Faculty of Engineering Ain Shams University

Credit Hours Engineering Programs

Computer Engineering and Software Systems

CSE 426 – Maintenance and Evolution

Assignment 2

Delivered to:

Dr. Ayman Bahaa

Made by:

Nagy Raouf 16P1021

Contents

List of figures

1. Introduction

This paper discusses an evolution of an open-source editor called **Anubis-IDE** that was made as a graduation project from the Faculty of Engineering Ain Shame University under the supervision of Prof. Dr. Ayman Bahaa, it was developed by Abanob Medhat and Ahmed Ashraf Mahmoud. The source code could be found on the is repo https://github.com/a1h2med/Anubis-IDE.

The goal of this tool was in the first place is to provide a simple environment where the users can write, edit, compile, and run micropython codes.

The tool was developed using Python and PYQT5 which is a library that allow using the Qt GUI framework from python and the Qt is written with C++ so the tool can have the speed of the C++.

In the past study a reverse engineering was made to the code in order to understand it and have the knowledge to evaluate it.

The main concern in this paper is to discusses the changes made to this system in order to give it the support of an additional language which is the C# so that the editor can support its format.

2. System requirement

2.1. Function requirement

F1: The tool should be able to open existing files (python code)

F2: The tool should be able apply edits to the opened file or a new created file

F3: The tool should be able to view the files aside to the coding tab

F4: The tool should be able to browse the files in the different directories

F5: The tool should be able to save the files after editing according to the used language

F6: The tool should be able to compile and run the snippets of code.

F7: The tool shall highlight the code in the coding tab according to the syntax of the code

F8: The tool should be able to view the port in order for the user to choose from it.

2.2. New function requirement

F9: The user is able to choose the language (python or C#)

F10: The tool should detect the language automatically of opened file according to its extension.

2.3. Non-functional requirement

1. The system must be written in python.
2. The system should be delivered in 30th of May 2021
3. The system must support different OSs
4. The tool must use the Qt which is made using C++ so the speed in the execution will be faster than using python only

3. Design

3.1. Enhancements made to the system

A new feature is added to the editor that is it can detect the language of a file from its extension weather it is C# or python and it also gives a support for the C# language and gives the proper highlighting for its coding.

In order to support the C# language a new dictionary with all the C# keyword was added to the tool and new rules that defines these keywords and operations were add also.

The users now have the ability to choose the language they wish to use weather it is C# or python and that can be made by choosing the desired language from the tool bar (The steps are shown in the screenshots below) so when writing the code that users now must provide the tool with the language, he/she is using so that the tool can respond to it and highlight the code according to it.

The user also can save the code is editing weather it was python code or C# (they are saved in the same directory of the IDE with the proper extension chosen by the user).

