Faculty of Engineering

Ain Shams University

Credit hours program

Spring 2021

Software Maintenance and Evolution

CSE 426

Evolving the Editor

Name: Mahmoud Mohammed Mustafa

ID: 17p8110

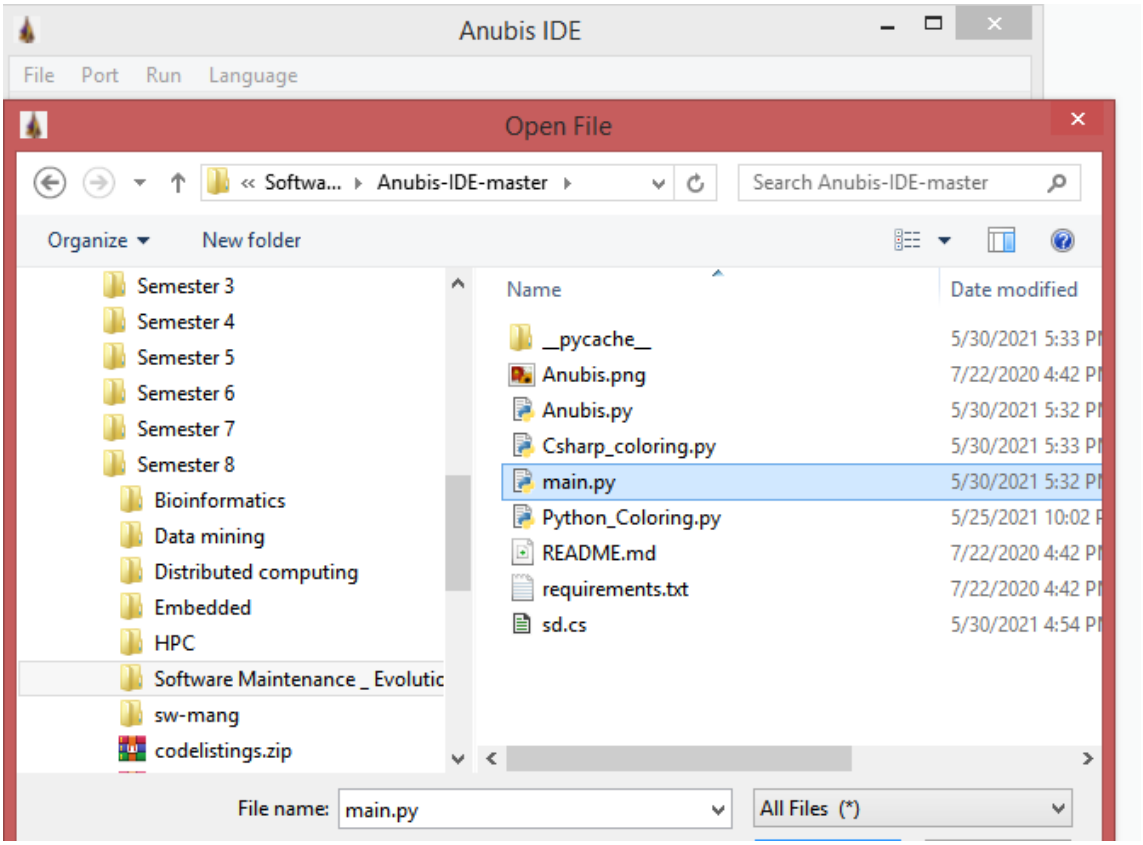
support for the c# format

the Implemented functionality is to add support for the c# format. The editor must automatically recognize which format to use based on the file extension. This done through three options

1. Open file from the top bar

   Code of this functionality:

