qutepart Documentation

Release 2.2.3

Andrei Kopats

November 11, 2015

Contents

1 Helper functions 7

2 Logging 9

3 API Version 11

4 Binary parser 13

Python Module Index 15

class `qutepart.Qutepart(*args)`

Bases: `QPlainTextEdit`

Qutepart is based on QPlainTextEdit, and you can use QPlainTextEdit methods, if you don't see some functionality here.

Text

`text` attribute holds current text. It may be read and written.:

```
qpart.text = readFile()
saveFile(qpart.text)
```

This attribute always returns text, separated with `\n`. Use `textForSaving()` for get original text.

It is recommended to use `lines` attribute whenever possible, because access to `text` might require long time on big files. Attribute is cached, only first read access after text has been changed in slow.

Selected text

`selectedText` attribute holds selected text. It may be read and written. Write operation replaces selection with new text. If nothing is selected - just inserts text:

```
print qpart.selectedText  # print selection
qpart.selectedText = 'new text'  # replace selection
```

Text lines

`lines` attribute, which represents text as list-of-strings like object and allows to modify it. Examples:

```
qpart.lines[0]  # get the first line of the text
qpart.lines[-1]  # get the last line of the text
qpart.lines[2] = 'new text'  # replace 3rd line value with 'new text'
qpart.lines[1:4]  # get 3 lines of text starting from the second line as list of strings
qpart.lines[1:4] = ['new line 2', 'new line3', 'new line 4']  # replace value of 3 lines
del qpart.lines[3]  # delete 4th line
del qpart.lines[3:5]  # delete lines 4, 5, 6

len(qpart.lines)  # get line count

qpart.lines.append('new line')  # append new line to the end
qpart.lines.insert(1, 'new line')  # insert new line before line 1

print qpart.lines  # print all text as list of strings

# iterate over lines.
for lineText in qpart.lines:
    doSomething(lineText)

qpart.lines = ['one', 'thow', 'three']  # replace whole text
```

Position and selection

- `cursorPosition` - cursor position as `(line, column)`. Lines are numerated from zero. If column is set to `None` - cursor will be placed before first non-whitespace character. If line or column is bigger, than actual file, cursor will be placed to the last line, to the last column
- `absCursorPosition` - cursor position as offset from the beginning of text.
- `selectedPosition` - selection coordinates as `((startLine, startCol), (cursorLine, cursorCol))`.

- `absSelectedPosition` - selection coordinates as `(startPosition, cursorPosition)` where position is offset from the beginning of text.

Rectangular selection is not available via API currently.

EOL, indentation, edge

- `eol` - End Of Line character. Supported values are `\n`, `\r`, `\r\n`. See comments for `textForSaving()`
- `indentWidth` - Width of `Tab` character, and width of one indentation level. Default is `4`.
- `indentUseTabs` - If True, `Tab` character inserts `\t`, otherwise - spaces. Default is `False`.
- `lineLengthEdge` - If not `None` - maximal allowed line width (i.e. 80 chars). Longer lines are marked with red (see `lineLengthEdgeColor`) line. Default is `None`.
- `lineLengthEdgeColor` - Color of line length edge line. Default is red.

Visible white spaces

- `drawIncorrectIndentation` - Draw trailing whitespaces, tabs if text is indented with spaces, spaces if text is indented with tabs. Default is `True`. Doesn't have any effect if `drawAnyWhitespace` is `True`.
- `drawAnyWhitespace` - Draw trailing and other whitespaces, used as indentation. Default is `False`.

Autocompletion

Qutepart supports autocompletion, based on document contents. It is enabled, if `completionEnabled` is `True`. `completionThreshold` is count of typed symbols, after which completion is shown.

Linters support

- `lintMarks` Linter messages as {lineNumber: (type, text)} dictionary. Cleared on any edit operation. Type is one of 'Qutepart.LINT_ERROR, Qutepart.LINT_WARNING, Qutepart.LINT_NOTE)

Vim mode

`vimModeEnabled` - read-write property switches Vim mode. See also `vimModeEnabledChanged`. `vimModeIndication` - An application shall display a label, which shows current Vim mode. This read-only property contains (QColor, str) to be displayed on the label. See also `vimModeIndicationChanged`.

Actions

Component contains list of actions (QAction instances). Actions can be insered to some menu, a shortcut and an icon can be configured.

Bookmarks:

- `toggleBookmarkAction` - Set/Clear bookmark on current block
- `nextBookmarkAction` - Jump to next bookmark
- `prevBookmarkAction` - Jump to previous bookmark

Scroll:

- `scrollUpAction` - Scroll viewport Up
- `scrollDownAction` - Scroll viewport Down
- `selectAndScrollUpAction` - Select 1 line Up and scroll
- `selectAndScrollDownAction` - Select 1 line Down and scroll

Indentation:

- `increaseIndentAction` - Increase indentation by 1 level
- `decreaseIndentAction` - Decrease indentation by 1 level
- `autoIndentLineAction` - Autoindent line
- `indentWithSpaceAction` - Indent all selected lines by 1 space symbol
- `unIndentWithSpaceAction` - Unindent all selected lines by 1 space symbol

Lines:

- `moveLineUpAction` - Move line Up
- `moveLineDownAction` - Move line Down
- `deleteLineAction` - Delete line
- `copyLineAction` - Copy line
- `pasteLineAction` - Paste line
- `cutLineAction` - Cut line
- `duplicateLineAction` - Duplicate line

Other: * `undoAction` - Undo * `redoAction` - Redo * `invokeCompletionAction` - Invoke completion * `printAction` - Print file

