buffer.caret_sticky = *setting*
: Changes the setting for the caret's preferred horizontal position when moving between lines to *setting*. The default value is buffer.CARETSTICKY_OFF, which uses the same position the caret had on the previous line. buffer.CARETSTICKY_ON uses the last position the caret was moved to, and buffer.CARETSTICKY_WHITESPACE uses the position the caret had on the previous line, but prior to any inserted indentation.

buffer.virtual_space_options = *option*
: Changes the virtual space mode to *option*. When virtual space is enabled, the caret may move into the space past end of line characters. The default value is buffer.VS_NONE, which disables virtual space. buffer.VS_USERACCESSIBLE enables virtual space while buffer.VS_RECTANGULARSELECTION enables virtual space only for rectangular selections.

Configure Selection Display

The user can alter how Textadept displays selected text.

`buffer.sel_eol_filled = `*`bool`*
: Extend the selection to the view's right margin. The default value is `false`.

`buffer:hide_selection(`*`hide`*`)`
: Disables the highlighting of selected text if *`hide`* is `true`.

Configure Whitespace Display

The user can vary how Textadept displays whitespace. Normally, tab, space, and end of line characters are invisible.

`buffer.view_ws = `*`mode`*
: Changes the whitespace visibility mode to *`mode`*. The default value is `buffer.WS_INVISIBLE`. `buffer.WS_VISIBLEALWAYS` displays all space characters as dots and tab characters as arrows, while `buffer.WS_VISIBLEAFTERINDENT` displays only non-indentation spaces and tabs.

  The terminal version cannot display tabs as arrows.

`buffer.whitespace_size = `*`pixels`*
: Specifies pixel length *`pixels`* as the width and height of the dots that represent space characters when whitespace is visible. The default value is `1`.

  The terminal version requires *`pixels`* to be `1`.

`buffer.view_eol = `*`bool`*
: Display end of line characters. The default value is `false`.

`buffer.extra_ascent = `*`pixels`*
`buffer.extra_descent = `*`pixels`*
: Extends the amount of pixel padding above or below lines to *`pixels`*. The default values are `0`.

  The terminal version cannot render extra ascent and descent properly.

Configure Scrollbar Display and Scrolling Behavior

The user can modify Textadept's scrolling behavior.

`buffer.h_scroll_bar = bool`
`buffer.v_scroll_bar = bool`
: Display the horizontal and vertical scroll bars. The default values are `true`.

  The terminal version cannot display scroll bars.

`buffer.scroll_width = pixels`
: Adjusts the horizontal scrolling pixel width to *`pixels`*. The default value is `2000`.

  The terminal version does not support this setting.

`buffer.scroll_width_tracking = bool`
: Continuously update the horizontal scrolling width to match the maximum width of a displayed line beyond `buffer.scroll_width`. The default value is `false`.

`buffer.end_at_last_line = bool`
: Disable scrolling past the last line. The default value is `true`.

`buffer:set_x_caret_policy(policy, x)`
`buffer:set_y_caret_policy(policy, y)`
: Defines scrolling policy bit-mask *`policy`* as the policy for keeping the caret *`x`* number of pixels away from the horizontal margins or *`y`* number of lines away from the vertical margins. *`policy`* is an additive combination of the flags listed in Table 11.

  The terminal version's unit of measure is a character instead of a pixel or line.

`buffer:set_visible_policy(policy, y)`
: Defines scrolling policy bit-mask *`policy`* as the policy for keeping the caret *`y`* number of lines away from the vertical margins as `buffer:ensure_visible_enforce_policy()` redisplays hidden or folded lines. *`policy`* is an additive combination of the `buffer.VISIBLE_SLOP` and `buffer.VISIBLE_STRICT` flags, which are similar in function to the flags listed in Table 11.

  The terminal version's unit of measure is a character instead of a line.

Table 11. Caret policy flags

Bit Flag	Description
`buffer.CARET_SLOP`	When the caret goes out of view, scroll the view so the caret is *x* pixels from the right margin or *y* lines below the top margin.
`buffer.CARET_STRICT`	Scroll the view to ensure the caret stays *x* pixels away from the right margin or *y* lines below the top margin.
`buffer.CARET_EVEN`	Consider both horizontal margins instead of just the right one for *x* or consider both vertical margins instead of just the top one for *y*.
`buffer.CARET_JUMPS`	Scroll the view more than usual in order to scroll less often.

Configure Mouse Cursor Display

The user can choose which mouse cursor Textadept displays from the list in Table 12.

Table 12. Mouse cursor types

Type	Description
`buffer.CURSORNORMAL`	The text insert cursor.
`buffer.CURSORARROW`	The arrow cursor.
`3`	The up arrow cursor.
`buffer.CURSORWAIT`	The wait cursor.
`5`	The arrow cursor.
`6`	The arrow cursor.
`buffer.CURSORREVERSEARROW`	The reversed arrow cursor.
`8`	The link cursor.

`buffer.cursor = type`
: Displays mouse cursor type *type*. The default value is `buffer.CURSORNORMAL`, which displays the text insert cursor. Table 12 lists all available cursor types.

  The terminal version cannot change the mouse cursor.

Configure Wrapped Line Display

The user can specify if and how Textadept displays wrapped lines. By default, lines that contain more characters than the view can show do not wrap into view and onto sub-lines.

`buffer.wrap_mode = mode`
: Changes the long line wrap mode to *mode*. The default value is `buffer.WRAP_NONE`. `buffer.WRAP_WORD` wraps long lines at word boundaries and `buffer.WRAP_CHAR` wraps them at character boundaries.

