Faculty of

Engineering Ain

Shams University Credit

hours program

CSE426 - Software Maintenance and Evolution

Evolving The Editor Project

Submitted to: Prof. Ayman Bahaa-Eldin
Submitted by: Mariam Hassan Nassar
ID: 17P6075

Brief Description

Anubis-IDE is an open-source desktop text editor that helps provide a simple integrated development environment to write, edit, run and compile python & C# code on microcontrollers. This project is supposed to facilitate development for embedded-systems engineers. It will help them compile, build and run their code directly on the microcontroller as efficiently as possible.

GitHub Repo: https://github.com/MiraNassar134/Anubis-IDE

System Requirements

A. Functional Requirements

The software must be able to:

1) Support opening and editing any text files
2) Allow writing micro-python codes to files
3) Support code highlighting
4) Support syntax checking
5) Support auto-completion
6) Provide debugging tool
7) Provide list of all available ports to select one
8) Allow selection of attached microcontroller port before run & compile
9) Compile, flush and run code on selected microcontroller
10) Have panel to display class hierarchy
11) Have panel to display project structure in files and folders directory format
12) Have code editor panel with code highlighting for reserved words, comments & variables
13) Save files in currently opened directory

B. Added Functional Requirements

The software must be able to:

1) Support C# programming language
2) Automatically recognize which format to use based on file extension

C. Non-Functional Requirements

The software must:

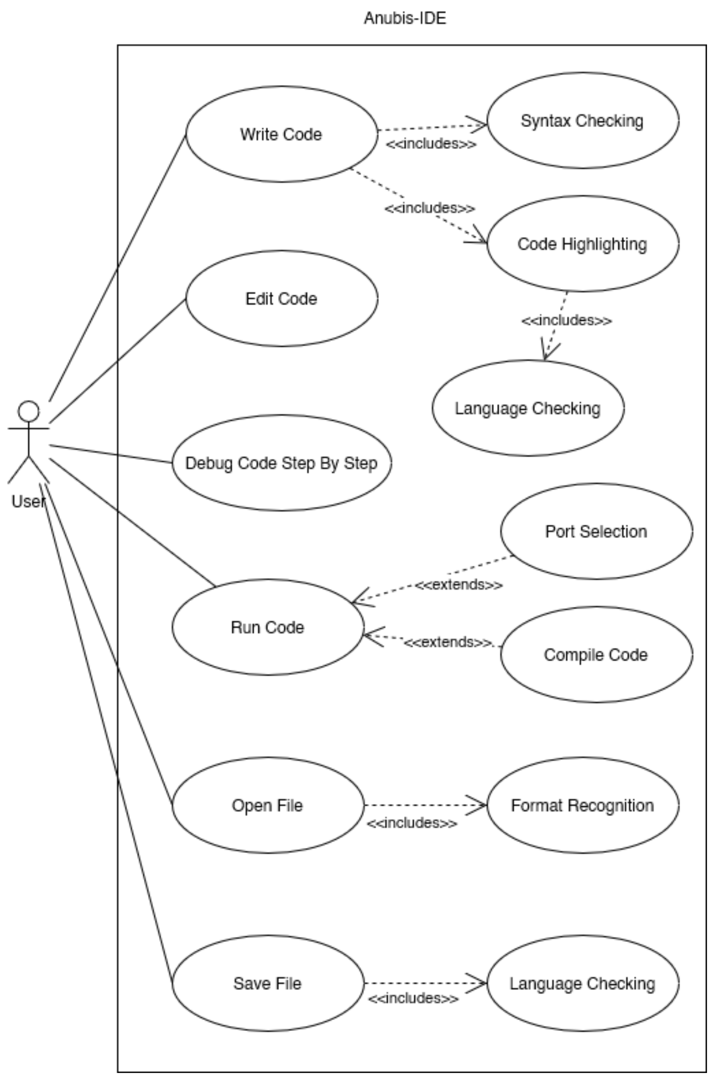
1) Be written in python
2) Be compatible with different operating systems
3) Provide error detection and feedback within a second of request

4) Not exceed 4GB of RAM usage
5) Use Git for version control
6) Have a public repo on GitHub
7) Follow agile process model
8) Be delivered within 3 main releases that are 3 months-long each
9) Declare dependencies
10) Provide installation instructions

Use Case Diagram

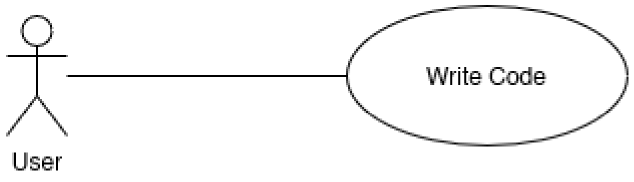
Updates:

- Open file includes existing format recognition based on file extension
- Save file includes checking the format for the programming language chosen
- Code highlighting includes checking programming language



Use Case Description

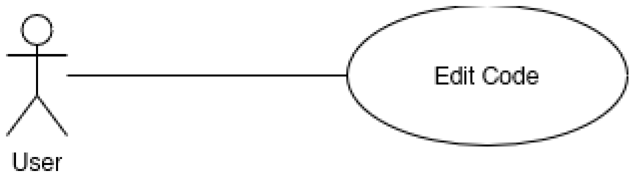
1) Write Code



Description: User should be able to write python code in text-editing panel
Primary Actor: User
Main Flow:

1. User opens IDE
2. User starts typing in text-editing panel

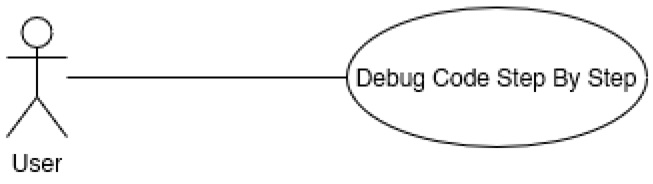
2) Edit Code



Description: User should be able to edit existing python code in text-editing panel
Primary Actor: User
Main Flow:

1. User selects python file from tree view
2. Code opens in text-editing panel
3. User starts editing code

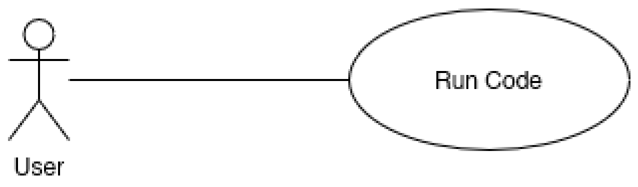
3) Debug Code Step By Step



Description: User should be able to use break points to debug python code
Primary Actor: User
Main Flow:

1. User selects line of code and adds break point to it
2. User specify running port
3. User runs code
4. New panel appears to show used variables, values should appear to user

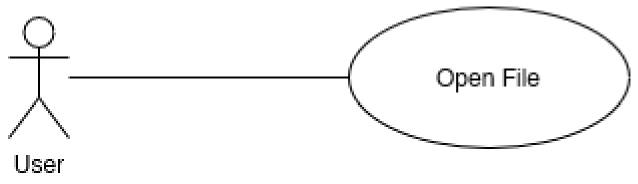
4) Run Code



Description: User should be able to run python code on a microcontroller
Primary Actor: User
Main Flow:

1. User specify running port
2. User chooses to run code
3. Feedback message should appear to user showing if run is successful or not

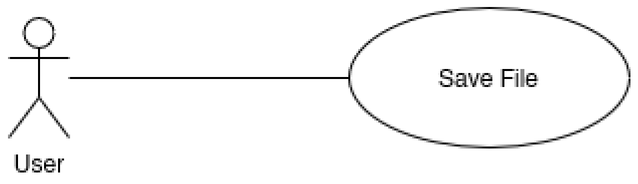
5) Open File



Description: User should be able to open existing code file from the tree view or file menu
Primary Actor: User
Main Flow:

1. User opens file menu
2. User selects “Open”
3. User selects desired file through the file explorer

6) Save File



Description: User should be able to save current progress
Primary Actor: User
Main Flow:

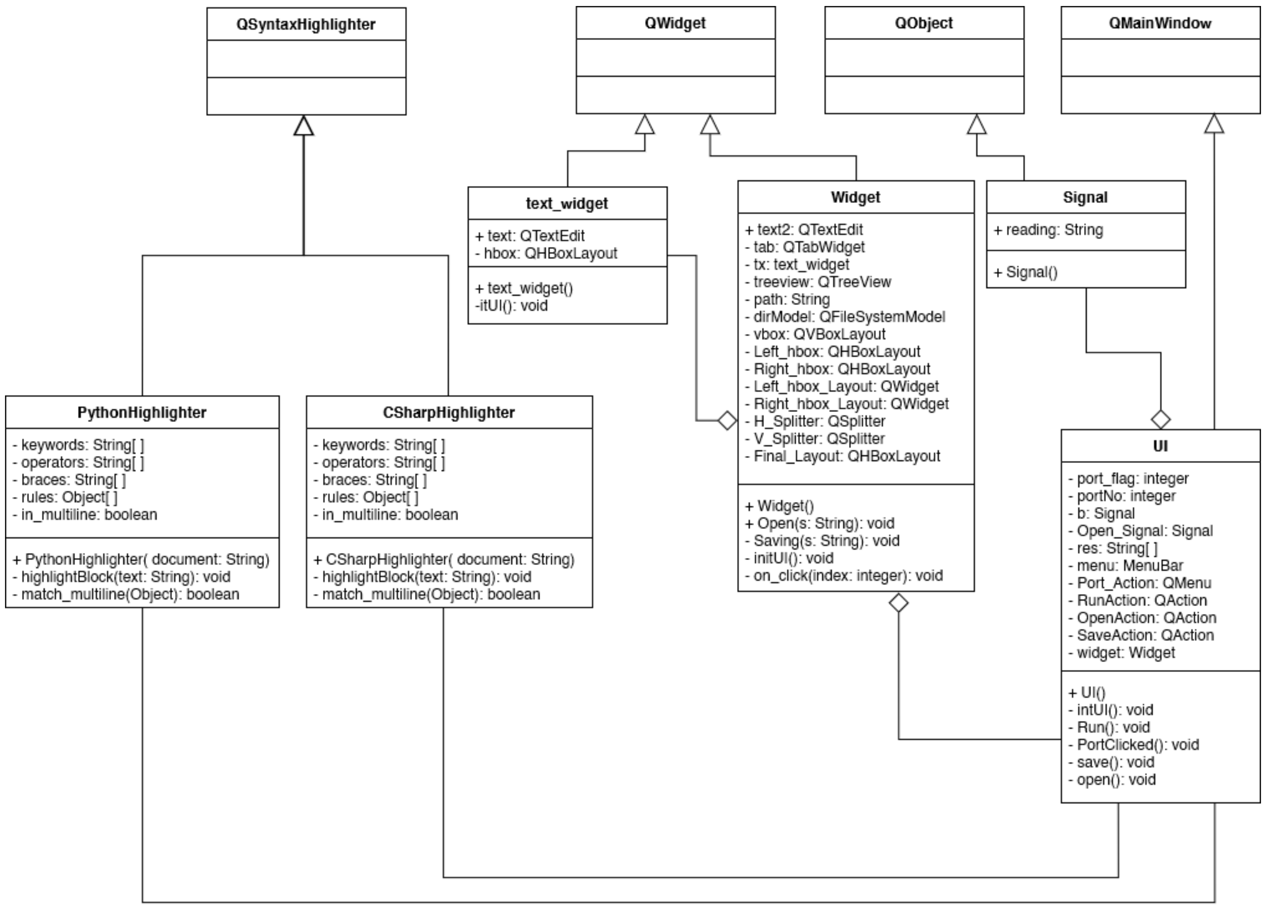
1. User opens file menu
2. User selects “Save”