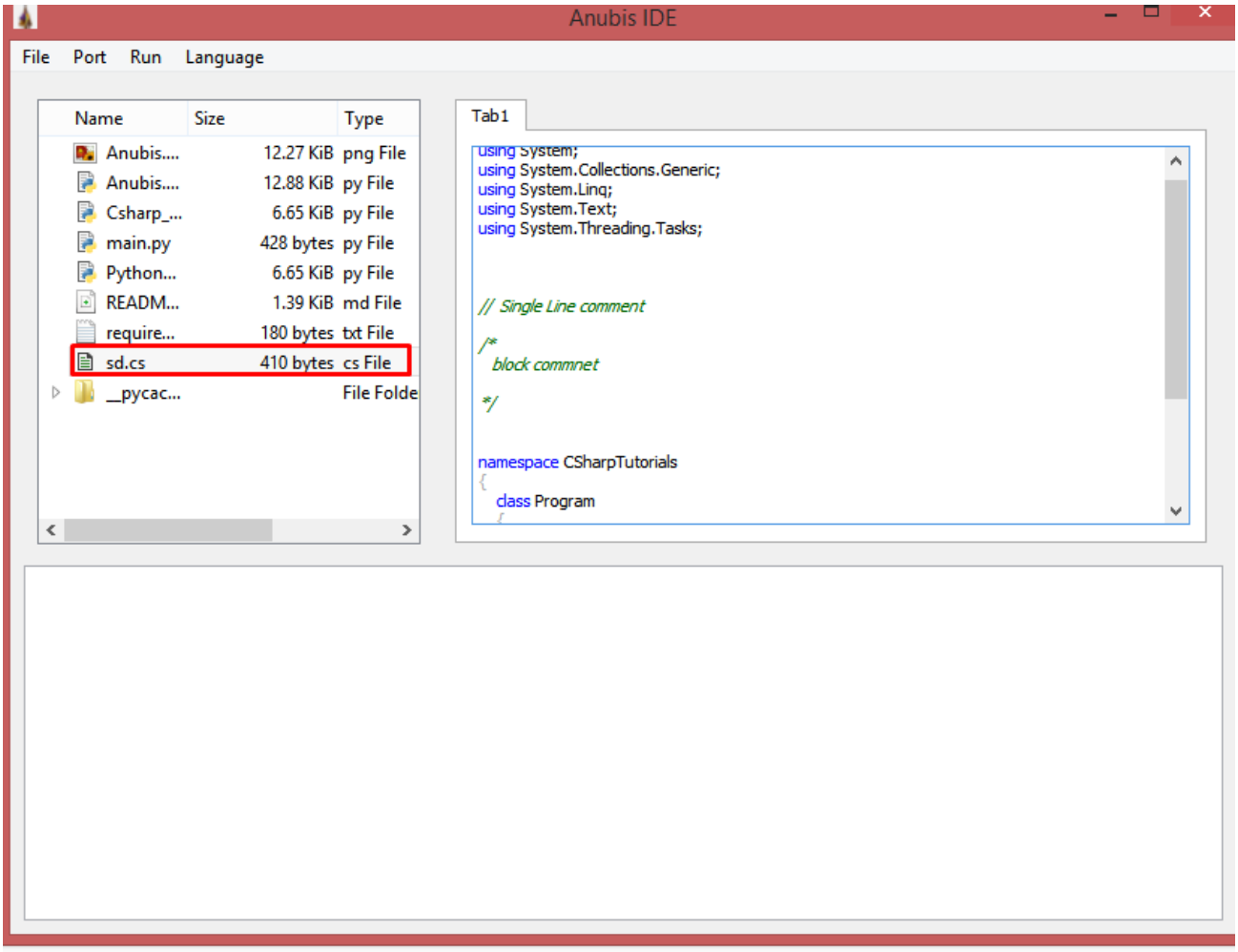
```
@pyqtSlot(str)
def Open(s):
    global text
    text.setText(s)
    # ******************************** 17p8110****************************************
    if '.cs' == file_name[0][-3:]:
        Csharp_coloring.CsharpHighlighter(text)
    else:
        print("python enabled")
        Python_Coloring.PythonHighlighter(text)
```



2. Click on file from the directory view

Code of this functionality:

```
def on_clicked(self, index):
    nn = self.sender().model().filePath(index)
    nn = tuple([nn])

    # ******************************** 17p8110*****************************************
    if '.cs' == nn[0][-3:]:
        Csharp_coloring.CsharpHighlighter(text)
    else:
        print("python enabled")
        Python_Coloring.PythonHighlighter(text)
    # ********************************** end*************************************
    if nn[0]:
        f = open(nn[0], 'r')
        with f:
            data = f.read()
            text.setText(data)
```



