CUTE User-friendly Text Editor

Heiko Köhler

CUTE User-friendly Text Editor
by Heiko Köhler

CUTE is a scintilla based, scriptable code editor using python for extensions and configuration. Its main purpose is to provide an user-friendly text editor for programmers with a common user interface.

Table of Contents

Chapter 1. Overview

CUTE is designed for programmers, so it supports many programming languages. It also integrates with external commands such as ’make’ in order to make the edit compile cycle as fast as possible. Custom menu entries can be inserted with the python interface of CUTE. For fast code navigation, bookmarks and ctags can be used.

The main concept of CUTE is its ability to be extended and configured with python. Indeed there is a built-in python interpreter in CUTE. Each option must be set with the python interface, but this is made easier by the inclusion of a options dialog to set most options. When CUTE exits a config file is generated, holding all options and shortcuts including a recent file list. In most cases you don’t even have to know about the python interpreter.

There is a command box on the tool bar. It can execute shell commands and special CUTE commands, which are described later. All output of a command is shown in a message docklet but stderr and stdout messages are also viewable separately in the Stdout and Stderr docklets. If you want to jump to a specified line, given in the output, you can just double-click the item and CUTE will position the cursor at that line in the source file.

Most programmers work with several files at a time, so CUTE provides a multi document interface. A file browser in the side panel lets you select more files for editing. If you want to search through several files you just have to open the grep front end dialog. All search results are shown in the search docklet, from which a result can be double-clicked in order to jump to that line.

Using CUTE, it is possible for related files to be grouped together to form a Project; any of the files can then easily be reloaded for editing. Within a project it is possible to set your own Compile, Build and Go commands. A Project can also contain one or more configurations, each of which can define its own environment variables or commands. For example, a Project could contain both a debug and a release configuration, readily switching between them. As an extension to Projects, Sessions can be used to store the whole workspace with the current project.

Of course you can search within single files too. CUTE also supports regular expressions for search and replace commands.

Chapter 2. The User Interface

CUTE's user interface is made up of a main window with some docklets and an area for child views. A child view is a scintilla text widget, which can load a file, highlight source code, fold text, auto-complete text and so on. Only one instance of a file can be loaded at a time, to avoid overwriting previously changed files. There is a command box for running CUTE and shell commands. Available commands can be listed by pulling the list box down; it is also capable of auto-completion in the same way. CUTE commands will be described in detail later.

2.1. Dialogs

2.1.1. Options Dialog

The most important options can be set with the options dialog.

2.1.2. Shortcuts Dialog

Each shortcut can be set using this dialog. On double-clicking on a shortcut, a little dialog will be shown. Just type in the new key sequence. If the new shortcut is unique and not part of another one it will be accepted by the shortcut dialog. Press the OK button to confirm the new shortcut, or Cancel to undo the alteration.

2.1.3. Find Dialog

Searching for a string or a regular expression is done with the find dialog. After the first occurrence is found, the "Find Next" and "Find Previous" commands can be used to repeat the search.

2.1.4. Replace Dialog

Replacing one or more strings is done with the replace dialog. The user has to confirm each substitution (unless the "Replace All" or "Replace in Selection" button is pressed, when no confirmation is needed).

2.1.5. Find in Files Dialog

This dialog is a front end for grep. The search results are displayed in the search docklet with line numbers; you can double-click a result to jump to that line.

2.1.6. Save Dialog

When CUTE exits or a Project is closed, if necessary a Save dialog will be shown. Any unsaved modified files are listed and the user can (de)select those he wishes to save.

2.1.7. Project Options Dialog

Called from the Project > Options menu, this dialog allows a Project to be configured. In the General tab you can set the working directory of a Project and Select, Add or Delete a configuration. The selected configuration can be modified in the Environment and Commands sections. The project's files are listed in the Files section. Note that all file paths are displayed relative to the directory containing the project file.

Chapter 3. Editing

There are three ways to edit text in CUTE. First you can use keyboard commands, either directly or by means of macros. Second you can use a CUTE command, which you type in the command box on the toolbar. Finally you can write python scripts, which are discussed in the 'Programming CUTE' chapter.

A CUTE command can be undone by typing Ctrl+Z. The available editing commands can be seen in the 'Options > Configure Shortcuts' menu item.

3.1. Editing Commands

CUTE is designed for supporting as much programming languages as possible, thus many common editing commands are defined. The python interface is for new language specific commands.