System Design

A. Class Diagram

Updates:

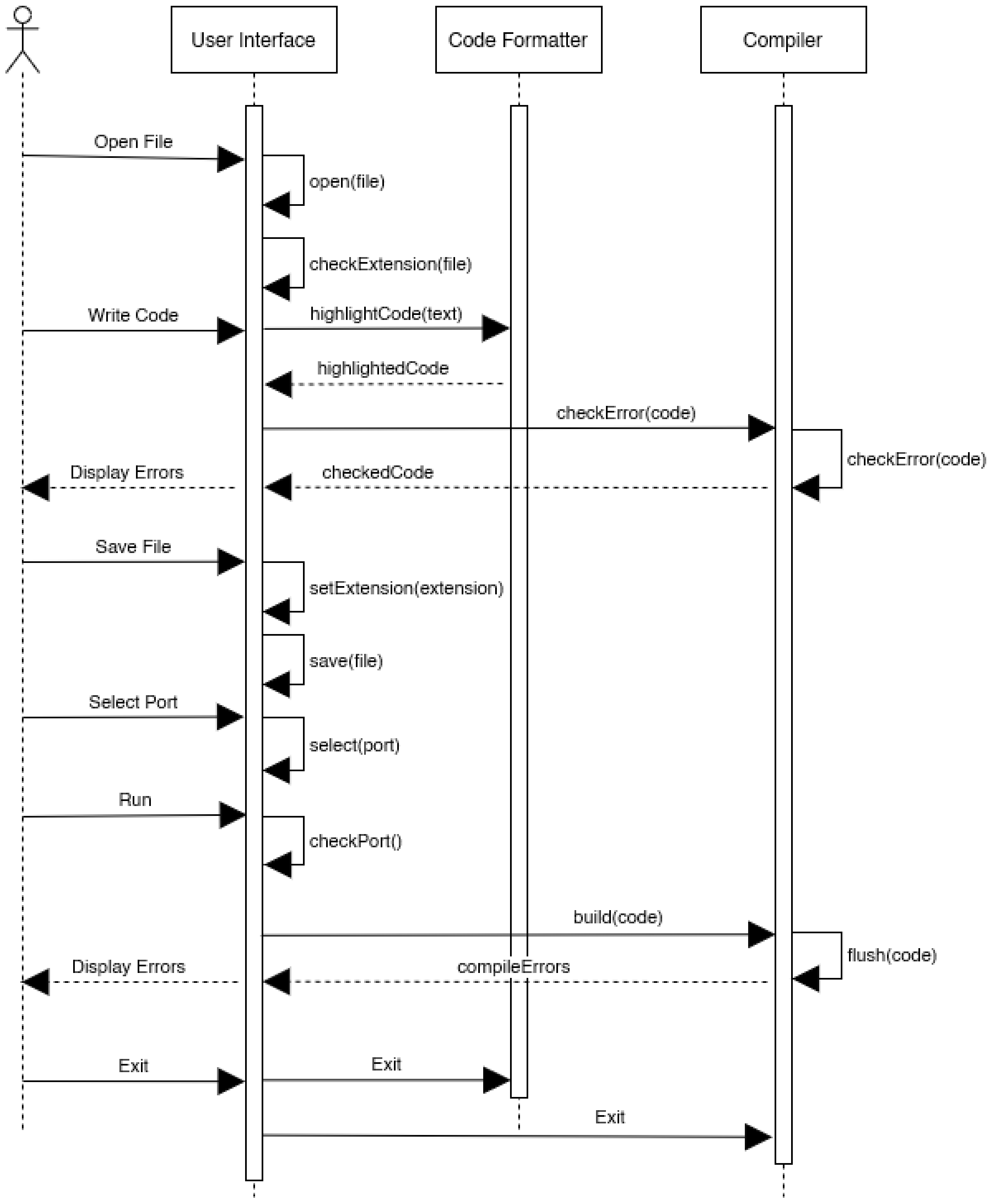
- Added CSharpHighlighter class that inherits from QSyntaxHighlighter and is responsible for C# code formatting



