Qscintilla documentation

(A learn-by-example guide)

This documentation assumes a basic knowledge of the Python programming language. But even if you are a complete beginner in the language do not get discouraged, the examples should be quite understandable regardless.

All examples will use the Python 3 programming language and the PyQt5 application framework. To change the examples to Python 2 or the PyQt4 is quite simple and in most cases, trivial.

For in-depth information about Qt, PyQt, Scintilla and QScintilla go to the official documentation websites.

Table of Contents

1. Introduction

This chapter assumes you know nothing about QScintilla or PyQt and will guide you step by step to understanding the basics of the PyQt framework and the QScintilla editing component.

1.1. What is QScintilla?

QScintilla is a text editing component for the Qt application framework written in the C++ programming language. It is a wrapper for the Scintilla text editing component created by Neil Hodgson also written in the C++ programming language. The Qt application framework is a set of objects, also referred to as widgets, that help making GUI (Graphical-User-Interface) and other types of applications easier and is cross-platform.

PyQt is a set of Python bindings to the Qt application framework, which also includes bindings to the QScintilla component. So hurray, we can use QScintilla in Python. This introduction chapter will focus solely on the PyQt's QScintilla component, other PyQt components will only be mentioned when they are needed with regards to the QScintilla component.

As PyQt is a wrapper for Qt framework, it cannot be avoided that some parts of Qt will have to be described in order to understand the documentation. Sometimes I will even mention some C++ code, but only when it is absolutely necessary.

The referencing of QScintilla / Scintilla will seem confusing at first but do not get discouraged, just keep in mind that QScintilla component is just the Scintilla text editing component wrapped in a Qt QWidget object so it can be used in the Qt framework, and PyQt's QScintilla component is just the QScintilla component wrapped in Python!

Also later in the document I will use the terms QScintilla, Qscintilla document, editor, object or widget interchangeably. All terms mean the QScintilla editing component.

1.2. Some QScintilla features

- Built-in syntax highlighting for more than 30 programming languages
- Create syntax highlighting for custom purposes
- Text styling: underlining, highlighting, …
- Clickable text (called **Hotspots** in QScintilla)
- Word wrapping
- Autocompletion functionality and call tips
- Error indicating, bookmarks, … using the margins of the document
- Code folding

- Selecting various font styles and colors

- Mixing font styles in the same document

- Customizing keyboard commands

- Block selection of text

- Zooming text in / out

- UTF-8 support

- Command macros (call sequences of commands with one command)

- Customizing the **Caret** (the blinking marker that shows where the cursor is located)

- Built-in search and replace functionality

- …

Plus with Python and the entire PyQt framework at your disposal, you will be able to do much, much more.

1.3. QScintilla object overview

All QScintilla objects are part of the PyQt framework under the **PyQt5.Qsci** module. The below listed objects will be covered in depth (some more than others) in this documentation.

1.3.1. PyQt5.Qsci.QsciScintillaBase

The base object for text editing. This is a more direct low level wrapper to access all of the underlying Scintilla functionality. It is used when you cannot implement something in the higher level **PyQt5.Qsci.QsciScintilla** object.

1.3.2. PyQt5.Qsci.QsciScintilla

The high level object for text editing. It is a subclass of the low level **PyQt5.Qsci.QsciScintillaBase** object, which just means it has access to all of the **PyQt5.Qsci.QsciScintillaBase** methods and attributes. This object will be used most of the time for all text editing purposes you will need and has a very Qt-like API. **When using the name QScintilla, I will always be referring to this object**.

1.3.3. PyQt5.Qsci.QsciLexer

The abstract object used for styling text (syntax highlighting) usually in the context of a programming language. This is an abstract object and has to be sub-classed to make a custom lexer. There are many built-in lexers already available such as: QsciLexerPython, QsciLexerRuby, … The lexer needs to be applied to an instance of a **PyQt5.Qsci.QsciScintilla** object for it to start styling the text.

1.3.4. PyQt5.Qsci.QsciAPIs

The object used for storing the custom autocompletion and call tip information. In a nutshell, you assign it to an instance of the sub-classed **PyQt5.Qsci.QsciLexer** with autocompletions / call tips enabled, add keywords to it and you get custom autocompletions in the editor.

1.3.5. PyQt5.Qsci.QsciStyle

The object used for storing the options of a style for styling text with a **PyQt5.Qsci.QsciLexer**. This object does not have to be explicitly used, it's options are selected using the **PyQt5.Qsci.QsciLexer's** setFont, setPaper, ... methods (all of the lexer's methods that take *style* as the second argument).

There are only a few situations that you need to use this object directly. One is setting the margin text, the other is when adding annotations to lines.

1.4. PyQt objects used by QScintilla

There are various PyQt objects needed to customize the QScintilla component, while some are needed to initialize the PyQt application.

1.4.1. PyQt5.QtWidgets.QApplication

The QApplication object manages the GUI application's control flow and main settings. It's the object that needs to be initialized and executed to get QScintilla shown on the screen.

1.4.2. PyQt5.QtWidgets.QMainWindow

This object provides the main application window. It should always be used a the main window widget. If you are creating a simple QScintilla editor without any bells and whistles, just create

1.4.3. PyQt5.QtWidgets.QWidget

The QWidget object is the base object of all user interface objects. The QScintilla object and all other GUI object inherit from this object, that is why it is relevant.

From the official Qt 5 documentation:

> The widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Every widget is rectangular, and they are sorted in a Z-order. A widget is clipped by its parent and by the widgets in front of it.

> A widget that is not embedded in a parent widget is called a window. Usually, windows have a frame and a title bar, although it is also possible to create windows without such decoration using suitable window flags). In Qt, QMainWindow and the various subclasses of QDialog are the most common window types.

1.4.4 PyQt5.QtGui.QFont

QScintilla uses PyQt's QFont object for describing fonts. The QFont object resides in the **PyQt5.QtGui** module. The relevant attributes of the QFont object with regards to QScintilla are:

- family: style of the font. Some examples are: Courier, Times, Helvetica, ...

- pointSize: size of the font in points. If it is set to lower than 1, it will default to the system default size

- weight: font thickness from 0 (ultralight) to 99 (extremely black). The predefined values are QFont.Light, QFont.Normal, QFont.DemiBold, QFont.Bold and Qfont.Black.

- italic: tilted text, True or False.

For more detailed information on this object, see the PyQt's web documentation.

You may be wondering how do you change the **color** of the font, right? This is what styling does, which will be described in a later chapter.
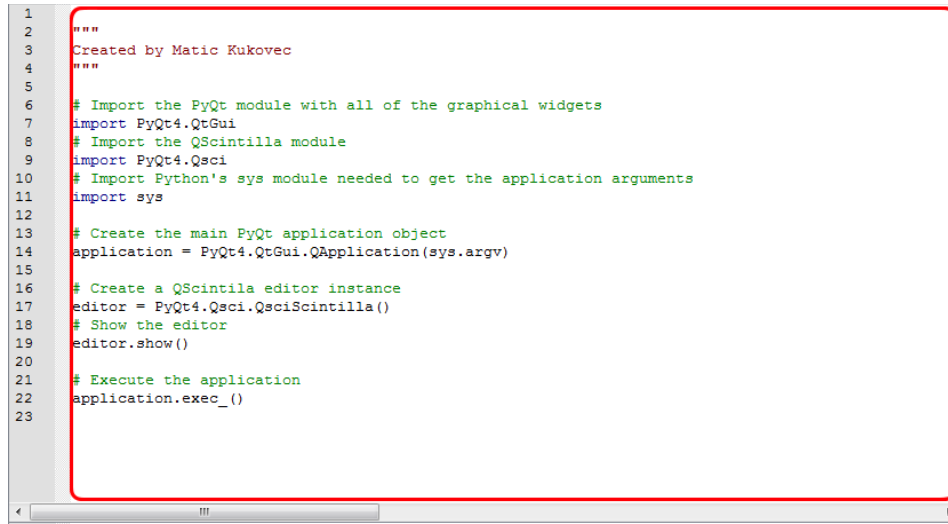
1.5. QScintilla visual description

Here I will describe the parts of the QScintilla component that will be discussed throughout this document, so you will have a clear idea of what goes where. **A more detailed look into these parts will follow in a later chapter.**

1.5.1. Editing area
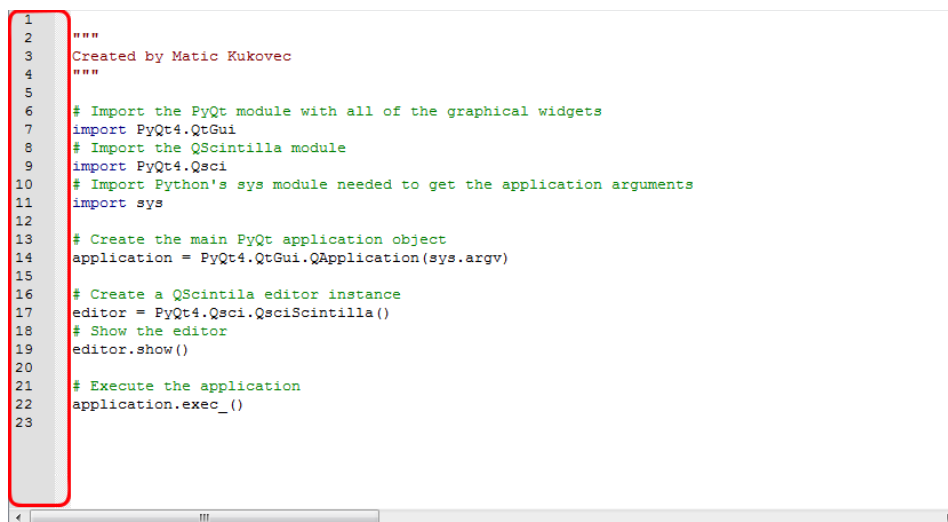
This is where all of the editing and styling (coloring, ...) of text happens. Text is edited using the mouse and keyboard, but it may also be manipulated in Python code. The editing area is swhon in the red rectangle in the image below.

1.5.2. Margin area

This is where the margins of the QScintilla document are located. Margins are the sidebars that can show line numbers, where there is the option for code folding, showing error indicators, showing bookmarks and anything else you can think of. The QScintilla can have up to 7 margins (I think?). The margin area is shown in the image below.

1.5.3. Scroll bar area

This is is the area where the scroll bars are shown. The horizontal vertical bar is always present, while the vertical scroll bars is shown depending on the number of lines in the document and the documents window size. The scroll bar area is shown in the image below.
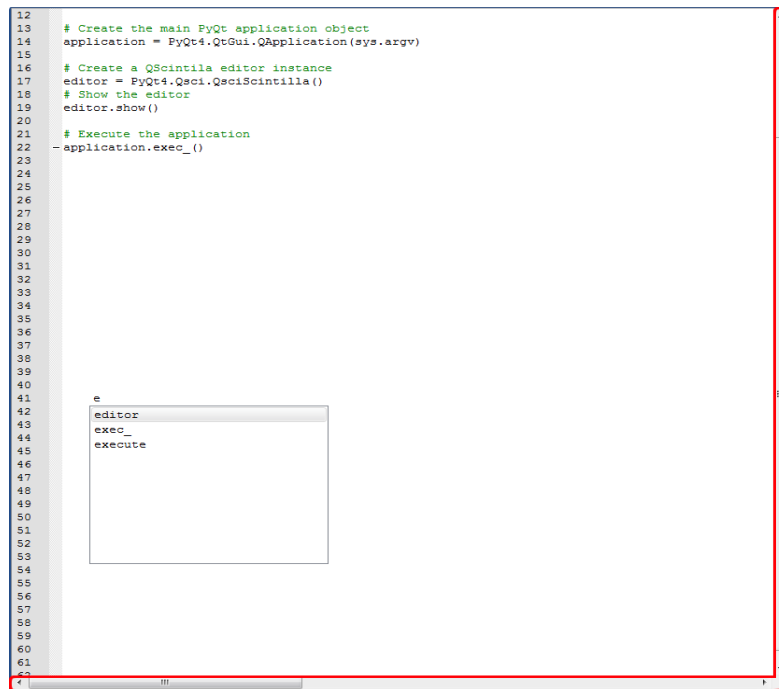
1.5.4. Autocompletion windows

When autocompletions are active in the document, the documentation window show sthe current autocompletion suggestions. By default, autocompletion is not enabled. An example autocompletion window is shown in the image below.
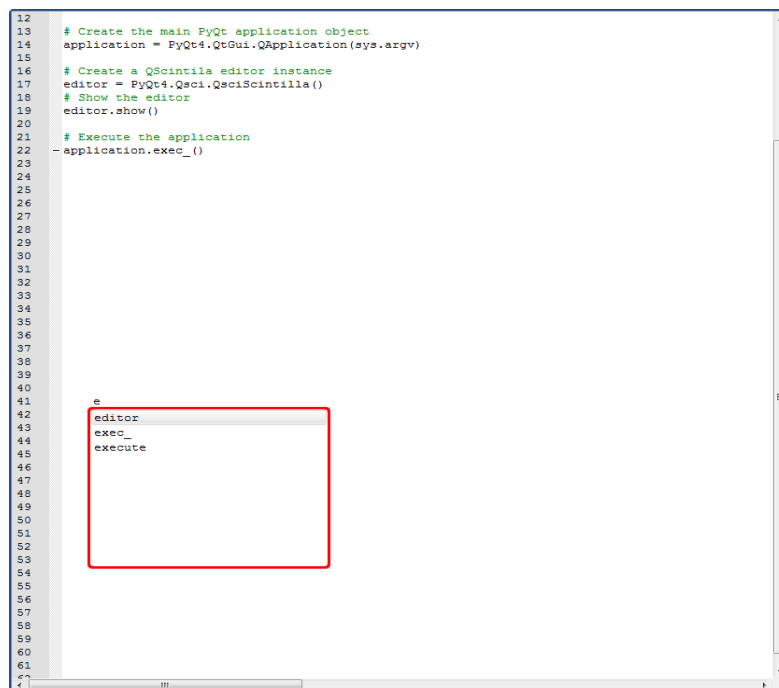
1.5.5. Context menu

By default there is only the right-click context menu that shows some standard editing options in the QScintilla component. Custom context menus can be added in Python code, but they use other parts of the PyQt framework that will not be covered in this documentation. QScintilla's default context menu is shown in the image below.
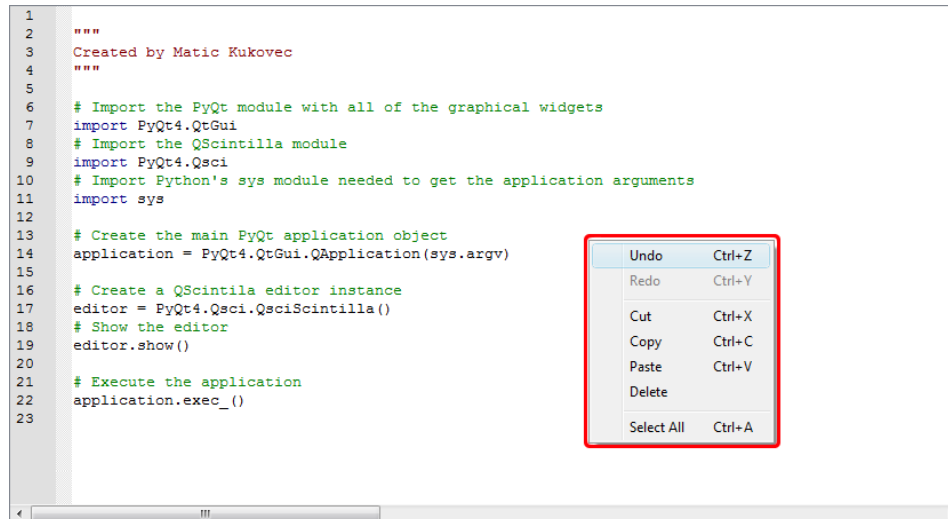
1.6. QScintilla's default settings

Below are QScintilla's default settings. These are the setting when you create an instance of the QScintilla component without setting any of it's options.

1.6.1. Default lexer

By default, there is **NO** lexer set for the QScintilla editing component when it is first created. Sometimes I will also say that the lexer is disabled, but it means the same thing, that no lexer is set for the editor.

1.6.2. Default font

The default font used by QScintilla when **NO** lexer is set depends on the operating system you are using. On Windows it is **MS Shell Dlg 2**, while on GNU/Linux Debian Jesse it is **Roboto**.