`buffer.wrap_visual_flags = mode`
: Changes the visual flag display mode for wrapped lines to *mode*. The default value is `buffer.WRAPVISUALFLAG_NONE`. `buffer.WRAPVISUALFLAG_END` shows a visual flag at the end of a wrapped line, `buffer.WRAPVISUALFLAG_START` shows the flag at the beginning of a sub-line, and `buffer.WRAPVISUALFLAG_MARGIN` shows the flag in the sub-line's line number margin.

  The terminal version cannot draw visual wrap flags.

`buffer.wrap_visual_flags_location = mode`
: Changes the visual flag drawing mode for wrapped lines to *mode*. The default value is `buffer.WRAPVISUALFLAGLOC_DEFAULT`, which draws a visual flag near the view's right margin. `buffer.WRAPVISUALFLAGLOC_END_BY_TEXT` draws the flag near text at the end of a wrapped line, and `buffer.WRAPVISUALFLAGLOC_START_BY_TEXT` draws the flag near text at the beginning of a sub-line.

`buffer.wrap_start_indent = spaces`
: Displays wrapped lines with *spaces* number of spaces of indentation if `buffer.wrap_indent_mode` is `buffer.WRAPINDENT_FIXED`. The default value is `0`.

`buffer.wrap_indent_mode = mode`
: Changes the indent mode for wrapped lines to *mode*. The default value is `buffer.WRAPINDENT_FIXED`, which indents wrapped lines by `buffer.wrap_start_indent`. `buffer.WRAPINDENT_SAME` indents wrapped lines by the same amount as the first line, and `buffer.WRAPINDENT_INDENT` indents wrapped lines by one more level than the level of the first line.

Configure Text Zoom

The user can temporarily increase or decrease the font size in a view.

NOTE

The terminal version does not support text zoom.

`buffer:zoom_in()`
`buffer:zoom_out()`
: Increases or decreases the size of all fonts by one point, up to 20 or down to -10, respectively.

`buffer.zoom = points`
: Adds *`points`* number of points to the size of all fonts. Negative values are allowed. The default value is `0`.

Configure Long Line Display

The user can configure Textadept to visually identify long lines, since the editor does not enforce a maximum line length.

`buffer.edge_column = column`
: Specifies column number *`column`* as the column to mark long lines at.

`buffer.edge_mode = mode`
: Changes the long line mark mode to *`mode`*. The default value is `buffer.EDGE_NONE`, which does not mark long lines. `buffer.EDGE_LINE` draws a vertical line at column number `buffer.edge_column`, and `buffer.EDGE_BACKGROUND` changes the background color of text after `buffer.edge_column`.

  The terminal version cannot draw edge lines properly.

Configure Fold Settings and Folded Line Display

The user can specify how Textadept folds lines and how it displays lines that contain fold points.

`buffer.property['fold'] = '0'`
: Disables code folding.

`buffer.property['fold.by.indentation'] = '1'`
: Utilizes changes in indentation to fold plain text and source code that Textadept does not know how to fold.

`buffer.property['fold.line.comments'] = '1'`
: Allows the user to fold groups of source code line comments for the lexers that support it.

`buffer.fold_flags = options`
: Specifies fold line bit-mask *`options`* as the set of fold lines to draw on lines that contain fold points. The default value is `buffer.FOLDFLAG_LINE_AFTER_CONTRACTED`, which draws lines below collapsed folds. `buffer.FOLDFLAG_LINE_BEFORE_CONTRACTED` draws lines above collapsed folds, `buffer.FOLDFLAG_LINE_BEFORE_EXPANDED` draws lines above expanded folds, and `buffer.FOLDFLAG_LINE_BEFORE_EXPANDED` draws lines below expanded folds.

  The terminal version does not draw fold lines properly.

Configure Highlighted Matching Brace Display

The user can decide which brace characters to highlight, if any.

`textadept.editing.HIGHLIGHT_BRACES = bool`
: Highlight matching brace characters like parentheses, brackets, and curly braces. Highlighting uses the `textadept.editing.INDIC_BRACEMATCH` indicator. The default value is `true`.

`textadept.editing.braces[byte] = true`
`textadept.editing.braces.lexer[byte] = true`
: Highlights the character represented by byte value *`byte`* as a brace character globally or in language *`lexer`*. The default byte values are `40` (‘(’), `41` (‘)’), `91` (‘[’), `93` (‘]’), `123` (‘{’), and `125` (‘}’).

Configure Indentation Guide Display

The user can choose how Textadept displays indentation guides and which guide to highlight.

`buffer.indentation_guides = mode`
: Changes the indentation guide drawing mode to *`mode`*. The default value is `buffer.IV_LOOKBOTH`, which draws

guides above and below the current line up to either the indentation level of the previous or next non-empty line, whichever is greater. `buffer.IV_LOOKFORWARD` draws guides below the current line up to the next non-empty line's indentation level, but with an additional level if the previous non-empty line is a fold point. `buffer.IV_REAL` draws guides only within indentation whitespace. `buffer.IV_NONE` does not draw any guides.

The terminal version cannot draw indentation guides.

`buffer.highlight_guide = column`
: Highlights the indentation guide at column number *`column`*. When `0`, stops highlighting.

Configure Hotspot Display

The user can change how Textadept displays hotspots (regions of clickable text that behave like web browser links).

`buffer.hotspot_active_underline = bool`
: Underline active hotspots. The default value is `true`.

`buffer.hotspot_single_line = bool`
: Limit hotspots to a single line. The default value is `true`.

Configure Textadept's Window

The user can alter Textadept's window title, state, menus, and status.

`ui.title = title` (Write-only)
: Changes Textadept's window title to string *`title`*.

`ui.size = {width, height}`
: Resizes Textadept's window to pixel width *`width`* and pixel height *`height`*.

The terminal version cannot alter its window size.

`ui.maximized = bool`
: Whether or not Textadept's window is maximized.