3.2. Use case diagram

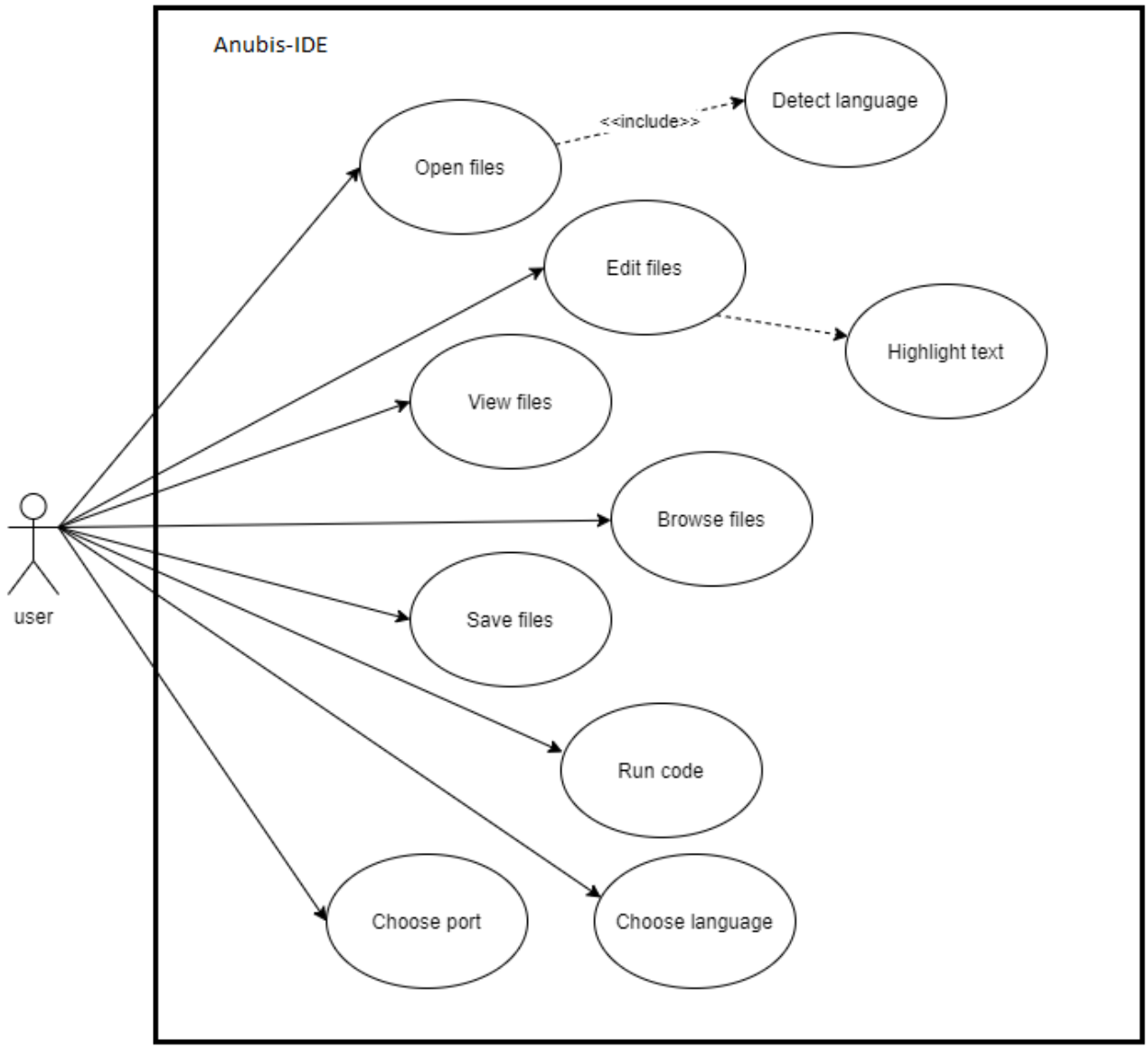


Figure 1: Use case diagram

3.3. Narrative Description of the use cases

Use Case #1 (Open files - U1)

Author	Nagy Raouf
Use case	Open files.
Goal in context	User opens an existing file in the ide
Precondition	User chooses open from the files in the tool bar.
Successful end condition	User opens the file in the ide.
Failed end condition	User cannot open the file in the ide.
Primary actor	User.
Secondary actor	IDE.
Trigger	The user clicks on the open button or short cut “Ctrl + o”
Main flow	1. The user selects file from the tool bar 2. The user selects open under the file in the tool bar 3. The user selects a supported file to open it. 4. The user presses open button.
Extensions	3.1 The user selects an unsupported file. 3.2 The system shows an error message

Use Case #2 (Edit file – U2)

Author	Nagy Raouf
Use case	Edit files.
Goal in context	User edits on a new or existing file
Precondition	User opened a new or existing file.
Successful end condition	User edits the file in the ide.
Failed end condition	User cannot edit the file in the ide.
Primary actor	User.
Secondary actor	IDE.
Trigger	The user opens a new or existing file.
Main flow	1. The user opens a new of existing file. 2. The user edits the file by writing a code to it.

Use Case #3 (View files – U3)

Author	Nagy Raouf
Use case	View files.
Goal in context	View files at the file tab
Precondition	User opens the ide
Successful end condition	User is able to view files.
Failed end condition	The files are not shown.
Primary actor	IDE.
Secondary actor	User.
Trigger	The user opens the ide.
Main flow	1. The user opens the ide. 2. The system loads the file in the same directory 3. The files are shown to the user in the display tab.
Extension	2.1 The file is not supported 2.2. The system shows error message.