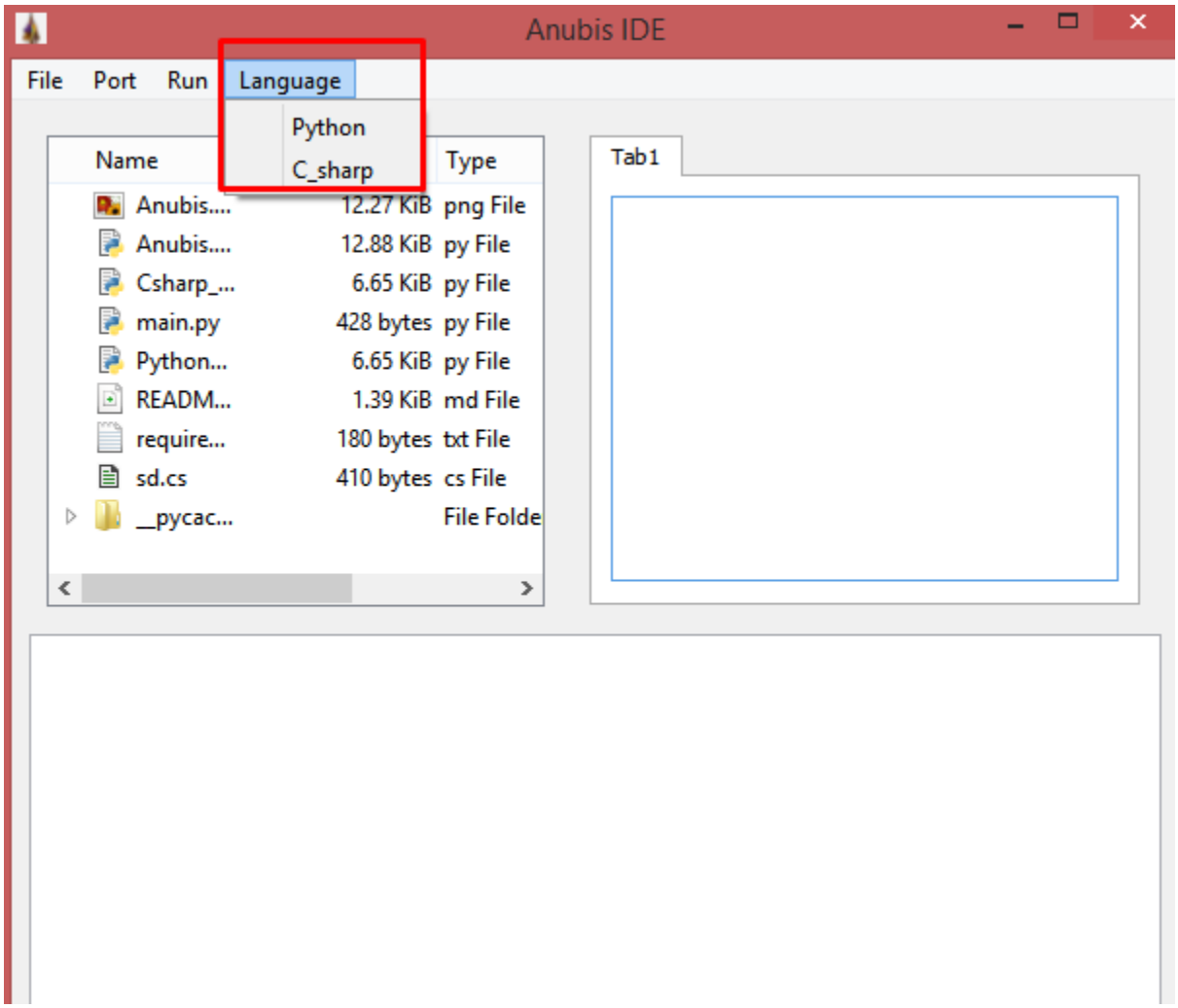
3. Choose a language from Language button in the top bar

Code of this functionality:

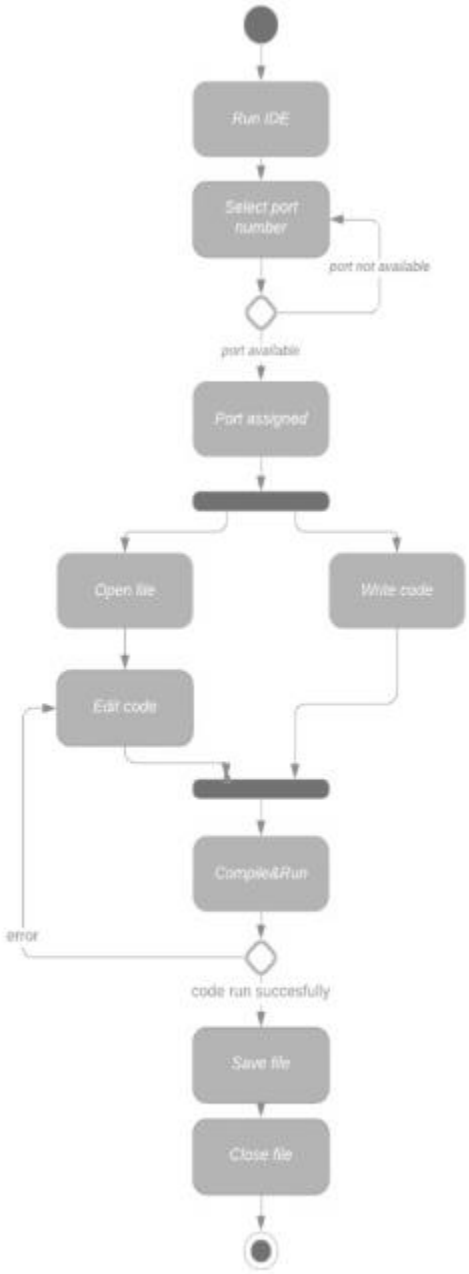
```
Language = menu.addMenu('Language')
 python_Action=QAction("Python",self)
 python_Action.triggered.connect(self.python)
 Csharp_Action=(QAction("C_sharp",self))
 Csharp_Action.triggered.connect(self.Csharp)

 Language.addAction(python_Action)
 Language.addAction(Csharp_Action)
```

```
def python(self):
  Python_Coloring.PythonHighlighter(text)
```