`textadept.menu.set_menubar(menubar)`
: Replaces Textadept's menubar with menu list *`menubar`*. Menus have `title` keys for their title text. Each menu item is a table that contains a label and a Lua command.

`ui.tabs = bool`
: Whether or not to display the tab bar when multiple buffers are open. The default value is `true`.

`buffer.tab_label = label`
: Changes the buffer's label in the tab bar to string *`label`*.

`textadept.menu.set_contextmenu(menu)`
: Replaces the buffer's context menu with menu item list *`menu`*. Each menu item is a table that contains a label and a Lua command.

`view.size = size`
: Changes the split resizer's pixel position to *`size`* if the view is a split one.

`ui.statusbar_text = text` (Write-only)
`ui.bufstatusbar_text = text` (Write-only)
: Displays string *`text`* in the statusbar or buffer statusbar.

Configure File Types

The user's *~/.textadept/init.lua* file may add new file types or alter Textadept's existing ones. Textadept recognizes a wide range of programming language files by their file extensions, by a keyword in their shebang lines, or by a Lua pattern that matches the text on their first lines.

`buffer:set_lexer([lexer])`
: Associates language *`lexer`* or the auto-detected language with the buffer and then loads the appropriate language module if that module exists.

`textadept.file_types.extension[ext] = lexer`
: Associates string file extension *`ext`* with language *`lexer`*.

`textadept.file_types.shebangs[word] = lexer`
: Associates string *`word`*, found in a file's shebang line (e.g. "#!/usr/bin/lua"), with language *`lexer`*.

`textadept.file_types.patterns[patt] = lexer`
: Associates pattern *`patt`* with language *`lexer`*. Textadept attempts to match *`patt`* against the first line of a file.

`textadept.file_types.select_lexer()`
: Prompts the user to select a language for the current buffer.

Query File Type Information

The user can obtain the current buffer's language, and learn of all the languages Textadept knows about.

`buffer:get_lexer(`*`current`*`)`
: Returns the buffer's language. If *`current`* is `true`, returns the language under the caret in a multiple-language lexer.

`textadept.file_types.lexers`
: The list of available languages. The section "Define a Lexer" on page 109 describes how to create new lexers for languages.

Configure Key Bindings

The user's *~/.textadept/init.lua* file may define custom *key bindings* or redefine Textadept's existing ones. A key binding consists of a *key sequence* assigned to either a *command* or *key chain* in the global `keys` table. The user may group key bindings into *language-specific keys* or *key modes*. Textadept supplies key bindings for many of its API functions.

NOTE

The terminal version can only recognize a subset of key sequences, which varies by platform.

Key Sequence
: A string built from an ordered combination of modifier keys and a key's inserted character. Table 13 on page 105 lists all available modifier keys. For unprintable or special keys, Textadept refers to the `keys.KEYSYMS` lookup table represented in Table 14 on page 105. Example 22 shows some sample key sequences.

Example 22. Sample key sequences

```
keys['ca']        -- Ctrl+A
keys['caa']       -- Ctrl+Alt+A
keys['caA']       -- Ctrl+Alt+Shift+A
keys['cs\t']      -- Ctrl+Shift+Tab
keys['c<']        -- Ctrl+Shift+Comma (U.S. English)
keys['cright']    -- Ctrl+Right Arrow
```

Command
: Either a Lua function by itself or a table that contains a Lua function along with a set of arguments to be passed. Example 23 shows some sample global key command bindings.

Example 23. Sample global key command bindings

```
keys['cn'] = buffer.new
keys['cs'] = buffer.save
keys['c('] = {textadept.editing.enclose, '(', ')'}
keys['cu'] = function() io.snapopen(_USERHOME) end
```

Key Chain
: A table of additional key bindings. Key chains allow the user to assign multiple key bindings to one prefix key sequence. By default, the Escape key cancels a key chain. Example 24 shows a sample global key chain binding.

Example 24. Sample global key chain binding

```
keys['af'] = {
  ['n'] = buffer.new,
  ['o'] = io.open_file,
  ['s'] = buffer.save,
  ['q'] = quit
}
```

Language-specific Keys
: Key bindings that are only active when editing source code in a particular programming language. Example 25 shows a sample language-specific key. Language modules typically define language-specific keys.

Example 25. Sample language-specific key

```
if not keys.cpp then keys.cpp = {} end
keys.cpp['\n'] = function()
  buffer:line_end()
  buffer:add_text(';')
  buffer:new_line()
end
```

Key Mode
: A group of key bindings such that when a key mode is

active, Textadept ignores all key bindings defined outside that mode until that mode is unset. Example 26 shows a sample key mode.

Example 26. Sample key mode

```
keys.command_mode = {
  ['h'] = buffer.char_left,
  ['j'] = buffer.line_up,
  ['k'] = buffer.line_down,
  ['l'] = buffer.char_right,
  ['i'] = function()
    keys.MODE = nil
    ui.statusbar_text = 'INSERT MODE'
  end
}
```

Key binding precedence is as follows:

1. Bindings in the current key mode.
2. Language-specific bindings.
3. Global bindings.

If a language-specific binding returns the boolean value `false`, Textadept also runs the applicable global binding if it exists.