B. Sequence Diagram

Updates:

- User can select preferred programming language; either Python or C#
- User can open and edit C# files



Program Installation

To successfully run program, user must set up the environment then clone the project repo. If the environment is already set up, user can skip part (A).

A. Environment Set Up

1) Install any text editor, preferably VS Code
2) Install Python, recommended version 3.0 or above
3) Add Python extension to VS Code

B. Anubis-IDE Installation

1) Clone project repo from this link: https://github.com/MiraNassar134/Anubis-IDE
2) Open project director using any text editor, preferably VS Code
3) Open terminal and run the following commands to install project dependencies:

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

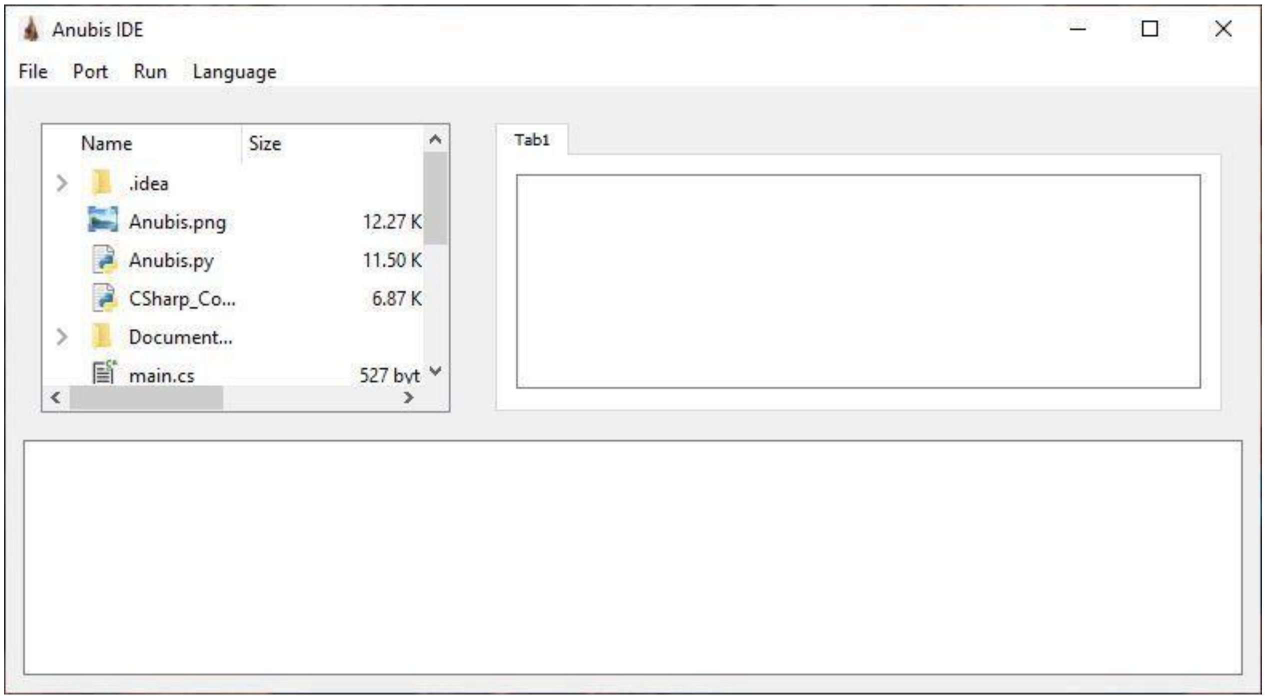
Or

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

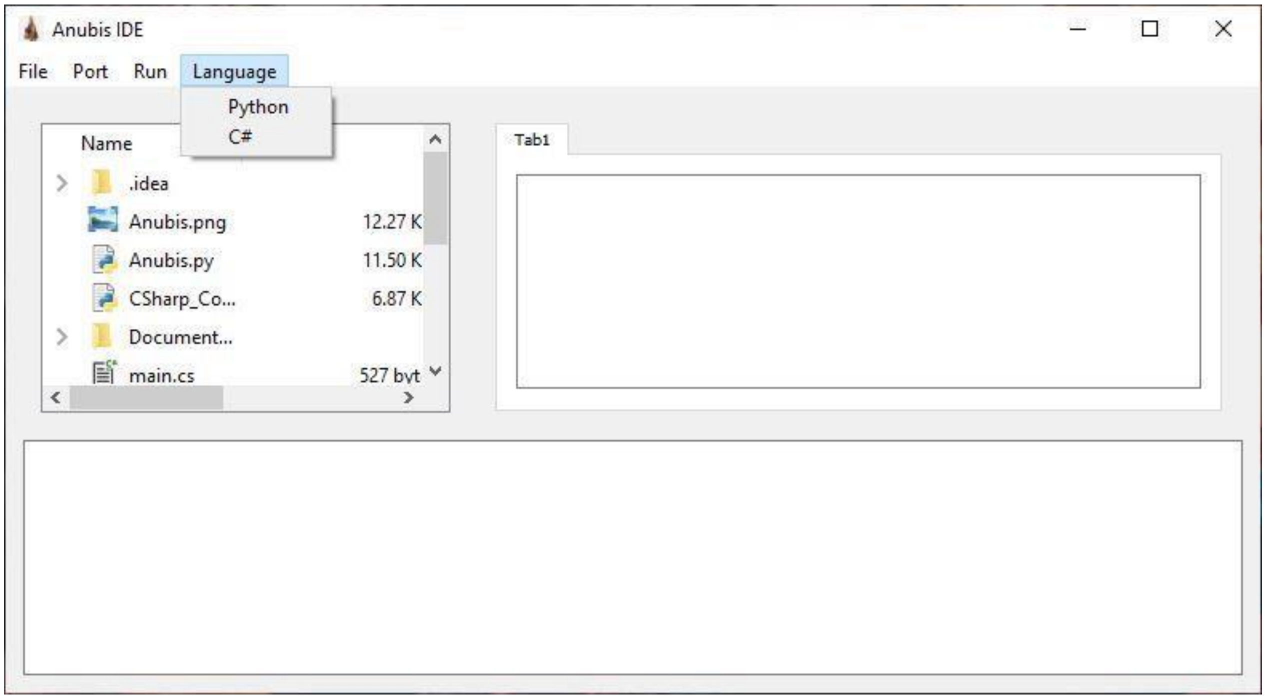
4) Run "Anubis.py"

Program Screenshots

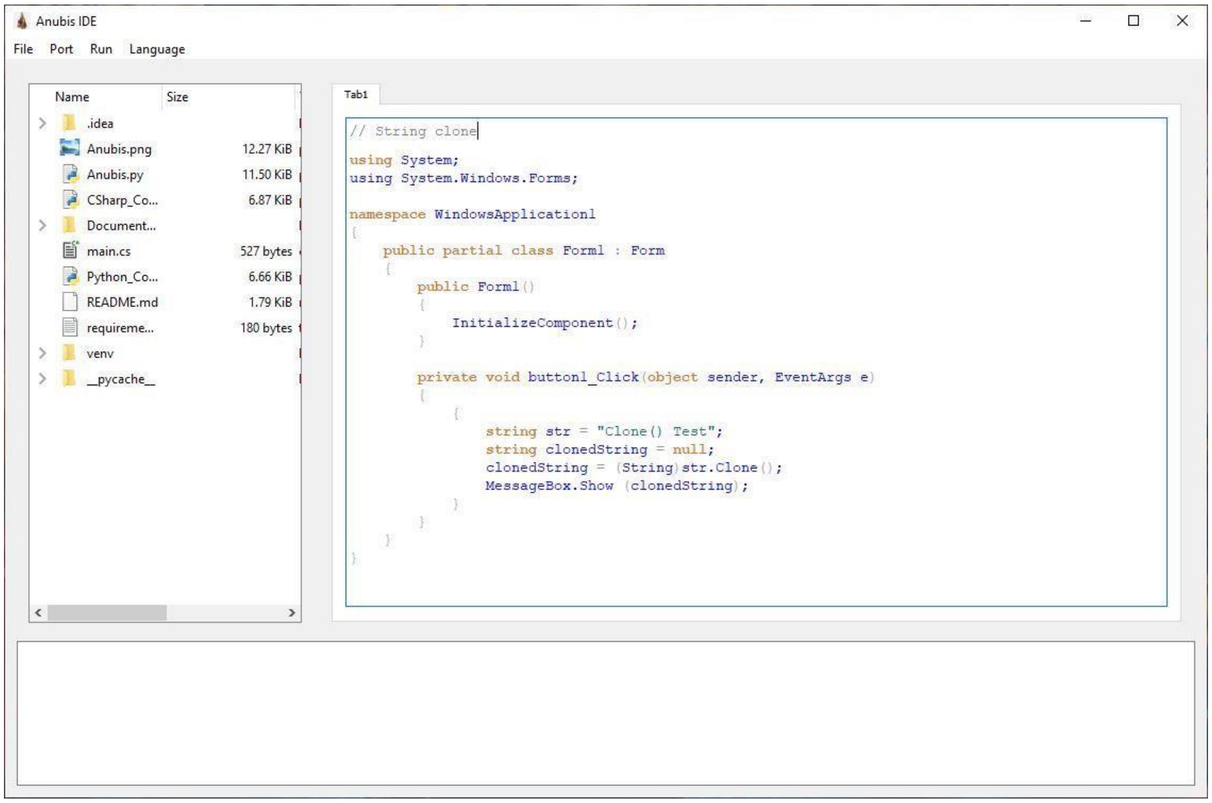
Program on Start Up:



Language Selection (New Feature):

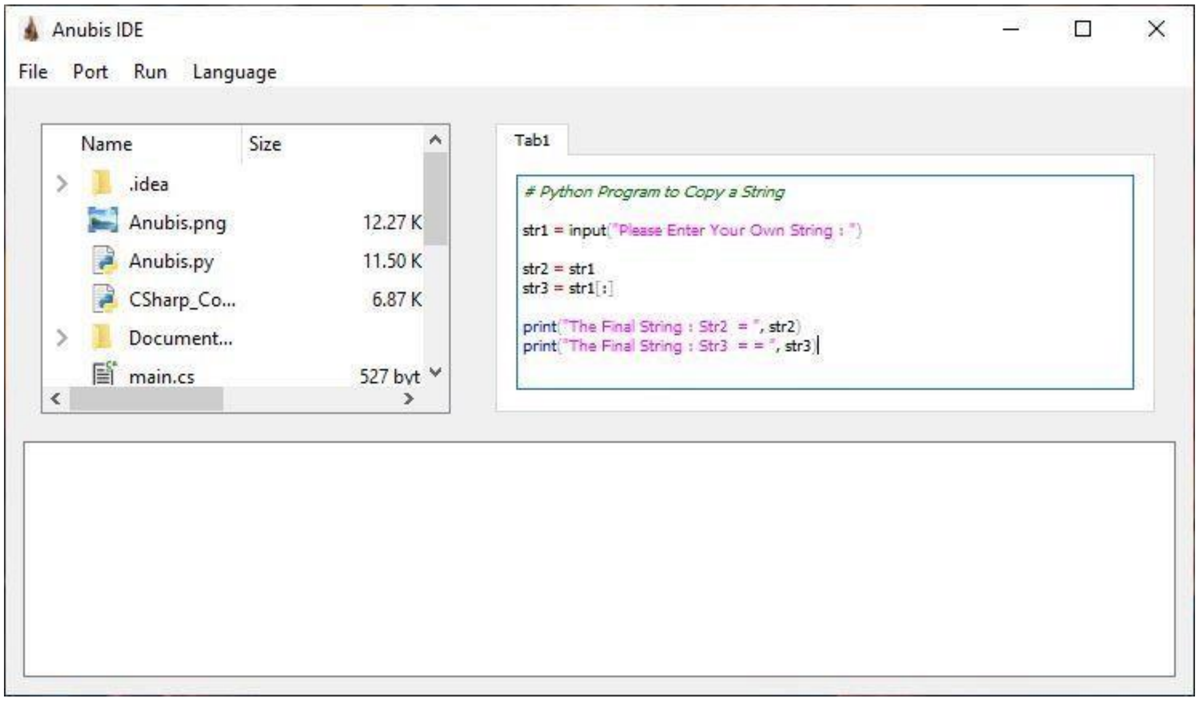


Syntax Analysis of C# string cloning code (New Feature):



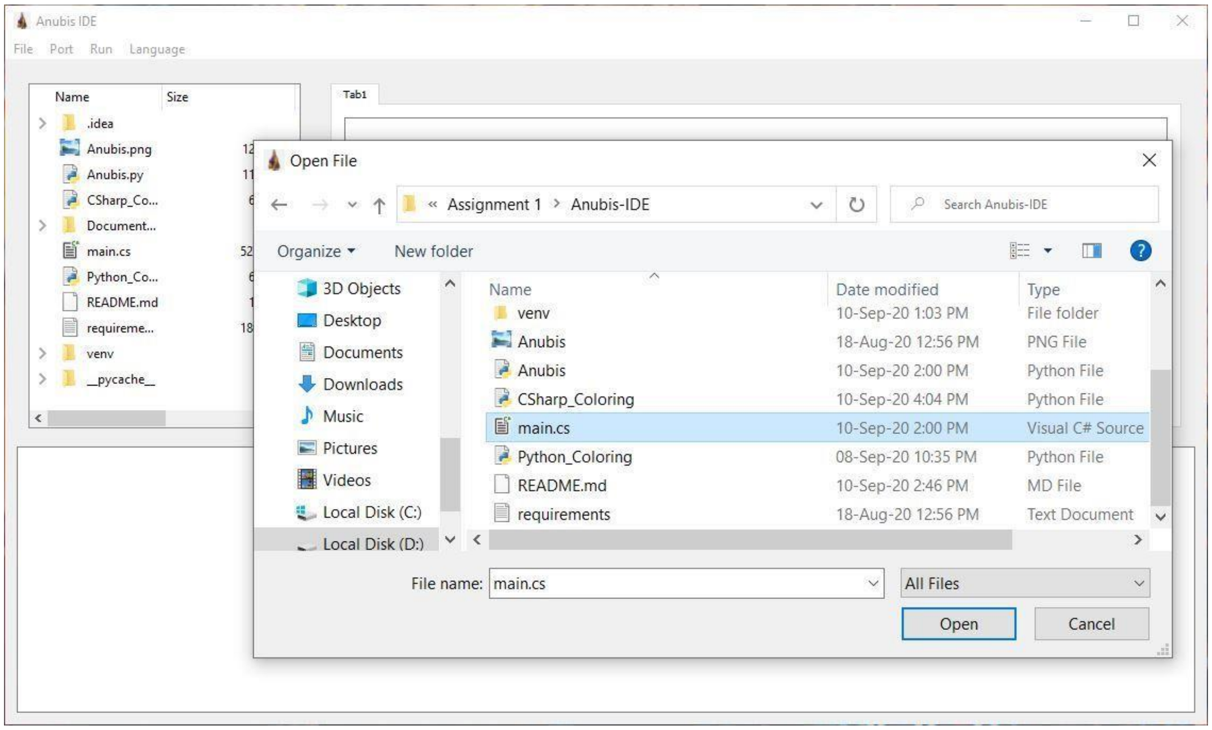
```
// String clone

using System;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            {
                string str = "Clone() Test";
                string clonedString = null;
                clonedString = (String)str.Clone();
                MessageBox.Show (clonedString);
            }
        }
    }
}
```