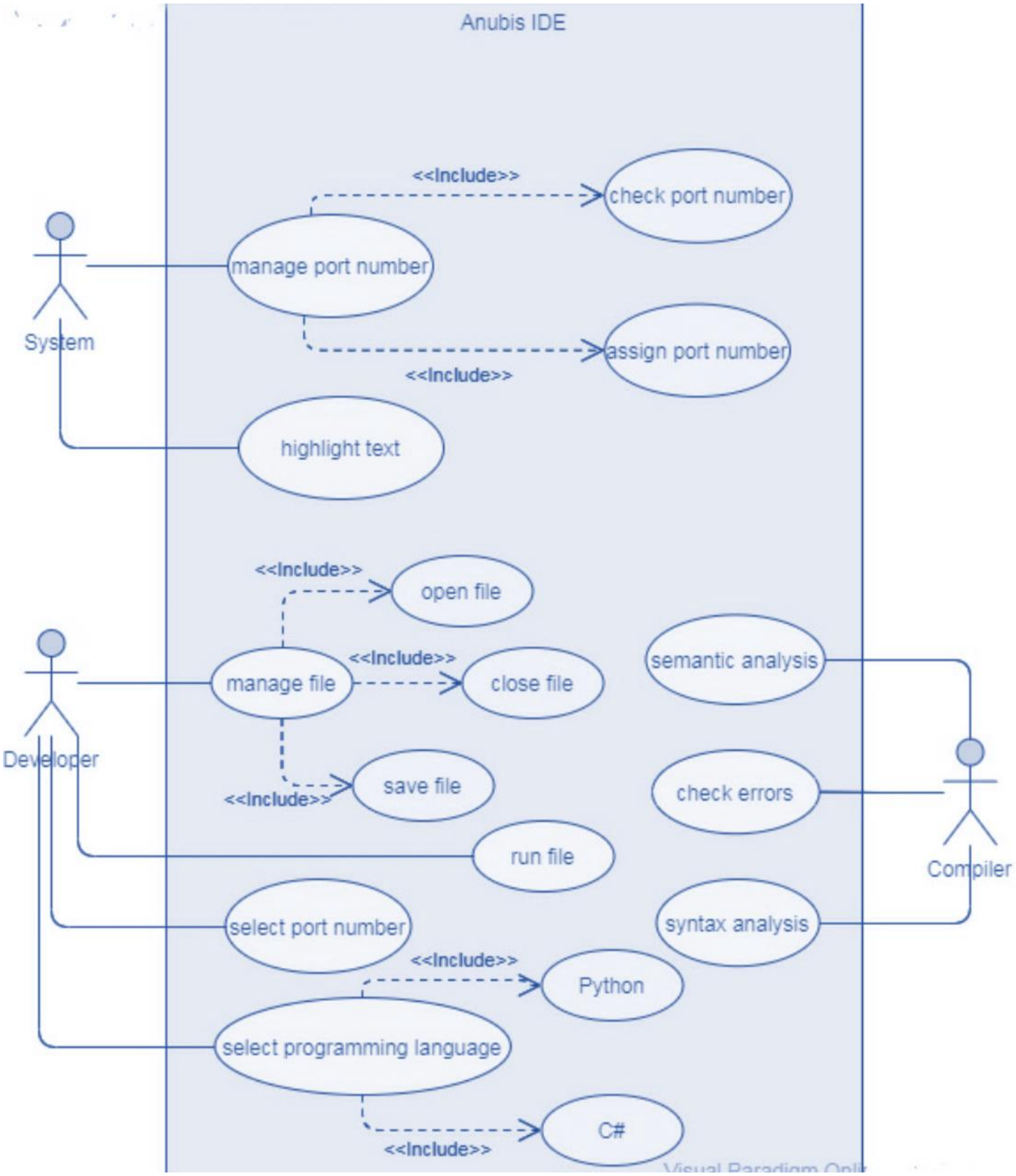
```
def Csharp(self):
  Csharp_coloring.CsharpHighlighter(text)

  print("Csharp now is used")
```