The default font color is black. The size of the font depends on your system's settings. I have not had the opportunity to test on other systems.

When there is a lexer set for the document, it overrides any settings that you have manually set previously. If you are trying to set some font style or color directly and nothing is happening, it means that a lexer is set on the document and is always overriding the changes you are trying to make.

1.6.3. Default paper

In QScintilla the background of the document is called the **paper**. The paper is the background color of the document. By default the paper is white.

Here it's the same as with the font. If you are trying to set the paper color directly and nothing is happening, it means that a lexer is set on the document and is always overriding the changes you are trying to make.

1.6.4. Default keyboard shortcuts

Below is the list of the default shortcuts used by QScintilla.
(NOTE: FIX ALL OF THE DESCRIPTIONS BELOW)

- 'Down': Move one line down
- 'Down+Shift': Extend selected text one line down
- 'Down+Ctrl': Scroll the view one line down
- 'Down+Alt+Shift': Block extend selection one line down
- 'Up': Move one line up
- 'Up+Shift': Extend selected text one line up
- 'Up+Ctrl': Scroll the view one line up
- 'Up+Alt+Shift': Block extend selection one line up
- '[+Ctrl': Move paragraph up
- '[+Ctrl+Shift': Extend selection one paragraph up
- ']+Ctrl': Move paragraph down
- ']+Ctrl+Shift': Extend selection one paragraph down
- 'Left': SCI_CHARLEFT
- 'Left+Shift': SCI_CHARLEFTEXTEND
- 'Left+Ctrl': SCI_WORDLEFT
- 'Left+Shift+Ctrl': SCI_WORDLEFTEXTEND
- 'Left+Alt+Shift': SCI_CHARLEFTRECTEXTEND
- 'Right': SCI_CHARRIGHT
- 'Right+Shift': SCI_CHARRIGHTEXTEND
- 'Right+Ctrl': SCI_WORDRIGHT
- 'Right+Shift+Ctrl': SCI_WORDRIGHTEXTEND
- 'Right+Alt+Shift': SCI_CHARRIGHTRECTEXTEND
- '/+Ctrl': SCI_WORDPARTLEFT
- '/+Ctrl+Shift': SCI_WORDPARTLEFTEXTEND
- '\\+Ctrl': SCI_WORDPARTRIGHT
- '\\+Ctrl+Shift': SCI_WORDPARTRIGHTEXTEND
- 'Home': SCI_VCHOME
- 'Home+Shift': SCI_VCHOMEEXTEND
- 'Ctrl+Home': SCI_DOCUMENTSTART
- 'Ctrl+End': SCI_DOCUMENTSTARTEXTEND
- 'Home+Alt': SCI_HOMEDISPLAY
- 'Home+Alt+Shift': SCI_VCHOMERECTEXTEND
- 'End': SCI_LINEEND
- 'End+Shift': SCI_LINEENDEXTEND
- 'Ctrl+End': SCI_DOCUMENTEND
- 'Ctrl+Shift+End': SCI_DOCUMENTENDEXTEND
- 'End+Alt': SCI_LINEENDDISPLAY
- 'End+Alt+Shift': SCI_LINEENDRECTEXTEND
- 'PageUp': SCI_PAGEUP
- 'Shift+PageUp': SCI_PAGEUPEXTEND
- 'PageUp+Alt+Shift': SCI_PAGEUPRECTEXTEND
- 'PageDown': SCI_PAGEDOWN
- 'Shift+PageDown': SCI_PAGEDOWNEXTEND
- 'PageDown+Alt+Shift': SCI_PAGEDOWNRECTEXTEND
- 'Delete': SCI_CLEAR
- 'Delete+Shift': SCI_CUT
- 'Ctrl+Delete': SCI_DELWORDRIGHT
- 'Ctrl+Shift+BackSpace': SCI_DELLINERIGHT
- 'Insert': SCI_EDITTOGGLEOVERTYPE
- 'Insert+Shift': SCI_PASTE

- 'Insert+Ctrl': SCI_COPY
- 'Escape': SCI_CANCEL
- 'Backspace': SCI_DELETEBACK
- 'Backspace+Shift': SCI_DELETEBACK
- 'Ctrl+BackSpace': SCI_DELWORDLEFT
- 'Backspace+Alt': SCI_UNDO
- 'Ctrl+Shift+BackSpace': SCI_DELLINELEFT
- 'Ctrl+Z': SCI_UNDO
- 'Ctrl+Y': SCI_REDO
- 'Ctrl+X': SCI_CUT
- 'Ctrl+C': SCI_COPY
- 'Ctrl+V': SCI_PASTE
- 'Ctrl+A': SCI_SELECTALL
- 'Tab': SCI_TAB
- 'Shift+Tab': SCI_BACKTAB
- 'Return': SCI_NEWLINE
- 'Return+Shift': SCI_NEWLINE
- 'Add+Ctrl': SCI_ZOOMIN
- 'Subtract+Ctrl': SCI_ZOOMOUT
- 'Divide+Ctrl': SCI_SETZOOM
- 'Ctrl+L': SCI_LINECUT
- 'Ctrl+Shift+L': SCI_LINEDELETE
- 'Ctrl+Shift+T': SCI_LINECOPY
- 'Ctrl+T': SCI_LINETRANSPOSE
- 'Ctrl+D': SCI_SELECTIONDUPLICATE
- 'U+Ctrl': SCI_LOWERCASE
- 'U+Ctrl+Shift': SCI_UPPERCASE

1.6.5. Default scroll bar behaviour

By default, only the horizontal scroll bar is shown and it has a default starting length. The horizontal scroll bar automatically grows as you enter more text and adjusts itself to the longest line in the document. The vertical scroll bar also shrinks automatically when you delete text form the lines, but it shrinks back to it's default size.

The vertical scroll bar appears only when there are more lines than can be shown in the current QScintilla document. When you add lines the vertical scroll bar grows automatically, while when you delete lines it shrinks automatically.

1.6.6. Default margin

The QScintilla shows one margin by default. It's behaviour is, that it selects the text of the line next to the position where the user has left-clicked on the margin using the mouse.

1.6.7. Default autocompletion behaviour

By default, autocompletion are disabled.

1.6.8. Default mouse behaviour

The default mouse behaviour is text selection with holding the left mouse button and dragging the mouse. Right clicking anywhere in the document shows the contex menu. By default the context menu has the following options: Undo, Redo, Cut, Copy, Paste, Delete and Select All.

When there is text selected in the editor

1.6.9. Default encoding

The default encoding is ASCII. Unknown characters will appear as the questionmark (**?**) symbol.

2. Installation guide

Note that in this installation guide I mention **QScintilla2**. QScintilla is at version 2.9.3 at the moment, this is where the **2** comes from.

Installing PyQt5 using **pip** as described below can be done **ONLY** with Python3.5 at the moment.

2.1. Windows

Install the latest PyQt5 library for your version of Python 3, this can be done easily using the **pip** package manager (if you have it installed) with the following command in the windows console:

```
pip install PyQt5
```

The other option is to install the Visual Studio version that your Python 3 version was compiled with and compile the from source from their official website. You will also need to download the SIP library source code. Download the source code and follow the instructions in the readme/install files. You'll also need the Qt5 C++ source code.

2.2. GNU/Linux

If you are on Lubuntu, Raspbian or probably most Debian derivatives, install the following libraries using **apt-get**:

- python3.x (Probably already installed on the system)
- python3-pyqt5
- python3-pyqt5.qsci

Another option is as on Windows using **pip** with the following command in your favourite terminal:

```
pip3 install PyQt5
```

Notice it's **pip3** on GNU/Linux as it usually has both Python2 and Python3 installed.

Otherwise you can install PyQt5 and QScintilla2 (you will also need the SIP library) from source from their official website. Download the source code and follow the instructions in the readme/install files. You'll also need the Qt5 C++ source code.

2.3. Mac OS

Install the latest Python3 version and the pip package manager and use the following command in the terminal:

```
pip install PyQt5
```

Another thing you can try is using Anaconda Python 3 and it's package manager to install all dependencies. Here is the more information.
I don't know much about Mac's, but you can try using the default Mac package manager to find the PyQt5 and QScintilla2 libraries or install the libraries from source, same as on GNU/Linux.

3. QScintilla options

This chapter will be an in depth description of all QScintilla options.

3.1. Text wrapping

Text wrapping disables or enables multiple types of text wrapping, which means breaking lines that are longer than what the QScintilla editor can show in the editor screen into multiple lines.

3.1.1. Text wrapping mode

Set with method: **setWrapMode(wrap_mode)**
Queried with method: **wrapMode()**
wrap_mode parameter options:
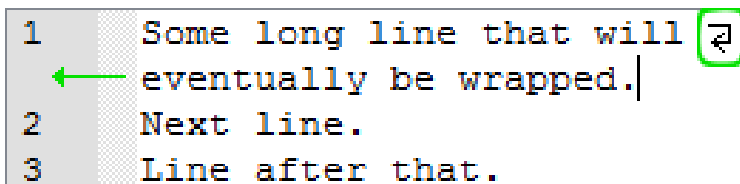
- **PyQt5.Qsci.QsciScintilla.WrapNone**

  No wrapping. Lines that exceed the editor's screen width cannot be seen unless you scroll horizonstally using the mouse or with the cursor.

- **PyQt5.Qsci.QsciScintilla.WrapWord**

  Lines are wrapped at words. Example:
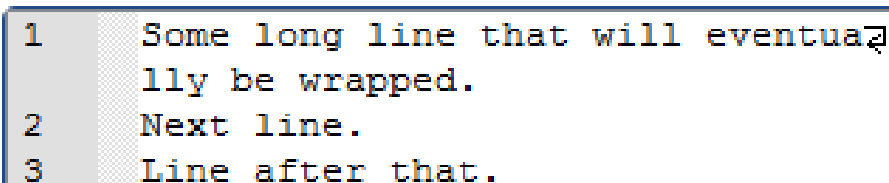
  Note the line wrap character and that the wrapped line has no line number (if there is a line margin present in the QScintilla editor). The other wrap visualisation options will be shown later.

- **PyQt5.Qsci.QsciScintilla.WrapCharacter**
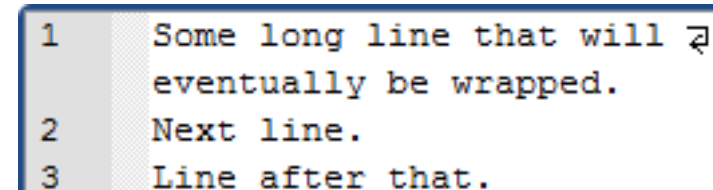
  Lines are wrapped at the character boundaries. Example:

- **PyQt5.Qsci.QsciScintilla.WrapWhitespace**

  Lines are wrapped at the whitespace boundaries. Example:

3.1.2. Text wrapping visual flags

These method selects how the text wrapping will be indicated in the QScintilla editor.

Set with method: **setWrapVisualFlags(endFlag, startflag, indent)**

endFlag parameter options:

- **PyQt5.Qsci.QsciScintilla.WrapFlagNone**

```
1    Some long line that will
     eventually be wrapped
2    Next line.
3    Line after that.|
```

- **PyQt5.Qsci.QsciScintilla.WrapFlagByText**

```
1    Some long line that will ⮠
     eventually be wrapped
2    Next line.
3    Line after that.
```

- **PyQt5.Qsci.QsciScintilla.WrapFlagByBorder**

```
1    Some long line that will      ⮠
     eventually be wrapped
2    Next line.
3    Line after that.
```

- **PyQt5.Qsci.QsciScintilla.WrapFlagInMargin**

```
1    Some long line that will
⮡    eventually be wrapped
2    Next line.
3    Line after that.
```

startflag parameter options:

- **PyQt5.Qsci.QsciScintilla.WrapFlagNone**

```
1    Some long line that will
     eventually be wrapped
2    Next line.
3    Line after that.|
```

- **PyQt5.Qsci.QsciScintilla.WrapFlagByText** or **PyQt5.Qsci.QsciScintilla.WrapFlagByBorder**

```
1    Some long line that will
⤶    eventually be wrapped
2    Next line.
3    Line after that.
```

- **PyQt5.Qsci.QsciScintilla.WrapFlagInMargin (same effect as the endflag parameter)**

```
1    Some long line that will
⤶    eventually be wrapped
2    Next line.
3    Line after that.
```

indent parameter options:

- This parameter sets the number of spaces each wrapped line is indented by. It has to be an **int**. **It effects can be seen ONLY if setWrapIndentMode is set to PyQt5.Qsci.QsciScintilla.WrapIndentFixed!**

3.1.3. Text wrapping indent mode

Selects how wrapped lines are indented.

Set with method: **setWrapIndentMode (indent_mode)**

indent_mode parameter options:

- **PyQt5.Qsci.QsciScintilla.WrapIndentFixed (with setWrapVisualFlags indent parameter set to 4!)**

```
1    Some long line that will ⤶
         eventually be wrapped
2    Next line.
3    Line after that.
```

- **PyQt5.Qsci.QsciScintilla.WrapIndentSame**
  Indents the same as the first wrapped line. In the below example, the first line is indented by two whitespaces.

```
1      Some long line that ⤶
       will eventually be ⤶
       wrapped
2    Next line.
3    Line after that.
```

- **PyQt5.Qsci.QsciScintilla.WrapIndentIndented**
  Indents the same as the first wrapped line **PLUS** one more indentation level. Indentation level is by the **setTabWidth** method.

```
1          Some long line that ⤶
                 will eventually ⤶
                 be wrapped
2       Next line.
3       Line after that.
```

3.2. End-Of-Line (EOL) options

These options effect the End-Of-Line settings like the End-Of-Line character and mode for the QScintilla editor. By default the line endings are invisible but can be made visible with the **setEolVisibility** method, described below.

3.2.1. End-Of-Line mode

Set with method: **setEolMode(eol_mode)**

Queried with method: **eolMode()**

indent_mode parameter options:

- **PyQt5.Qsci.QsciScintilla.EolWindows:** Carrige-Return + Line-Feed (\r\n)
- **PyQt5.Qsci.QsciScintilla.EolUnix:** Line-Feed (\n)
- **PyQt5.Qsci.QsciScintilla.EolMac:** Carrige-Return (\r)

3.2.2. End-Of-Line character/s visibility

This selects the visibility of the EOL character/s in the editor window. By default it is not visible.

Set with method: **setEolVisibility(visibility)**

Queried with method: **eolVisibility()**

visibility parameter options:

- **False**: The EOL character/s is/are **NOT** visible

```
1       No EOL character at the end.
2       New line.|
```

- **True:** The EOL character/s is/are visible (marked in green in the image below)

```
1       This line shows the EOL characterLF
2       New line.|
```

3.3. Indentation options

These options select the indentation character, indentation size, ...

3.3.1. Indentation character (tabs or spaces)
This selects whether the **indent/unindent** functions use either the **TAB ('\t')** character or the **WHITESPACE (' ')** character.

Set with method: **setIndentationsUseTabs(use_tabs)**
Queried with method: **indentationsUseTabs()**

use_tabs parameter options:

- **False**: Indentation uses the whitspace characters
- **True:** Indentation uses the tab character

3.3.2. Indentation size
Selects the number of space characters to indent by.
This means the number of space characters when indenting with WHITESPACES or the width of the TAB character in spaces when using TABS for indentation.
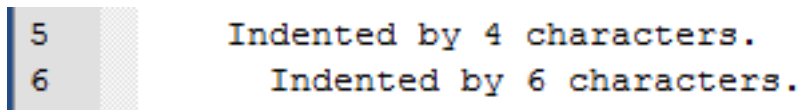
Set with method: **setTabWidth(indentation_size)**
Queried with method: **tabWidth()**

indentation_size parameter options:

- **A number greater than zero:** Number of space characters to indent by

```
5          Indented by 4 characters.
6              Indented by 6 characters.
```

3.3.3. Indentation guides
QScintilla can show vertical guiding lines at indentation columns as dotted lines. These guides are only visual guides, they do not effect the editor's text.
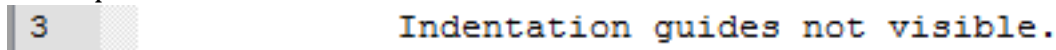
Set with method: **setIndentationGuides (indents)**
Queried with method: **indentationGuides ()**

- **False**: indentation guides not visible
  Example:
  ```
  3                    Indentation guides not visible.
  ```
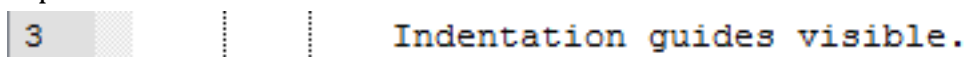
- **True**: indentation guides visible
  Example:
  ```
  3          ┊      ┊      Indentation guides visible.
  ```

3.3.4. Indentation *at spaces* options

This one is a little confusing to explain without pictures. When indenting at a place in a line where there are ONLY whitespaces/tabs, if this option it set to **True**, then QScintilla indents and aligns the next NON-whitespace/tab character and the rest of the line to the indentation level and moves the cursor to the NON-whitespace/tab character.