Table 13. Modifier keys for key sequences

Modifier	Win32/Linux	Mac OSX	Terminal
Control	'c'	'c'	'c'
Alt	'a'	'a'	'm'
Command (Meta)		'm'	
Shift	's'	's'	's'

Table 14. Unprintable or special keys for key sequences

Key	Representation	Key	Representation
Tab	'\t'	Keypad Left	'kpleft'
Bksp	'\b'	Keypad Right	'kpright'
Enter	'\n'	Keypad Up	'kpup'
Esc	'esc'	Keypad Down	'kpdown'

Key	Representation	Key	Representation
Left	'left'	Keypad Home	'kphome'
Right	'right'	Keypad End	'kpend'
Up	'up'	Keypad PgUp	'kppgup'
Down	'down'	Keypad PgDn	'kppgdn'
Home	'home'	Keypad Add	'kpadd'
End	'end'	Keypad Subtract	'kpsub'
PgUp	'pgup'	Keypad Multiply	'kpmult'
PgDn	'pgdn'	Keypad Divide	'kpdiv'
Del	'del'	Keypad Decimal	'kpdec'
Ins	'ins'	Keypad 0-9	'kp0' - 'kp9'
F1-F12	'f1' - 'f12'		

`keys[`*`key`*`] = `*`command`*
`keys.`*`lexer`*`[`*`key`*`] = `*`command`*
: Binds command *`command`* to key sequence *`key`* globally or in language *`lexer`*.

`keys[`*`key`*`] = {`*`key1 = command1, key2 = command2,`* `...}`
`keys.`*`lexer`*`[`*`key`*`] = {`*`key1 = command1, key2 = command2,`* `...}`
: Creates a global or *`lexer`*-specific key chain with prefix key sequence *`key`*.

`keys.`*`mode`* `= {`*`key1 = command1, key2 = command2,`* `...}`
: Creates a set of key bindings that apply only to mode name *`mode`*.

Configure Key Settings

The user's *~/.textadept/init.lua* file may redefine the key sequence that clears a key chain, change the key sequence reserved for language-specific keys, and specify additional unprintable keys.

`keys.MODE = `*`mode`*
: Changes the current key mode to mode name *`mode`*, or clears the current key mode if *`mode`* is `nil`. The default value is `nil`.

`keys.CLEAR = key`
: Defines key sequence *key* as the sequence that clears the current key chain. *key* cannot be part of a key chain. The default value is `'esc'`.

`keys.LANGUAGE_MODULE_PREFIX = key`
: Reserves key sequence *key* as the prefix for language modules' key chains. The default value is `'cl'` on Windows and Linux platforms, `'ml'` on the Mac OSX platform, and `'ml'` for the terminal version.

`keys.KEYSYMS[code] = key`
: Assigns string representation *key* to numeric key code *code*. Table 14 lists all default representations.

Configure Snippets

The user's *~/.textadept/init.lua* file may define text *snippets*. A snippet consists of a *trigger word* assigned to *snippet text* in the global `snippets` table. The user may group snippets into *language-specific snippets*.

Trigger Word
: A word the user types before manually prompting Textadept to insert the word's assigned snippet text.

Snippet Text
: A simple plain text fragment or a dynamic source code template that contains placeholders, mirrors, transforms, and executable code. Table 15 lists all special characters in snippet text. Example 27 shows some sample global snippets.

Example 27. Sample global snippets

```
snippets['foo'] = 'foobar%1(baz)'
snippets['bar'] = 'start\n\t%0\nend'
snippets['baz'] = '%1(mirror), %1, on the wall'
snippets['add'] = '%1(1) + %2(2) = %3<%1 + %2>'
snippets['env'] = '$%1(HOME) = %2[echo $%1]'
```

Language-specific Snippets
: Snippets that are only available when editing source code in a particular programming language. Example 28 shows some sample language-specific snippets. Lan-

guage modules typically define language-specific snippets.

Example 28. Sample language-specific snippets

```
snippets.lua = {
  l = "local %1(variable)%2( = %3(value))",
  p = "print(%0)",
  f = "function %1(name)(%2(args))\n\t%0\nend"
}
```

Snippet precedence is as follows:

1. Language-specific snippets.
2. Global snippets.

Table 15. Special characters in snippet text

Characters	Meaning
%*n*(*text*)	Represents an *n*-numbered placeholder with default text *text*. The caret moves to placeholders in numeric order, finishing at "%0" or the end of the snippet.
%*n*	Represents an *n*-numbered mirror that mirrors any user input into the *n*-numbered placeholder. If no *n*-numbered placeholder exists, the first *n*-numbered mirror becomes the placeholder, but with no default text.
%*n*<*Lua*> %*n*[*Shell*]	Represents an *n*-numbered transform that executes Lua code *Lua* or shell code *Shell* when the caret visits the *n*-numbered placeholder. If the transform omits *n*, the code is executed upon snippet insertion. *Lua* runs within Textadept's Lua State and inserts the code's return value. *Lua* may use a temporary `selected_text` global variable that contains the currently selected text. *Shell* runs within the platform's shell environment and inserts any standard output.
%%	Represents a single '%' in the instances where '%' by itself has a special meaning.
\t	A single level of indentation based on the buffer's indentation settings.
\n	A single set of end of line characters based on the buffer's end of line mode.

`snippets[`*`trigger`*`] = `*`text`*
`snippets.`*`lexer`*`[`*`trigger`*`] = `*`text`*
: Assigns trigger word *`trigger`* to snippet text *`text`* globally or in language *`lexer`*. Table 15 lists all special characters in snippet text.

Configure Miscellaneous Settings

The user's *~/.textadept/properties.lua* file may change other various settings.

`buffer.read_only = `*`bool`*
: Whether or not the buffer is read-only. The default value is `false`.

`buffer.multiple_selection = `*`bool`*
: Enable multiple selection. The default value is `true`.

`buffer.additional_selection_typing = `*`bool`*
: Type into multiple selections. The default value is `true`.

`buffer.mouse_dwell_time = `*`time`*
: Specifies integer *`time`* as the number of milliseconds the mouse must idle before generating an `events.DWELL_START` event. A time of `buffer.TIME_FOREVER` will never generate one.