Text modification and Undo/Redo

Sometimes, it is required to make few text modifications, which are Undo-Redoble as atomic operation. i.e. you want to indent (insert indentation) few lines of text, but user shall be able to Undo it in one step. In this case, you can use Qutepart as a context manager.:

```
with qpart:
    qpart.modifySomeText()
    qpart.modifyOtherText()
```

Nested atomic operations are joined in one operation

Signals

- `userWarning(text)` ` Warning, which shall be shown to the user on status bar. I.e. 'Rectangular selection area is too big'
- `languageChanged(langName)` ` Language has changed. See also `language()`
- `indentWidthChanged(int)` Indentation width changed. See also `indentWidth`
- `indentUseTabsChanged(bool)` Indentation uses tab property changed. See also `indentUseTabs`
- `eolChanged(eol)` EOL mode changed. See also `eol`.
- ``vimModeEnabledChanged(enabled) Vim mode has been enabled or disabled.
- `vimModeIndicationChanged(color, text)` Vim mode changed. Parameters contain color and text to be displayed on an indicator. See also `vimModeIndication`

Public methods

`terminate`()

Terminate Qutepart instance. This method MUST be called before application stop to avoid crashes and some other interesting effects Call it on close to free memory and stop background highlighting

`textForSaving`()

Get text with correct EOL symbols. Use this method for saving a file to storage

resetSelection()
Reset selection. Nothing will be selected.

replaceText(*pos*, *length*, *text*)
Replace length symbols from `pos` with new text.

If `pos` is an integer, it is interpreted as absolute position, if a tuple - as `(line, column)`

insertText(*pos*, *text*)
Insert text at position

If `pos` is an integer, it is interpreted as absolute position, if a tuple - as `(line, column)`

detectSyntax(*xmlFileName=None*, *mimeType=None*, *language=None*, *sourceFilePath=None*, *firstLine=None*)
Get syntax by next parameters (fill as many, as known):

- name of XML file with syntax definition
- MIME type of source file
- Programming language name
- Source file path
- First line of source file

First parameter in the list has the hightest priority. Old syntax is always cleared, even if failed to detect new.

Method returns `True`, if syntax is detected, and `False` otherwise

clearSyntax()
Clear syntax. Disables syntax highlighting

This method might take long time, if document is big. Don't call it if you don't have to (i.e. in destructor)

language()
Get current language name. Return `None` for plain text

isHighlightingInProgress()
Check if text highlighting is still in progress

isCode(*blockOrBlockNumber*, *column*)
Check if text at given position is a code.

If language is not known, or text is not parsed yet, `True` is returned

isComment(*line*, *column*)
Check if text at given position is a comment. Including block comments and here documents.

If language is not known, or text is not parsed yet, `False` is returned

isBlockComment(*line*, *column*)
Check if text at given position is a block comment.

If language is not known, or text is not parsed yet, `False` is returned

isHereDoc(*line*, *column*)
Check if text at given position is a here document.

If language is not known, or text is not parsed yet, `False` is returned

setExtraSelections(*selections*)
Set list of extra selections. Selections are list of tuples `(startAbsolutePosition, length)`. Extra selections are reset on any text modification.

This is reimplemented method of QPlainTextEdit, it has different signature. Do not use QPlainTextEdit method

mapToAbsPosition(*line*, *column*)
Convert line and column number to absolute position

mapToLineCol(*absPosition*)
Convert absolute position to `(line, column)`

CHAPTER 1

Helper functions

`qutepart.`**`iterateBlocksFrom`**`(`*block*`)`

Generator, which iterates QTextBlocks from block until the End of a document

`qutepart.`**`iterateBlocksBackFrom`**`(`*block*`)`

Generator, which iterates QTextBlocks from block until the Start of a document

CHAPTER 2

Logging

Package logs it's debug information to `qutepart` logger. By default logger prints messages to stderr with `qutepart:` prefix. For performance reasons, parser in C:

- checks, if DEBUG logs are enabled only when created
- always prints logs to stderr
- always prints ERROR logs
- always uses `qutepart:` prefix

Logging handler is available as `consoleHandler` attribute of the package

CHAPTER 3

API Version

API version is available as `qutepart.VERSION`. It is a tuple `(major, minor, patch)`

- Major versions might be incompatible.
- Minor versions add new API, but remain backward compatilbe
- Patch versions does not change API

CHAPTER 4

Binary parser

Qutepart has 2 text parsers - slower in Python, and quicker in C. If C parser has been successfully loaded, `qutepart.binaryParserAvailable` is True

q

qutepart, 1

C

clearSyntax() (qutepart.Qutepart method), 4

D

detectSyntax() (qutepart.Qutepart method), 4

I

insertText() (qutepart.Qutepart method), 4
isBlockComment() (qutepart.Qutepart method), 4
isCode() (qutepart.Qutepart method), 4
isComment() (qutepart.Qutepart method), 4
isHereDoc() (qutepart.Qutepart method), 4
isHighlightingInProgress() (qutepart.Qutepart method), 4
iterateBlocksBackFrom() (in module qutepart), 7
iterateBlocksFrom() (in module qutepart), 7

L

language() (qutepart.Qutepart method), 4

M

mapToAbsPosition() (qutepart.Qutepart method), 5
mapToLineCol() (qutepart.Qutepart method), 5

Q

Qutepart (class in qutepart), 1
qutepart (module), 1

R

replaceText() (qutepart.Qutepart method), 4
resetSelection() (qutepart.Qutepart method), 4

S

setExtraSelections() (qutepart.Qutepart method), 4

T

terminate() (qutepart.Qutepart method), 3
textForSaving() (qutepart.Qutepart method), 3