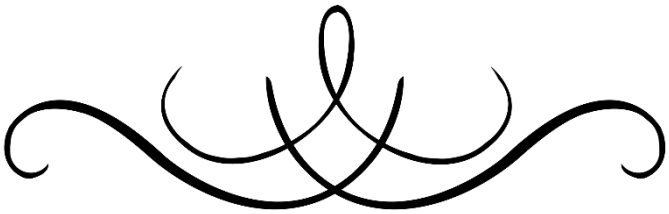


EVOLVING ANUBIS-IDE

SEPTEMBER 10, 2020

AIN SHAMS UNIVERSITY – FACULTY OF ENGINEERING
1 Al-Sarayyat st, Abbassiya, Cairo 11517, Egypt

FACULTY OF ENGINEERING

AIN SHAMS UNIVERSITY

CSE426: Maintenance and Evolution

EVOLVING ANUBIS-IDE

Submitted By:

Abdelrahman Ibrahim ELGhamry

16P3043@eng.asu.edu.eg
Ghamry1998@gmail.com

Submitted to:

Prof. Dr. Ayman Bahaa

SEPTEMBER 10, 2020

A REPORT FOR SOFTWARE MAINTENANCE AND EVOLUTION COURSE CODDED CSE426 WITH THE REQUIREMENTS OF AIN SHAMS UNIVERSITY

TABLE OF CONTENTS

1. PURPOSE

The aim of this document is to build a desktop-based open-source Integrated development environment (IDE) which enables its users to write, edit, compile, and run python and C# codes on micro-controllers.

It will also explain the aim and the key features of the system, the various scenarios of the system, and the constraints under which the system must operate.

Throughout this document, architecture and procedure, constraints, application sequence, and updates are going to be explained in detail.

GitHub Repo: https://github.com/Ghamry98/Anubis-IDE

2. BRIEF DESCRIPTION

Anubis-IDE is an open source desktop-based text editor, which aims to provide a simple integrated development environment to write, edit, compile, and run python scripts and C# codes on various micro-controllers.

This new product will assist a huge community of embedded-systems engineers to compile, build and run their python and C# codes directly on micro-controllers in an efficient and manageable manner.

3. SYSTEM REQUIREMENTS

3.1 Functional Requirements

1. The software should support opening and editing any text file.
2. The user can write micro-python codes to files.
3. The IDE must support code highlighting, syntax checking and auto-completion.
4. Debugging tools must be provided to the user.
5. The IDE must provide a list of all available ports on the running host in order to select one of them.

6. The user must select the attached micro-controller port before compiling and running the code.
7. The IDE must compile, flush, and run the code on the selected micro-controller.
8. The IDE has a panel to display the class hierarchy view.
9. The IDE has a panel to display the project structure as folders and files in the current directory.
10. The IDE has a code editor panel that supports code highlighting for all reserved words, numbers, comments and variables in the currently supported programming language.
11. The IDE should save files in the currently opened directory.

3.2 Added Functional Requirements

1. The IDE should support C# programming language.
2. The editor must automatically recognize which format to use based on the file extension.

