Software Maintenance and Evolution

CSE 423

Assignment 1

Submitted by: Nada Tarek Ahmed

ID :16P6053

Submitted to: Dr. Ayman Bahaa

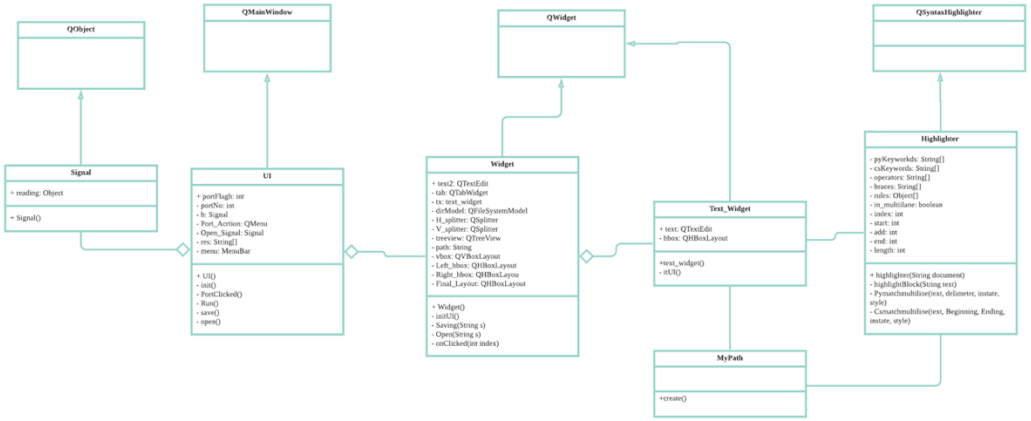
Table of Contents

Introduction

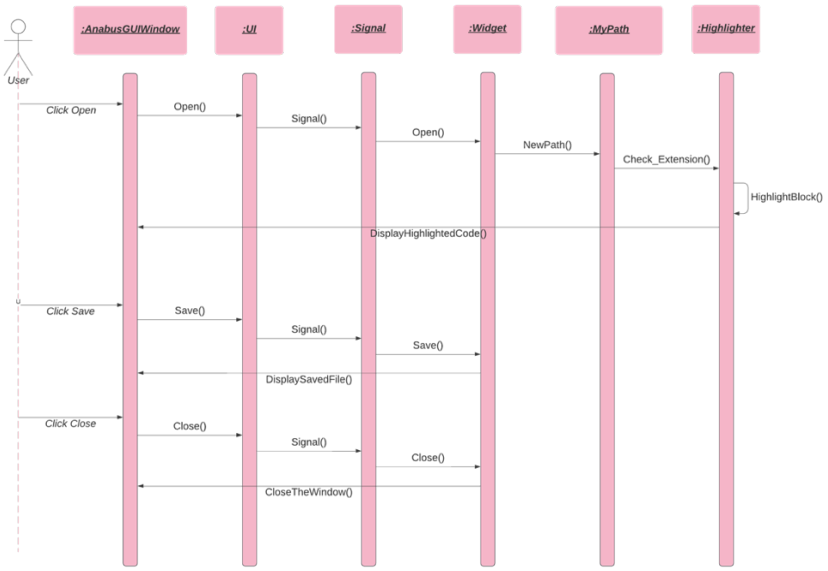
For assignment 2, I chose adding support for C# language. This feature is automated based on the file format, and not explicitly specified by the user, so the user can just choose a C# file (.cs extension), and the editor will support it as expected.

Please refer to this repository to find where the modified source code is:

New Class Diagram



New Sequence Diagram



Modifications

PythonColoring.py

- Refactoring:
  Changed the name of the file to the coloring, as the context of the application now is not strictly tied to python language, and the compiler supports coloring for different languages
- Highlighter class:
  The variable keywords that was used to define MicroPython is changed because we now support 2 languages: C#, which has its keywords in the vartiable “csKeywords[]”, and MicroPython, which has its keywords in variable “pyKeywords[]”.
- Init function:
  Changed the variable rules[] and split it into csRules[] and pyRules[] to support the different languages we now have
- HighlightBlock function:
  The current path is now obtained from the file “MyPath.py” and the file extension is checked to know whether this is a python file (.py) or C# file (.cs) to do the highlighting.
- matchMultiline function:
  Again, since we now support multiple languages, 2 versions of this function replaced it. Pymatchmultiling for python, and Csmatchmultiline for C#. Csmatchmultiline has additional parameters beginning and ending, and added

MyPath.py

Created this file which is referenced for the path file to use for the editor. It contains a global variable that represents the path of the file to work on and used by the coloring and highlighting classes. They use it do get data and decide which mode of operations to use (Python or C#)