`buffer.undo_collection = `*`bool`*
: Collect undo information. The default value is `true`. When setting to `false`, call `buffer:empty_undo_buffer()` to synchronize the undo buffer with the buffer text.

Define a Lexer

Textadept employs a lexer to highlight the syntax of source code written in a particular programming language. Lexers reside in the user's *~/.textadept/lexers/* directory. Each lexer is a single Lua file composed of lexer options, LPeg[5] patterns, *tokens*, *rules*, token style assignments, and optional fold point definitions. Lexers may be *embedded* within one another. All of these aspects are described in the following sections. Example 5 on page 7 shows a simple lexer for a non-existent language.

5 *http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html*

Declare the Lexer Configuration

The user can alter a lexer's properties and thus affect how a lexer operates.

lexer._NAME = *name*
: Identifies lexer *lexer* by string *name*. This is mandatory.

lexer._LEXBYLINE = true
: Indicates lexer *lexer* can only process one whole line of text (instead of an arbitrary chunk of text) at a time.

lexer.property['fold.by.indentation'] = '1'
: Declares that the lexer being defined does not define fold points and that fold points should be calculated based on changes in indentation.

Construct Patterns

The user can construct LPeg patterns that match the programming language's syntax by using combinations of the simple and complex patterns listed in the next three sections.

Simple Patterns

The user can utilize a number of simple, predefined LPeg patterns.

lpeg.P(*string*)
: Returns a pattern that matches string *string* literally.

lpeg.P(*n*)
: Returns a pattern that matches any *n* characters.

lpeg.S(*set*)
: Returns a pattern that matches any character in string set *set*.

lpeg.R('*xy*')
: Returns a pattern that matches any character in the range of characters *x* and *y*, including *x* and *y*.

lexer.lower
lexer.upper
: A pattern that matches any lower case character ('a'-'z') or upper case character ('A'-'Z').

`lexer.alpha`
`lexer.alnum`
: A pattern that matches any alphabetic character ('A'-'Z', 'a'-'z') or alphanumeric character ('A'-'Z', 'a'-'z', '0'-'9').

`lexer.word`
: A pattern that matches a typical word. Words begin with a letter or underscore, and consist of alphanumeric and underscore characters.

`lexer.digit`
`lexer.xdigit`
: A pattern that matches any digit ('0'-'9') or hexadecimal digit ('0'-'9', 'A'-'F', 'a'-'f').

`lexer.dec_num`
`lexer.hex_num`
`lexer.oct_num`
: A pattern that matches a decimal, hexadecimal, or octal number.

`lexer.integer`
: A pattern that matches either a decimal, hexadecimal, or octal number.

`lexer.float`
: A pattern that matches a floating point number.

`lexer.space`
: A pattern that matches any whitespace character ('\t', '\v', '\f', '\n', '\r', ' ').

`lexer.newline`
`lexer.nonnewline`
: A pattern that matches any set of end of line characters or any single, non-newline character.

`lexer.nonnewline_esc`
: A pattern that matches any single, non-newline character or any set of end of line characters escaped with '\'.

`lexer.cntrl`
: A pattern that matches any control character (ASCII codes 0 to 31).

`lexer.graph`
: A pattern that matches any graphical character ('!' to '~').

`lexer.punct`
: A pattern that matches any punctuation character ('!' to '/', ':' to '@', '[' to '`', '{' to '~').

`lexer.print`
: A pattern that matches any printable character (' ' to '~').

`lexer.ascii`
`lexer.extend`
: A pattern that matches any ASCII character (codes 0 to 127) or ASCII extended character (codes 0 to 255).

`lexer.any`
: A pattern that matches any single character.

Complex Patterns

The user can harness a number of functions to generate more complex, language-specific LPeg patterns.

`lexer.word_match(`*`words`*`[, `*`word_chars`*`[, `*`case_insensitive`*`]])`
: Returns a pattern that matches any single word in list *`words`*. Words consist of alphanumeric and underscore characters, as well as the characters in string set *`word_chars`*. *`case_insensitive`* indicates whether or not to ignore case when matching words.

`lexer.delimited_range('`*`x`*`'[, `*`single_line`*`[, `*`no_escape`*`[, `*`balanced`*`]]])`
`lexer.delimited_range('`*`xy`*`'[, `*`single_line`*`[, `*`no_escape`*`[, `*`balanced`*`]]])`
: Returns a pattern that matches a range of text bounded by character *`x`* or a range of text that starts with *`x`* and ends with *`y`*. *`single_line`* indicates whether or not the range must be on a single line, *`no_escape`* indicates whether or not to ignore '\' as an escape character, and *`balanced`* indicates whether or not to handle balanced ranges like parentheses. If *`balanced`* is `true`, *`y`* must be given.

`lexer.nested_pair(`*`start_chars`*`, `*`end_chars`*`)`
: Returns a pattern that matches a balanced range of text that starts with string *`start_chars`* and ends with string *`end_chars`*.

`lexer.starts_line(`*`patt`*`)`
: Returns a pattern that matches pattern *`patt`* only at the

beginning of a line.

`lexer.last_char_includes(s)`
: Returns a pattern that verifies that string set *`s`* contains the first non-whitespace character behind the current match position.

Pattern Operators

The user can combine LPeg patterns with pattern operators.

`patt^n`
: Returns a pattern that matches at least *`n`* repetitions of pattern *`patt`*.

`patt^-n`
: Returns a pattern that matches at most *`n`* repetitions of pattern *`patt`*.

`patt1 * patt2`
: Returns a pattern that matches pattern *`patt1`* followed by pattern *`patt2`*.

`patt1 + patt2`
: Returns a pattern that matches pattern *`patt1`* or pattern *`patt2`*.

`patt1 - patt2`
: Returns a pattern that matches pattern *`patt1`* if pattern *`patt2`* does not match.