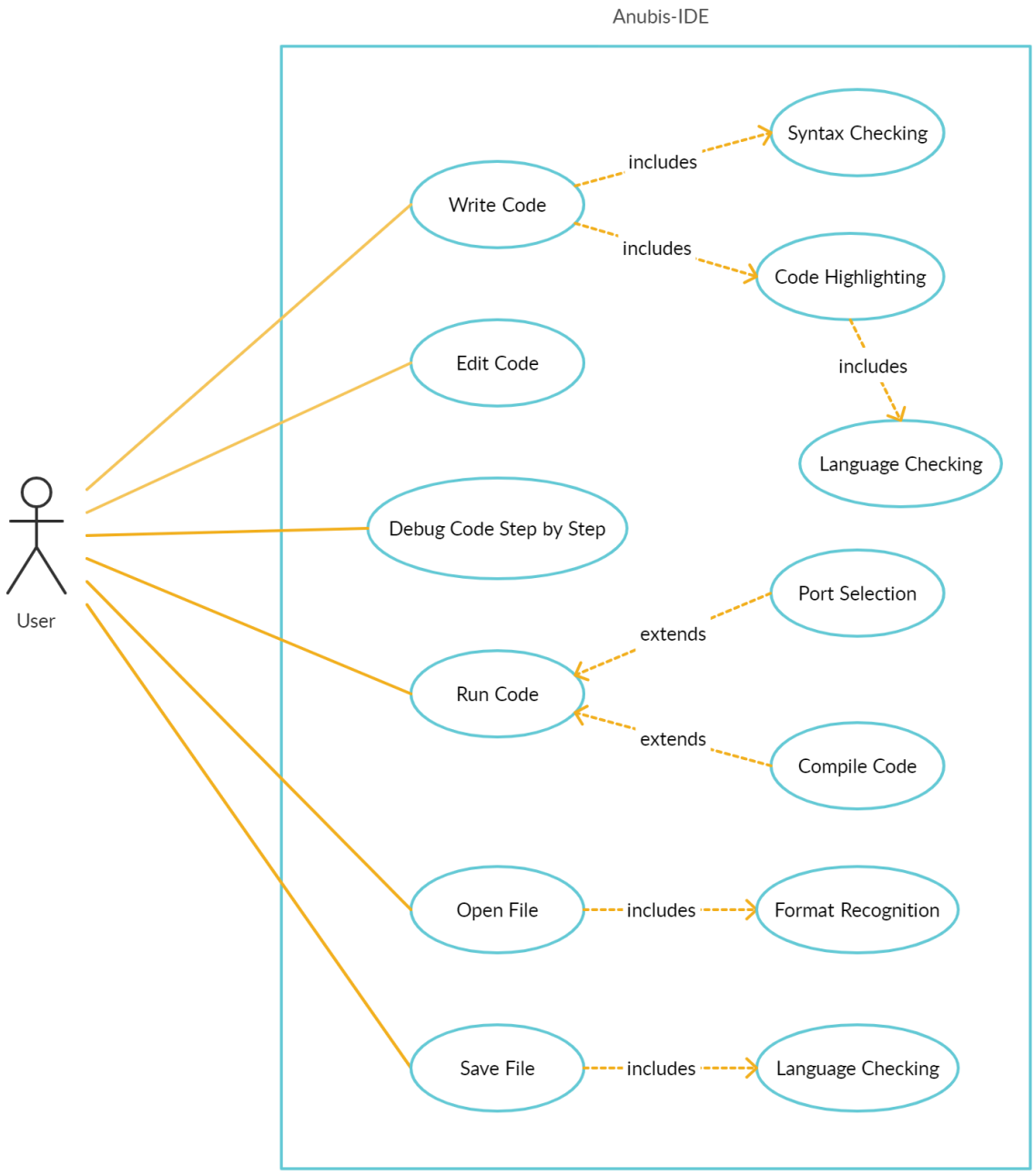
3.3 Non-Functional Requirements

3. The software must be written using python.
4. The software must be compatible with various operating systems: Windows, macOS and Linux.
5. Syntax errors detection and feedback must happen within one second.
6. The software running requirements must not exceed 4GB of RAM.
7. The software must use Git for version controlling and the repo must be public on GitHub.
8. The software must follow agile process model and deliver increments within 3 weeks cycles.
9. The software must be delivered within 9 months on 3 main releases where 3 months is considered for each one.
10. The software must declare dependencies and instructions to install them.

4. USE CASE DIAGRAM

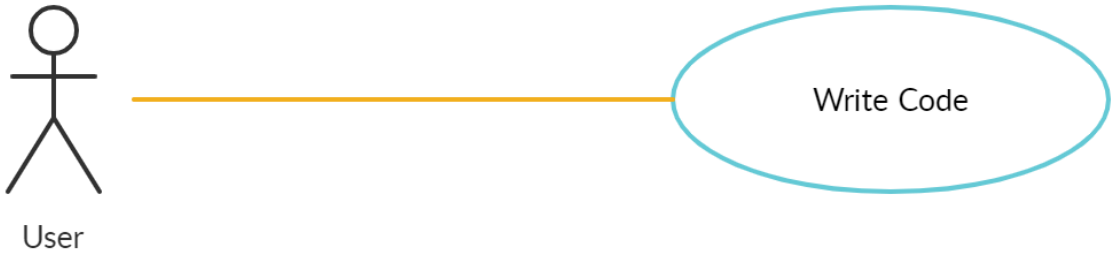
Update:

- Opening a file includes recognizing which format to use based on file extension.
- Saving a file includes checking the chosen programming language format.
- Code highlighting includes programming language checking.



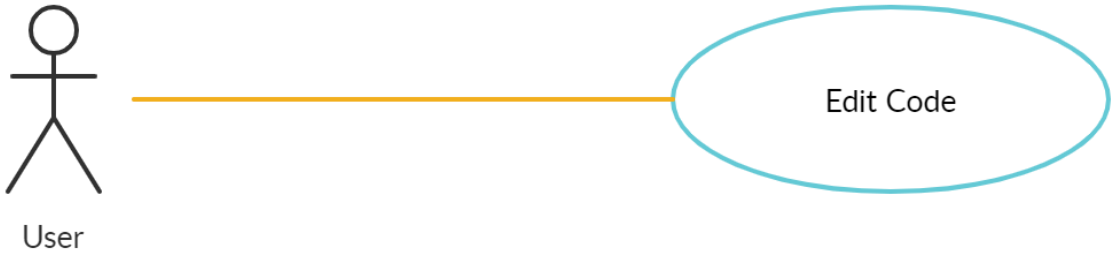
5. NARRATIVE DESCRIPTION

1. Write Code



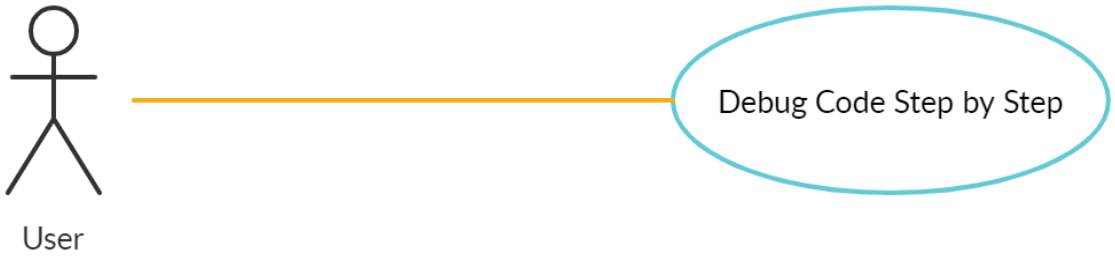
Description	The user should be able to write python code in the text-editing panel.
Primary Actor	The user.
Main Flow	1. User opens the IDE 2. User starts typing in the text-editing panel.