Anubis.py

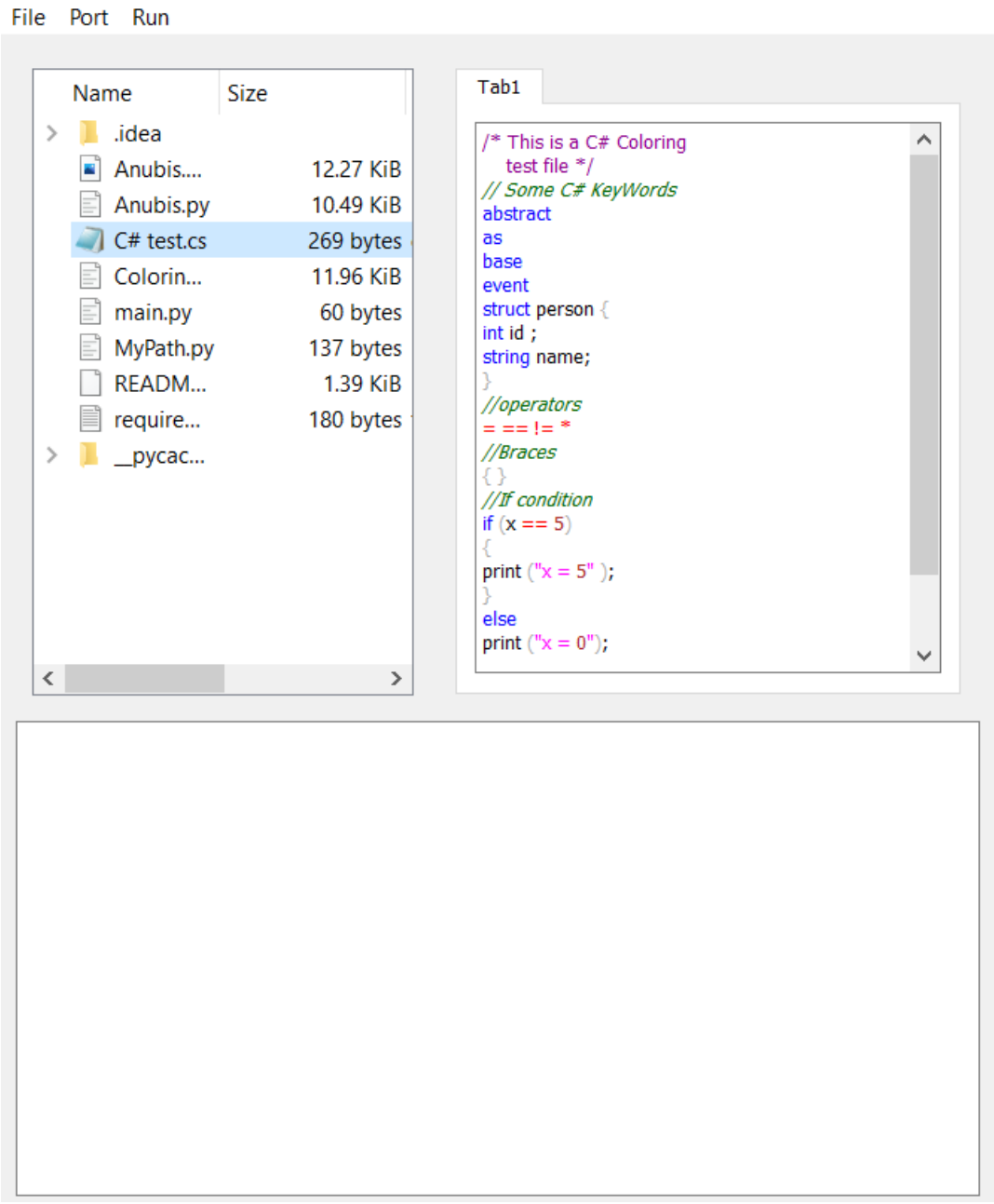
on_clicked function: As explained before in MyPath, this is the module that is referenced for the pathfile. The function is changed to use MyPath global variable to get the path. This module (MyPath) is also imported here.

Initialize MyPath by calling the create() function inside the module to initialize the global variable at the start of running.