But if set to **False**, then QScintilla just inserts an indentation, whether whitespaces or tabs. This sounds confusing, but take a look at the examples shown in the images below.

Set with method: **setTabIndents(indents)**
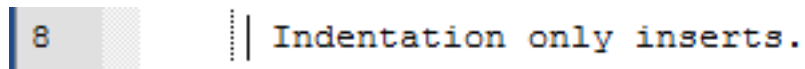
Queried with method: **tabIndents()**

indents parameter options:
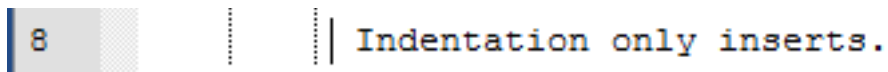
- **False**: Indentation simply inserts an indentation (whitespaces/tab) if indenting in a place in a line where there are only whitespaces to each side of the cursor.

  Example:

  Before indentation
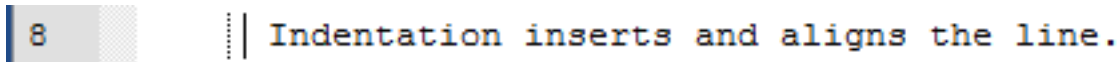
  After indentation (after pressing the **TAB** key). An indentation (whitespaces/tab) is inserted and the cursor moved the width of the indentation forward.
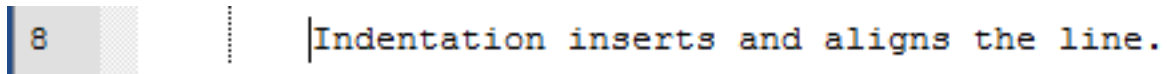
- **True:** Indentation indents and aligns the next non-whitespace/tab character to that indentation level and also moves the cursor to that indentation level.

  Example:

  Before indentation

  After indentation (after pressing the **TAB** key). The next non-whitespace/tab character is aligned to the indentation and the cursor has aligned to the same indentation level.

3.3.5. Automatic indentation

When set to **True**, automatic indentation moves the cursor to the same indentation as the previous line when adding a new line by either pressing **Enter** or **Return**. But when set to **False**, the cursor will always move to the start of the new line.
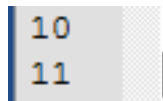
NOTE: A lexer can modify this behaviour with it's **setAutoIndentStyle** method!

Set with method: **setAutoIndent(autoindent_on)**

Queried with method: **autoIndent()**
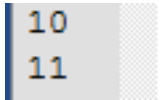
autoindent_on parameter options:

- **False**: The cursor moves to the start of the line when creating a new line.

```
10              Previous line.
11       |
```

- **True**: The cursor moves to the same indentation as the previous line.

```
10              Previous line.
11            |
```

3.4. Caret options (cursor representation)

These option set the way the cursor symbol looks and behaves.

3.4.1. Caret foreground color
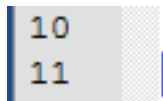
This option sets the color of the cursor.

Set with method: **setCaretForegroundColor(fg_color)**
Queried with method: **Not available directly**

fg_color parameter options:

- **PyQt5.QtGui.QColor:** To set the caret color to for example blue, use PyQt5.QtGui.Qcolor("#ff0000ff").

```
10      It's hard to see but the caret is blue.
11      |
```

3.4.2. Caret line visibility
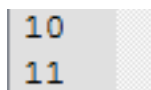
This option enables or disables the coloring of the line that the cursor is on.

Set with method: **setCaretLineVisible(visibility)**
Queried with method: **Not available directly**
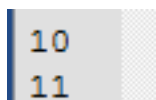
visibility parameter options:

- **False**: The line the cursor is on is not colored.

```
10      Normal line.|
11
```

- **True**: The line the cursor is on is indicated with the caret background color.

```
10      Caret line indicated.|
11
```

3.4.3. Caret line background color

This option sets the background color of the line that the cursor is on. The caret line visibility option has to be set to **True** for this option to be visible!

Set with method: **setCaretLineBackgroundColor(bg_color)**

Queried with method: **Not available directly**

bg_color parameter options:

- **PyQt5.QtGui.QColor:** To set the caret color to for example light blue, use PyQt5.QtGui.Qcolor("#1f0000ff").

```
10     Caret line indicated.|
11
```
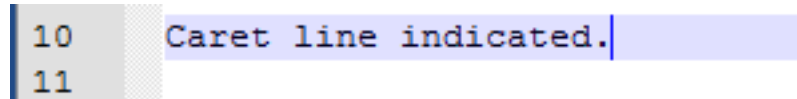
3.4.3. Caret width

This option sets the caret width in pixels. **0 makes the caret invisible!**

Set with method: **setCaretWidth(size)**

Queried with method: **Not available directly**

size parameter options:

- **Integer**: Size of the caret in pixels.

```
10     Caret size is 4 pixels.
11     |
```
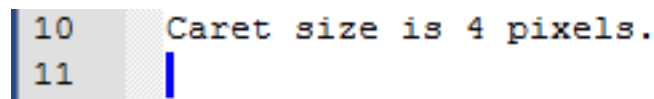
3.5. Autocompletion – Basic

Autocompletion is the functionality of the QScintilla editor to show suggestions for words from an autocompletion source while you are typing characters into the editor. The autocompletion source is selectable. The suggestions are shown in Autocompletion windows, which are described in 1.5.4. Autocompletion windows.

These are the basic options that do not need a lexer to be set for the editor and no **API**'s loaded (lexer/API autocompletion options will be explained in the **Advanced** chapter).

3.5.1. Autocompletion case sensitivity

Set with method: **setAutoCompletionCaseSensitivity(case_sensitivity)**

Queried with method: **autoCompletionCaseSensitivity()**

case_sensitivity parameter options:

- **False**: Autocompletion is case **IN**-sensitive. This means that the letters in the word you are typing do not have to match the case of the words in the autocompletion source. Example (the autocompletion source contains ["will", "wrapped", "Word"]):

```
w
will
word
wrapped
```

```
W
Will
Word
Wrapped
```
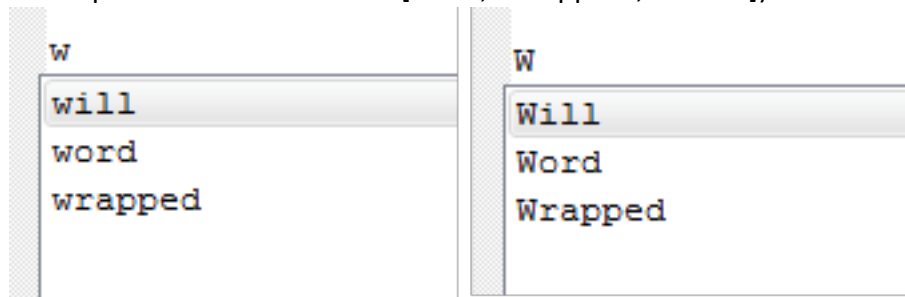
- **True**: Autocompletion in case sensitive. If you type a word that is in the autocompletion sources but does not match the case of the autocompletion source word, the word will not appear in the suggestion window. Example (the autocompletion source contains ["will", "wrapped", "Word"]):

```
w
will
wrapped
```

```
W
Word
```
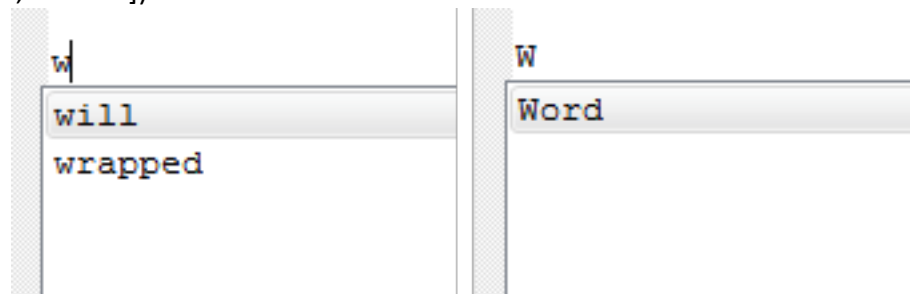
3.6. Margins

Margins are the sidebars on the right of every QScintilla editor. They are shown in the image in chapter 1.5.2. Margin area.

If margins are not visible, which is the default, means that every margin has a width of **0**.

By default margin 0 (the leftmost margin) is the line number margin, margin 1 is used to display non-folding (custom) symbols and margin 2 displays the folding symbols. But you can customize the order of the margins as you wish.

You can have up to 5 margins: 0, 1, 2, 3 and 4.

Margins have customizable type, width, background and foreground color, font type, sensitivity to mouse clicks, ...

3.6.1. Margin types

- **Line number margin**

    The line number margin (by default margin **0**) is the margin that displays the line numbers. The line margin does not resize automatically when lines are added to the

```
1
2  # Import t
3  import PyQ
4  # Import t
5  import PyQ
6  # Import P
```
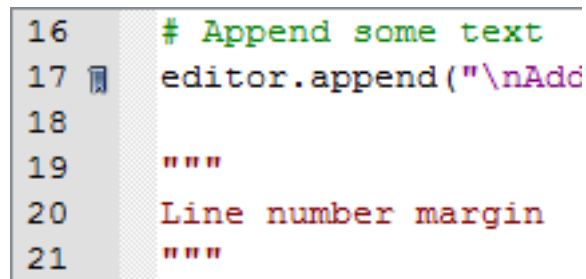
document, so you have to resize it manually with the **setMarginWidth** function. An example is shown in the image below:

- **Symbol margin**

  The symbol margin (by default margin **1**) can be used for displaying custom symbols (images) on each line of the margin.

  Symbol margins have **marker masks**, which are used to filter which symbols can be shown on which line of the margin. Each symbol is bound to a **marker** which can be added to a margin. Think of markers as a wrapper around a symbol which you can add to a line number, then the margins determine if the marker can be shown on it using their marker masks. Markers will be explained in more detail in the next chapter.
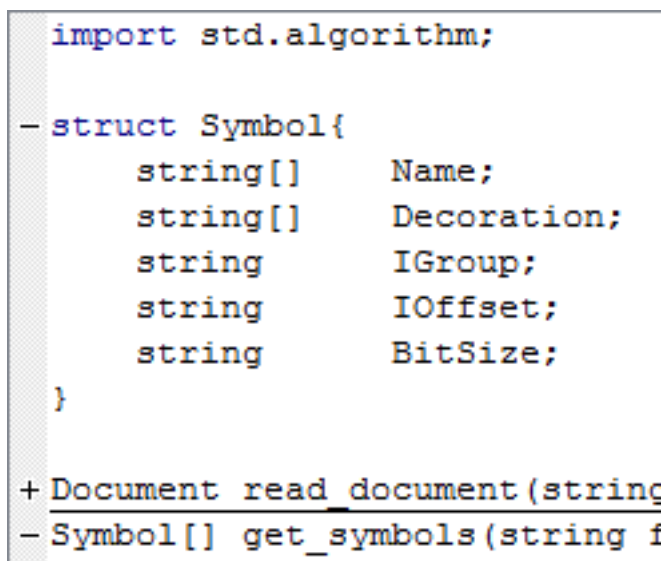
  In the image below is an example of a bookmark symbol on margin **1** in my Ex.Co. editor:

```
16        # Append some text
17 🔖     editor.append("\nAdd
18
19        """
20        Line number margin
21        """
```

- **Folding margin**

  The folding margin (by default margin **2**) displays the folding symbols that are used for folding and unfolding lines that have been styled for folding. In the image below you can see the **+** and **–** symbols which are used to fold and unfold lines:
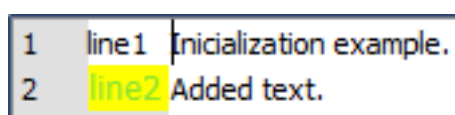
```
  import std.algorithm;

─ struct Symbol{
      string[]    Name;
      string[]    Decoration;
      string      IGroup;
      string      IOffset;
      string      BitSize;
  }

+ Document read_document(string
─ Symbol[] get_symbols(string f
```

- **Text margin**

  This margin is used for displaying text in the margin lines. The text style (size and

```
1    line1   Inicialization example.
2    line2   Added text.
```

family), font and paper color are selectable. In the image below is an example of a text margin with the first line set to the default text and styled green text on a yellow background on the second line:

3.6.2. Choosing a margin's type

Set with method: **setMarginType(margin_number, margin_type)**
Queried with method: **marginType()**

margin_number parameter options:

- **Integer**: Number of the margin

margin_type parameter options:

- **PyQt5.Qsci.QsciScintilla.NumberMargin**: Margin for displaying line numbers

- **PyQt5.Qsci.QsciScintilla.SymbolMargin:** Margin for displaying custom symbols (images)
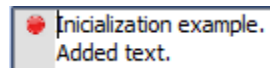
- **PyQt5.Qsci.QsciScintilla.SymbolMarginDefaultBackgroundColor:** Margin for displaying custom symbols (images) with the background color set to what the default paper color of the editor is. Below is an example of an white editor paper color and a white margin. **NOTE:** This margin is **NOT** affected by the **setMarginsBackgroundColor** and
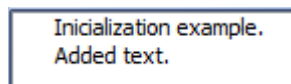
    setMarginsForegroundColor!

- **PyQt5.Qsci.QsciScintilla.SymbolMarginDefaultForegroundColor:** Margin for displaying custom symbols (images) with the background color set to what the default font color of the editor is. Below is an example of an editor with black font color and a black margin. **NOTE:** This margin is **NOT** affected by the **setMarginsBackgroundColor** and **setMarginsForegroundColor!**
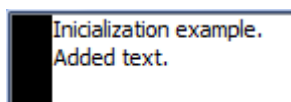
- **PyQt5.Qsci.QsciScintilla.TextMargin:** Margin for displaying text. The text can be styled, by default it is the same as the editor's default font and paper color.
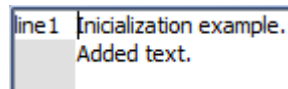
- **PyQt5.Qsci.QsciScintilla.TextMarginRightJustified:** Margin for displaying text. Same as the standard text margin with the text justified to the right.
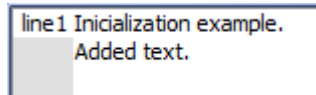
- **PyQt5.Qsci.QsciScintilla.SymbolMarginColor (**NOT AVAILABLE PRE VERSION 2.10**):** Margin for displaying symbols, but it's background color is set with **setMarginBackgroundColor**.

3.6.2.1. Special line numbers method

This is a special method which can set the visibility of line numbers on a margin, even if the margin is not the **PyQt5.Qsci.QsciScintilla.NumberMargin** type. This can either overwrite or combine the contents of the margin. An example of a symbol margin with line numbers visible and overlapping is shown below:
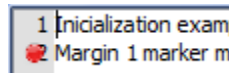
Set with method: **setMarginLineNumbers(margin_number, line_numbers_visible)**
Queried with method: **marginLineNumbers (margin_number)**

margin_number parameter options:

- **Integer**: Selected margin

line_numbers_visible parameter options:

- **True**: Line numbers visible
- **False**: Line numbers hidden

3.6.3. Set the number of margins (NOT AVAILABLE PRE VERSION 2.10)
Set with method: **setMargins(number_of_margins)**
Queried with method: **margins()**

number_of_margins parameter options:

- **Integer**: Number of used margins

3.6.4. Margin foreground color
By default the foreground (text) color of every margin is black.

Set with method: **setMargin*s*ForegroundColor(color) (**ON QScintilla PRE VERSION 2.10 IT'S *setMarginForegroundColor***)**

color parameter options:

- **PyQt5.QtGui.QColor**: To change the margin foreground (text) color to blue for example, set it to PyQt5.QtGui.QColor("#0000ffff")
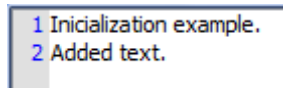
3.6.5. Margin background color

By default the background color of every margin (except for the SymbolMarginDefaultForegroundColor and SymbolMarginDefaultBackgroundColor) is grey.