2. Edit Code



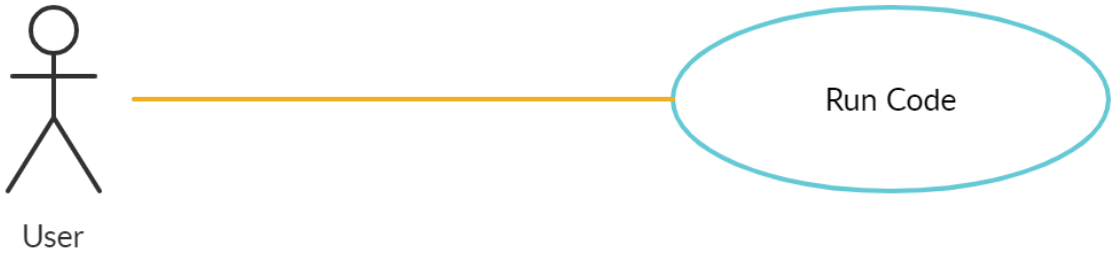
Description	The user should be able to edit an existing python code in the text-editing panel.
Primary Actor	The user.
Main Flow	1. User selects a python file from the tree view. 2. Code opens in the text-editing panel. 3. User starts editing code.

3. Debug Code Step by Step



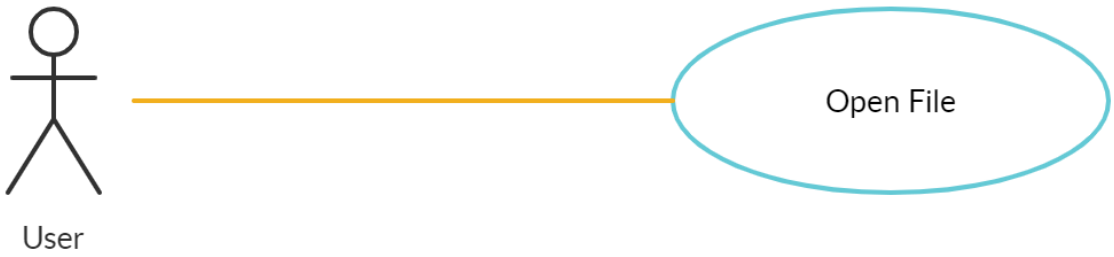
Description	The user should be able to use break points to debug his python code.
Primary Actor	The user.
Main Flow	1. User selects a line of code and adds a break point to it. 2. User specify the running port. 3. User runs the code. 4. A new panel that shows used variables values should appear to the user.

4. Run Code



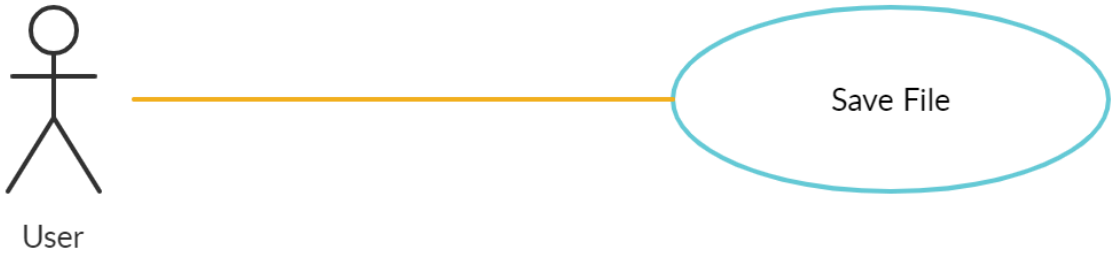
Description	The user should be able to run his python code on the micro-controller.
Primary Actor	The user.
Main Flow	1. User specify the running port. 2. User chooses to run the code. 3. A feedback message should appear to the user whether it's a successful run or not.

5. Open File



Description	The user should be able to open an existing file using the tree view panel or the file menu.
Primary Actor	The user.
Main Flow	1. User opens the file menu. 2. User selects the “Open” option. 3. User selects the desired file to open using the file explorer.