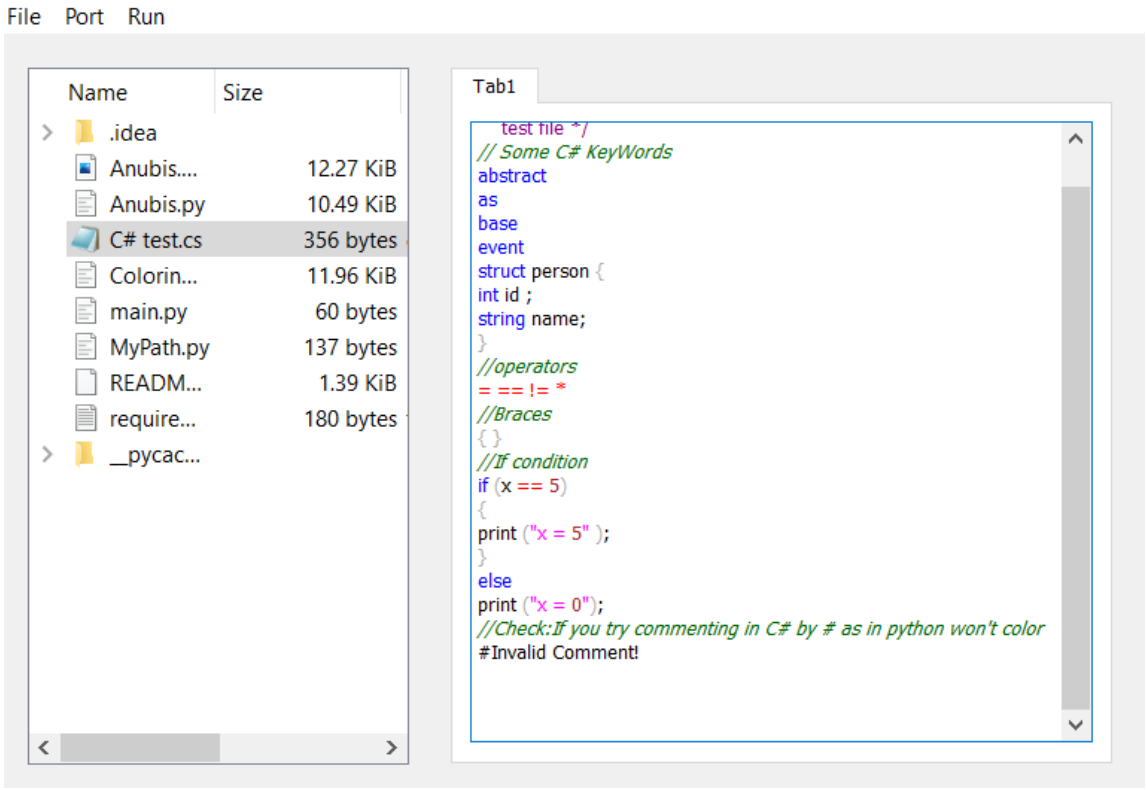
Screenshots

These screenshots are after modification.

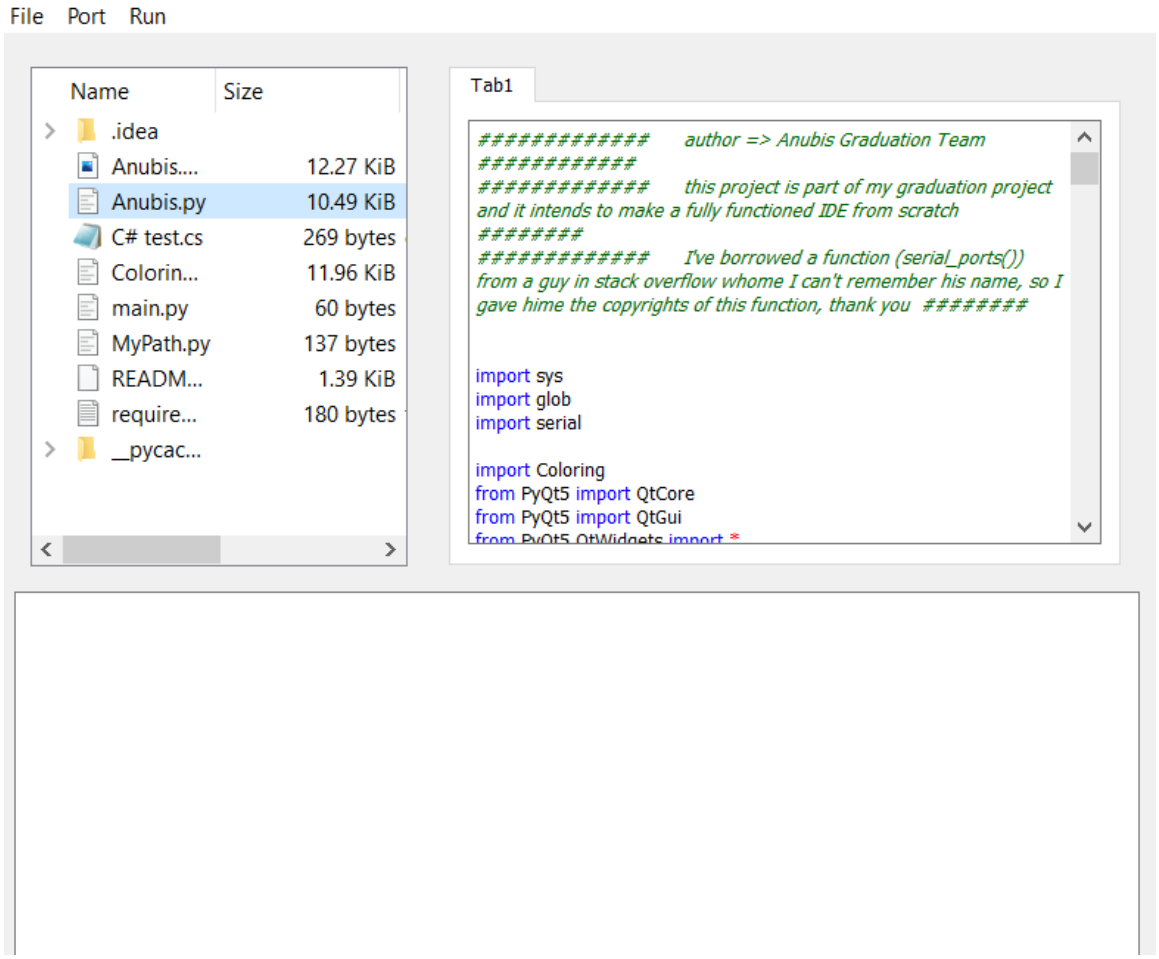
The following screenshot shows and example of supporting C# file, with text highlighting and coloring.