Use Case #4 (Browse files – U4)

Author	Nagy Raouf
Use case	Browse files.
Goal in context	Browsing the files in the directories
Precondition	User chooses a folder to open.
Successful end condition	User can see the files and folders.
Failed end condition	User cannot browse the files.
Primary actor	User.
Trigger	The user presses on the folder or file to browse it.
Main flow	1. The user chooses a file or folder to browse it. 2. The IDE loads the file contents in the tab area for coding.

Use Case #5 (Save files – U5)

Author	Nagy Raouf
Use case	Save files
Goal in context	Save files that are edited.
Precondition	User has a file to save.
Successful end condition	User saves the file.
Failed end condition	User cannot save the file.
Primary actor	User.
Secondary actor	IDE.
Trigger	The user presses the save button.
Main flow	1. The user opens a new or existing file. 2. The user edits the file by writing a code to it. 3. The user presses the save button or ctrl+s 4. The user choose the programming language 5. The IDE saves the file under the name “main” with the proper extension (.cs for C# and .py for python)
Extension	4.1 The user did not choose a programing language 4.2 The system displays an error message

Use Case #6 (Run code – U6)

Author	Nagy Raouf
Use case	Run code
Goal in context	Compile and run snippets of code.
Precondition	User writes a snippet of code in the tab of coding.
Successful end condition	The ide compiles and run the code
Failed end condition	The code does not compile or run.
Primary actor	IDE.
Secondary actor	User.
Trigger	The user presses on the run button.
Main flow	1. The user loads or writes the code he/she wants to run. 2. The user chooses a port 3. The user chooses the programing language. 4. The user presses on the run button. 5. The system compile the code 6. The system runs the code and display the output
Extension	2.1. The users did not choose a port. 2.2. The system displays an error message 3.1 The user did not choose a language 3.2 The system shows error message 4.1. The code has syntax error. 4.1 The system displays an error message.

Use Case #7 (Highlight text – U7)

Author	Nagy Raouf
Use case	Highlight text
Goal in context	The code is highlighted according to its syntax of the programming language
Precondition	Snippets of code in the tab of coding and the user chooses the programming language he/she desired
Successful end condition	The code is highlighted according to its syntax
Failed end condition	The code is not highlighted
Primary actor	IDE.
Trigger	Snippets of code is written in the tab of coding and the user chooses the programming language he/she desired Or The user opens a code file
Main flow	1. The IDE capture the text in the code snippets written in the tab of coding 2. The system gets the selected language or detects the language from the extension of the file 3. The IDE detects the keywords in the text according to the syntax 4. The Keywords are highlighted according to syntax and display it.
Extension	2.1. There is no selected language or file does have a supported extension 2.2. The system doesn't highlight the code

Use Case #8 (Choose port – U8)

Author	Nagy Raouf
Use case	Choose port
Goal in context	The user chooses a port
Precondition	The ide is running
Successful end condition	A port is selected
Failed end condition	No port to select
Primary actor	User
Secondary actor	IDE.
Trigger	The user presses on the port button in the tool bar
Main flow	1. The user presses on the port button in the tool bar 2. The tool provide the user with a set of ports to choose from 3. The user choose the port he/she desire

Use Case #9 (Detect language – U9)

Author	Nagy Raouf
Use case	Detect language
Goal in context	The system detects the language from the file extension
Precondition	The ide is running
Successful end condition	The system detects the language from the file extension and use the appropriate highlighter
Failed end condition	The system does not detect the file extension
Primary actor	IDE
Trigger	The user opens an existing file
Main flow	1. The Ide read the file name 2. The system get the file extension 3. The system selects the highlighter according to this extension

Extension	2.1. The extension is not supported by the IDE 2.2. The system will not choose an highlighter

Use Case #10 (Choose language – U10)