6. Save File



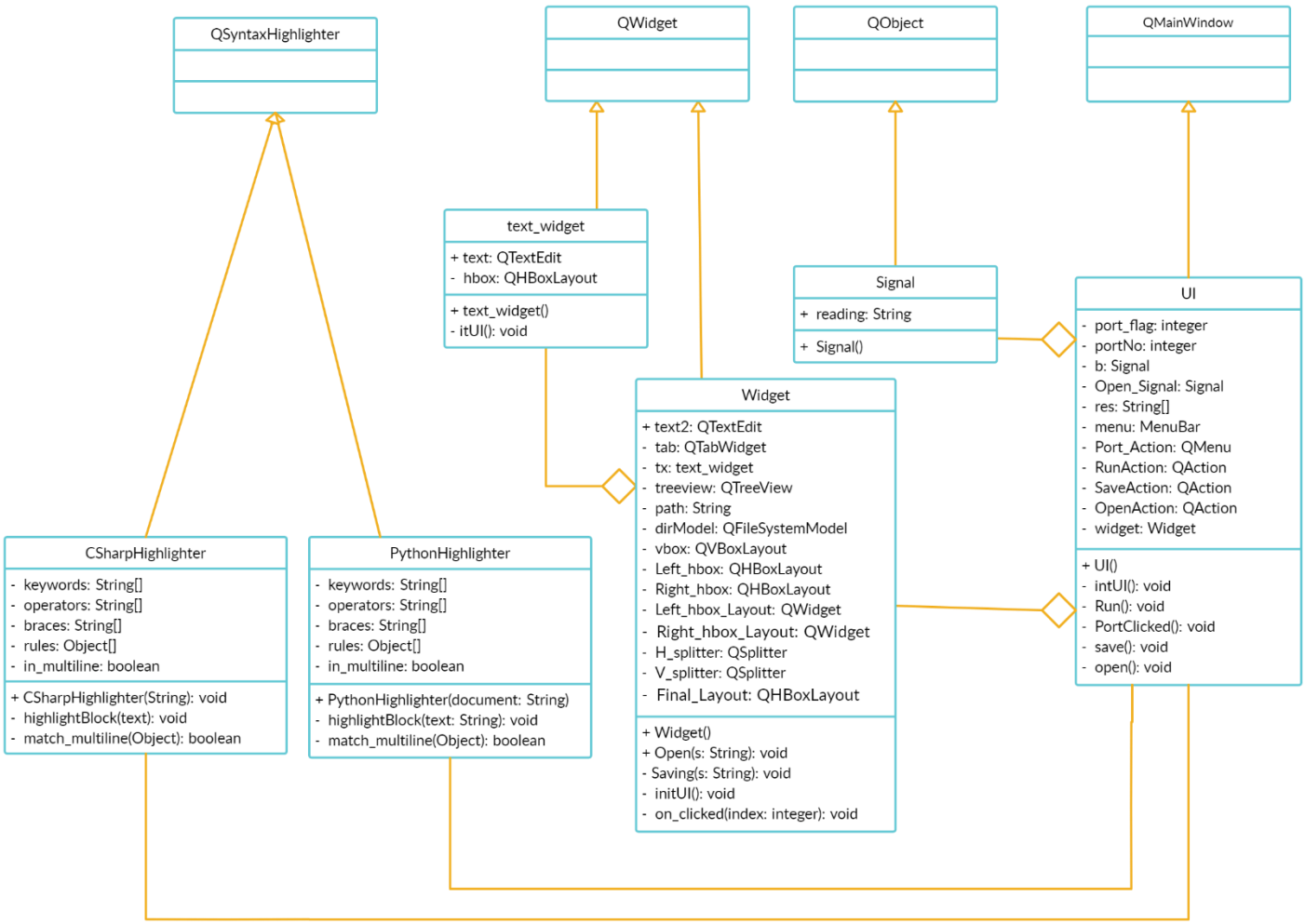
Description	The user should be able to save his current progress.
Primary Actor	The user.
Main Flow	1. User opens the file menu. 2. User selects the “Save” option.

6. SYSTEM DESIGN

6.1 Class Diagram

Update:

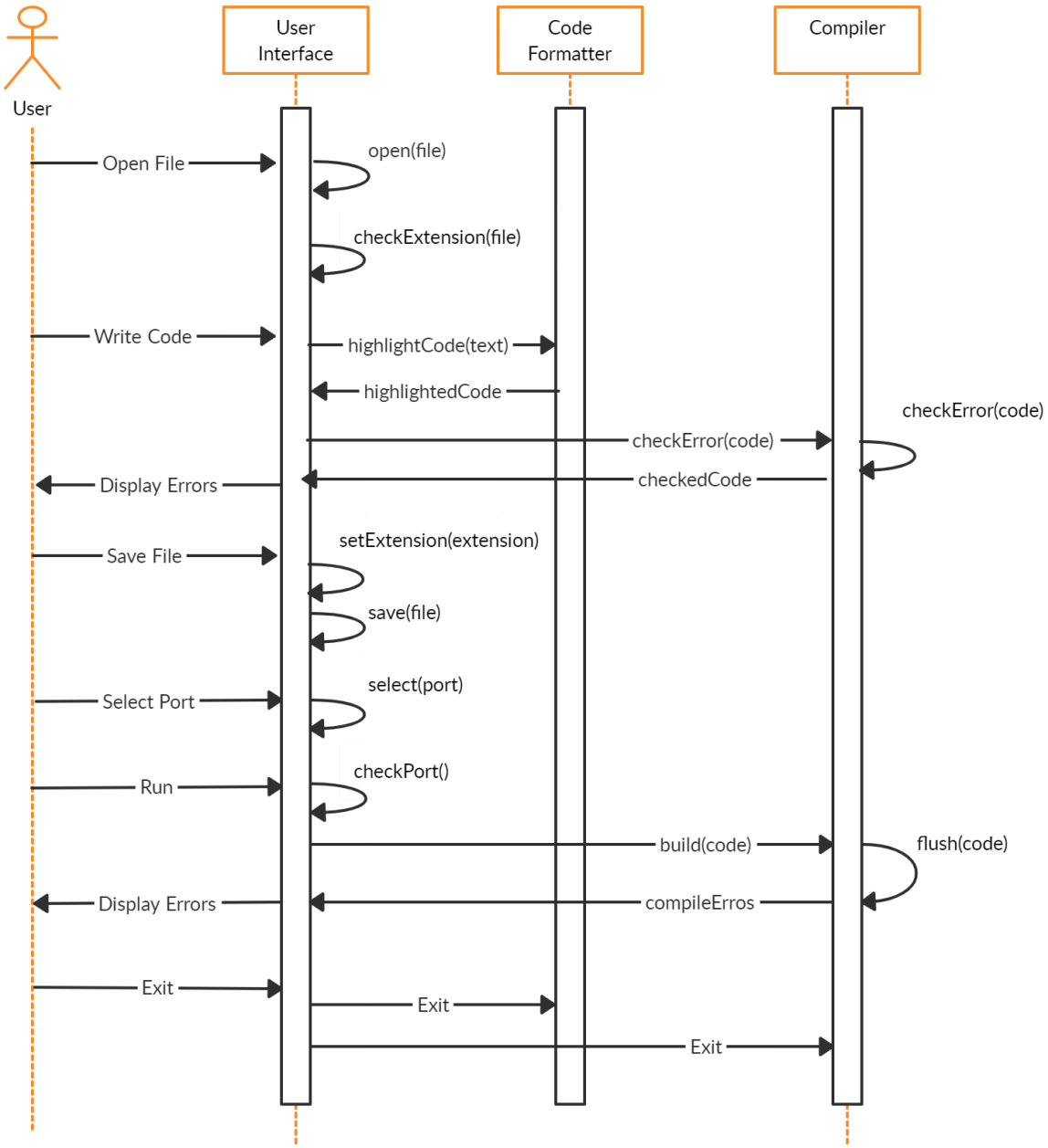
- Adding CSharpHighlighter class, CSharpHighlighter is responsible for formatting the C# code.