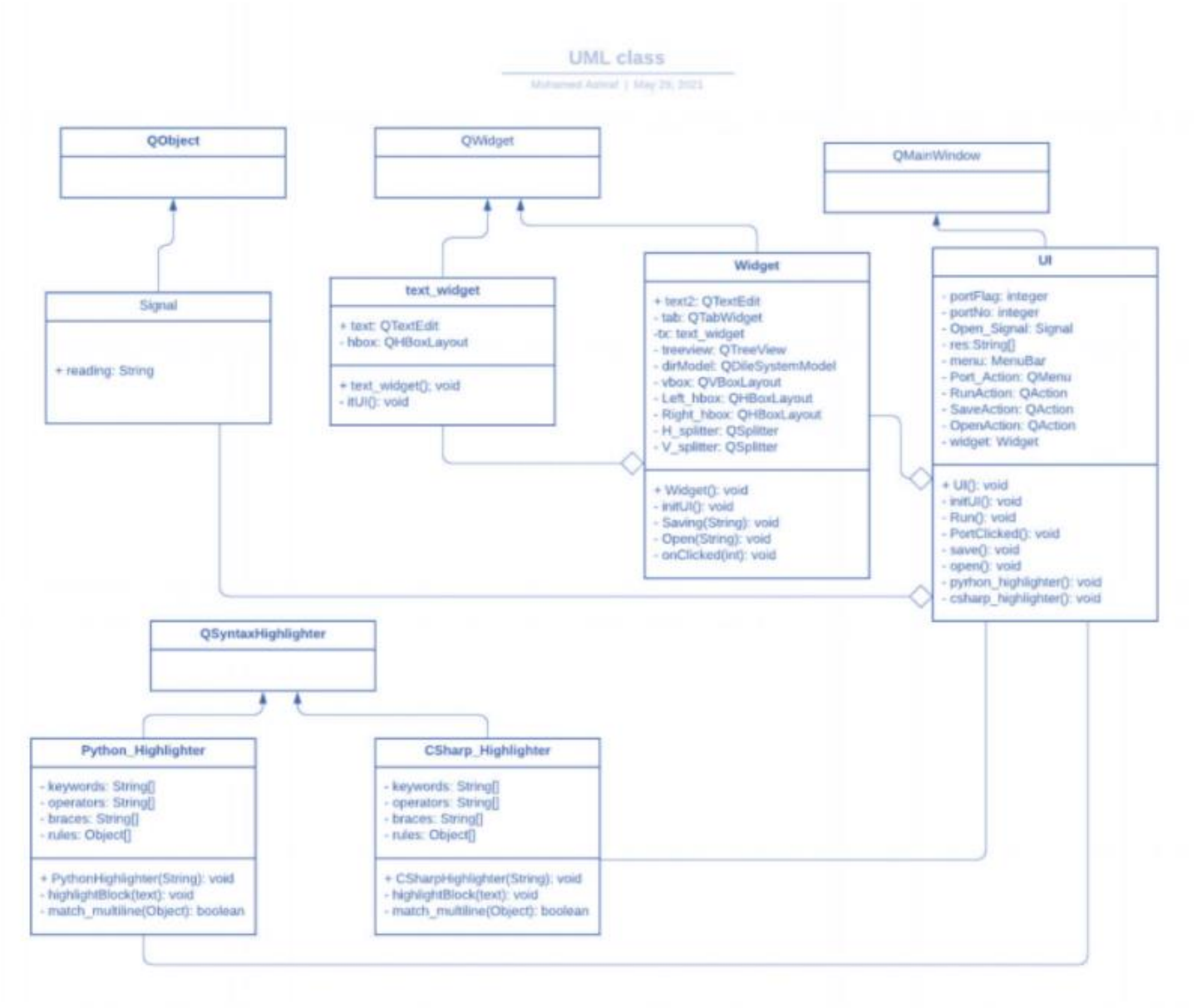
Activity Diagram



Use Case


Anubis IDE
<<Include>>
check port number
manage port number
System
assign port number
<<Include>>
highlight text
<<Include>>
open file
manage file
<<Include>>
close file
semantic analysis
Developer
save file
<<Include>>
check errors
Compiler
run file
select port number
syntax analysis
<<Include>>
Python
select programming language
C#
<<Include>>