3.1.1. Text Modification, Cut, Copy and Paste

You can edit text in CUTE just like in any text editor which is compatible with common GUI standards. Two editing modes are supported: inserting and overwriting. Switch between them by pressing Ins. If there is text selected, Cut it with Shift+Del and Paste it with Shift+Ins (the key-bindings may be altered to those of your choosing). Please note that there are two different clipboards available: the X Window clipboard and the clipboard of the current view. When you mark text with the mouse it is copied to the X clipboard and can be pasted by a middle click of the mouse. If your mouse has only two buttons, middle clicks can usually be emulated by clicking the left and right buttons at the same time.

3.1.2. Selection

There are two text selection modes available. The default is Stream selection (where the mouse selects all the text between left button down and left up), but you can switch to rectangle mode by pressing Ctrl+Alt during the selection process (this selects only the text within the rectangular area defined by the mouse). Double-clicking selects a word, where a word is defined as the text between two delimiters (what constitutes a delimiter depends on the current language).

3.1.3. Bookmarks

A bookmark is a highlit line in a file and is created by clicking the margin. The bookmark adjusts automatically when lines are deleted or inserted. You can jump to a bookmark by right clicking the mouse and selecting one from the Bookmarks sub menu (or use the Search menu). To delete one, just click on the margin. If a project is loaded, any bookmarks are stored in the project file.

3.1.4. Autocompletion

Autocompletion completes the current word if a similar word is found. In the options dialog you can select either the current file or the current tags file as the autocompletion source. After you have typed several characters of a word, the autocompletion list will automatically pop up. The number of chars to type before this happens can be defined in the Threshold option in the options dialog. If you prefer to control autocompletion yourself, put a huge number in Threshold so that it never triggers; then when you want to autocomplete a word, type (by default) Ctrl+Return.

3.1.5. Indentation

Indentation is configured by four options: indentationsUseTabs, autoIndent, tabIndents and backspaceUnindents. Normally pressing Tab inserts a tab, but if you set the indentationsUseTabs option you can press Tab anywhere in the line and the whole line will indent one tabs-worth. If autoindent is enabled, when a new line is created by pressing Return, it inherits the current indentation level. indentationsUseTabs decides whether tabs or spaces are used for indentation. The number of spaces is set with the tabWidth option, which you can also set in the Options > Tab Size menu.

3.1.6. Folding and Zooming

Depending on the current programming language, Folding can be used to hide parts of the code e.g. the body of a function or an if statement can be hidden. CUTE supports multifont editing, thus it is laborious to set each font size separately. Zooming increments or decrements each font size.

3.2. CUTE Commands

Each CUTE command starts with a specific string; for example if you want to execute an external program, you have to type :!program. “:!” is the start string, indicating that command is an execution command (nobody dies ;-)). A % is substituted by the file name of the current document. More CUTE commands are listed below:

!

filter selected text with the given program; for example “!indent” causes the selected text to be indented.

>

insert output of a given program at the cursor position.

/

search text forward. Possible arguments are: re, cs, word, wrap. If one of these is given the search will match a regular expression, will be case sensitive, will search a word or will wrap around lines. For example: “/switch(.*) re”, will find a switch statement in C.

?

same as above, but search backward.

:

jump to given line number. For example ":22"

:py

run a python command. For example ":py clear()" will clear current view, as clear() is a built-in CUTE/python function.

:pythony

same as above.

:s

this is a sed-like substitution command. Options are g or gc. g will replace all found strings, gc will replace all found strings, but the user must confirm each substitution. Without an option CUTE will replace only the first occurrence. For example: “:s/true/TRUE gc”

:q

exit CUTE.

:quit

same as above.

:w

save current file.

:write

same as above.

:e

open given file. For example ":e /home/heiko/CUTE/cute.cpp".

:edit

same as above.

:wq

save all files and exit.

:x

same as above.

:close

close current file.

:cd

change current directory. For example “:cd /home/heiko/CUTE”.

:char

insert the given integer value as a character at the current cursor position. For example “:char 190”, will insert 3/4 char.

Chapter 4. Searching and Replacing

There are two way to search and replace a string. You can use the find and replace dialogs, or the CUTE command box as described in the previous chapter. Regular expressions are supported too.