Set with method: **setMargin*s*BackgroundColor(color) (**ON QScintilla PRE VERSION 2.10 IT'S *setMarginBackgroundColor***)**

Queried with method: **marginBackgroundColor** (DOESN'T SEEM TO WORK IN QScintilla PRE VERSION 2.10)

color parameter options:

- **PyQt5.QtGui.QColor**: To change the margin background color to red for example, set it to PyQt5.QtGui.QColor("#ff0000ff")
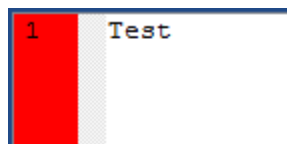
3.6.6. Margin width

Set with method:

- **setMarginWidth(margin_number, int_width)**

- **setMarginWidth(margin_number, width_string)**

Queried with method: **marginType(margin_number)**

margin_number parameter options:

- **Integer**: Selected margin

int_width parameter options:

- **Integer**: Width of the margin in pixels.

width_string parameter options:

- **String**: A string (for example *"0000")* which will be used to automatically calculate the needed margin width based on the margin's font's style.

3.6.6. Margin sensitivity to mouse clicks

This option enables or disables mouse click emitting the ***marginClicked()*** signal (see PyQt documentation on signals for more details).

Set with method: **setMarginSensitivity(margin_number, sensitivity)**
Queried with method: **marginSensitivity(margin_number)**

margin_number parameter options:

- **Integer**: Selected margin

sensitivity parameter options:

- **True:** Enables the emitting of the ***marginClicked()*** and ***marginRightClicked()*** signal
- **False:** Disables the emitting of the ***marginClicked()*** and ***marginRightClicked()*** signal

Here is an example of how to connect to the margin click signal. To connect to the ***marginClicked()*** signal use:

```python
def margin_click(margin, line, state):
    # margin parameter = Integer (margin_number)
    # line parameter = Integer (line_number)
    # state parameter = Qt.KeyboardModifiers (OR-ed together)
    print("Margin {} clicked at line {}!".format(margin, line))

editor.marginClicked.connect(margin_click)
```

To connect to the ***marginRightClicked()*** signal it is completely the same, except the signal name to:

```python
editor.marginRightClicked.connect(margin_click)
```

3.6.7. Margin marker mask

A margin's marker mask is 32-bit integer number of which every bit represents the **shown/hidden** state of the marker (marker = symbol that can be displayed on the margin) on that margin. If a bit is **0** the marker is hidden and if it is **1** it is shown. Examples:

All markers enabled: 0b11111111111111111111111111111111

Marker 0 and 3 disabled: 0b11111111111111111111111111110110

Set with method: **setMarginMarkerMask(margin_number, mask)**
Queried with method: **marginMarkerMask(margin_number)**

margin_number parameter options:

- **Integer**: Selected margin

mask parameter options:

- **Integer**: A 32-bit integer that will be used as the mask

Example:

The below image has one margin (margin number **0**) with marker number **1** added to it and the marker mask **0b11111111111111111111111**:
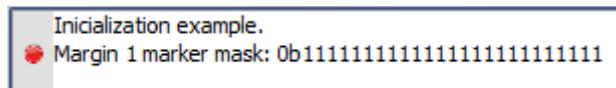
Now we change the margin's marker mask to **0b11111111111111111111101** and the marker will be hidden:
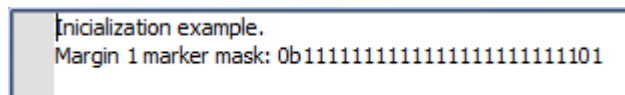
3.6.8 Markers

Markers are a wrapper around the symbols (images) that you wish to display in a margin. In short, you put an image into a marker and add it to one or more margin's line.

Markers can have custom images or you can choose from a number of built-in symbols.

You can have a total of 32 markers, which is the total number of bits in a marker mask of every margin, **but you can assign the same marker multiple times to multiple lines**.

3.6.8.1 Defining a marker

There a four ways to define (create) a marker. You can use a built-in symbol, a single ASCII character, a QPixmap or QImage.

Defined with methods:

- **markerDefine(builtin_symbol, marker_number)**

  builtin_symbol parameter options:

  - **PyQt5.Qsci.QsciScintilla.MarginSymbol:** Built-in symbols, which can be the following

    - PyQt5.Qsci.QsciScintilla.Circle

    - PyQt5.Qsci.QsciScintilla.Rectangle

    - PyQt5.Qsci.QsciScintilla.RightTriangle

    - PyQt5.Qsci.QsciScintilla.SmallRectangle

    - PyQt5.Qsci.QsciScintilla.RightArrow

    - PyQt5.Qsci.QsciScintilla.Invisible

    - PyQt5.Qsci.QsciScintilla.DownTriangle

    - PyQt5.Qsci.QsciScintilla.Minus

- PyQt5.Qsci.QsciScintilla.Plus

- PyQt5.Qsci.QsciScintilla.VerticalLine

- PyQt5.Qsci.QsciScintilla.BottomLeftCorner

- PyQt5.Qsci.QsciScintilla.LeftSideSplitter

- PyQt5.Qsci.QsciScintilla.BoxedPlus

- PyQt5.Qsci.QsciScintilla.BoxedPlusConnected

- PyQt5.Qsci.QsciScintilla.BoxedMinus

- PyQt5.Qsci.QsciScintilla.BoxedMinusConnected

- PyQt5.Qsci.QsciScintilla.RoundedBottomLeftCorner

- PyQt5.Qsci.QsciScintilla.LeftSideRoundedSplitter

- PyQt5.Qsci.QsciScintilla.CircledPlus

- PyQt5.Qsci.QsciScintilla.CircledPlusConnected

- PyQt5.Qsci.QsciScintilla.CircledMinus

- PyQt5.Qsci.QsciScintilla.CircledMinusConnected

- PyQt5.Qsci.QsciScintilla.Background

- PyQt5.Qsci.QsciScintilla.ThreeDots

- PyQt5.Qsci.QsciScintilla.ThreeRightArrows

- PyQt5.Qsci.QsciScintilla.FullRectangle

- PyQt5.Qsci.QsciScintilla.LeftRectangle

- PyQt5.Qsci.QsciScintilla.Underline

- PyQt5.Qsci.QsciScintilla.Bookmark

Below is the image from the official Scintilla documentation displaying all the available built-in markers (as can be seen, the names of the markers differ from the QScintilla names):

```
○   SC_MARK_CIRCLE,
○   SC_MARK_ROUNDRECT,
□   SC_MARK_SMALLRECT,
▷   SC_MARK_SHORTARROW,
◀   SC_MARK_BOOKMARK,
    SC_MARK_FULLRECT,
    SC_MARK_LEFTRECT,
    SC_MARK_BACKGROUND,
    SC_MARK_UNDERLINE,
... SC_MARK_DOTDOTDOT,
»»  SC_MARK_ARROWS,
▷   SC_MARK_ARROW,
▽   SC_MARK_ARROWDOWN,
✚   SC_MARK_PLUS,
─   SC_MARK_MINUS,
⊞   SC_MARK_BOXPLUS,
⊞   SC_MARK_BOXPLUSCONNECTED,
⊟   SC_MARK_BOXMINUS,
⊟   SC_MARK_BOXMINUSCONNECTED,
⊕   SC_MARK_CIRCLEPLUS,
⊕   SC_MARK_CIRCLEPLUSCONNECTED,
⊖   SC_MARK_CIRCLEMINUS,
⊖   SC_MARK_CIRCLEMINUSCONNECTED,
│   SC_MARK_VLINE,
└   SC_MARK_LCORNER,
├   SC_MARK_TCORNER,
└   SC_MARK_LCORNERCURVE,
├   SC_MARK_TCORNERCURVE
```

marker_number parameter options:

○ **Integer**: The number of the marker, which can be 0 – 31. **If this value is set to -1, then the first free marker will be used. If the marker number is invalid the method will return -1.**

Below is an image showing a marker **PyQt5.Qsci.QsciScintilla.Rectangle** marker:
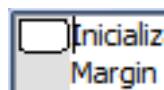
- **markerDefine(ascii_character, marker_number)**
  ascii_character parameter options:

  ○ **Char**: a – Z ASCII character

  marker_number parameter options:

  Integer: The number of the marker, which can be 0 – 31. **If this value is set to -1, then the first free marker will be used. If the marker number is invalid the method will return -1.**

Below is an image of a character marker "T":
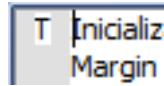
- **markerDefine(qpixmap_image, marker_number)**
  qpixmap_image parameter options:

  - **PyQt5.QtGui.QPixmap**: A custom QPixmap image

  marker_number parameter options:

  Integer: The number of the marker, which can be 0 – 31. **If this value is set to -1, then the first free marker will be used. If the marker number is invalid the method will return -1.**

- **markerDefine(qimage_image, marker_number)**
  qpixmap_image parameter options:

  - **PyQt5.QtGui.QImage**: A custom QImage image

  marker_number parameter options:

  Integer: The number of the marker, which can be 0 – 31. **If this value is set to -1, then the first free marker will be used. If the marker number is invalid the method will return -1.**

Return value:

- **Integer**: The marker number. **If the marker number is invalid the method will return -1.**

3.6.8.2 Adding a marker to margins

Remeber that markers are added to **ALL** margins on the specified lines, margin marker masks determine if the marker is visible on a specific margin.

Added with method: **markerAdd(line_number, marker_number)**

line_number parameter options:

- **Integer**: Line number which the marker will be added to

marker_number parameter options:

- **Integer**: The number of the marker, which can be 0 – 31. **If the marker number is invalid the method will return -1.**

Return value:

- **Integer**: A handle that is used to track the marker's position. If you assign the same marker to multiple lines, this method will return multiple handles so you can modify each one seperately.

3.6.8.3 Deleting a marker from margins

Like adding a marker to a line, deleting also deletes the marker from **ALL** margins.

Deleted with method: **markerDelete(line_number, marker_number)**

line_number parameter options:

- **Integer**: Line number on which the marker will be deleted. **If this parameter is -1, then all markers will be deleted from this line.**

marker_number parameter options:

- **Integer**: The number of the marker, which can be 0 – 31. **If the marker number is invalid the method will return -1.**

3.6.8.4 Deleting a marker by it's handle

Same as the above method **markerDelete**, but this method takes the marker handle as the parameter and deletes the marker from the line that it was added to.

Deleted with method: **markerDeleteHandle(marker_handle)**

marker_handle parameter options:

- **Integer**: Marker handle of the marker that will be deleted.

3.6.8.5 Deleting all markers

Deleted with method: **markerDeleteAll(marker_number)**

marker_number parameter options:

- **Integer**: The number of the marker, which can be 0 – 31. **If the marker number is invalid the method will return -1.**

3.6.8.6 Get all markers on a line

Returns a integer that represents a 32 bit mask of the markers defined on a line.

Querried with method: **markersAtLine (line_number)**

line_number parameter options:

- **Integer**: The line number that you wish to querry

Example:

> If there are no markers on the line, the method returns 0b0.
> But if there is marker 2 on that line, the method return 0b10.

3.6.8.7 Get line number that a marker is on by the markers handle

This method returns the line number that the marker was added to using the markers handle.

Querried with method: **markerLine(marker_handle)**

marker_handle parameter options:

- **Integer**: Marker handle of the marker that will be deleted.

Return value:

- **Integer**: The line number that the marker is on or *-1* if the marker is not present on any line.

3.6.8.7 Find markers

You can search for markers forwards and backwards using a marker mask with these methods.

Search forward with method: **markerFindNext (starting_line_number, marker_mask)**

starting_line_number parameter options:

- **Integer**: Starting line number from which the next marker will be searched for.

marker_mask parameter options:

- **Integer**: A 32 bit mask with the bits that represent the markers to search for set to *1*. Example: To search for markers 1 and 3, set the mask to: 0b1010 (the upper part of the mask are all *0*'s)

Search backwards with method: **markerFindPrevious(starting_line_number, marker_mask)**

starting_line_number parameter options:

- **Integer**: Starting line number from which the next marker will be searched for.

marker_mask parameter options:

- **Integer**: A 32 bit mask with the bits that represent the markers to search for set to *1*. Example: To search for markers 1 and 3, set the mask to: 0b1010 (the upper part of the mask are all *0*'s)

3.7. Hotspots (clickable text)

Hotspots are a feature of QScintilla text styling that make parts of the editor's text clickable with the mouse. This can be used for a multitude of options, like ***jump-to-declaration*** functionality.

Below is an example of a hotspot:

```
 94        def style_hotspot(self, index_from, length, color=0xff0000):
 95            """Style the text from/to with a hotspot"""asdfasdf
 96            #Use the scintilla low level messaging system to set the hotspot
 97            style_number = 2
 98            self.SendScintilla(QsciScintillaBase.SCI_STYLESETHOTSPOT, style_number, True)
 99            self.SendScintilla(QsciScintillaBase.SCI_SETHOTSPOTACTIVEFORE, True, color)
100            self.SendScintilla(QsciScintillaBase.SCI_SETHOTSPOTACTIVEUNDERLINE, True)
101            self.SendScintilla(QsciScintillaBase.SCI_STARTSTYLING, index_from, style_number)
102            self.SendScintilla(QsciScintillaBase.SCI_SETSTYLING, length, style_number)
```

Hotspots are an attribute of the **PyQt5.Qsci.QsciStyle** object, but usually you will just want to set a specific style to be a hotspot style, meaning every time a lexer styles a piece of text with a certain style, it will be a hotspot. This is done by sending a message to a editor using the **SendScintilla** method using the **PyQt5.Qsci.QsciScintillaBase.SCI_STYLESETHOTSPOT** message parameter, which will be shown in the examples. It is possible to style hotspots manually using the **SendScintilla** method, **BUT THE EDITOR MUST NOT HAVE A LEXER SET**, otherwise the lexer will overwrite what you manually styled. You can quickly realize if this is the case, when you are trying to create a hotspot manually and the hotspot does not work, which usually means that the lexers is overwriting your manual hotspot styling (or your code is not correct).

A hotspot becomes visible when you hover the mouse over it and you can choose the hotspots foreground color, background colors and underlining when the mouse is over it.

3.7.1. Hotspot foreground color

This setting changes the **active** (when the mouse is over the hotspot) foreground color of all hotspots.

Set with method: **setHotspotForegroundColor(color)**
Reset with method (resets to the default style color): **resetHotspotForegroundColor()**

color parameter options:

- **PyQt5.QtGui.QColor**: To change the **active** hotspot foreground color to yellow for example, set it to PyQt5.QtGui.QColor("#ffffff00")

```
HOTSPOT, style_number, True)
OTACTIVEFORE, True, color
OTACTIVEUNDERLINE, True)
LING, index_from, style_number)
NG, length, style_number)
```

3.7.2. Hotspot background color

This setting changes the **active** (when the mouse is over the hotspot) background color of all hotspots. THIS DOESN'T SEEM TO BE WORKING ON QSCINTILLA 2.9.2 AND BELOW! USE THE SendScintilla METHOD WITH THE *SCI_SETHOTSPOTACTIVEBACK* PARAMETER IN THIS CASE.

Set with method: **setHotspotBackgroundColor(color)**
Reset with method (resets to the default style color): **resetHotspotBackgroundColor()**

color parameter options:

- **PyQt5.QtGui.QColor**: To change the **active** hotspot background color to torquoise for
  example, set it to PyQt5.QtGui.QColor("#ffff00").

```
THOTSPOT, style_number, True)
POTACTIVEFORE, True, colo
POTACTIVEUNDERLINE, True)
YLING, index_from, style_number)
ING, length, style_number)
```

3.7.3. Hotspot underlining

This setting selects the visibility of underlining on an **active** (when the mouse is over the hotspot) hotspot.