Class Diagram



Anubis IDE code

```
#############     author => Anubis Graduation Team       ############
#############     this project is part of my graduation project and it intends to make a fully functioned IDE from scratch
########
#############     I've borrowed a function (serial_ports()) from a guy in stack overflow whome I can't remember his name, so
I gave hime the copyrights of this function, thank you  ########


import sys
import glob
import serial
import Python_Coloring
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from pathlib import Path
#      -------added by Mahmoud Mustafa------------
import os
import Csharp_coloring

file_name = " "


# csh_enable=0    # boolean to detect c# code

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()
        # *********************************
        #   Csharp_coloring.CsharpHighlighter(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)
        # ********************************* 17p8110*****************************************
        if '.cs' == file_name[0][-3:]:
            Csharp_coloring.CsharpHighlighter(text)
        else:
            print("python enabled")
            Python_Coloring.PythonHighlighter(text)

    # ************************************end***********************************

    def on_clicked(self, index):

        nn = self.sender().model().filePath(index)
        nn = tuple([nn])

        # ********************************* 17p8110*****************************************
        if '.cs' == nn[0][-3:]:
            Csharp_coloring.CsharpHighlighter(text)
        else:
            print("python enabled")
            Python_Coloring.PythonHighlighter(text)
        # ********************************** end***************************************
        if nn[0]:
            f = open(nn[0], 'r')
            with f:
                data = f.read()
                text.setText(data)


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

# 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')
        Language = menu.addMenu('Language')  # ****************************** 17p8110 **********************************

        # 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)
    # ************************************ 17p8110**********************************************************************
    python_Action = QAction("Python", self)
    python_Action.triggered.connect(self.python)
    Csharp_Action = (QAction("C_sharp", self))
    Csharp_Action.triggered.connect(self.Csharp)

    Language.addAction(python_Action)
    Language.addAction(Csharp_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 python(self):

    Python_Coloring.PythonHighlighter(text)

def Csharp(self):

    Csharp_coloring.CsharpHighlighter(text)

    print("Csharp now is used")

def Run(self):
```

```
    if self.port_flag == 0:
        mytext = text.toPlainText()

        f = open("temp.py", "w")
        f.write(mytext)
        f.close()

        #
        ##### Compiler Part
        #
        #           ide.create_file(mytext)
        #           ide.upload_file(self.portNo)
        # ---------------- Mahmoud Mustafa 17p8110-----------------------------------
        try:

            #               os.system(f'python "{file_name[0]}" > out.txt 2> errors.txt')
            os.system(f'python temp.py > out.txt 2> errors.txt')

            f = open('out.txt', 'r')
            d = open('errors.txt', 'r')
            with f, d:
                data = f.read()
                errors = d.read()


        except:
            text2.append(" Exception error")

        os.remove("out.txt")
        os.remove("errors.txt")
        os.remove("temp.py")

        if not data:
            text2.append(errors)
        else:
            text2.append(data)
    # ------------------------------------------------------------------------
    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):
    global file_name
    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__':
    app = QApplication(sys.argv)

    ex = UI()
    # ex = Widget()
    sys.exit(app.exec_())
```

C# format code

```
import sys
from PyQt5.QtCore import QRegExp
from PyQt5.QtGui import QColor, QTextCharFormat, QFont, QSyntaxHighlighter


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]),
    'this': 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'),
      'this': format('black', 'italic'),
      'numbers': format('brown'),
   }



class CsharpHighlighter(QSyntaxHighlighter):
    """Syntax highlighter for the C# language.
    """
    # C# keywords
```

```
 keywords = [
'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', 'addv', 'and',
'alias', 'ascending', 'async', 'await', 'by', 'descending', 'dynamic', 'equals', 'from', 'get', 'global', 'group',
'init', 'into', 'join', 'let', 'managed', 'nameof', 'nint', 'not', 'notnull', 'nuint', 'on', 'or', 'orderby', 'partial',
'record', 'remove', 'select', 'set', 'unmanaged', 'value', 'var', 'when', 'where', 'with', 'yield',
]
 operators = [
    '=',
    # Comparison
    '==', '!=', '<', '<=', '>', '>=',
    # Arithmetic
    '\+', '-', '\*', '/', '//', '\%', '\*\*',
    # In-place
    '\+=', '-=', '\*=', '/=', '\%=',
    # Bitwise
    '\^', '\|', '\&', '\~', '>>', '<<',
 ]

 # c# braces
 braces = [
    '\{', '\}', '\(', '\)', '\[', '\]',
 ]
 def __init__(self,document):
      QSyntaxHighlighter.__init__(self, document)
    # Block comments
    self.block_start = (QRegExp("/[*]"), 1, STYLES['comment'])
    self.block_end = (QRegExp("[*]/"), 1, STYLES['comment'])

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

    # All other rules
    rules += [
        # 'this'
        (r'\bthis\b', 0, STYLES['this']),
        # Double-quoted string, possibly containing escape sequences
        (r'"[^"\\]*(\\.[^"\\]*)*"', 0, STYLES['string']),
        # Single-quoted string, possibly containing escape sequences
        (r"'[^'\\]*(\\.[^'\\]*)*'", 0, STYLES['string']),


        # From // comment
        (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.rules = [(QRegExp(pat), index, fmt)
                  for (pat, index, fmt) in rules]

def highlightBlock(self, text):
    """Apply syntax highlighting to the given block of text.
    """
    # Do other syntax formatting
    for expression, nth, format in self.rules:
        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 multi-line comments
    in_multiline = self.match_multiline(text, *self.block_start,*self.block_end)

def match_multiline(self, text, delimiter, in_state, style,end_delimiter, end_state, end_style):
    """Do highlighting of multi-line comments
    """
    # 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 = end_delimiter.indexIn(text, start + add)
        # Ending delimiter on this line?
        if end >= add:
            length = end - start + add + end_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
```