4.1. Regular Expressions

In a regular expression, special characters interpreted are:

.
: Matches any character

\(
: This marks the start of a region for tagging a match.

\)
: This marks the end of a tagged region.

\n
: Where n is 1 through 9 refers to the first through ninth tagged region when replacing. For example, if the search string was Fred\([1-9];\)XXX and the replace string was Sam\1YYY, when applied to Fred2XXX this would generate Sam2YYY.

\<
: This matches the start of a word using Scintilla's definitions of words.

\>
: This matches the end of a word using Scintilla's definition of words.

\x
: This allows you to use a character x that would otherwise have a special meaning. For example, \[would be interpreted as [and not as the start of a character set.

[...]
: This indicates a set of characters, for example, [abc] means any of the characters a, b or c. You can also use ranges, for example [a-z] for any lower case character.

[^...]
: The complement of the characters in the set. For example, [^A-Za-z] means any character except an alphabetic character.

^
: This matches the start of a line (unless used inside a set, see above).

$
: This matches the end of a line.

*

This matches 0 or more times. For example, Sa*m matches Sm, Sam, Saam, Saaam and so on.

+

This matches 1 or more times. For example, Sa+m matches Sam, Saam, Saaam and so on.

4.2. Tags

4.2.1. Building tags files

When the code is parsed by ctags it generates a file called tags. This file includes locations of all functions, methods and so forth. By clicking the tags tool button a dialog will appear asking which directory to search through and whether to recurse into subdirectories.

4.2.2. Jumping to tags

After building the tags file the tag docklet has several (programming language specific) category entries with tag subentries. On clicking a tag entry in the docklet, the cursor is positioned at the location of the tag in the file where it is defined. Alternatively a selection can be used as a tag: click the right mouse button to bring up the context menu, and select "Find Tag".

Chapter 5. Configuration

5.1. Main Configuration

The whole autogenerated configuration is stored in $HOME/.cuterc. This is a python file where the special variables of the built-in python interpreter of CUTE are held. Most of these can be configured with the ’Options > Configure CUTE’ menu item. It is possible to modify the rest of the variables manually: load the file with ’Options > Show .cuterc file’, adjust the values of any of the predefined variables, and then run the file using ’Extra > Run current file’ (you cannot just Save the file as it would be at first ignored, and then overwritten when CUTE exits). All config variables are defined in python modules. There are 4 modules: general, edit, view and lang. The .cuterc file will execute $HOME/cute.pre_config (if it exists) before any options are set. After setting all options, .cuterc will execute $HOME/cute.post_config (if it exists).

Therefore any user configuration should be placed in either cute.pre_config or cute.post_config. Put definitions of new commands in cute.pre_config if they are to be configured in .cuterc.

5.2. Language Configuration

Each language has its own python module and each module has its own file in $HOME/.cute/langs/. For example, the C++ options are defined in $HOME/.cute/langs/cpp.py. In each language module various properties are defined. A property is a python class, which stores the foreground and background color, font, size, whether end of line is filled with background color, whether the font is bold, underlined or italic. In addition there are several variables for comments defined: blockCommentStart, streamCommentStart and streamCommentEnd, which are used for the "Block Un/Comment" and "Stream Comment" edit command.

Chapter 6. Projects

6.1. Overview

Projects are used to store files, bookmarks, working dir and configurations. A configuration consists of environment variables and commands for compiling the current file, building and running the program.

6.2. Creating Projects

An empty project can be created (Project/New). After creating the project, further files can be added by using Open from the menu or toolbar, and then confirming in the subsequent pop-up dialog that they should become part of the project. All file or directory paths are stored relative to the directory containing the project file, which means that only files in the project directory or below can be added.

But a project isn't made up only of files. Bookmarks too are stored, and now one or more configurations can be added to a project. From the Project menu, open the Options dialog. First you have to add a configuration. After that the Commands and Environment fields are enabled and you can now edit them. This feature is useful when working with two compilers, e.g. you can switch between the compiler for your system and another cross compiler. It is also possible to create two configurations each with a different build command: one to generate the debug, the other the release version of your program.

Chapter 7. Programming CUTE

The most important feature of CUTE is the built-in python interpreter. It is responsible for configuring and extending CUTE. There are tasks which a user will wish to automate in a text editor. One way to do this is to record a macro once and execute it several times. The alternative is to write a python script using CUTE functions.