Syntax Analysis of Python string cloning code:



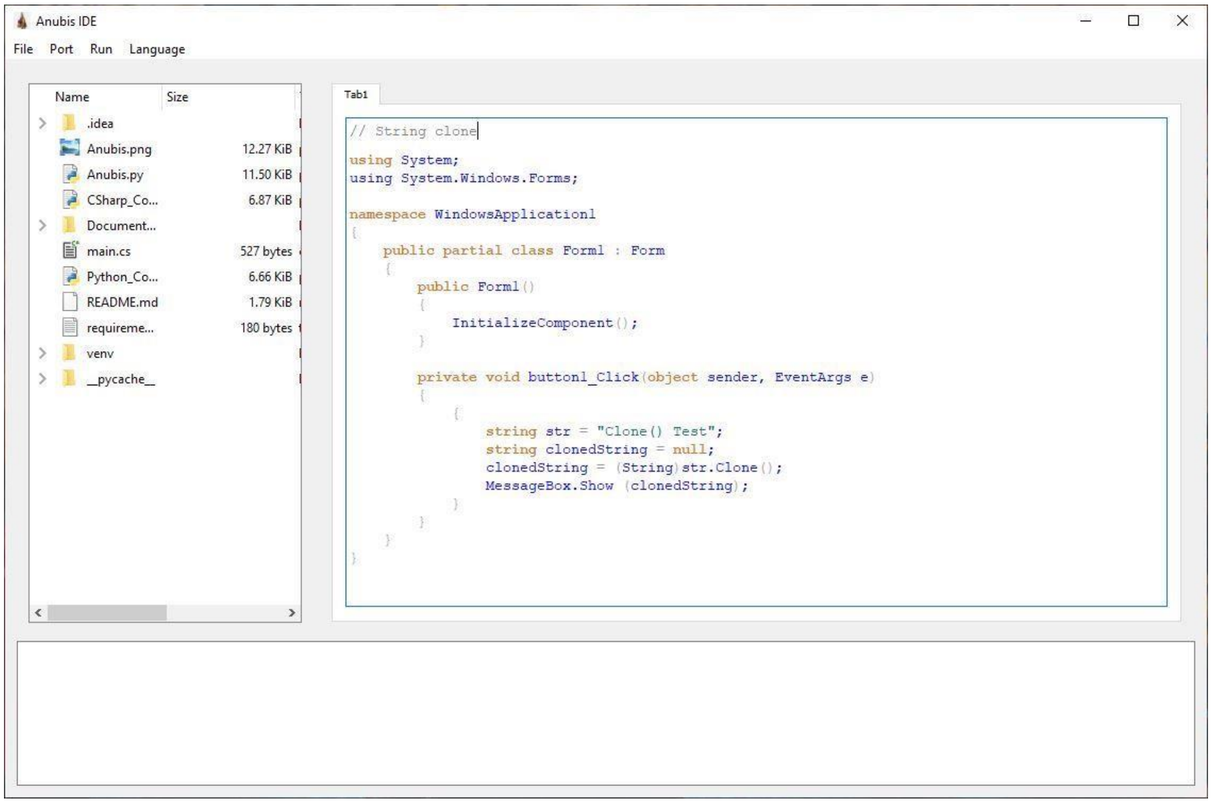
```
# Python Program to Copy a String

str1 = input("Please Enter Your Own String : ")

str2 = str1
str3 = str1[:]

print("The Final String : Str2  = ", str2)
print("The Final String : Str3  = = ", str3)
```

Opening & Editing File/Language Detection (New Feature):



Name	Date modified	Type
venv	10-Sep-20 1:03 PM	File folder
Anubis	18-Aug-20 12:56 PM	PNG File
Anubis	10-Sep-20 2:00 PM	Python File
CSharp_Coloring	10-Sep-20 4:04 PM	Python File
main.cs	10-Sep-20 2:00 PM	Visual C# Source
Python_Coloring	08-Sep-20 10:35 PM	Python File
README.md	10-Sep-20 2:46 PM	MD File
requirements	18-Aug-20 12:56 PM	Text Document