6.2 Sequence Diagram

Update:

- The user can now select his preferred programming language to write; He can choose between Python and C#.
- Also, the user can now open and edit C# files.



7. PROJECT INSTALLATION

Firstly, we need to get our environment ready to run the project, then we need to clone the project repo. Follow up the following steps for more details:

(Ps: if your environment is ready to run a python project, skip part (1))

7.1 Part (1)

1. Install any text editor (VS Code is recommended)
2. Install Python (3.0 or above is recommended)
3. Add Python's extension to your vs code.

7.2 Part (2)

1. Clone the project repo from the following link
   https://github.com/Ghamry98/Anubis-IDE

2. Open the project directory using any text editor
   (VS Code is recommended).

3. Open the terminal and run the following command(s) to download the project dependencies (packages):

   ```
   $ pip install -r requirements.txt
   ```

   Or

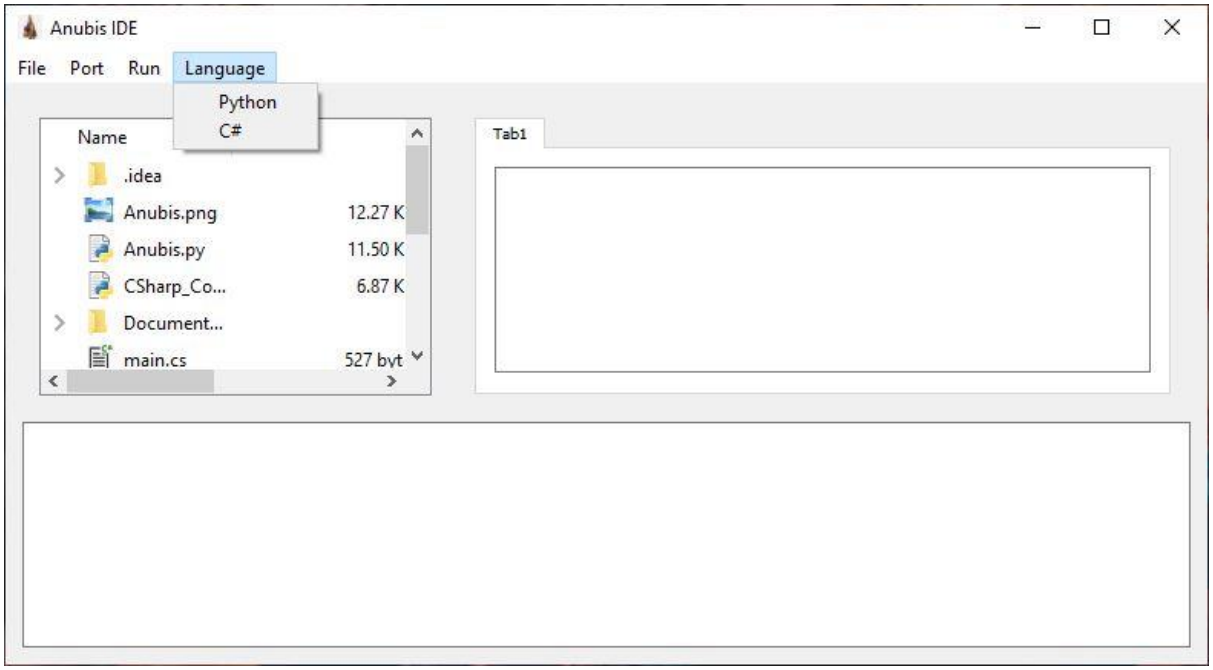
   ```
   $ pip install pyserial
   $ pip install PyQt5
   ```