Python format code

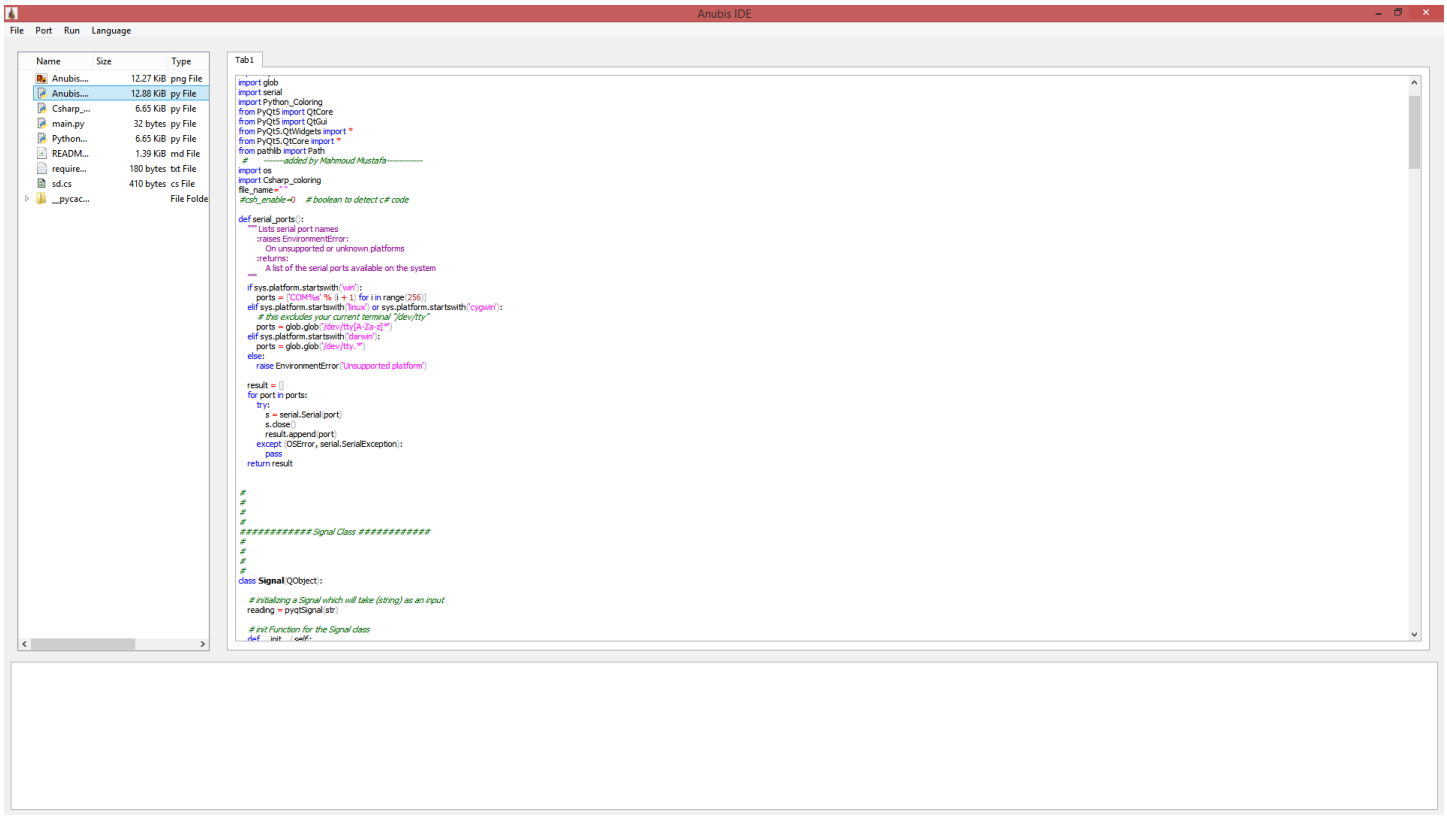
```
import sys
from PyQt5.QtCore import QRegExp
from PyQt5.QtGui import QColor, QTextCharFormat, QFont, QSyntaxHighlighter


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 PythonHighlighter(QSyntaxHighlighter):
    """Syntax highlighter for the Python language.
    """
    # Python keywords
```