`-patt`
: Returns a pattern that matches anything other than pattern *`patt`*, but does not advance the match position.

`#patt`
: Returns a pattern that matches pattern *`patt`*, but does not advance the match position.

Define Tokens

The user can group LPeg patterns into *tokens*. A token consists of a token name and an associated pattern. Table 16 lists Textadept's predefined token names. The user may also choose his or her own.

Table 16. Predefined token names

lexer.CLASS	lexer.COMMENT	lexer.CONSTANT
lexer.DEFAULT	lexer.ERROR	lexer.FUNCTION
lexer.IDENTIFIER	lexer.KEYWORD	lexer.LABEL
lexer.NUMBER	lexer.OPERATOR	lexer.PREPROCESSOR
lexer.REGEX	lexer.STRING	lexer.TYPE
lexer.VARIABLE	lexer.WHITESPACE	

`lexer.token(`*`name`*`, `*`patt`*`)`
: Returns a token pattern with token name *`name`* and pattern *`patt`*. If *`name`* is not one of the predefined token names in Table 16, its style must be defined in the lexer's `_tokenstyles` table. Example 5 on page 7 shows some sample token definitions.

Define Rules

The user can build the lexer's grammar by defining its *rules*. A rule consists of a rule name and an associated pattern.

`lexer``._rules = {{`*`name1`*`, `*`patt1`*`}, {`*`name2`*`, `*`patt2`*`}, ...}`
: Defines *`patt1`*, *`patt2`*, ..., *`pattN`* as the patterns of tokens composing lexer *`lexer`*. Their order matters, since *`lexer`* matches them sequentially. Arbitrary strings *`name1`*, *`name2`*, ..., *`nameN`* serve only to identify their respective patterns. Example 5 on page 7 shows a sample `_rules` list.

Assign Styles

The user can assign token styles to custom token names for syntax highlighting. Table 17 lists Textadept's predefined token styles. The user may also define his or her own. The section "Define and Assign Styles" on page 87 describes how to define styles.

NOTE

There is no need to assign styles to Textadept's predefined tokens because the editor does it automatically.

Table 17. Predefined token styles

lexer.STYLE_CLASS	lexer.STYLE_COMMENT
lexer.STYLE_CONSTANT	lexer.STYLE_EMBEDDED
lexer.STYLE_ERROR	lexer.STYLE_FUNCTION
lexer.STYLE_IDENTIFIER	lexer.STYLE_KEYWORD
lexer.STYLE_LABEL	lexer.STYLE_NUMBER
lexer.STYLE_OPERATOR	lexer.STYLE_PREPROCESSOR
lexer.STYLE_REGEX	lexer.STYLE_STRING
lexer.STYLE_TYPE	lexer.STYLE_VARIABLE
lexer.STYLE_WHITESPACE	

lexer._tokenstyles = {*token1* = *style1*, *token2* = *style2*, ...}
: Assigns styles *style1*, *style2*, ..., *styleN* to non-predefined token names *token1*, *token2*, ..., *tokenN*, respectively, for lexer *lexer*. Example 5 on page 7 shows a sample _tokenstyles table.

Specify Fold Points

The user can specify source code fold points with a high degree of granularity.

lexer._foldsymbols = {_patterns = {...}, *token1* = {*string1* = *value1*, *string2* = *value2*, ...}, *token2* = {...}, ...}
: Designates *string1*, *string2*, ..., *stringN* within the tables assigned to token names *token1*, *token2*, ..., *tokenN* as fold points for lexer *lexer*. _patterns must contain patterns that match *string1*, *string2*, ..., *stringN*. *value1*, *value2*, ..., *valueN* are integers of value 1, 0, or -1. 1 indicates a beginning fold point, 0 indicates a non-fold point, and -1 indicates an ending fold point. Example 5 on page 7 shows a sample _foldsymbols table.

  value1, *value2*, ..., *valueN* may also be functions returning 1, 0, or -1. The signature for these functions is:

```
function(text, pos, line, s, match)
```

  text is the text to identify fold points in, pos is the begin-

ning position of the current line in `text`, `line` is the current line's text, `s` is the position in the current line the matched text starts at, and `match` is the matched text itself.

`lexer.fold_line_comments(`*`prefix`*`)`
: Returns a fold function (to be used within the lexer's `_foldsymbols` table) that folds consecutive line comments that start with string *`prefix`*.

Embed Lexers

The user can *embed* lexers within one another after loading the appropriate lexers and defining their start and end rules. The resultant multiple-language lexer can highlight the source code of each of its component languages. For example, the RHTML lexer can highlight HTML, CSS, Javascript, and Ruby, all within the same file.

`lexer.load(`*`name`*`[, `*`alt_name`*`])`
: Loads and returns the lexer of string name *`name`*. *`alt_name`* is the alternate name to give the lexer.

`lexer.embed_lexer(`*`parent, child, start_rule, end_rule`*`)`
: Embeds child lexer *`child`* in parent lexer *`parent`* using patterns *`start_rule`* and *`end_rule`*, which signal the beginning and end of the embedded lexer, respectively.

`lexer``._RULES[`*`name`*`] = `*`patt`*
: Reassigns pattern *`patt`* to rule name *`name`* in embedded lexer *`lexer`*.

Query Lexer Properties and Rules

The user can fetch various lexer properties and rules.

`lexer.property[`*`key`*`]`
: The string value assigned to string *`key`*.

`lexer.property_int[`*`key`*`]` (Read-only)
: The integer value of the string assigned to string *`key`*, or `0` if *`key`* does not exist.

`lexer``._RULES['`*`name`*`']`
: The pattern for rule name *`name`* in embedded lexer *`lexer`*.