4. Run "Anubis.py" using your text editor.

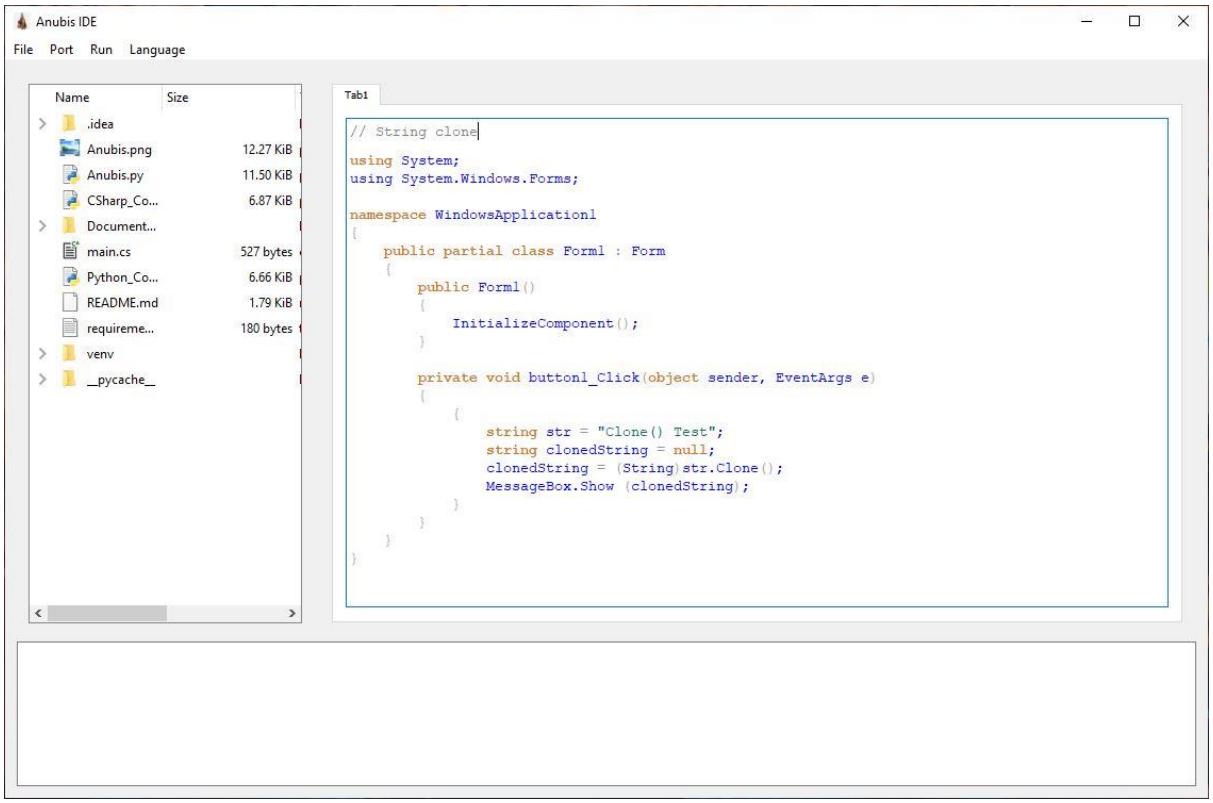
8. RUNNING PROGRAM SCREENSHOTS

Program on start up

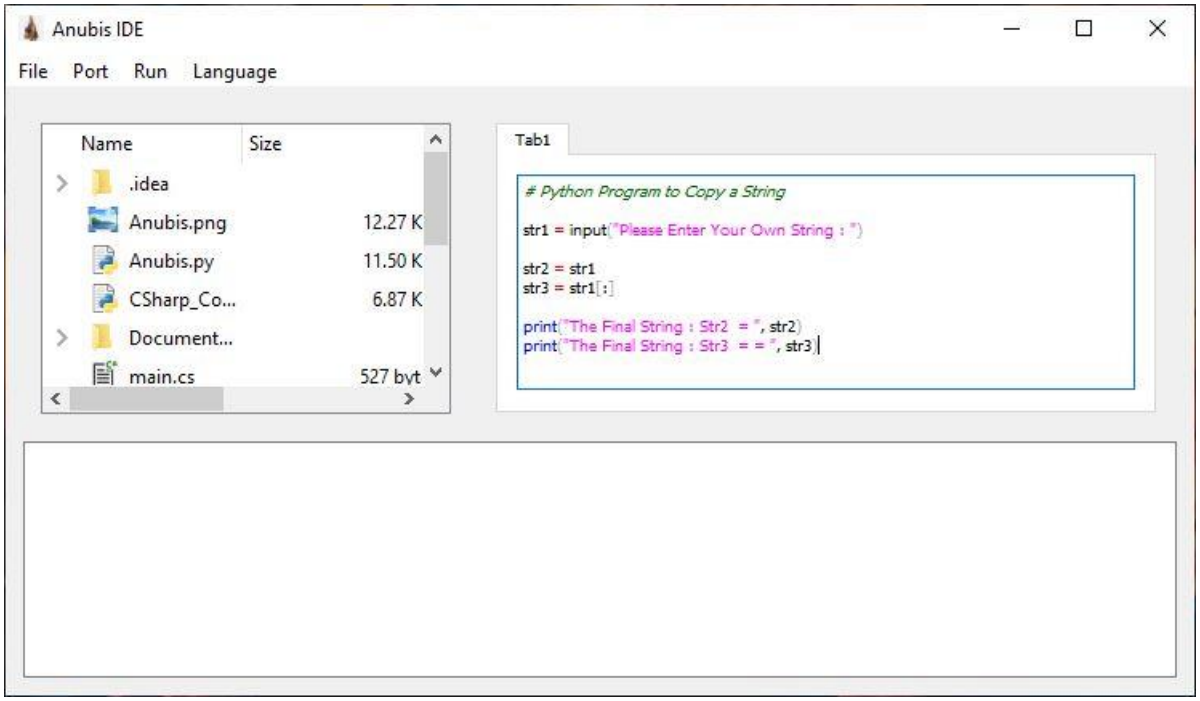
Language selection (new added feature).