Set with method: **setHotspotUnderline(underlined)**

underlined parameter options:

- **True (default value)**: The undeline is visible.

```
THOTSPOT, style_number, True)
POTACTIVEFORE, True, colo
POTACTIVEUNDERLINE, True)
YLING, index_from, style_number)
ING, length, style_number)
```

- **False** : The undeline is not visible.

```
THOTSPOT, style_number, True)
POTACTIVEFORE, True, colo
POTACTIVEUNDERLINE, True)
YLING, index_from, style_number)
ING, length, style_number)
```

3.7.4. Hotspot wrapping

This setting selects if an active hotspot wraps to a new line or not. I DON'T KNOW WHAT THIS DOES! I TRIED BOTH OPTIONS AND IT DOES NOT SEEM TO EFFECT ANY HOTSPOT! IF ANYONE FIGURES OUT WHAT THIS DOES, PLEASE LET ME KNOW!

Set with method: **setHotspotWrap(wrapping)**

underlined parameter options:

- **True (default value)**: Wrapping enabled.

- **False**: Wrapping disabled.

3.7.5. Connecting to the hotspot click signal

To connect to the event when a hotspot is click to actually respond to the click, you need to connect to the **SCN_HOTSPOTCLICK, SCN_HOTSPOTDOUBLECLICK** or **SCN_HOTSPOTRELEASECLICK** signal. All three signals have the same signature. For example:

```
1.  # Connecting to a hotspot signal example
2.  def hotspot_click(position, modifiers):
3.      print("Hotspot click at position: ", str(position))
4.  editor.SCN_HOTSPOTCLICK.connect(hotspot_click)
```

This will connect the hotspot signal **SCN_HOTSPOTCLICK** (single mouse click) to the **hotspot_click** function. Now when a hotspot is clicked, it will print the character position where the editor was clicked.

Hotspot signal function signature description (same for all three signals):

```
def hotspot_click(position, modifiers)
```

- first parameter: **position (Integer)**
  The position in the editor where the click signal was emmited. It is the absolute character position in the editor's document. To get the line number and column position of from the absolute position use the editor's **lineIndexFromPosition** method.

  Example:
  ```
  # position is obtained from the function parameter
  line_number = None
  column_position = None
  editor.lineIndexFromPosition(
      position, line_number, column_position
  )
  print(line_number)
  print(column_position)
  # The line_number and column_position variables have
  # been filled with the values
  ```

- second value: **modifiers (Integer)**
  Keyboard modifiers logically **OR**-ed together that were pressed when the hotspot was clicked.

4. Lexers

The lexer is an object used for automatic styling of an editor's text everytime the editor's text changes (typing, cutting, pasting, ...), usually used for coloring text according to a programming language. For the lexer to style text, it needs to be applied/set for an instance of the **PyQt5.Qsci.QsciScintilla** object using the it's **setLexer** method.

This chapter will be an in depth description and creation of the **PyQt5.Qsci.QsciLexerCustom** object and how it is used with the QScintilla editor. **PyQt5.Qsci.QsciLexerCustom** is a predefined subclass of **PyQt5.Qsci.QsciLexer** that is used to create a custom lexer.

There are a number of predefined lexers for various languages like Python, C/C++, C#, ... (PyQt5.Qsci.QsciLexerPython, PyQt5.Qsci.QsciLexerCPP, PyQt5.Qsci.QsciLexerCSharp, ...) which can be used out of the box. Just instantiate an instance of the desired lexer and set it as the editor's lexer using the **setLexer** method and the lexer will do the rest.

But to create a custom lexer, it is needed to subclass a **PyQt5.Qsci.QsciLexerCustom** and override the necessary methods, initialize styles and create some attributes. This will also be a focus of this chapter, implementing a custom lexer and later also how to enhance the lexer with **Cython**.

The further sub-chapters will describe all the steps in creating a **Nim programming language** lexer. THIS WILL BE A BASIC LEXER EXAMPLE, IT'S UP TO THE READER TO IMPLEMENT EVERY NUANCE OF THE PROGRAMMING LANGUAGE.
For more information on the language see: https://nim-lang.org/

4.1. Creating a custom lexer

As described in the previous chapter, first it is needed to subclass the **PyQt5.Qsci.QsciLexerCustom** object, define the used styles as a dictionary and some keywords:

```python
class LexerNim(data.PyQt.Qsci.QsciLexerCustom):
    styles = {
        "Default" : 0,
        "Comment" : 1,
        "Keyword" : 2,
        "String" :  3,
        "Number" :  4,
        "Pragma" : 5,
        "Operator" : 6,
        "Unsafe" : 7,
        "Type" : 8,
    }
```

```
keyword_list = [
    "block", "const", "export", "import", "include",
    "let",
    "static", "type", "using", "var", "when",
    "as", "atomic", "bind", "sizeof",
    "break", "case", "continue", "converter",
    "discard", "distinct", "do", "echo", "elif", "else",
    "end",
    "except", "finally", "for", "from", "defined",
    "if", "interface", "iterator", "macro", "method",
    "mixin",
    "of", "out", "proc", "func", "raise", "ref", "result",
    "return", "template", "try", "inc", "dec", "new",
    "quit",
    "while", "with", "without", "yield", "true", "false",
    "assert", "min", "max", "newseq", "len", "pred",
    "succ",
    "contains", "cmp", "add", "del", "deepcopy",
    "shallowcopy",
    "abs", "clamp", "isnil", "open", "reopen",
    "close", "readall",
    "readfile", "writefile", "endoffile", "readline",
    "writeline",
]
unsafe_keyword_list = [
    "asm", "addr", "cast", "ptr", "pointer", "alloc",
    "alloc0",
    "allocshared0", "dealloc", "realloc", "nil", "gc_ref",
    "gc_unref", "copymem", "zeromem", "equalmem",
    "movemem",
    "gc_disable", "gc_enable",
]
```

Now we need to add the initialization methods and some needed method overloads:

```
def __init__(self, parent):
    # Initialize superclass
    super().__init__(parent)
    # Set the default style values
    self.setDefaultColor(
        PyQt5.QtGui.QColor(0x00, 0x00, 0x00)
    )
    self.setDefaultPaper(
        PyQt5.QtGui.QColor(0xff, 0xff, 0xff)
    )
```

```python
        self.setDefaultFont(PyQt5.QtGui.QFont("Courier", 8))
        # Initialize all style colors
        self.init_colors()
        # Init the fonts
        for i in range(len(self.styles)):
            if i == self.styles["Keyword"]:
                # Make keywords bold
                self.setFont(
                    PyQt5.QtGui.QFont("Courier", 8,
weight=PyQt5.QtGui.QFont.Black),
                    i
                )
            else:
                self.setFont(
                    PyQt5.QtGui.QFont("Courier", 8),
                    i
                )

    def init_colors(self):
        # Font color
        self.setColor(
            PyQt5.QtGui.QColor(0x00, 0x00, 0x00),
            self.styles["Default"]
        )
        self.setColor(
            PyQt5.QtGui.QColor(0x00, 0x7f, 0x00),
            self.styles["Comment"]
        )
        self.setColor(
            PyQt5.QtGui.QColor(0x00, 0x00, 0x7f),
            self.styles["Keyword"]
        )
        self.setColor(
            PyQt5.QtGui.QColor(0x7f, 0x00, 0x7f),
            self.styles["String"]
        )
        self.setColor(
            PyQt5.QtGui.QColor(0x00, 0x7f, 0x7f),
            self.styles["Number"]
        )
        self.setColor(
            PyQt5.QtGui.QColor(0x00, 0x7f, 0x40),
            self.styles["Pragma"]
        )
```

```python
        self.setColor(
            PyQt5.QtGui.QColor(0x7f, 0x7f, 0x7f),
            self.styles["Operator"]
        )
        self.setColor(
            PyQt5.QtGui.QColor(0x7f, 0x00, 0x00),
            self.styles["Unsafe"]
        )
        # Paper color
        for i in range(len(self.styles)):
            self.setPaper(
                PyQt5.QtGui.QColor(0xff, 0xff, 0xff),
                i
            )

    def language(self):
        return "Nim"

    def description(self, style):
        if style < len(self.styles):
            description = "Custom lexer for the Nim
    programming languages"
        else:
            description = ""
        return description
```

Let us first look in detail at the used methods:

- **lexer.setDefaultColor(color)**: sets the default text color

  - parameter **color: PyQt5.QtGui.QColor**

- **lexer.setDefaultPaper(color)**: sets the default paper (background) color

  - parameter **color: PyQt5.QtGui.QColor**

- **lexer.setDefaultFont(font)**: sets the default font style, check the official PyQt documentation for more details

  - parameter **font: PyQt5.QtGui.QFont**

- **lexer.setColor(color, style):** sets the text color for a specified style

  - parameter **color: PyQt5.QtGui.QColor**

  - parameter **style: Integer**, the number of the style that the text color is set for

- **lexer.setPaper(color, style):** sets the paper (background) color for a specified style

- parameter **color: PyQt5.QtGui.QColor**

- parameter **style: Integer**, the number of the style that the paper color is set for

With the above method descriptions it is now clear what the lexers **_init_** method does. It simply sets the default values for the text color, font style and paper color, and initializes the text color, font style and paper color for every style that was defined.
Then we see the **description** and **language** methods. These are needed for the lexer to operate correctly and are therefore necessary.

Now we come to the method where all the styling magic happens, the **styleText** method.

```python
def styleText(self, start, end):
    # Initialize the styling
    self.startStyling(start)
    # Tokenize the text that needs to be styled using
    # regular expressions.
    # To style a sequence of characters you need to know
    # the length of the sequence
    # and which style you wish to apply to the sequence.
    # It is up to the implementer
    # to figure out which style the sequence belongs to.
    # THE PROCEDURE SHOWN BELOW IS JUST ONE OF MANY!

    # Scintilla works with bytes, so we have to adjust the
    # start and end boundaries.
    # Like all Qt objects the lexers parent is the
    # QScintilla editor.
    text = self.parent().text()[start:end]
    # Tokenize the text using a list comprehension and
    # regular expressions
    splitter = re.compile(
        r"(\{\.|\.\}|\#|\'|\"\"\"|\n|\s+|\w+|\W)"
    )
    tokens = [
        (token, len(bytearray(token, "utf-8")))
            for token in splitter.findall(text)
    ]
    # Style the text in a loop
    for i, token in enumerate(tokens):
        if token[0] in self.keyword_list:
            # Keyword
            self.setStyling(
                token[1],
                self.styles["Keyword"]
            )
```

```python
            elif token[0] in self.unsafe_keyword_list:
                # Keyword
                self.setStyling(
                    token[1],
                    self.styles["Unsafe"]
                )
            else:
                # Style with the default style
                self.setStyling(
                    token[1],
                    self.styles["Default"]
                )
```

And that's all there is for a simple example. Let us break down the *styleText* method.
The *styleText* method has a **start** and an **end** parameter, which are integers and show from which index (absolute character position) of the editors **ENTIRE** text the text styling starts and at which index the styling ends. THIS METHOD IS EXECUTED EVERY TIME THE EDITOR'S TEXT IS CHANGED.
The **startStyling** method initializes the styling procedure to begin at the position of the **start** parameter. Then the text is sliced, so that it will only be parsed and styled from **start** to **end**.

```python
text = self.parent().text()[start:end]
```

It is allowed to parse and style the entire text every time the text changes, but for performance reasons it is not advisable to do this, especially since the parsing is done in pure Python.
Then we use a **list comprehension** and **regular expressions** to tokenize the text.

```python
splitter = re.compile(
    r"(\{\.|\.\}|\#|\'|\"\"\"|\n|\s+|\w+|\W)"
)
tokens = [
    (token, len(bytearray(token, "utf-8")))
        for token in splitter.findall(text)
]
```

The created list is a list of tuples of (token_name: **String**, token_length: **Integer**). So for example if the text contains the string **"proc"**, which is a Nim programming language keyword, the list comprehension will add a tuple ("proc", 4) to the list.
Now it is possible to style the text token-by-token. This is done by looping over the tokens in a for loop and styling the tokens as needed with the **setStyling** method. The **setStyling** method parameters are:

lexer.setStyling (number_of_characters, style)

- parameter **number_of_characters: Integer**, the number of characters that will be styled

- parameter **style: Integer**, the style number **WHICH HAS TO BE PREDEFINED OTHERWISE IF YOU CHOOSE A STYLE NUMBER THAT DOES NOT EXIST THE DEFAULT TEXT AND PAPER COLOR AND FONT STYLE WILL BE USED!**

The whole example is in the example directory *custom_lexer_basic.py* and in chapter X.4 !

4.2. Detailed description of the *setStyling* method and it's operation

The setStyling method is the main method for styling text so we will look at it in detail, because the above example may be hard to visualize what the method does.

Let us say we have the below text in an QScintilla editor which we want to style:

```
"QScintilla is a great tool."
```

And as an example in our lexer we have keywords *is* and ***tool*** that we wish to style with style number **1**, which we predefined with the **setColor**, **setPaper**, … methods. The other tokens will be styled with the style number **0**, which is our default style.

We will go through the tokens and characters in the above text manually for easier understanding. **NOTE THAT HOW YOU SPLIT THE TEXT INTO TOKENS IS COMPLETELY UP TO YOU!**
The tokens are:

```
"QScintilla"      - starts at index 0, has a length of 10 characters
" "               - starts at index 10, has a length of 1 character
"is"              - starts at index 11, has a length of 2 characters
" "               - starts at index 13, has a length of 1 character
"a"               - starts at index 14, has a length of 1 character
" "               - starts at index 15, has a length of 1 character
"great"           - starts at index 16, has a length of 5 characters
" "               - starts at index 21, has a length of 1 character
"tool"            - starts at index 22, has a length of 4 character
"."               - starts at index 26, has a length of 1 character
```

So now we can style these tokens. One important state that is not directly accessible is the **styling index**. The **styling index** is the position in the editor's text at which the styling starts when executing the **setStyling** method. The **setStyling** method also moves the **styling index** forward the number of characters that was passed as the first parameter **setStyling**'s method. So for example **lexer.setStyling(4, 0)** moves the **styling index** forward by **4** characters.

Here is the explanation of how the styling works:
- To start styling, it is required to execute the **startStyling** method! For the current example, it is required to execute:

- **lexer.startStyling(0)** – This sets the **styling index** to **0** because we are starting styling at the beginning of the text
- `QScintilla` is not a keyword so we will style it with the default number **0** style: `"QScintilla"` **lexer.setStyling(10, 0) –** style **10** characters with style **0**
- Now the **styling index** has 10 characters forward to **10**!
- Spaces are also styled with the default number **0** style: `" "` **lexer.setStyling(1, 0) –** style **1** character with style **0**
- Now the **styling index** has 1 character forward to **11**!
- `is` is a keyword so we style it with the style 1: `"is"` **lexer.setStyling(2, 1) –** style **2** characters with style **1**
- Now the **styling index** has 2 characters forward to **13**!
- And so on …

4.3. Advanced lexer functionality

4.3.1 Multiline styling

Sometime it is needed to style a piece of text across multiple lines with the same style. This appears trivial and it usually is. Just add a state variable (Boolean) that when set, marks that it is needed to style the text in a specific style until you reach a certain token.

But styling does not have to start at the beginning of the text! When you scroll through or change text the editor's **styleText** method gets executed which has the **start** and **end** parameters which determine from where the styling starts and where it ends. This becomes an issue when you are styling a multiline style (e.g.: multiline comment in C) but the **end** parameter of the **styleText** method is before the ending token for the multiline style is found. So the next time the text is for example scrolled down, the styling starts in the middle of a multiline comment and styling text normally, where correctly it should continue multiline styling.

To solve this problem, QScintilla has a feature to check the style of a character in the editor's text. This is done by using the **SCI_GETSTYLEAT** message with the **SendScintilla** method, which return the style number of a selected character.