Handle Lexer Events

The user can connect to the following lexer event.

`events.LEXER_LOADED` *`lexer`*
: Emitted after loading a lexer. *`lexer`* is that lexer's name.

Manually Style Text

Textadept allows the user to refresh source code syntax highlighting and fold point markers. Additionally, he or she can manually style plain text.

Refresh Styling

The user can tell the lexer to reprocess a range of text if that range has incorrect highlighting or incorrect fold points.

`buffer:colourise(`*`start_pos`*`, `*`end_pos`*`)`
: Instructs the lexer to style and mark fold points in the range of text between positions *`start_pos`* and *`end_pos`*. If *`end_pos`* is `-1`, styles and marks to the end of the buffer.

Assign Plain Text Styles

The user can apply custom fonts, colors, and attributes to plain text styles. Textadept offers 256 styles, numbered from 0 to 255, to style text with.

`buffer:style_reset_default()`
: Resets style number `buffer.STYLE_DEFAULT` to its initial state.

`buffer:style_clear_all()`
: Reverts all styles to having the same properties as style number `buffer.STYLE_DEFAULT`.

TIP

The user can have all styles inherit from the style whose number is `buffer.STYLE_DEFAULT` by first setting that style's properties and then calling `buffer:style_clear_all()`.

`buffer.style_font[`*`style`*`] = `*`font_name`*

Specifies string *`font_name`* as the font name of text with style number *`style`*.

The terminal cannot change font names.

`buffer.style_size[`*`style`*`] = `*`size`*

Specifies integer *`size`* as the font size of text with style number *`style`*.

The terminal version cannot adjust font size.

`buffer.style_fore[`*`style`*`] = `*`color`*
`buffer.style_back[`*`style`*`] = `*`color`*

Specifies color *`color`* as the foreground or background color of text with style number *`style`*.

`buffer.style_bold[`*`style`*`] = `*`bool`*
`buffer.style_italic[`*`style`*`] = `*`bool`*
`buffer.style_underline[`*`style`*`] = `*`bool`*

Whether or not text with style number *`style`* is bold, italic, or underlined. The default values are `false`.

The terminal version cannot set an italic attribute.

`buffer.style_eol_filled[`*`style`*`] = `*`bool`*

Whether or not the background color of text with style number *`style`* extends all the way to the view's right margin. The default value is `false`.

`buffer.style_case[`*`style`*`] = `*`mode`*

Changes the letter case mode of text with style number *`style`* to *`mode`*. The default value is `buffer.CASE_MIXED`, which displays text in normal, mixed case. `buffer.CASE_UPPER` displays text in upper case while `buffer.CASE_LOWER` displays text in lower case.

`buffer.style_visible[`*`style`*`] = `*`bool`*

Whether or not text with style number *`style`* is visible. The default value is `true`.

`buffer.style_changeable[`*`style`*`] = `*`bool`*

Whether or not text with style number *`style`* is changeable. The default value is `true`.

`buffer.style_hot_spot[`*`style`*`] = `*`bool`*

Whether or not text with style number *`style`* is clickable. The default value is `false`. Hotspots behave like web

browser links.

The terminal version does not support hotspots.

Style Plain Text

The user can style plain text and store line state information.

`buffer:clear_document_style()`
: Clears all styling and folding information.

`buffer:start_styling(`*`pos`*`, buffer.STYLE_MAX)`
: Begins styling at position *`pos`*.

`buffer:set_styling(`*`length`*`, `*`style`*`)`
: Assigns style number *`style`* to the next *`length`* characters, starting from the current styling position, and increments the styling position by *`length`*.

`buffer.line_state[`*`line`*`] = `*`state`*
: Stores integer *`state`* in line number *`line`*. Line states are unaffected by styling changes.

Query Style Information

The user can retrieve styling information for positions and lines.

`buffer.style_at[`*`pos`*`]` (Read-only)
: The style number at position *`pos`*.

`buffer.style_name[`*`style`*`]` (Read-only)
: The name of style number *`style`*.

`buffer.end_styled` (Read-only)
: The current styling position or the last correctly styled character's position.

`buffer.line_state[`*`line`*`]`
: The integer line state for line number *`line`*.

`buffer.max_line_state` (Read-only)
: The last line number with a non-zero line state.

Handle Hotspot Style Events

The user can connect to the following hotspot style click events.

`events.HOTSPOT_CLICK` *`position, modifiers`*
`events.HOTSPOT_DOUBLE_CLICK` *`position, modifiers`*
: Emitted when clicking or double-clicking on text that is in a style that has the hotspot attribute set. *`position`* is the clicked or double-clicked text's position and *`modifiers`* is a bit-mask of any modifier keys used (`buffer.MOD_CTRL` for Control, `buffer.MOD_SHIFT` for Shift, `buffer.MOD_ALT` for Alt, and `buffer.MOD_META` for Command).

  The terminal version cannot detect mouse clicks.

`events.HOTSPOT_RELEASE_CLICK` *`position`*
: Emitted when releasing the mouse after clicking on text that is in a style that has the hotspot attribute set. *`position`* is the clicked text's position.

  The terminal version cannot detect mouse releases.

Miscellaneous

Textadept provides many other miscellaneous facilities.

`timeout(`*`interval, f`*`[, ...])`
: Calls function *`f`* with the given arguments after *`interval`* seconds. If *`f`* returns `true`, calls *`f`* repeatedly every *`interval`* seconds as long as *`f`* returns `true`.

`reset()`
: Resets the Lua state by reloading all initialization scripts. `arg` is set to `nil` during this process.

`quit()`
: Emits an `events.QUIT` event and, unless any handler returns `false`, quits Textadept.

`buffer:edit_toggle_overtype()`
: Toggles overtype mode, where typed characters overwrite existing ones.