7.1. The CUTE/python API

7.1.1. General Functions

General functions are defined in the cute python module.

load(filename)

Load a file into the editor.

inputInteger(text)

Get an integer value from the user via a dialog.

inputText(text)

Get a string from the user via a dialog.

message(text)

Show a message dialog.

question(text)

Show a message dialog asking a question with yes and no button.

getOpenFileName()

Show an open file dialog and returns file name. You can also use a filter. For example: getOpenFileName(“*.cpp”).

getExistingDirectory()

Show an open dir dialog and returns dir name.

loadSession(filename)

Load a previously-saved session called filename.

viewList()

Retrieve list containing all views.

activateView(view)

The given view will become the current one.

7.1.2. Configuration Functions

These functions are defined in the config module.

addRecentFile(filename)

Add an item to the recent file menu.

addBookmark(name,line_number)

Add an item to the bookmark menu.

addTool(name,command)

Add an item to the tools menu.

addPythonTool(command)

Add an item to the tools menu, after the command has been created with createCommand()

setCurrentDir()

Set working directory.

setMainWidgetGeometry(x,y,width,height)

Set main window geometry: width, height and position.

map(command,shortcut)

Map an action to a shortcut.

createCommand(function,name)

creates a new command implemented in function with the given name; it can be bound to a shortcut and inserted into a menu

Color(red,green,blue)

Construct new color object.

7.1.3. Scintilla Functions

Scintilla functions are defined in the cute module.

These functions can be called with or without a view object: for example it is possible to call my_view.clear() in order to clear a certain view or just clear() in order to clear the current view.

clear()

Deletes all the text in the text edit.

copy()

Copies selected text to the clipboard.

cut()

Copies selected text to the clipboard and then deletes the text.

foldAll()

If any lines are currently folded then they are all unfolded. Otherwise all lines are folded. This has the same effect as clicking in the fold margin with the shift and control keys pressed.

paste()

Pastes text from the clipboard into the text edit at the current cursor position.

undo()

Undo the last change or sequence of changes.

redo()

Redo previously-Undone changes or sequences of changes.

selectToMatchingBrace()

If the cursor is either side of a brace character then move it to the position of the corresponding brace and select the text between the braces.

zoomIn()

Zooms in on the text by making the base font size one point larger and recalculating all font sizes.

zoomOut()

Zooms out on the text by making the base font size range points smaller and recalculating all font sizes.

beginUndoAction()

Mark the beginning of a sequence of actions that can be undone by a single call to undo().

endUndoAction()

Mark the end of a sequence of actions that can be undone by a single call to undo().

autoIndent()

Returns 1 if auto-indentation is enabled.

backspaceUnindents()

Returns 1 if the backspace key unindents a line instead of deleting a character.

braceMatching()

Returns the brace matching mode.

eolMode()

Returns the end-of-line mode.

eolVisibility()

Returns the visibility of end-of-lines.

findNext()

Find the next occurrence of the string found using findFirst().

folding()

Returns the current folding style.

hasSelectedText()

Returns 1 if some text is selected.

indentationGuides()

Returns 1 if the display of indentation guides is enabled.

indentationsUseTabs()

Returns 1 if indentations are created using tabs and spaces, rather than just spaces.

indentationWidth()

Returns the indentation width in characters. The default is 0 which means that the value returned by tabWidth() is actually used.

isModified()

Returns 1 if the text has been modified.

isReadOnly()

Returns 1 if the text edit is read-only.

isRedoAvailable()

Returns 1 if there is something that can be redone.

isUndoAvailable()

Returns 1 if there is something that can be undone.

isUtf8()

Returns 1 if text is interpreted as being UTF8 encoded. The default is to interpret the text as Latin1 encoded.

lines()

Returns the number of lines of text.

length()

Returns the length of the text edit’s text.

tabIndents()

Returns 1 if the tab key indents a line instead of inserting a tab character.

tabWidth()

Returns the tab width in characters.

whitespaceVisibility()

Returns the visibility of whitespace.

ensureLineVisible()

Ensures that the line number line is visible.

setAutoIndent(bool)

If autoindent is 1 then auto-indentation is enabled.

setBackspaceUnindents(bool)

If bool is 1, the backspace key will unindent a line rather then delete a character.

setEolMode(eolMode)

Sets the end-of-line mode to eolMode.

setEolVisibility(eolVisibility)