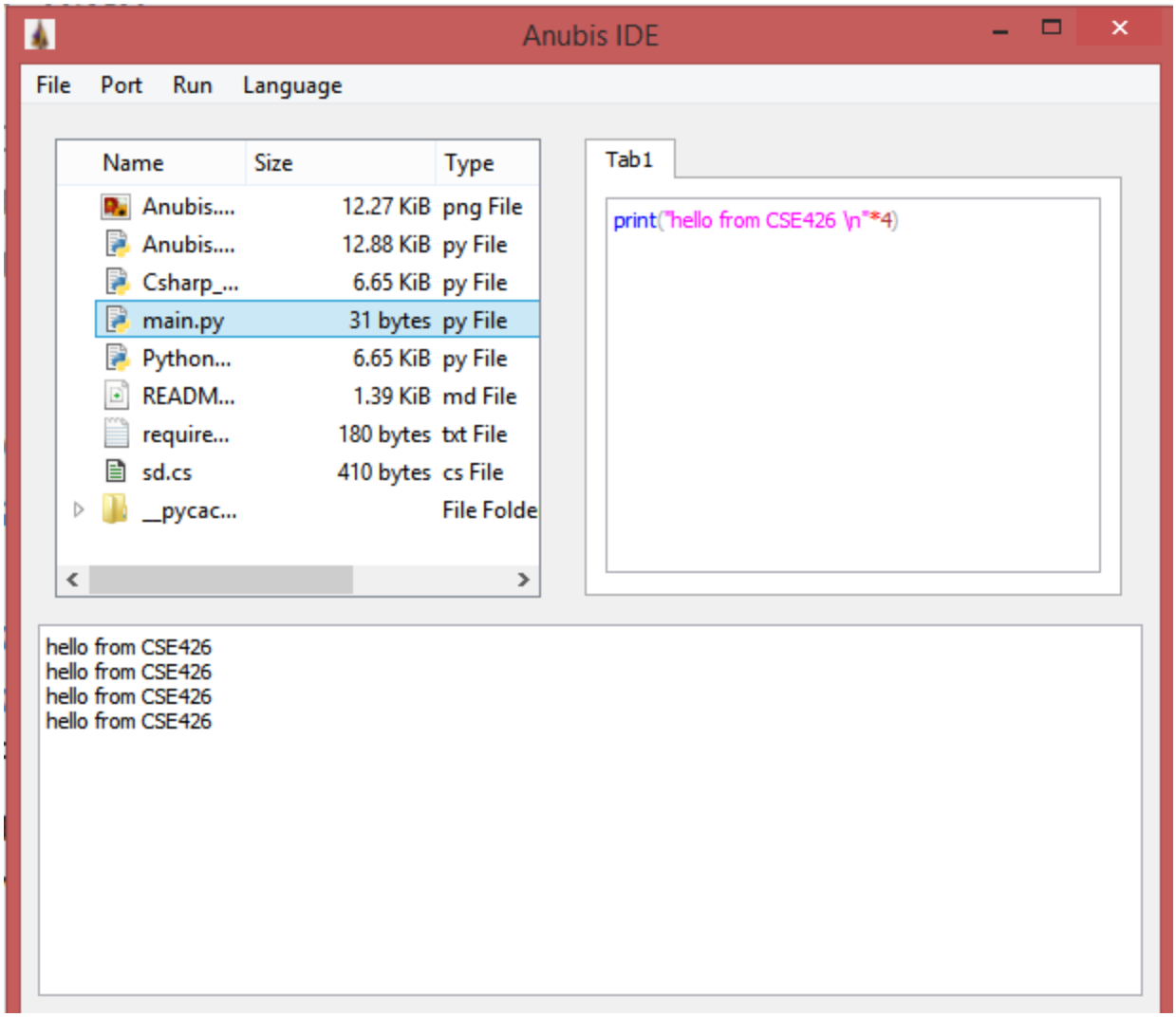
```
keywords = [
    'and', 'assert', 'break', 'class', 'continue', 'def',
    'del', 'elif', 'else', 'except', 'exec', 'finally',
    'for', 'from', 'global', 'if', 'import', 'in',
    'is', 'lambda', 'not', 'or', 'pass', 'print',
    'raise', 'return', 'try', 'while', 'yield',
    'None', 'True', 'False',
]

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

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

    rules = []

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

    # All other rules
    rules += [
        # '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']),

        # block comment
        (r'"/\*.*?\*/|\/{2}.*?\n|\s"', 1, STYLES['comment']),
        # '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.rules = [(QRegExp(pat), index, fmt)
                      for (pat, index, fmt) in rules]

    def highlightBlock(self, text):
        """Apply syntax highlighting to the given block of text.
        """
        # Do other syntax formatting
        for expression, nth, format in self.rules:
            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 multi-line strings
        in_multiline = self.match_multiline(text, *self.tri_single)
        if not in_multiline:
            in_multiline = self.match_multiline(text, *self.tri_double)

    def match_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
```

Screen shots of python formatting




Anubis IDE
File
Port
Run
Language
Name
Size
Type
Anubis....
12.27 KiB
png File
Anubis....
12.88 KiB
py File
Csharp_...
6.65 KiB
py File
main.py
31 bytes
py File
Python...
6.65 KiB
py File
READM...
1.39 KiB
md File
require...
180 bytes
txt File
sd.cs
410 bytes
cs File
__pycac...
File Folde
Tab1
print("hello from CSE426 \n"*4)
hello from CSE426
hello from CSE426
hello from CSE426
hello from CSE426

C# format example