`buffer:cancel()`
: Cancels the active selection mode, autocompletion or

user list, call tip, etc.

`buffer:choose_caret_x()`
: Identifies the current horizontal caret position as the caret's preferred horizontal position when moving between lines.

`buffer:toggle_caret_sticky()`
: Toggles between `buffer.caret_sticky` option settings `buffer.CARETSTICKY_ON` and `buffer.CARETSTICKY_OFF`.

`buffer:set_save_point()`
: Indicates the buffer has no unsaved changes.

Handle Miscellaneous Events

The user can connect to the following miscellaneous events.

`events.ARG_NONE`
: Emitted when no command line arguments are passed to Textadept on startup.

`events.DOUBLE_CLICK` *`position, line, modifiers`*
: Emitted after double-clicking the mouse button. *`position`* is the position double-clicked, *`line`* is the line number of *`position`*, and *`modifiers`* is a bit-mask of any modifier keys used (`buffer.MOD_CTRL` for Control, `buffer.MOD_SHIFT` for Shift, `buffer.MOD_ALT` for Alt, and `buffer.MOD_META` for Command).

  The terminal version cannot detect mouse clicks.

`events.DWELL_START` *`position`*
: Emitted when the mouse is stationary for `buffer.mouse_dwell_time` milliseconds. *`position`* is the position closest to the mouse.

  The terminal version cannot detect mouse movements.

`events.DWELL_END` *`position`*
: Emitted after `events.DWELL_START` when the user moves the mouse, presses a key, or scrolls the view. *`position`* is the position closest to the mouse.

`events.ERROR` *`text`*
: Emitted when an error occurs. *`text`* is the error message text.

`events.KEYPRESS` *`code, shift, ctrl, alt, meta`*
: Emitted when pressing a key. *`code`* is the numeric key code and *`shift`*, *`ctrl`*, *`alt`*, and *`meta`* indicate whether or not those modifier keys (Shift, Control, Alt, and Command, respectively) are pressed.

`events.INITIALIZED`
: Emitted after Textadept finishes initializing.

`events.QUIT`
: Emitted when quitting Textadept. When connecting to this event, connect with an index of `1` or the handler will be ignored.

`events.RESET_BEFORE`
`events.RESET_AFTER`
: Emitted before or after resetting the Lua state.

`events.UPDATE_UI`
: Emitted after the view is visually updated.

Appendix: Image Formats

Textadept supports two image formats for marker and interactive list images: XPM and RGBA.

XPM Image Format

Textadept's XPM format is a C-style string array within a Lua string. The following components make up that string array.

`/* XPM */`
`static char *`*`name`*`[] = {`
: Declares arbitrary string *`name`* as the image's name. *`name`* does not have to be unique.

WARNING

The XPM image must start with "/* XPM */" and no leading whitespace or Textadept will crash.

"*`width height ncolors`* `1`",
: Specifies pixel width *`width`*, pixel height *`height`*, and integer *`ncolors`* as the image's width, height, and number of colors, respectively.

"*`char`* `c None`",
"*`char`* `c #RRGGBB`",
: Specifies character *`char`* as the character in the image that represents a transparent pixel or a pixel of hexadecimal color *`RRGGBB`*. The image must have as many of these color definitions as colors declared.

"*`pixels`*",
: Specifies string *`pixels`* as a row of pixel data in the image. *`pixels`* must have as many characters as the image's width and the image must have as many of these lines as its declared height.

};
: Marks the end of the image.

Example 29 shows a sample XPM image of a black dot.

Example 29. Sample XPM image

```
local image = [[
/* XPM */
static char *black_dot[] = {
  "12 12 3 1",
  "  c None",
  ". c #4C4C4C",
  "+ c #000000",
  "            ",
  "            ",
  "            ",
  "            ",
  "     .++.   ",
  "    .++++.  ",
  "    ++++++  ",
  "    ++++++  ",
  "    .++++.  ",
  "     .++.   ",
  "            ",
  "            ",
  "            "
};]]
```

RGBA Image Format

Textadept's RGBA format is simply a stream of an image's pixels within a Lua string. The image's dimensions are defined by the `buffer.rgba_image_width` and `buffer.rgba_image_height` properties. Each pixel consists of three RGB color byte values and an alpha transparency value. Alpha transparency values are integers that range from 0 to 0xFF (transparent to opaque). Example 30 shows a sample RGBA image of a 3x6 black rectangle.

Example 30. Sample RGBA image

```
local image = table.concat{
  "\x4C\x4C\x4C\xFF\x4C\x4C\x4C\xFF\x4C\x4C\x4C\xFF",
  "\x4C\x4C\x4C\xFF\x00\x00\x00\xFF\x4C\x4C\x4C\xFF",
  "\x4C\x4C\x4C\xFF\x00\x00\x00\xFF\x4C\x4C\x4C\xFF",
  "\x4C\x4C\x4C\xFF\x00\x00\x00\xFF\x4C\x4C\x4C\xFF",
  "\x4C\x4C\x4C\xFF\x00\x00\x00\xFF\x4C\x4C\x4C\xFF",
  "\x4C\x4C\x4C\xFF\x4C\x4C\x4C\xFF\x4C\x4C\x4C\xFF",
}
```

Index of Key and Mouse Bindings

Each binding contains a set of three key or mouse sequences for the Windows and Linux, Mac OSX, and terminal platforms, respectively. Some bindings only apply to specific modes, which are indicated parenthetically. For a particular platform, '∅' indicates nothing is currently assigned to the command, while '∉' indicates the command does not exist.

T

U

V

W

Z

† *xxxx* represents the unicode character's four hexadecimal digits.

Lua API Index

io

keys

lexer

Concept Index

H

I

K

L