Below I check whether or not there is an intersection between the C# and the python keywords, and whether the editor can detect a python comment in a C# file (The answer is no, which is expected).



And below is the default example of the python code correctly highlighted and colored, which should work from before.



Coloring.py

```
import sys
from PyQt5.QtCore import QRegExp
from PyQt5.QtGui import QColor, QTextCharFormat, QFont, QSyntaxHighlighter
import MyPath
# A Module created for sharing a Global Variable for Path to be used based
# on extension in Coloring


def format(color, style=''):
    """
    Return a QTextCharFormat with the given attributes.
    """
    _color = QColor()
    if type(color) is not str:
        _color.setRgb(color[0], color[1], color[2])
    else:
        _color.setNamedColor(color)

    _format = QTextCharFormat()
    _format.setForeground(_color)
    if 'bold' in style:
        _format.setFontWeight(QFont.Bold)
    if 'italic' in style:
        _format.setFontItalic(True)

    return _format


# Syntax styles that can be shared by all languages

STYLES2 = {
    'keyword': format([200, 120, 50], 'bold'),
    'operator': format([150, 150, 150]),
    'brace': format('darkGray'),
    'defclass': format([220, 220, 255], 'bold'),
    'string': format([20, 110, 100]),
    'string2': format([30, 120, 110]),
    'comment': format([128, 128, 128]),
    'self': format([150, 85, 140], 'italic'),
    'numbers': format([100, 150, 190]),
}
STYLES = {
    'keyword': format('blue'),
    'operator': format('red'),
    'brace': format('darkGray'),
    'defclass': format('black', 'bold'),
    'string': format('magenta'),
    'string2': format('darkMagenta'),
    'comment': format('darkGreen', 'italic'),
    'self': format('black', 'italic'),
    'numbers': format('brown'),
}
```

```
class Highlighter(QSyntaxHighlighter):
    """Syntax highlighter for the Python and C# languages.
    """
    # Python keywords
    pyKeywords = [
        'and', 'assert', 'break', 'class', 'continue', 'def',
        'del', 'elif', 'else', 'except', 'exec', 'finally',
        'for', 'from', 'global', 'if', 'import', 'in',
        'is', 'lambda', 'not', 'or', 'pass', 'print',
        'raise', 'return', 'try', 'while', 'yield',
        'None', 'True', 'False',
    ]

    # C# keywords

    csKeywords = ['abstract', 'as', 'base', 'bool'
        , 'break', 'byte', 'case', 'catch'
        , 'char', 'checked', 'class', 'const'
        , 'continue', 'decimal', 'default','delegate'
        , 'do', 'double', 'else', 'enum'
        , 'event', 'explicit', 'extern', 'false'
        , 'finally', 'fixed', 'float', 'for'
        , 'foreach', 'goto', 'if', 'implicit', 'in', 'int', 'interface',
'internal'
        , 'is', 'lock', 'long', 'namespace', 'new', 'null', 'object', 'operator'
        , 'out', 'override', 'params', 'private', 'protected', 'public',
'readonly', 'ref'',return', 'sbyte', 'sealed','short', 'sizeof', 'stackalloc',
'static', 'string'
        , 'struct', 'switch', 'this', 'throw'
        , 'true', 'try', 'typeof', 'uint'
        , 'ulong', 'unchecked', 'unsafe', 'ushort'
        , 'using', 'virtual', 'void', 'volatile','while', 'var']

    #operators
    operators = [
        '=',
        # Comparison
        '==', '!=', '<', '<=', '>', '>=',
        # Arithmetic
        '\+', '-', '\*', '/', '//', '\%', '\*\*',
        # In-place
        '\+=', '-=', '\*=', '/=', '\%=',
        # Bitwise
        '\^', '\|', '\&', '\~', '>>', '<<',
    ]

    #  braces
    braces = [
        '\{', '\}', '\(', '\)', '\[', '\]',
    ]
```

```
def __init__(self, document):


    QSyntaxHighlighter.__init__(self, document)

    # Multi-line strings (expression, flag, style)
    # FIXME: The triple-quotes in these two lines will mess up the
    # syntax highlighting from this point onward

    # For Python Commmenting
    self.tri_single = (QRegExp("'''"), 1, STYLES['string2'])