If eolVisibility is 1 then ends-of-lines are made visible.

setFolding(bool)

Sets the folding style for margin 2 to fold.

setIndentationGuidesBackgroundColor(color)

Set the background color of indentation guides.

setIndentationGuidesForegroundColor(color)

Set the foreground color of indentation guides.

setMarginsForegroundColor(color)

Set the foreground color of all margins.

setMarginsBackgroundColor(color)

Set the background color of all margins.

setMarkerBackgroundColor(color,mnr)

Set the background color of the marker mnr. If mnr is -1 then the color of all markers is set.

setMarkerForegroundColor(color,mnr)

Set the foreground color of the marker mnr. If mnr is -1 then the color of all markers is set.

setMatchedBraceBackgroundColor(color)

Set the background color used to display matched braces.

setMatchedBraceForegroundColor(color)

Set the foreground color used to display matched braces.

setUnmatchedBraceBackgroundColor(color)

Set the background color used to display unmatched braces.

setUnmatchedBraceForegroundColor(color)

Set the foreground color used to display unmatched braces.

setIndentationGuides(indentationsGuides)

Enables or disables the display of indentation guides.

setIndentationsUseTabs(bool)

If bool is 1 then indentations are created using tabs and spaces, rather than just spaces.

setIndentationWidth(width)

Sets the indentation width. If width is 0 then the value returned by tabWidth() is used.

setModified(bool)

Sets the modified state of the text edit. Note that it is only possible to clear the modified state. Attempts to set the modified state are ignored.

setReadOnly(bool)

Sets the read-only state of the text edit.

setTabIndents(bool)

If bool is 1 then the tab key will indent a line rather then insert a tab character.

setTabWidth(width)

Sets the tab width.

setUtf8(bool)

Sets the current text encoding. If bool is 1 then UTF8 is used, otherwise Latin1 is used.

setWhitespaceVisibility(bool)

Sets the visibility of whitespace.

unindent(line_number)

Decreases the indentation of line line_number.

zoomTo(percentage)

Zooms the text by making the base font size given points and recalculating all font sizes.

convertEol(eolMode)

All the lines of the text have their end-of-lines converted to given mode.

markerDeleteHandle(int_handle)

Delete the marker instance with the given marker handle.

indentation(line_number)

Returns the number of characters by which the given line is indented.

lineLength(line_number)

Returns the length of the given line, or -1 if there is no such line.

marginLineNumbers(margin_number)

Returns 1 if line numbers are enabled for the given margin.

marginSensitivity(margin_number)

Returns 1 if the given margin is sensitive to mouse clicks.

marginWidth(margin_number)

Returns the width in pixels of the given margin.

markerDefine(symbol,margin_number)

Define a marker using the given symbol with the given marker number. If mnr is -1 then the marker number is automatically allocated.

markerLine(marker_handle)

Return the line number that contains the marker instance with the given marker handle.

setCursorPosition(line,index)

Sets the cursor to the given line at the given position index.

setIndentation(line_number,string)

Sets the indentation of line line_number to match the given string.

setMarginLineNumbers(margin_number,bool)

Enables or disables, according to second arg, the display of line numbers in the given margin.

setMarginSensitivity(margin_number,bool)

Enables or disables, according to second arg, the sensitivity of the given margin to mouse clicks.

setMarginWidth(margin_number,width)

Sets the width of the given margin to the given pixels. If the width of a margin is 0 then it is not displayed.

insert(string,line,index)

Inserts text at the given position, or at the current position if none is supplied. The function must be given at least 1 argument.

find(string,isRegExp,isCaseSensitive,isWholeWord,wrap,forward,line,index)

Finds a string in the current view. The function must be given at least 1 argument.

currentLine()

Returns current line.

currentColumn()

Returns current index.

selection()

Returns current selection as Selection object. A selection object has four attributes: lineFrom, lineTo, columnFrom and columnTo.

selectedText()

Returns selected text.

moveCursor(int left, int down)

Moves cursor, negative values can also be used.

line(line_number)

Returns line at the given line number.

setMarginStringWidth(margin_number,string)

Sets the margin margin_number’s width with the given string.

setSelection(from_line,from_index,to_line,to_index)

Sets selection.

insertCommand(shell_command)

Inserts output of a shell command.

filter(shell_command)

Filters selection with a shell command.

fileName()

Retrieves the file name of the view.