How this is used is by checking if the styling starts at an index different front **0** and checking if the character **(start – 1)** has a multiline style and the setting the appropriate state flag. This will become clear in the below example, which is just an upgrade from the **styleText** method from chapter 4.1. Creating a custom lexer:

```python
def styleText(self, start, end):

    # Initialize the styling
    self.startStyling(start)
    # Set the text
    text = self.parent().text()[start:end]
    # Tokenize the text using a list comprehension and
    # regular expressions
    splitter = re.compile(
```

```python
        r"(\{\.|\.\}|\#|\'|\"\"\"|\n|\s+|\w+|\W)"
    )
    tokens = [
        (token, len(bytearray(token, "utf-8")))
            for token in splitter.findall(text)
    ]
    # Multiline styles
    multiline_comment_flag = False
    # Check previous style for a multiline style
    if start != 0:
        previous_style = editor.SendScintilla(
            editor.SCI_GETSTYLEAT,
            start - 1
        )
        if previous_style == self.styles["MultilineComment"]:
            multiline_comment_flag = True
    # Style the text in a loop
    for i, token in enumerate(tokens):
        if (multiline_comment_flag == False and
            token[0] == "#" and
            tokens[i+1][0] == "["):
                # Start of a multiline comment
                self.setStyling(
                    token[1], self.styles["MultilineComment"]
                )
                # Set the multiline comment flag
                multiline_comment_flag = True
        elif multiline_comment_flag == True:
            # Multiline comment flag is set
            self.setStyling(
                token[1], self.styles["MultilineComment"]
            )
            # Check if a multiline comment ends
            if token[0] == "#" and tokens[i-1][0] == "]":
                multiline_comment_flag = False
        elif token[0] in self.keyword_list:
            # Keyword
            self.setStyling(
                token[1],
                self.styles["Keyword"]
            )
        elif token[0] in self.unsafe_keyword_list:
            # Keyword
            self.setStyling(
```

```python
                token[1],
                self.styles["Unsafe"]
            )
        else:
            # Style with the default style
            self.setStyling(
                token[1],
                self.styles["Default"]
            )
```

The whole example is in the example directory *custom_lexer_advanced.py* and in chapter X.5 !

4.3.2 Code folding

Code folding is the ability of QScintilla to fold parts of the editor's text. All built-in lexers have code folding implemented for e.g.: functions, structures, classes, ...

Text can be folded and unfolded using the folding margin, shown in the image below:
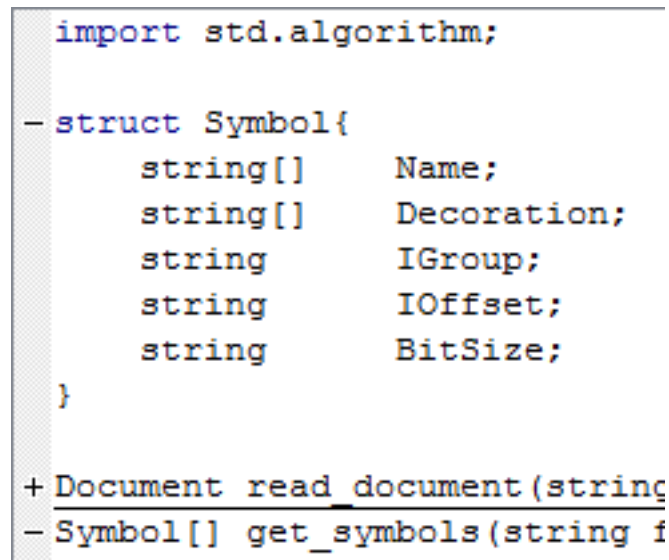
I do not use code folding so if someone is wished to detail this functionality it would be greatly appreciated.

4.3.2 Clickable styles

It is possible to add click functionality to a style by setting the style as a hotspot style. To do this it is only needed to set the style to be a hotspot style with the **SCI_STYLESETHOTSPOT** message with the **editor.SendScintilla** method, like so:

```python
editor.SendScintilla(
    PyQt5.Qsci.QsciScintillaBase.SCI_STYLESETHOTSPOT,
    style_number,
    True
)
```

The **True** parameter above enables the style to be a hotspot style.

What the clicks in a style do it completely up to the implementer. In the *custom_lexer_advanced.py* example, in the lexers **__init__** method the **Keywords** style is set as a hotspot style, with the clicks replacing the keyword with the **"CLICK"** string.

5. Basic examples

Below will be the basic learning examples to get you familiarized with how to setup and put the QScintilla editor component on the screen and start editing some text. Most of the parts of the QScintilla visual description chapter will be explained here briefly.

As this is learn-by-example guide, in this chapter the core functionalities will be described in much detail.

For all examples below on GNU/Linux operating systems substitute the executing command "**python script_name.py**" with "**python3 script_name.py**" if you have both Python2 and Python3 installed.

ALL THE EXAMPLES SHOULD BE PROVIDED WITH THIS DOCUMENTATION. Otherwise the code examples can be copied into your favourite editor, saved and executed with Python 3.

5.1. Qscintilla's "Hello World" example

So lets start with the mandatory "Hello World" example. The code is in chapter X, link here: X.1. Hello World.

Execute the example file ***hello_qscintilla.py*** with Python 3, provided along with this documentation. If you do not have the example files, save the above code to a text file called ***hello_qscintilla.py*** and run it using:

```
python hello_qscintilla.py
```

A new window should appear with a QScintilla text editing component inside it with the text "Hello World" already entered into the editor's editing area. Click inside the window and type something. You will see that you're already editing text.

5.1.1. Code breakdown
Pretty straightforward code, here are the key points:

- We import all of the needed modules *(lines 1 to 6)*

- Create a PyQt5.QtWidgets.QApplication object instance *(line 10)*. This is the main object which needs to be initialized and handles things like widget initialization and finalization.

- Create a PyQt5.Qsci.QsciScintilla object instance *(line 13)*. This is our main editing object.

- Show the QScintilla editor widget using the Qwidget's **show** method *(line 16)*. As QScintilla inherits from the QWidget object, it has all of QWidget's methods. The **show** method shows the widget and its child widgets on the screen in a new window. **This way of displaying the QScintilla editor on the screen in not recommended! Always use a PyQt5.QtWidgets.QMainWindow as the main**

widget and set it's central widget to be the editor! That way all events in the editor will be handeled properly. BUT FOR THIS SIMPLE EXAMPLE IT WILL BE OK.

- Next is an example of directly setting the QScintilla editor's text using the [setText](#) method *(line 18)*. This method sets the text to the string you enter as the argument **AND** also resets the undo/redo buffer of the editor. QScintilla has an internal buffer that you that stores all of the text changes until it runs out of memory and cannot store any more. I personally have never reached this limit.

- And lastly we execute the application *(line 21)*. This is called in all PyQt applications, but sometimes with a different style like `sys.exit(app.exec_())`, so that the application does not run any code after the PyQt application is closed.

5.2. Customization example

Let's show some customizing of various QScintilla settings by modifying the previous example. This example will disable the editor's lexer. Some customization features are only accessible with a lexer, but that will be shown in a later example. The entire code example is in chapter X, link here: X.2. Customization

Run the example ***customization.py*** or save the code to a file and execute it with Python 3. There is a lot going on in this example, so each of the sections will be explained in detail in the next paragraph.

5.2.1. Code breakdown

This example will be broken down into separate sections. The sections are devided with docstrings, the triple double-quoted strings at the top of each section.

- **Initialization:**

  Nothing special here, same basic initialization process. The only thing added was the **append** method, which just adds the parameter string to the editors text.

- **General custumizations:**

  ```
  """
  Customization - GENERAL
  """
  ```
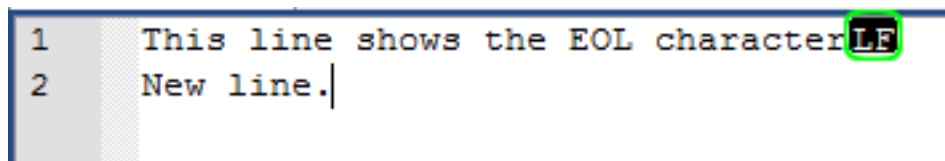
  1. `# Disable the lexer`
  2. `editor.setLexer(None)`
  3. `# Set the encoding to UTF-8`
  4. `editor.setUtf8(True)`
  5. `# Set the End-Of-Line character to Unix style ('\n')`
  6. `editor.setEolMode(PyQt5.Qsci.QsciScintilla.EolUnix)`
  7. `# Make End-Of-Line characters visible`
  8. `editor.setEolVisibility(True)`

```python
9.  # Make selecting text between multiple lines paint
10. # the selection field to the end of the editor, not
11. # just to the end of the line text.
12. editor.setSelectionToEol(True)
13. # Set the zoom factor, the factor is in points.
14. editor.zoomTo(4)
```

- Disabling the lexer is done in line 2. We simply set the lexer to **None** with the **setLexer** method. As described in chapter 1.6.1. Default lexer, by default the lexer is disabled, so this step can be skipped.

- In line 4 we set the editor's encoding to UTF-8 with the **setUtf8** method. If the **setUtf8** method's parameter is **False**, the QScintilla editor's encoding will be set to ASCII and will show non-ASCII characters as **questionmarks (?)**. Be careful when changing the encoding from UTF-8 to ASCII at runtime, you will get some interesting squiggles and symbols if you used UTF-8 characters.

- Line 6 set the End-Of-Line mode for the editor's document with the **setEolMode** `method`. This selects how line endings will be interpreted when using the **text** method. This comes into play when you will be saving or manipulating the editor's text in code. In the above example the line endings are set to **Unix** mode, which means the **Line-Feed (\n)** character. All End-Of-Line options are described in 3.2. End-Of-Line (EOL) options .

- Line 8 we set the End-Of-Line (EOL) character to be visible with the **setEolVisibility** method. In the below example the EOL character is circled in green:

- In line 10 we set the zoom factor for the editor with the **zoomTo** method. This sets the current base font size to the size you set the parameter of the method to. Experiment with different parameters to se the effects. This recalculates all the font sizes in the editor.

- **Line wrapping custumizations:**

```python
1. """
2. Customization – LINE WRAPPING
3. """
4. # Set the text wrapping mode to word wrap
5. editor.setWrapMode(PyQt5.Qsci.QsciScintilla.WrapWord)
6. # Set the text wrapping mode visual indication
7. editor.setWrapVisualFlags(PyQt5.Qsci.QsciScintilla.WrapFlagByText)
8. # Set the text wrapping to indent the wrapped lines
9. editor.setWrapIndentMode(PyQt5.Qsci.QsciScintilla.WrapIndentSame)
```

- ○ Wrapping mode is selected in line 5 with the **setWrapMode** method. There are 4 wrap modes to choose from, described in 3.1.1. Text wrapping mode. Here we selected to wrap text at word boundaries.
- ○ Choosing how wrapping is indicated is set in line 7 with **setWrapVisualFlags** methods.

- **Indentation customizations:**

  These customizations effect the indentation functionality when pressing the **Tab / Shift + Tab** buttons or using the **indent / unindent** methods.

```
1.  """
2.  Customization – INDENTATION
3.  """
4.  # Set indentation with spaces instead of tabs
5.  editor.setIndentationsUseTabs(False)
6.  # Set the tab width to 4 spaces
7.  editor.setTabWidth(4)
8.  # Set tab indent mode, True indents the with the next
9.  # non-whitespace character while False inserts
10. # a tab character
11. editor.setTabIndents(True)
12. # Set autoindentation mode to maintain the indentation
13. # level of the previous line (the editor's lexer HAS
14. # to be disabled)
15. editor.setAutoIndent(True)
16. # Make the backspace jump back to the tab width guides
17. # instead of deleting one character, but only when
18. # there are ONLY whitespaces / tab characters on the
19. # left side of the cursor
20. editor.setBackspaceUnindents(True)
21. # Set indentation guides to be visible
22. editor.setIndentationGuides(True)
```

- ○ In line 5 we choose the indentation functionality to use whitespace characters instead of the tab character (\t) with the **setIndentationsUseTabs** method.
- ○ Line 7 sets the number of characters the indentation functionality indents / unindents by with the **setTabWidth** method. In the above example the indentation is increased or decreased by 4 characters when indenting / unindenting (pressing **Tab** or **Shift+Tab**). More details in chapter 3.3.2. Indentation size.
- ○ Line 11 sets the special indentation functionality. When set to **False** and when the cursor is in set to a whitespace character in a line and then indent is executed (e.g.: by

pressing **Tab**), a Tab character is inserted. For more detailed explanation see chapter 3.3.4. Indentation at spaces options.

- Line 15 sets smart indentation on with the **setAutoIndent** method, which means that when inserting a new line character/s by pressing **Enter/Return**, the new line will be indented to the same indentation level as the previous NON empty line.
- Line 20 sets the **Backspace** key functionality with the **setBackspaceUnindents** method. When set to **True**, this makes pressing the backspace key unindent to columns aligned to the tab width property. Basicaly when pressing backspace and there are only whitespaces before the cursor position, the cursor will move exactly like as if it was unindenting.
- Line 22 makes the indentation guides (vertical dots where indentations align) visible if set to **True** with the **setIndentationGuides** method.

FUTURE CHAPTERS:

- ○ Show an example of a **Cython** compiled lexer

X. Appendix: code examples

X.1. Hello World

```python
# Import the PyQt module with all of the graphical widgets
import PyQt5.QtGui
# Import the QScintilla module
import PyQt5.Qsci
# Import Python's sys module needed to get the application arguments
import sys

# Create the main PyQt application object and give it the
# command line arguments passed to the Python application
application = PyQt5.QtWidgets.QApplication(sys.argv)

# Create a QScintilla editor instance
editor = PyQt5.Qsci.QsciScintilla()
# Put the "Hello World" text into the editing area of the editor
editor.setText("Hello World")
# As the QScintilla object inherits from the QWidget object it has
# the show method which displays the widget on the screen. THIS WAY
# OF DISPLAYING THE EDITOR SHOULD NOT BE USE FOR ANYTHING OTHER THAN
# TESTING PURPOSES! USE THE QMainWindow WIDGET AND SET IT'S CENTRAL
# WIDGET TO BE THE EDITOR, SO THAT THE EVENTS ARE PROPERLY HANDLED!
editor.show()

# Execute the application
application.exec_()
```

X.2. Customization

```python
"""
Initialization
"""
# Import the PyQt5 module with some of the GUI widgets
import PyQt5.QtWidgets
# Import the QScintilla module
import PyQt5.Qsci
# Import Python's sys module needed to get the
# application arguments
import sys

# Create the main PyQt application object
application = PyQt5.QtWidgets.QApplication(sys.argv)

# Create a QScintila editor instance
editor = PyQt5.Qsci.QsciScintilla()
# Set the initial text
editor.setText("Inicialization example.")
# Append some text
editor.append("\nAdded text.")

"""
Customization - GENERAL
"""
# Disable the lexer
editor.setLexer(None)
# Set the encoding to UTF-8
editor.setUtf8(True)
# Set the End-Of-Line character to Unix style ('\n')
editor.setEolMode(PyQt5.Qsci.QsciScintilla.EolUnix)
# Make End-Of-Line characters visible
editor.setEolVisibility(True)
# Set the zoom factor, the factor is in points.
editor.zoomTo(4)

"""
Customization - LINE WRAPPING
"""
# Set the text wrapping mode to word wrap
editor.setWrapMode(PyQt5.Qsci.QsciScintilla.WrapWord)
# Set the text wrapping mode visual indication
```

```python
editor.setWrapVisualFlags(PyQt5.Qsci.QsciScintilla.WrapFlagByText)
# Set the text wrapping to indent the wrapped lines
editor.setWrapIndentMode(PyQt5.Qsci.QsciScintilla.WrapIndentSame)

"""
Customization - EDGE MARKER
"""
# Set the edge marker's position and set it to color the
background
# when a line goes over the limit of 50 characters
editor.setEdgeMode(PyQt5.Qsci.QsciScintilla.EdgeBackground)
editor.setEdgeColumn(50)
edge_color = caret_fg_color = PyQt5.QtGui.QColor("#ff00ff00")
editor.setEdgeColor(edge_color)
# Add a long line that will display the edge marker coloring
editor.append("\nSome long line that will display the edge
marker's functionality.")

"""
Customization - INDENTATION
"""
# Set indentation with spaces instead of tabs
editor.setIndentationsUseTabs(False)
# Set the tab width to 4 spaces
editor.setTabWidth(4)
# Set tab indent mode, see the 3.3.4 chapter in QSciDocs
# for a detailed explanation
editor.setTabIndents(True)
# Set autoindentation mode to maintain the indentation
# level of the previous line (the editor's lexer HAS
# to be disabled)
editor.setAutoIndent(True)
# Make the backspace jump back to the tab width guides
# instead of deleting one character, but only when
# there are ONLY whitespaces on the left side of the
# cursor
editor.setBackspaceUnindents(True)
# Set indentation guides to be visible
editor.setIndentationGuides(True)

"""
Customization - CARET (the blinking cursor indicator)
"""
# Set the caret color to red
```

```python
caret_fg_color = PyQt5.QtGui.QColor("#ffff0000")
editor.setCaretForegroundColor(caret_fg_color)
# Enable and set the caret line background color to slightly
transparent blue
editor.setCaretLineVisible(True)
caret_bg_color = PyQt5.QtGui.QColor("#7f0000ff")
editor.setCaretLineBackgroundColor(caret_bg_color)
# Set the caret width of 4 pixels
editor.setCaretWidth(4)

"""
Customization - AUTOCOMPLETION (Partially usable without a lexer)
"""
# Set the autocompletions to case INsensitive
editor.setAutoCompletionCaseSensitivity(False)
# Set the autocompletion to not replace the word to the right
of the cursor
editor.setAutoCompletionReplaceWord(False)
# Set the autocompletion source to be the words in the
# document
editor.setAutoCompletionSource(PyQt5.Qsci.QsciScintilla.AcsDocument)
# Set the autocompletion dialog to appear as soon as 1
character is typed
editor.setAutoCompletionThreshold(1)

# For the QScintilla editor to properly process events we
# need to add it to a QMainWindow object.
main_window = PyQt5.QtWidgets.QMainWindow()
# Set the central widget of the main window to
# be the editor
main_window.setCentralWidget(editor)
# Resize the main window and show it on the screen
main_window.resize(800, 600)
main_window.show()

# Execute the application
application.exec_()
```