    self.tri_double = (QRegExp('"""'), 2, STYLES['string2'])

    # For CS Commmenting
    self.CS_Comment = (QRegExp('/\*'),QRegExp('\*/'), 3, STYLES['string2'])


    # Python regular Expression Rules
    pyRules = []

    # Keyword, operator, and brace pyRules
    pyRules += [(r'\b%s\b' % w, 0, STYLES['keyword'])
              for w in Highlighter.pyKeywords]
    pyRules += [(r'%s' % o, 0, STYLES['operator'])
              for o in Highlighter.operators]
    pyRules += [(r'%s' % b, 0, STYLES['brace'])
              for b in Highlighter.braces]

    # All other pyRules
    pyRules += [
        # 'self'
        (r'\bself\b', 0, STYLES['self']),

        # Double-quoted string, possibly containing escape sequences
        (r'"[^"\\]*(\\.[^"\\]*)*"', 0, STYLES['string']),
        # Single-quoted string, possibly containing escape sequences
        (r"'[^'\\]*(\\.[^'\\]*)*'", 0, STYLES['string']),

        # 'def' followed by an identifier
        (r'\bdef\b\s*(\w+)', 1, STYLES['defclass']),
        # 'class' followed by an identifier
        (r'\bclass\b\s*(\w+)', 1, STYLES['defclass']),

        # From '#' until a newline
        (r'#[^\n]*', 0, STYLES['comment']),

        # Numeric literals
        (r'\b[+-]?[0-9]+[lL]?\b', 0, STYLES['numbers']),
        (r'\b[+-]?0[xX][0-9A-Fa-f]+[lL]?\b', 0, STYLES['numbers']),
        (r'\b[+-]?[0-9]+(?:\.[0-9]+)?(?:[eE][+-]?[0-9]+)?\b', 0, STYLES[
'numbers']),
    ]

    # Build a QRegExp for each pattern
    self.pyRules = [(QRegExp(pat), index, fmt)
                    for (pat, index, fmt) in pyRules]
```

```
# C# regular Expression Rules
        csRules = []

        # Keyword, operator, and brace C# Rules
        csRules += [(r'\b%s\b' % w, 0, STYLES['keyword'])
                    for w in Highlighter.csKeywords]
        csRules += [(r'%s' % o, 0, STYLES['operator'])
                    for o in Highlighter.operators]
        csRules += [(r'%s' % b, 0, STYLES['brace'])
                    for b in Highlighter.braces]

        # All other C# Rules
        csRules += [
            # 'self'
            (r'\bself\b', 0, STYLES['self']),

            # Double-quoted string, possibly containing escape sequences
            (r'"[^"\\]*(\\.[^"\\]*)*"', 0, STYLES['string']),
            # Single-quoted string, possibly containing escape sequences
            (r"'[^'\\]*(\\.[^'\\]*)*'", 0, STYLES['string']),

            # 'def' followed by an identifier
            (r'\bdef\b\s*(\w+)', 1, STYLES['defclass']),
            # 'class' followed by an identifier
            (r'\bclass\b\s*(\w+)', 1, STYLES['defclass']),

            # From '//' until a newline
            (r'//[^\n]*', 0, STYLES['comment']),

            # Numeric literals
            (r'\b[+-]?[0-9]+[lL]?\b', 0, STYLES['numbers']),
            (r'\b[+-]?0[xX][0-9A-Fa-f]+[lL]?\b', 0, STYLES['numbers']),
            (r'\b[+-]?[0-9]+(?:\.[0-9]+)?(?:[eE][+-]?[0-9]+)?\b', 0, STYLES[
'numbers']),
        ]

        # Build a QRegExp for each pattern
        self.csRules = [(QRegExp(pat), index, fmt)
                        for (pat, index, fmt) in csRules]
```

```
def highlightBlock(self, text):
        """Apply syntax highlighting to the given block of text.
        """
        # Python Highlighting

        if MyPath.nn[0][-3:] == '.py':


         for expression, nth, format in self.pyRules:
            index = expression.indexIn(text, 0)

            while index >= 0:
                # We actually want the index of the nth match
                index = expression.pos(nth)
                length = len(expression.cap(nth))
                self.setFormat(index, length, format)
                index = expression.indexIn(text, index + length)

         self.setCurrentBlockState(0)

         # Do Python multi-line strings
         in_multiline = self.Pymatch_multiline(text, *self.tri_single)
         if not in_multiline:
             in_multiline = self.Pymatch_multiline(text, *self.tri_double)