File name: main.cs
All Files



Name	Size
.idea	
Anubis.png	12.27 KiB
Anubis.py	11.50 KiB
CSharp_Co...	6.87 KiB
Document...	
main.cs	527 bytes
Python_Co...	6.66 KiB
README.md	1.79 KiB
requireme...	180 bytes
venv	
pycache	

```
// String clone

using System;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            {
                string str = "Clone() Test";
                string clonedString = null;
                clonedString = (String)str.Clone();
                MessageBox.Show (clonedString);
            }
        }
    }
}
```

Code

A. Anubis.py

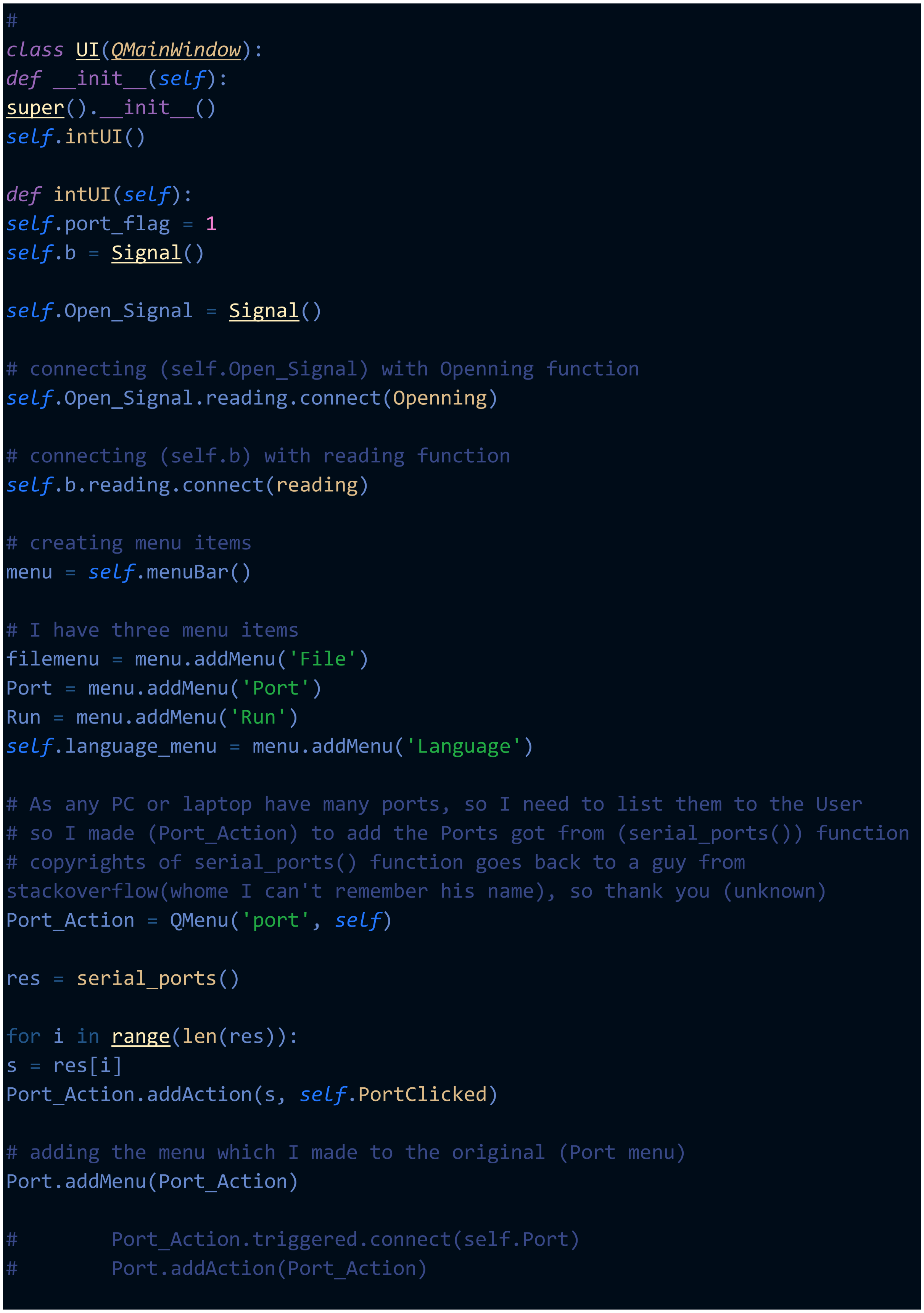
```
#############      author => Anubis Graduation Team        ############
#############      this project is part of my graduation project and it
intends to make a fully functioned IDE from scratch    ########
#############      I've borrowed a function (serial_ports()) from a guy in
stack overflow 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
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
to (self) in widget class)
text = QTextEdit
text2 = QTextEdit
language = "Python"

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

```python
# 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 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):
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
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')
self.language_menu = menu.addMenu('Language')

# 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)
```

```python
# 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 text
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_())
```

B. CSharp_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
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...
```

```python
        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. main.cs

```csharp
using System;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            {
                string str = "Clone() Test";
                string clonedString = null;
                clonedString = (String)str.Clone();
                MessageBox.Show (clonedString);
            }
        }
    }
}
```

D. 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'),
```

```python
        '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
```