X.3. Margins

```python
# Import the PyQt5 module with some of the GUI widgets
import PyQt5.QtWidgets
# Import the QScintilla module
import PyQt5.Qsci
# Import Python's sys module needed to get the application
arguments
import sys

# Create the main PyQt application object
application = PyQt5.QtWidgets.QApplication(sys.argv)

# Create a QScintila editor instance
editor = PyQt5.Qsci.QsciScintilla()
# Set the initial text
editor.setText("Inicialization example.")
# Append some text
editor.append("\nAdded text.")

"""
Line number margin
"""
# Set margin 0 as the line margin (this is the default, so you
can skip this step)
editor.setMarginType(0, PyQt5.Qsci.QsciScintilla.NumberMargin)
# Set the width of the line number margin with a string, which
sets the width
# to the width of the entered string text. There is also an
alternative function
# with the same name which you can use to set the width
directly in number of pixels.
editor.setMarginWidth(0, "00")

"""
Symbol margin - Used to display custom symbols
"""
# Set margin 1 as a symbol margin (this is the default, so you
can skip this step)
editor.setMarginType(1, PyQt5.Qsci.QsciScintilla.SymbolMargin)
# Set the margin's width
editor.setMarginWidth(1, 20)
# Prepare an image for the marker
```

```python
image_scale_size = PyQt5.QtCore.QSize(16, 16)
marker_image = PyQt5.QtGui.QPixmap("marker_image.png")
scaled_image = marker_image.scaled(image_scale_size)
# Set the margin mask (mask constant ~SC_MASK_FOLDERS enables
all markers!)
# This sets which of the 32 markers will be visible on margin
1
editor.setMarginMarkerMask(
  1, ~PyQt5.Qsci.QsciScintillaBase.SC_MASK_FOLDERS
)
# Just for info we display the margin mask, which should be:
0b11111111111111111111111111
# which means all 32 markers are enabled.
print(bin(editor.marginMarkerMask(1)))
# Create and add marker on margin 1
marker = editor.markerDefine(scaled_image, 1)
editor.markerAdd(1, 1)


"""
Symbol margin with the background color set to the editor's default
paper (background) color
"""
# Set margin 2 as a symbol margin with customizable background
color
editor.setMarginType(2,
PyQt5.Qsci.QsciScintilla.SymbolMarginDefaultBackgroundColor)
# Set the margin's width
editor.setMarginWidth(2, "000000")


"""
Symbol margin with the background color set to the editor's default
font color
"""
# Set margin 4 as a symbol margin with customizable background
color
editor.setMarginType(3,
PyQt5.Qsci.QsciScintilla.SymbolMarginDefaultForegroundColor)
# Set the margin's width
editor.setMarginWidth(3, "0000")
# Set the margin mask to display all markers
editor.setMarginMarkerMask(
    3, 0b11111111111111111111111111
)
```

```python
# Add a marker that is built into QScintilla.
# Note that the marker will be displayed on all symbol margins
with the
# third marker bit set to '1'!
marker =
editor.markerDefine(PyQt5.Qsci.QsciScintilla.Rectangle, 2)
editor.markerAdd(0, 2)


"""
Text margin
"""
# Set margin 5 as a symbol margin with customizable background
color
editor.setMarginType(4, PyQt5.Qsci.QsciScintilla.TextMargin)
# Set the margin's width
editor.setMarginWidth(4, "00000")
# Set the margin's text on line 1 font style 0
# Style 0 is taken from the current lexer, I think!
editor.setMarginText(0, "line1", 0)
# Create a new style and set it on line 2 with some text
style = PyQt5.Qsci.QsciStyle(
  2, # style (This has to be set to something other than 0, as
that is the default)
  "new style", # description
  PyQt5.QtGui.QColor(0x8a, 0xe2, 0x34, 80), # color (font)
  PyQt5.QtGui.QColor(0xff, 0xff, 0xff, 0xff), # paper
(background)
  PyQt5.QtGui.QFont('Helvetica', 10), # font
  eolFill=False, # End-Of-Line Fill
)
editor.setMarginText(1, "line2", style)


# For the QScintilla editor to properly process events we need
to add it to
# a QMainWindow object.
main_window = PyQt5.QtWidgets.QMainWindow()
# Set the central widget of the main window to be the editor
main_window.setCentralWidget(editor)
# Resize the main window and show it on the screen
main_window.resize(800, 600)
main_window.show()


# Execute the application
```

```python
    application.exec_()
```

X.4. Custom lexer – basic

```python
# Import the PyQt5 module with some of the GUI widgets
import PyQt5.QtWidgets
# Import the QScintilla module
import PyQt5.Qsci
# Import Python's sys module needed to get the application
arguments
import sys
import re

# Create a custom Nim lexer
class LexerNim(PyQt5.Qsci.QsciLexerCustom):
  styles = {
    "Default" : 0,
    "Keyword" : 1,
    "Unsafe" : 2,
    "MultilineComment" : 3,
  }
  keyword_list = [
    "block", "const", "export", "import", "include", "let",
    "static", "type", "using", "var", "when",
    "as", "atomic", "bind", "sizeof",
    "break", "case", "continue", "converter",
    "discard", "distinct", "do", "echo", "elif", "else", "end",
    "except", "finally", "for", "from", "defined",
    "if", "interface", "iterator", "macro", "method", "mixin",
    "of", "out", "proc", "func", "raise", "ref", "result",
    "return", "template", "try", "inc", "dec", "new", "quit",
    "while", "with", "without", "yield", "true", "false",
    "assert", "min", "max", "newseq", "len", "pred", "succ",
    "contains", "cmp", "add", "del","deepcopy", "shallowcopy",
    "abs", "clamp", "isnil", "open", "reopen", "close","readall",
```

```python
        "readfile", "writefile", "endoffile", "readline",
"writeline",
    ]
    unsafe_keyword_list = [
        "asm", "addr", "cast", "ptr", "pointer", "alloc", "alloc0",
        "allocshared0", "dealloc", "realloc", "nil", "gc_ref",
        "gc_unref", "copymem", "zeromem", "equalmem", "movemem",
        "gc_disable", "gc_enable",
    ]

    def __init__(self, parent=None):
        # Initialize superclass
        super().__init__(parent)
        # Set the default style values
        self.setDefaultColor(PyQt5.QtGui.QColor(0x00, 0x00, 0x00))
        self.setDefaultPaper(PyQt5.QtGui.QColor(0xff, 0xff, 0xff))
        self.setDefaultFont(PyQt5.QtGui.QFont("Courier", 8))
        # Initialize all style colors
        self.init_colors()
        # Init the fonts
        for i in range(len(self.styles)):
            if i == self.styles["Keyword"]:
                self.setFont(PyQt5.QtGui.QFont("Courier", 8,
weight=PyQt5.QtGui.QFont.Black), i)
            else:
                self.setFont(PyQt5.QtGui.QFont("Courier", 8), i)

    def init_colors(self):
        # Font color
        self.setColor(PyQt5.QtGui.QColor(0x00, 0x00, 0x00),
self.styles["Default"])
        self.setColor(PyQt5.QtGui.QColor(0x00, 0x00, 0x7f),
self.styles["Keyword"])
        self.setColor(PyQt5.QtGui.QColor(0x7f, 0x00, 0x00),
self.styles["Unsafe"])
        self.setColor(PyQt5.QtGui.QColor(0x7f, 0x7f, 0x00),
self.styles["MultilineComment"])
        # Paper color
        for i in range(len(self.styles)):
            self.setPaper(PyQt5.QtGui.QColor(0xff, 0xff, 0xff), i)
```

```python
    def language(self):
        return "Nim"

    def description(self, style):
        if style < len(self.styles):
            description = "Custom lexer for the Nim programming
languages"
        else:
            description = ""
        return description

    def styleText(self, start, end):
        # Initialize the styling
        self.startStyling(start)
        # Tokenize the text that needs to be styled using regular
expressions.
        # To style a sequence of characters you need to know the
length of the sequence
        # and which style you wish to apply to the sequence. It is
up to the implementer
        # to figure out which style the sequence belongs to.
        # THE PROCEDURE SHOWN BELOW IS JUST ONE OF MANY!
        splitter = re.compile(r"(\
{\.|\.\}|\#|\'|\"\"\"\"|\n|\s+|\w+|\W)")
        # Scintilla works with bytes, so we have to adjust the
start and end boundaries.
        # Like all Qt objects the lexers parent is the QScintilla
editor.
        text = self.parent().text()[start:end]
        tokens = [(token, len(bytearray(token, "utf-8"))) for token in
splitter.findall(text)]
        # Style the text in a loop
        for i, token in enumerate(tokens):
            if token[0] in self.keyword_list:
                # Keyword
                self.setStyling(token[1], self.styles["Keyword"])
            elif token[0] in self.unsafe_keyword_list:
                # Keyword
                self.setStyling(token[1], self.styles["Unsafe"])
            else:
                # Style with the default style
```

```
        self.setStyling(token[1], self.styles["Default"])


# Create the main PyQt application object
application = PyQt5.QtWidgets.QApplication(sys.argv)

# Create a QScintila editor instance
editor = PyQt5.Qsci.QsciScintilla()
# Set the lexer to the custom Nim lexer
nim_lexer = LexerNim(editor)
editor.setLexer(nim_lexer)
# Set the initial text
editor.setText(
"""
proc python_style_text*(self, args: PyObjectPtr): PyObjectPtr
{.exportc, cdecl.} =
  var
    result_object: PyObjectPtr
    value_start, value_end: cint
    lexer, editor: PyObjectPtr
    parse_result: cint

  parse_result = argParseTuple(
    args,
    "iiOO",
    addr(value_start),
    addr(value_end),
    addr(lexer),
    addr(editor),
  )
  if parse_result == 0:
    echo "Napaka v pretvarjanju argumentov v funkcijo!"
    returnNone()
  var
    text_method = objectGetAttr(editor, buildValue("s",
cstring("text")))
    text_object = objectCallObject(text_method, tupleNew(0))
    string_object = unicodeAsEncodedString(text_object, "utf-8",
nil)
    cstring_whole_text = bytesAsString(string_object)
    whole_text = $cstring_whole_text
```

```nim
    text = whole_text[int(value_start)..int(value_end-1)]
    text_length = len(text)
    current_token: string = ""
  # Prepare the objects that will be called as functions
  var
    start_styling_obj: PyObjectPtr
    start_args: PyObjectPtr
    set_styling_obj: PyObjectPtr
    set_args: PyObjectPtr
    send_scintilla_obj: PyObjectPtr
    send_args: PyObjectPtr
  start_styling_obj = objectGetAttr(lexer, buildValue("s",
cstring("startStyling")))
  start_args = tupleNew(1)
  set_styling_obj = objectGetAttr(lexer, buildValue("s",
cstring("setStyling")))
  set_args = tupleNew(2)
  send_scintilla_obj = objectGetAttr(editor, buildValue("s",
cstring("SendScintilla")))
  send_args = tupleNew(2)

  # Template for final cleanup
  template clean_up() =
    xDecref(text_method)
    xDecref(text_object)
    xDecref(string_object)
    xDecref(args)
    xDecref(result_object)

    xDecref(start_styling_obj)
    xDecref(start_args)
    xDecref(set_styling_obj)
    xDecref(set_args)
    xDecref(send_scintilla_obj)
    xDecref(send_args)
  # Template for the lexers setStyling function
  template set_styling(length: int, style: int) =
    discard tupleSetItem(set_args, 0, buildValue("i", length))
    discard tupleSetItem(set_args, 1, buildValue("i", style))
    discard objectCallObject(set_styling_obj, set_args)
  # Procedure for getting previous style
```

```nim
  proc get_previous_style(): int =
    discard tupleSetItem(send_args, 0, buildValue("i",
SCI_GETSTYLEAT))
    discard tupleSetItem(send_args, 1, buildValue("i",
value_start - 1))
    result = longAsLong(objectCallObject(send_scintilla_obj,
send_args))
    xDecref(send_args)
  # Template for starting styling
  template start_styling() =
    discard tupleSetItem(start_args, 0, buildValue("i",
value_start))
    discard objectCallObject(start_styling_obj, start_args)
  # Safety
  if set_styling_obj == nil:
    raise newException(FieldError, "Lexer doesn't contain the
'setStyling' method!")
  elif start_styling_obj == nil:
    raise newException(FieldError, "Lexer doesn't contain the
'startStyling' method!")
  elif send_scintilla_obj == nil:
    raise newException(FieldError, "Editor doesn't contain the
'SendScintilla' method!")
  # Styling initialization
  start_styling()
  #-------------------------------------------------------
  var
    actseq = SeqActive(active: false)
    token_name: string = ""
    previous_token: string = ""
    token_start: int = 0
    token_length: int = 0
  # Check previous style
  if value_start != 0:
    var previous_style = get_previous_style()
    for i in multiline_sequence_list:
      if previous_style == i.style:
        actseq.sequence = i
        actseq.active = true
        break
  # Style the tokens accordingly
```

```
proc check_start_sequence(pos: int, sequence: var SeqActive):
bool =
    for s in sequence_lists:
      var found = true
      for i, ch in s.start.pairs:
        if text[pos+i] != ch:
          found = false
          break
      if found == false:
        continue
      sequence.sequence = s
      return true
    return false

  proc check_stop_sequence(pos: int, actseq: SeqActive): bool =
    if text[pos] in actseq.sequence.stop_extra:
      return true
    if pos > 0 and (text[pos-1] in
actseq.sequence.negative_lookbehind):
      return false
    for i, s in actseq.sequence.stop.pairs:
      if text[pos+i] != s:
        return false
    return true

  template style_token(token_name: string, token_length: int) =
    if token_length > 0:
      if token_name in keywords:
        set_styling(token_length, styles["Keyword"])
        previous_token = token_name
      elif token_name in custom_keywords:
        set_styling(token_length, styles["CustomKeyword"])
      elif token_name[0].isdigit() or (token_name[0] == '.' and
token_name[1].isdigit()):
        set_styling(token_length, styles["Number"])
      elif previous_token == "class":
        set_styling(token_length, styles["ClassName"])
        previous_token = ""
      elif previous_token == "def":
        set_styling(token_length, styles["FunctionMethodName"])
```

```
        previous_token = ""
      else:
        set_styling(token_length, styles["Default"])

  var i = 0
  token_start = i
  while i < text_length:
    if actseq.active == true or check_start_sequence(i, actseq) ==
true:
      #[
        Multiline sequence already started in the previous line or
        a start sequence was found
      ]#
      if actseq.active == false:
        # Style the currently accumulated token
        token_name = text[token_start..i]
        token_length = i - token_start
        style_token(token_name, token_length)
      # Set the states and reset the flags
      token_start = i
      token_length = 0
      if actseq.active == false:
        i += len(actseq.sequence.start)
      while i < text_length:
        # Check for end of comment
        if check_stop_sequence(i, actseq) == true:
          i += len(actseq.sequence.stop)
          break
        i += 1
      # Style text
      token_length = i - token_start
      set_styling(token_length, actseq.sequence.style)
      # Style the separator tokens after the sequence, if any
      token_start = i
      while text[i] in extended_separators and i < text_length:
        i += 1
      token_length = i - token_start
      if token_length > 0:
        set_styling(token_length, styles["Default"])
      # Set the states and reset the flags
```

```
      token_start = i
      token_length = 0
      # Skip to the next iteration, because the index is
already
      # at the position at the end of the string
      actseq.active = false
      continue
    elif text[i] in extended_separators:
      #[
        Separator found
      ]#
      token_name = text[token_start..i-1]
      token_length = len(token_name)
      if token_length > 0:
        style_token(token_name, token_length)
        token_start = i
        while text[i] in extended_separators and i < text_length:
          i += 1
        token_length = i - token_start
        if token_length > 0:
          set_styling(token_length, styles["Default"])
        # State reset
        token_start = i
        token_length = 0
        continue
      else:
        while text[i] in extended_separators and i < text_length:
          i += 1
        token_length = i - token_start
        if token_length > 0:
          set_styling(token_length, styles["Default"])
        # State reset
        token_start = i
        token_length = 0
        continue
    # Update loop variables
    inc(i)
    # Check for end of text
    if i == text_length:
      token_name = text[token_start..i-1]
```

```python
            token_length = len(token_name)
            style_token(token_name, token_length)
        elif i > text_length:
            raise newException(
                IndexError,
                "Styling went over the text length limit!"
            )
    #--------------------------------------------------------
    clean_up()
    returnNone()
"""
)


# For the QScintilla editor to properly process events we need to add it to
# a QMainWindow object.
main_window = PyQt5.QtWidgets.QMainWindow()
# Set the central widget of the main window to be the editor
main_window.setCentralWidget(editor)
# Resize the main window and show it on the screen
main_window.resize(800, 600)
main_window.show()

# Execute the application
application.exec_()
```