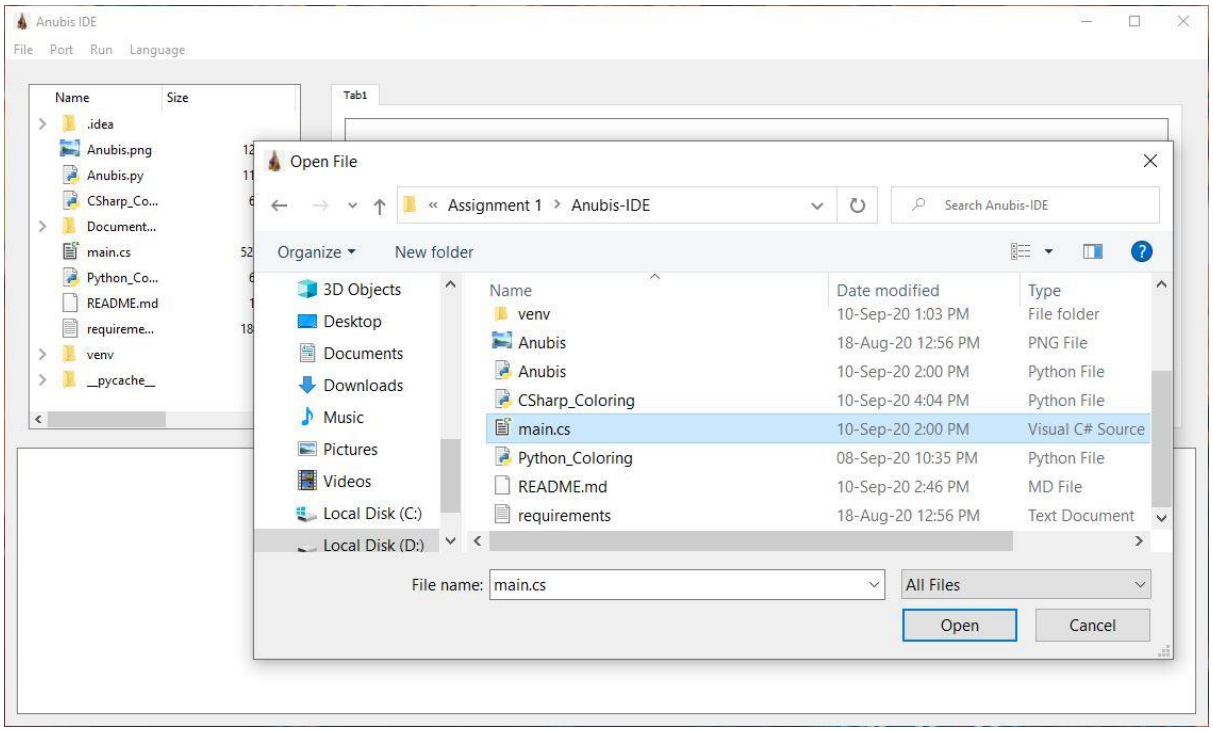
Syntax analyzing of C# string cloning code.

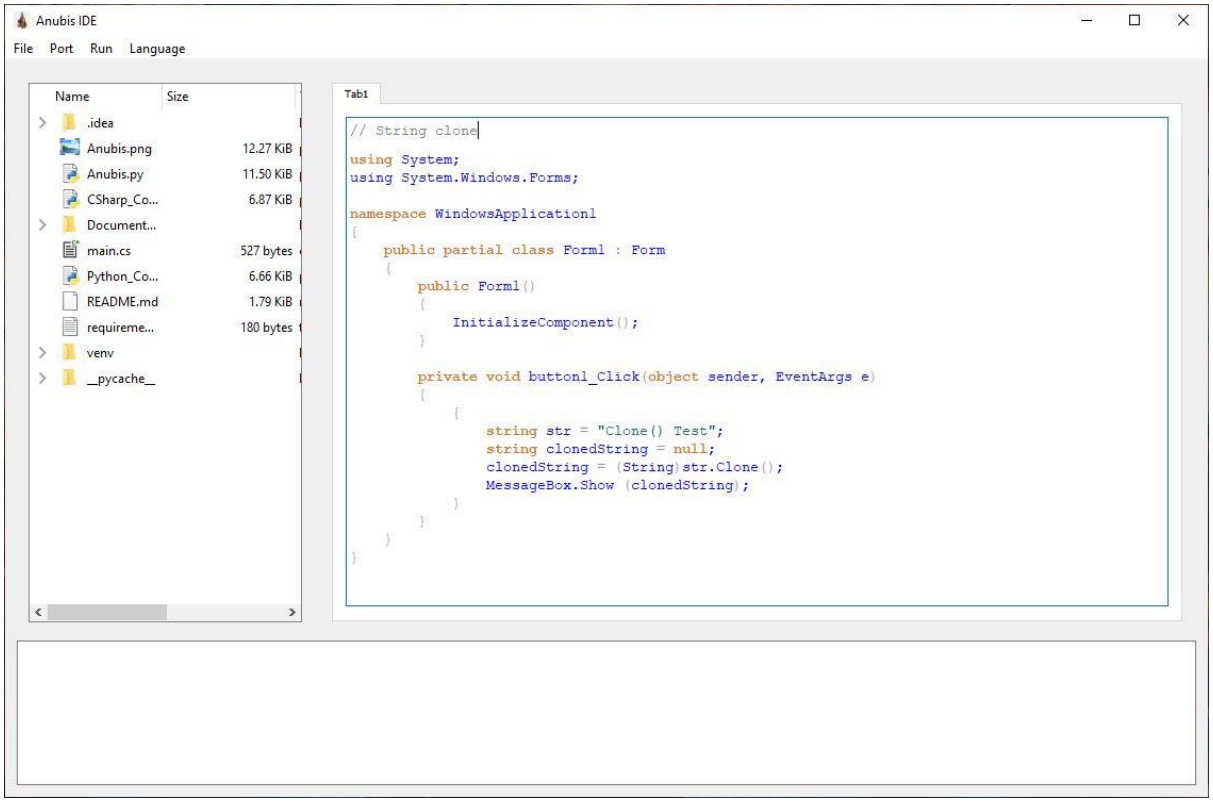


Syntax analyzing of python string cloning code.