     # C# Highlighting

        if MyPath.nn[0][-3:] == '.cs':

          for expression, nth, format in self.csRules:
            index = expression.indexIn(text, 0)

            while index >= 0:
                # We actually want the index of the nth match
                index = expression.pos(nth)
                length = len(expression.cap(nth))
                self.setFormat(index, length, format)
                index = expression.indexIn(text, index + length)

          self.setCurrentBlockState(0)

          # Do C# Multi-Line Strings
          self.CSmatch_multiline(text, *self.CS_Comment)
```

```
  #Python Multi Commenting Function

def Pymatch_multiline(self, text, delimiter, in_state, style):
    """Do highlighting of multi-line strings. ``delimiter`` should be a
    ``QRegExp`` for triple-single-quotes or triple-double-quotes, and
    ``in_state`` should be a unique integer to represent the corresponding
    state changes when inside those strings. Returns True if we're still
    inside a multi-line string when this function is finished.
    """
    # If inside triple-single quotes, start at 0
    if self.previousBlockState() == in_state:
        start = 0
        add = 0
    # Otherwise, look for the delimiter on this line
    else:
        start = delimiter.indexIn(text)
        # Move past this match
        add = delimiter.matchedLength()

    # As long as there's a delimiter match on this line...
    while start >= 0:
        # Look for the ending delimiter
        end = delimiter.indexIn(text, start + add)
        # Ending delimiter on this line?
        if end >= add:
            length = end - start + add + delimiter.matchedLength()
            self.setCurrentBlockState(0)
        # No; multi-line string
        else:
            self.setCurrentBlockState(in_state)
            length = len(text) - start + add
        # Apply formatting
        self.setFormat(start, length, style)
        # Look for the next match
        start = delimiter.indexIn(text, start + length)

    # Return True if still inside a multi-line string, False otherwise
    if self.currentBlockState() == in_state:
        return True
    else:
        return False
```

```
# C# Multi Commenting

    def CSmatch_multiline(self, text, Beginning, Ending, in_state, style):
        """Do highlighting of multi-line strings. There should be a
          ``QRegExp`` for /* as  Beginning and */ as Ending(Delimiter) , and
          ``in_state`` should be a unique integer to represent the corresponding
          state changes when inside those strings. Returns True if we're still
          inside a multi-line string when this function is finished.
          """
        # If inside /* , start at 0
        if self.previousBlockState() == in_state:
            start = 0
            add = 0
        # Otherwise, look for the Ending on this line
        else:
            start = Beginning.indexIn(text)
            # Move past this match
            add = Beginning.matchedLength()

        # As long as there's a Ending match on this line...
        while start >= 0:
            # Look for the ending delimiter
            end = Ending.indexIn(text, start + add)
            # Ending delimiter on this line?
            if end >= add:
                length = end - start + add + Ending.matchedLength()
                self.setCurrentBlockState(0)
            # No; multi-line string
            else:
                self.setCurrentBlockState(in_state)
                length = len(text) - start + add
            # Apply formatting
            self.setFormat(start, length, style)
            # Look for the next match
            start = Ending.indexIn(text, start + length)

        # Return True if still inside a multi-line string, False otherwise
        if self.currentBlockState() == in_state:
            return True
        else:
            return False
```