X.5. Custom lexer – advanced

```python
# Import the PyQt5 module with some of the GUI widgets
import PyQt5.QtWidgets
# Import the QScintilla module
import PyQt5.Qsci
# Import Python's sys module needed to get the application
arguments
import sys
import re

# Create a custom Nim lexer
class LexerNim(PyQt5.Qsci.QsciLexerCustom):
    styles = {
        "Default" : 0,
        "Keyword" : 1,
        "Unsafe" : 2,
        "MultilineComment" : 3,
    }
    keyword_list = [
        "block", "const", "export", "import", "include", "let",
        "static", "type", "using", "var", "when",
        "as", "atomic", "bind", "sizeof",
        "break", "case", "continue", "converter",
        "discard", "distinct", "do", "echo", "elif", "else", "end",
        "except", "finally", "for", "from", "defined",
        "if", "interface", "iterator", "macro", "method", "mixin",
        "of", "out", "proc", "func", "raise", "ref", "result",
        "return", "template", "try", "inc", "dec", "new", "quit",
        "while", "with", "without", "yield", "true", "false",
        "assert", "min", "max", "newseq", "len", "pred", "succ",
        "contains", "cmp", "add", "del","deepcopy", "shallowcopy",
        "abs", "clamp", "isnil", "open", "reopen", "close","readall",
        "readfile", "writefile", "endoffile", "readline",
"writeline",
    ]
    unsafe_keyword_list = [
        "asm", "addr", "cast", "ptr", "pointer", "alloc", "alloc0",
        "allocshared0", "dealloc", "realloc", "nil", "gc_ref",
        "gc_unref", "copymem", "zeromem", "equalmem", "movemem",
        "gc_disable", "gc_enable",
```

```python
    ]

    def __init__(self, parent=None):
        # Initialize superclass
        super().__init__(parent)
        # Set the default style values
        self.setDefaultColor(PyQt5.QtGui.QColor(0x00, 0x00, 0x00))
        self.setDefaultPaper(PyQt5.QtGui.QColor(0xff, 0xff, 0xff))
        self.setDefaultFont(PyQt5.QtGui.QFont("Courier", 8))
        # Initialize all style colors
        self.init_colors()
        # Init the fonts
        for i in range(len(self.styles)):
            if i == self.styles["Keyword"]:
                # Make keywords bold
                self.setFont(PyQt5.QtGui.QFont("Courier", 8,
weight=PyQt5.QtGui.QFont.Black), i)
            else:
                self.setFont(PyQt5.QtGui.QFont("Courier", 8), i)
        # Set the Keywords style to be clickable with hotspots
        # using the scintilla low level messaging system
        parent.SendScintilla(
            PyQt5.Qsci.QsciScintillaBase.SCI_STYLESETHOTSPOT,
            self.styles["Keyword"],
            True
        )
        parent.SendScintilla(
            PyQt5.Qsci.QsciScintillaBase.SCI_SETHOTSPOTACTIVEFORE,
            True,
            PyQt5.QtGui.QColor(0x00, 0x7f, 0xff)
        )
        parent.SendScintilla(

PyQt5.Qsci.QsciScintillaBase.SCI_SETHOTSPOTACTIVEUNDERLINE,
True
        )
        # Define a hotspot click function
        def hotspot_click(position, modifiers):
            """
            Simple example for getting the clicked token
```

```python
        """
        text = parent.text()
        delimiters = [
            '(', ')', '[', ']', '{', '}', ' ', '.', ',', ';', '-',
            '+', '=', '/', '*', '#'
        ]
        start = 0
        end = 0
        for i in range(position+1, len(text)):
            if text[i] in delimiters:
                end = i
                break
        for i in range(position, -1, -1):
            if text[i] in delimiters:
                start = i
                break
        clicked_token = text[start:end].strip()
        # Print the token and replace it with the string "CLICK"
        print("'" + clicked_token + "'")
        parent.setSelection(0, start+1, 0, end)
        parent.replaceSelectedText("CLICK")
    # Attach the hotspot click signal to a predefined function
    parent.SCN_HOTSPOTCLICK.connect(hotspot_click)


    def init_colors(self):
        # Font color
        self.setColor(PyQt5.QtGui.QColor(0x00, 0x00, 0x00),
self.styles["Default"])
        self.setColor(PyQt5.QtGui.QColor(0x00, 0x00, 0x7f),
self.styles["Keyword"])
        self.setColor(PyQt5.QtGui.QColor(0x7f, 0x00, 0x00),
self.styles["Unsafe"])
        self.setColor(PyQt5.QtGui.QColor(0x7f, 0x7f, 0x00),
self.styles["MultilineComment"])
        # Paper color
        for i in range(len(self.styles)):
            self.setPaper(PyQt5.QtGui.QColor(0xff, 0xff, 0xff), i)

    def language(self):
        return "Nim"
```

```python
    def description(self, style):
        if style < len(self.styles):
            description = "Custom lexer for the Nim programming
languages"
        else:
            description = ""
        return description


    def styleText(self, start, end):
        # Initialize the styling
        self.startStyling(start)
        # Tokenize the text that needs to be styled using regular
expressions.
        # To style a sequence of characters you need to know the
length of the sequence
        # and which style you wish to apply to the sequence. It is
up to the implementer
        # to figure out which style the sequence belongs to.
        # THE PROCEDURE SHOWN BELOW IS JUST ONE OF MANY!
        splitter = re.compile(r"(\
{\.|\.\}|\#|\'|\"\"\"\"|\n|\s+|\w+|\W)")
        # Scintilla works with bytes, so we have to adjust the
start and end boundaries.
        # Like all Qt objects the lexers parent is the QScintilla
editor.
        text = bytearray(self.parent().text(), "utf-8")
[start:end].decode("utf-8")
        tokens = [
            (token, len(bytearray(token, "utf-8")))
            for token in splitter.findall(text)
        ]
        # Multiline styles
        multiline_comment_flag = False
        # Check previous style for a multiline style
        if start != 0:
            previous_style =
editor.SendScintilla(editor.SCI_GETSTYLEAT, start - 1)
            if previous_style == self.styles["MultilineComment"]:
                multiline_comment_flag = True
        # Style the text in a loop
```

```python
    for i, token in enumerate(tokens):
      if multiline_comment_flag == False and token[0] == "#" and
tokens[i+1][0] == "[":
        # Start of a multiline comment
        self.setStyling(token[1],
self.styles["MultilineComment"])
        # Set the multiline comment flag
        multiline_comment_flag = True
      elif multiline_comment_flag == True:
        # Multiline comment flag is set
        self.setStyling(token[1],
self.styles["MultilineComment"])
        # Check if a multiline comment ends
        if token[0] == "#" and tokens[i-1][0] == "]":
          multiline_comment_flag = False
      elif token[0] in self.keyword_list:
        # Keyword
        self.setStyling(token[1], self.styles["Keyword"])
      elif token[0] in self.unsafe_keyword_list:
        # Keyword
        self.setStyling(token[1], self.styles["Unsafe"])
      else:
        # Style with the default style
        self.setStyling(token[1], self.styles["Default"])


# Create the main PyQt application object
application = PyQt5.QtWidgets.QApplication(sys.argv)

# Create a QScintila editor instance
editor = PyQt5.Qsci.QsciScintilla()
# Set the lexer to the custom Nim lexer
nim_lexer = LexerNim(editor)
editor.setLexer(nim_lexer)
# Set the initial text with some Nim code
editor.setText(
"""
proc python_style_text*(self, args: PyObjectPtr): PyObjectPtr
{.exportc, cdecl.} =
  var
    result_object: PyObjectPtr
```

```
      value_start, value_end: cint
      lexer, editor: PyObjectPtr
      parse_result: cint

  parse_result = argParseTuple(
    args,
    "iiOO",
    addr(value_start),
    addr(value_end),
    addr(lexer),
    addr(editor),
  )

  #[
      if parse_result == 0:
          echo "Napaka v pretvarjanju argumentov v funkcijo!"
          returnNone()
  ]#

  var
    text_method = objectGetAttr(editor, buildValue("s",
cstring("text")))
    text_object = objectCallObject(text_method, tupleNew(0))
    string_object = unicodeAsEncodedString(text_object, "utf-8",
nil)
    cstring_whole_text = bytesAsString(string_object)
    whole_text = $cstring_whole_text
    text = whole_text[int(value_start)..int(value_end-1)]
    text_length = len(text)
    current_token: string = ""
  # Prepare the objects that will be called as functions
  var
    start_styling_obj: PyObjectPtr
    start_args: PyObjectPtr
    set_styling_obj: PyObjectPtr
    set_args: PyObjectPtr
    send_scintilla_obj: PyObjectPtr
    send_args: PyObjectPtr
  start_styling_obj = objectGetAttr(lexer, buildValue("s",
cstring("startStyling")))
  start_args = tupleNew(1)
```

```nim
  set_styling_obj = objectGetAttr(lexer, buildValue("s",
cstring("setStyling")))
  set_args = tupleNew(2)
  send_scintilla_obj = objectGetAttr(editor, buildValue("s",
cstring("SendScintilla")))
  send_args = tupleNew(2)

  # Template for final cleanup
  template clean_up() =
    xDecref(text_method)
    xDecref(text_object)
    xDecref(string_object)
    xDecref(args)
    xDecref(result_object)

    xDecref(start_styling_obj)
    xDecref(start_args)
    xDecref(set_styling_obj)
    xDecref(set_args)
    xDecref(send_scintilla_obj)
    xDecref(send_args)
  # Template for the lexers setStyling function
  template set_styling(length: int, style: int) =
    discard tupleSetItem(set_args, 0, buildValue("i", length))
    discard tupleSetItem(set_args, 1, buildValue("i", style))
    discard objectCallObject(set_styling_obj, set_args)
  # Procedure for getting previous style
  proc get_previous_style(): int =
    discard tupleSetItem(send_args, 0, buildValue("i",
SCI_GETSTYLEAT))
    discard tupleSetItem(send_args, 1, buildValue("i",
value_start - 1))
    result = longAsLong(objectCallObject(send_scintilla_obj,
send_args))
    xDecref(send_args)
  # Template for starting styling
  template start_styling() =
    discard tupleSetItem(start_args, 0, buildValue("i",
value_start))
    discard objectCallObject(start_styling_obj, start_args)
  # Safety
```

```
  if set_styling_obj == nil:
    raise newException(FieldError, "Lexer doesn't contain the
'setStyling' method!")
  elif start_styling_obj == nil:
    raise newException(FieldError, "Lexer doesn't contain the
'startStyling' method!")
  elif send_scintilla_obj == nil:
    raise newException(FieldError, "Editor doesn't contain the
'SendScintilla' method!")
  # Styling initialization
  start_styling()
  #--------------------------------------------------------
  var
    actseq = SeqActive(active: false)
    token_name: string = ""
    previous_token: string = ""
    token_start: int = 0
    token_length: int = 0
  # Check previous style
  if value_start != 0:
    var previous_style = get_previous_style()
    for i in multiline_sequence_list:
      if previous_style == i.style:
        actseq.sequence = i
        actseq.active = true
        break
  # Style the tokens accordingly
  proc check_start_sequence(pos: int, sequence: var SeqActive):
bool =
    for s in sequence_lists:
      var found = true
      for i, ch in s.start.pairs:
        if text[pos+i] != ch:
          found = false
          break
      if found == false:
        continue
      sequence.sequence = s
      return true
    return false
```

```
proc check_stop_sequence(pos: int, actseq: SeqActive): bool =
  if text[pos] in actseq.sequence.stop_extra:
    return true
  if pos > 0 and (text[pos-1] in
actseq.sequence.negative_lookbehind):
    return false
  for i, s in actseq.sequence.stop.pairs:
    if text[pos+i] != s:
      return false
  return true


template style_token(token_name: string, token_length: int) =
  if token_length > 0:
    if token_name in keywords:
      set_styling(token_length, styles["Keyword"])
      previous_token = token_name
    elif token_name in custom_keywords:
      set_styling(token_length, styles["CustomKeyword"])
    elif token_name[0].isdigit() or (token_name[0] == '.' and
token_name[1].isdigit()):
      set_styling(token_length, styles["Number"])
    elif previous_token == "class":
      set_styling(token_length, styles["ClassName"])
      previous_token = ""
    elif previous_token == "def":
      set_styling(token_length, styles["FunctionMethodName"])
      previous_token = ""
    else:
      set_styling(token_length, styles["Default"])

var i = 0
token_start = i
while i < text_length:
  if actseq.active == true or check_start_sequence(i, actseq) ==
true:
    #[
      Multiline sequence already started in the previous line or
      a start sequence was found
    ]#
    if actseq.active == false:
      # Style the currently accumulated token
```

```
        token_name = text[token_start..i]
        token_length = i - token_start
        style_token(token_name, token_length)
      # Set the states and reset the flags
      token_start = i
      token_length = 0
      if actseq.active == false:
        i += len(actseq.sequence.start)
      while i < text_length:
        # Check for end of comment
        if check_stop_sequence(i, actseq) == true:
          i += len(actseq.sequence.stop)
          break
        i += 1
      # Style text
      token_length = i - token_start
      set_styling(token_length, actseq.sequence.style)
      # Style the separator tokens after the sequence, if any
      token_start = i
      while text[i] in extended_separators and i < text_length:
        i += 1
      token_length = i - token_start
      if token_length > 0:
        set_styling(token_length, styles["Default"])
      # Set the states and reset the flags
      token_start = i
      token_length = 0
      # Skip to the next iteration, because the index is
already
      # at the position at the end of the string
      actseq.active = false
      continue
    elif text[i] in extended_separators:
      #[
        Separator found
      ]#
      token_name = text[token_start..i-1]
      token_length = len(token_name)
      if token_length > 0:
        style_token(token_name, token_length)
```

```
            token_start = i
            while text[i] in extended_separators and i < text_length:
              i += 1
            token_length = i - token_start
            if token_length > 0:
              set_styling(token_length, styles["Default"])
            # State reset
            token_start = i
            token_length = 0
            continue
          else:
            while text[i] in extended_separators and i < text_length:
              i += 1
            token_length = i - token_start
            if token_length > 0:
              set_styling(token_length, styles["Default"])
            # State reset
            token_start = i
            token_length = 0
            continue
      # Update loop variables
      inc(i)
      # Check for end of text
      if i == text_length:
        token_name = text[token_start..i-1]
        token_length = len(token_name)
        style_token(token_name, token_length)
      elif i > text_length:
        raise newException(IndexError, "Styling went over the text
length limit!")
    #----------------------------------------------------
    clean_up()
    returnNone()
"""
)


# For the QScintilla editor to properly process events we need
to add it to
# a QMainWindow object.
main_window = PyQt5.QtWidgets.QMainWindow()
```

```python
# Set the central widget of the main window to be the editor
main_window.setCentralWidget(editor)
# Resize the main window and show it on the screen
main_window.resize(800, 600)
main_window.show()

# Execute the application
application.exec_()
```