Opening and editing a file (language detection).





9. CODE

9.1 Anubis.py

Anubis.py

```
##############      author => Anubis Graduation Team        ############
##############      this project is part of my graduation project and it intend
s to make a fully functioned IDE from scratch    ########
##############      I've borrowed a function (serial_ports()) from a guy in sta
ck 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
import CSharp_Coloring
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from pathlib import Path

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 t
o (self) in widget class)
text = QTextEdit
text2 = QTextEdit
language = "Python"

#
#
#
#
############ 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()
        Python_Coloring.PythonHighlighter(text)
        hbox = QHBoxLayout()
        hbox.addWidget(text)
        self.setLayout(hbox)



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



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

    def __init__(self, ui):
        super().__init__()
        self.initUI()
        self.ui = ui

    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 incl
uded files
```

```
        self.treeview = QTreeView()

        # making a variable (path) and setting it to the root path (surely I c
an 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 som
efilters (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 dire
ctory names.
        # Files => List files.
        self.dirModel.setFilter(QDir.NoDotAndDotDot | QDir.AllDirs | QDir.File
s)
        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 e
ditor 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 QH
BoxLayout
        # 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 sid
```

```
es 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 tex
t editor
    @pyqtSlot(str)
    def Saving(s):
        if language == "Python":
            with open('main.py', 'w') as f:
                TEXT = text.toPlainText()
                f.write(TEXT)
        else:
            with open('main.cs', '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):

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

        fileExtension = nn[0].split(".")[1]
        if fileExtension == "py":
            UI.python_analyzer(self.ui)
        else:
            UI.csharp_analyzer(self.ui)

        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 clas
s) 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')
        self.language_menu = menu.addMenu('Language')

        # As any PC or laptop have many ports, so I need to list them to the U
ser
        # so I made (Port_Action) to add the Ports got from (serial_ports()) f
unction
        # copyrights of serial_ports() function goes back to a guy from stacko
verflow(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)

        python_action = QAction('Python', self)
        python_action.triggered.connect(self.python_analyzer)
        csharp_action = QAction('C#', self)
        csharp_action.triggered.connect(self.csharp_analyzer)
```

```
        self.language_menu.addAction(python_action)
        self.language_menu.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)

        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 tex
t editor
    def open(self):
        file_name = QFileDialog.getOpenFileName(self,'Open File','/home')
        fileExtension = file_name[0].split(".")[1]
        if fileExtension == "py":
            self.python_analyzer()
        else:
            self.csharp_analyzer()
```

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

    def python_analyzer(self):
        global language
        language = "Python"
        Python_Coloring.PythonHighlighter(text)

    def csharp_analyzer(self):
        global language
        language = "C#"
        CSharp_Coloring.CSharpHighlighter(text)

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

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = UI()
    # ex = Widget()
    sys.exit(app.exec_())
```

9.2 CSharp_Coloring.py

Anubis.py

```
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
STYLES = {
    '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]),
}

class CSharpHighlighter(QSyntaxHighlighter):
    """Syntax highlighter for the C Sharp language.
    """
    # C Sharp keywords
    keywords = [
        'abstract', 'bool',     'continue', 'decimal',  'default',
        'event',    'explicit', 'extern',   'char',     'checked',
        'class',    'const',    'break',    'as',       'base',
        'delegate', 'is,'       'lock',     'long',     'num',
        'byte',     'case',     'catch',    'false',    'finally',
        'fixed',    'float',    'for',      'foreach',  'static',
        'goto',     'if',       'implicit', 'in',       'int',
        'interface','internal', 'do',       'double',   'else',
        'namespace','new',      'null',     'object',   'operator',
        'out',      'override', 'params',   'private',  'protected',
        'public',   'readonly', 'sealed',   'short',    'sizeof',
        'ref',      'return',   'sbyte',    'stackalloc','static',
        'string',   'struct',   'void',     'volatile', 'while',
        'true',     'try',      'switch',   'this',     'throw',
        'unchecked' 'unsafe',   'ushort',   'using',    'using',
        'virtual',  'typeof',   'uint',     'ulong',    'out',
        'add',      'alias',    'async',    'await',    'dynamic',
        'from',     'get',      'orderby',  'ascending','decending',
        'group',    'into',     'join',     'let',      'nameof',
        'global',   'partial',  'set',      'remove',   'select',
        'value',    'var',      'when',     'Where',    'yield'
    ]

    # C Sharp operators
```

```
    operators = [
        '=',
        # logical
        '!', '?', ':',
        # Comparison
        '==', '!=', '<', '<=', '>', '>=',
        # Arithmetic
        '\+', '-', '\*', '/', '\%', '\+\+', '--',
        # Assignment
        '\+=', '-=', '\*=', '/=', '\%=', '<<=', '>>=', '\&=', '\^=', '\|=',
        # 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
        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 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 += [
            # Double-quoted string, possibly containing escape sequences
            (r'"[^"\\]*(\\.[^"\\]*)*"', 0, STYLES['string']),
            # Single-quoted string, possibly containing escape sequences
            (r"'[^'\\]*(\\.[^'\\]*)*'", 0, STYLES['string']),

            # Comments. 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
```

9.3 Python_Coloring.py

Python_Coloring.py

```
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']),

            # '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.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
```