Anubis.py

```
import sys
import glob
import serial

import Coloring
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from pathlib import Path
import MyPath

def serial_ports():
    """ Lists serial port names
        :raises EnvironmentError:
            On unsupported or unknown platforms
        :returns:
            A list of the serial ports available on the system
    """
    if sys.platform.startswith('win'):
        ports = ['COM%s' % (i + 1) for i in range(256)]
    elif sys.platform.startswith('linux') or sys.platform.startswith('cygwin'):
        # this excludes your current terminal "/dev/tty"
        ports = glob.glob('/dev/tty[A-Za-z]*')
    elif sys.platform.startswith('darwin'):
        ports = glob.glob('/dev/tty.*')
    else:
        raise EnvironmentError('Unsupported platform')

    result = []
    for port in ports:
        try:
            s = serial.Serial(port)
            s.close()
            result.append(port)
        except (OSError, serial.SerialException):
            pass
    return result
```

```
#
#
#
#
############ Signal Class ############
#
#
#
#
class Signal(QObject):

    # initializing a Signal which will take (string) as an input
    reading = pyqtSignal(str)

    # init Function for the Signal class
    def __init__(self):
        QObject.__init__(self)

#
#
############ end of Class ############
#
#

# Making text editor as A global variable
# (to solve the issue of being local to (self) in widget class)
text = QTextEdit
text2 = QTextEdit

#
#
#
#
############ Text Widget Class ############
#
#
#
#

# this class is made to connect the QTab with the necessary layouts
class text_widget(QWidget):
    def __init__(self):
        super().__init__()
        self.itUI()
    def itUI(self):
        global text
        text = QTextEdit()
        Coloring.Highlighter(text)
        hbox = QHBoxLayout()
        hbox.addWidget(text)
        self.setLayout(hbox)




#
#
############ end of Class ############
#
#
```

```
#
#
#
#
############ Widget Class ############
#
#
#
#
class Widget(QWidget):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):

        # This widget is responsible of making Tab in IDE which makes the
        # Text editor looks nice
        tab = QTabWidget()
        tx = text_widget()
        tab.addTab(tx, "Tab"+"1")


# second editor in which the error messeges and succeeded connections will be shown
        global text2
        text2 = QTextEdit()
        text2.setReadOnly(True)

# defining a Treeview variable to use it in showing the directory included files
        self.treeview = QTreeView()

        # making a variable (path) and setting it to the root path
        # (surely I can set it to whatever the root I want, not the default)
        #path = QDir.rootPath()

        path = QDir.currentPath()


# making a Filesystem variable, setting its root path and applying somefilters
        # (which I need) on it
        self.dirModel = QFileSystemModel()
        self.dirModel.setRootPath(QDir.rootPath())

        # NoDotAndDotDot => Do not list the special entries "." and "..".

# AllDirs =>List all directories; i.e. don't apply the filters to directory names.
        # Files => List files.
        self.dirModel.setFilter(QDir.NoDotAndDotDot | QDir.AllDirs | QDir.Files)
        self.treeview.setModel(self.dirModel)
        self.treeview.setRootIndex(self.dirModel.index(path))
        self.treeview.clicked.connect(self.on_clicked)

        vbox = QVBoxLayout()
        Left_hbox = QHBoxLayout()
        Right_hbox = QHBoxLayout()

        # after defining variables of type QVBox and QHBox

# I will Assign treevies variable to the left one and the first text editor
        # in which the code will be written to the right one
        Left_hbox.addWidget(self.treeview)
        Right_hbox.addWidget(tab)


# defining another variable of type Qwidget to set its layout as an QHBoxLayout
        # I will do the same with the right one
        Left_hbox_Layout = QWidget()
        Left_hbox_Layout.setLayout(Left_hbox)

        Right_hbox_Layout = QWidget()
        Right_hbox_Layout.setLayout(Right_hbox)
```

```
 # I defined a splitter to seperate the two variables (left, right) and
        # make it more easily to change the space between them
        H_splitter = QSplitter(Qt.Horizontal)
        H_splitter.addWidget(Left_hbox_Layout)
        H_splitter.addWidget(Right_hbox_Layout)
        H_splitter.setStretchFactor(1, 1)

        # I defined a new splitter to seperate between the upper
        # and lower sides of the window
        V_splitter = QSplitter(Qt.Vertical)
        V_splitter.addWidget(H_splitter)
        V_splitter.addWidget(text2)

        Final_Layout = QHBoxLayout(self)
        Final_Layout.addWidget(V_splitter)

        self.setLayout(Final_Layout)

    # defining a new Slot (takes string) to save the text inside
    # the first text editor
    @pyqtSlot(str)
    def Saving(s):
        with open('main.py', 'w') as f:
            TEXT = text.toPlainText()
            f.write(TEXT)

    # defining a new Slot (takes string) to set the string to the text editor
    @pyqtSlot(str)
    def Open(s):
        global text
        text.setText(s)

    def on_clicked(self, index):

        #Getting Path in a shared module for Extension deffering in Coloring
        MyPath.nn = self.sender().model().filePath(index)
        MyPath.nn = tuple([MyPath.nn])

        if MyPath.nn[0]:
            f = open(MyPath.nn[0],'r')
            with f:
                data = f.read()
                text.setText(data)

#
#
############ end of Class ############
#
#
```

```python
# defining a new Slot (takes string)
# Actually I could connect the (mainwindow) class directly
# to the (widget class) but I've made this function in between for futuer use
# All what it do is to take the (input string) and establish
# a connection with the widget class, send the string to it
@pyqtSlot(str)
def reading(s):
    b = Signal()
    b.reading.connect(Widget.Saving)
    b.reading.emit(s)

# same as reading Function
@pyqtSlot(str)
def Openning(s):
    b = Signal()
    b.reading.connect(Widget.Open)
    b.reading.emit(s)
#
#
#
#
############ MainWindow Class ############
#
#
#
#
class UI(QMainWindow):
    def __init__(self):
        super().__init__()
        self.intUI()

    def intUI(self):
        self.port_flag = 1
        self.b = Signal()

        self.Open_Signal = Signal()

        # connecting (self.Open_Signal) with Openning function
        self.Open_Signal.reading.connect(Openning)

        # connecting (self.b) with reading function
        self.b.reading.connect(reading)

        # creating menu items
        menu = self.menuBar()

        # I have three menu items
        filemenu = menu.addMenu('File')
        Port = menu.addMenu('Port')
        Run = menu.addMenu('Run')


# As any PC or laptop have many ports, so I need to list them to the User

# so I made (Port_Action) to add the Ports got from (serial_ports()) function

# copyrights of serial_ports() function goes back to a guy from stackoverflow
        # (whome I can't remember his name), so thank you (unknown)
        Port_Action = QMenu('port', self)

        res = serial_ports()

        for i in range(len(res)):
            s = res[i]
            Port_Action.addAction(s, self.PortClicked)

        # adding the menu which I made to the original (Port menu)
        Port.addMenu(Port_Action)
```

```
#        Port_Action.triggered.connect(self.Port)
#        Port.addAction(Port_Action)

        # Making and adding Run Actions
        RunAction = QAction("Run", self)
        RunAction.triggered.connect(self.Run)
        Run.addAction(RunAction)

        # Making and adding File Features
        Save_Action = QAction("Save", self)
        Save_Action.triggered.connect(self.save)
        Save_Action.setShortcut("Ctrl+S")
        Close_Action = QAction("Close", self)
        Close_Action.setShortcut("Alt+c")
        Close_Action.triggered.connect(self.close)
        Open_Action = QAction("Open", self)
        Open_Action.setShortcut("Ctrl+O")
        Open_Action.triggered.connect(self.open)

        filemenu.addAction(Save_Action)
        filemenu.addAction(Close_Action)
        filemenu.addAction(Open_Action)

        # Seting the window Geometry
        self.setGeometry(200, 150, 600, 500)
        self.setWindowTitle('Anubis IDE')
        self.setWindowIcon(QtGui.QIcon('Anubis.png'))

        widget = Widget()

        self.setCentralWidget(widget)
        self.show()

    ##########################Start OF the Functions#################
    def Run(self):
        if self.port_flag == 0:
            mytext = text.toPlainText()
        #
        ##### Compiler Part
        #
#            ide.create_file(mytext)
#            ide.upload_file(self.portNo)
            text2.append("Sorry, there is no attached compiler.")

        else:
            text2.append("Please Select Your Port Number First")

    # this function is made to get which port was selected by the user
    @QtCore.pyqtSlot()
    def PortClicked(self):
        action = self.sender()
        self.portNo = action.text()
        self.port_flag = 0

    # I made this function to save the code into a file
    def save(self):
        self.b.reading.emit("name")


# I made this function to open a file and exhibits it to the user in a text editor
    def open(self):
        file_name = QFileDialog.getOpenFileName(self,'Open File','/home')

        if file_name[0]:
            f = open(file_name[0],'r')
            with f:
                data = f.read()
            self.Open_Signal.reading.emit(data)

#
#
############ end of Class ############
#
#

if __name__ == '__main__':
    MyPath.create()
#To create the global shared variable in the beginning of the program
    app = QApplication(sys.argv)
    ex = UI()
    # ex = Widget()
    sys.exit(app.exec_())
```

MyPath.py

```
# global variable for file path to know the file extension for highlighting
# in Coloring
def create() :
    global nn
    nn = ""
```