Author	Nagy Raouf
Use case	Choose language
Goal in context	The user chooses a language
Precondition	The ide is running
Successful end condition	A language is selected
Failed end condition	No language to select
Primary actor	User
Secondary actor	IDE.
Trigger	The user presses on the port button in the tool bar
Main flow	1. The user presses on the language button in the tool bar 2. The tool provide the user with the supported languages (C# and python) to choose from 3. The user choose the language he/she desire

3.4. Sequence diagram

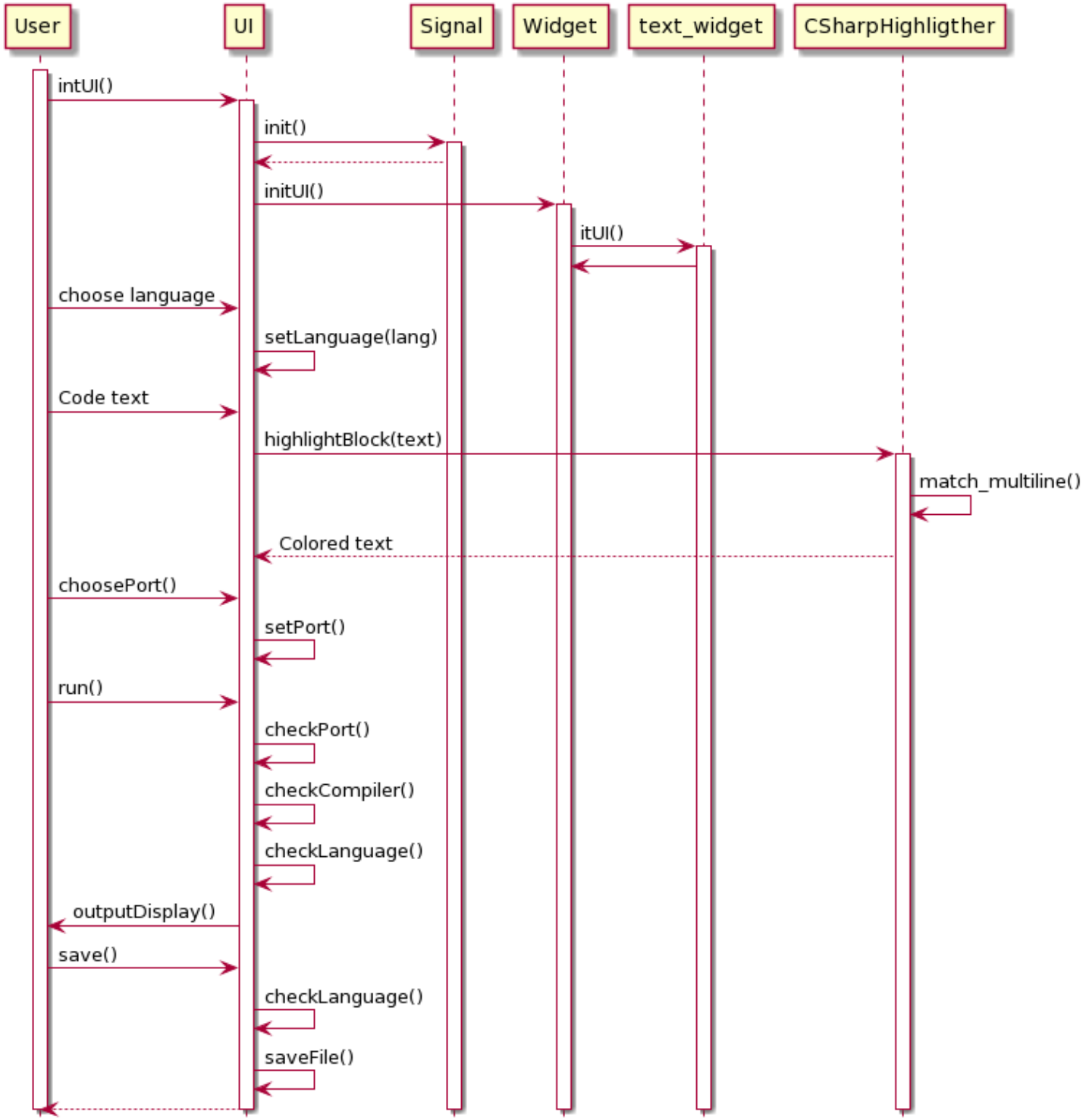


Figure 2: Sequence diagram for writing C# code

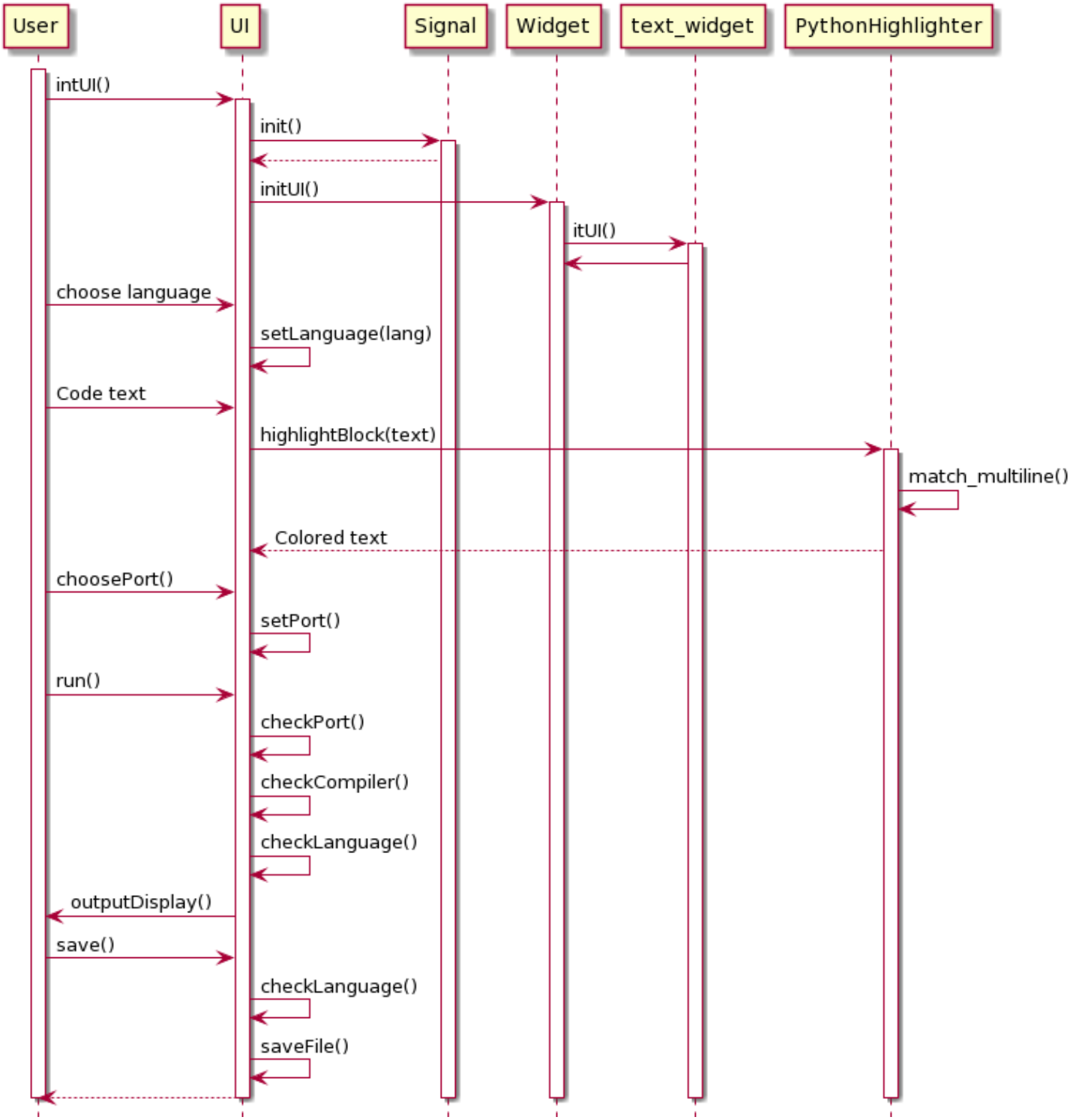


Figure 3: Sequence diagram for writing python code

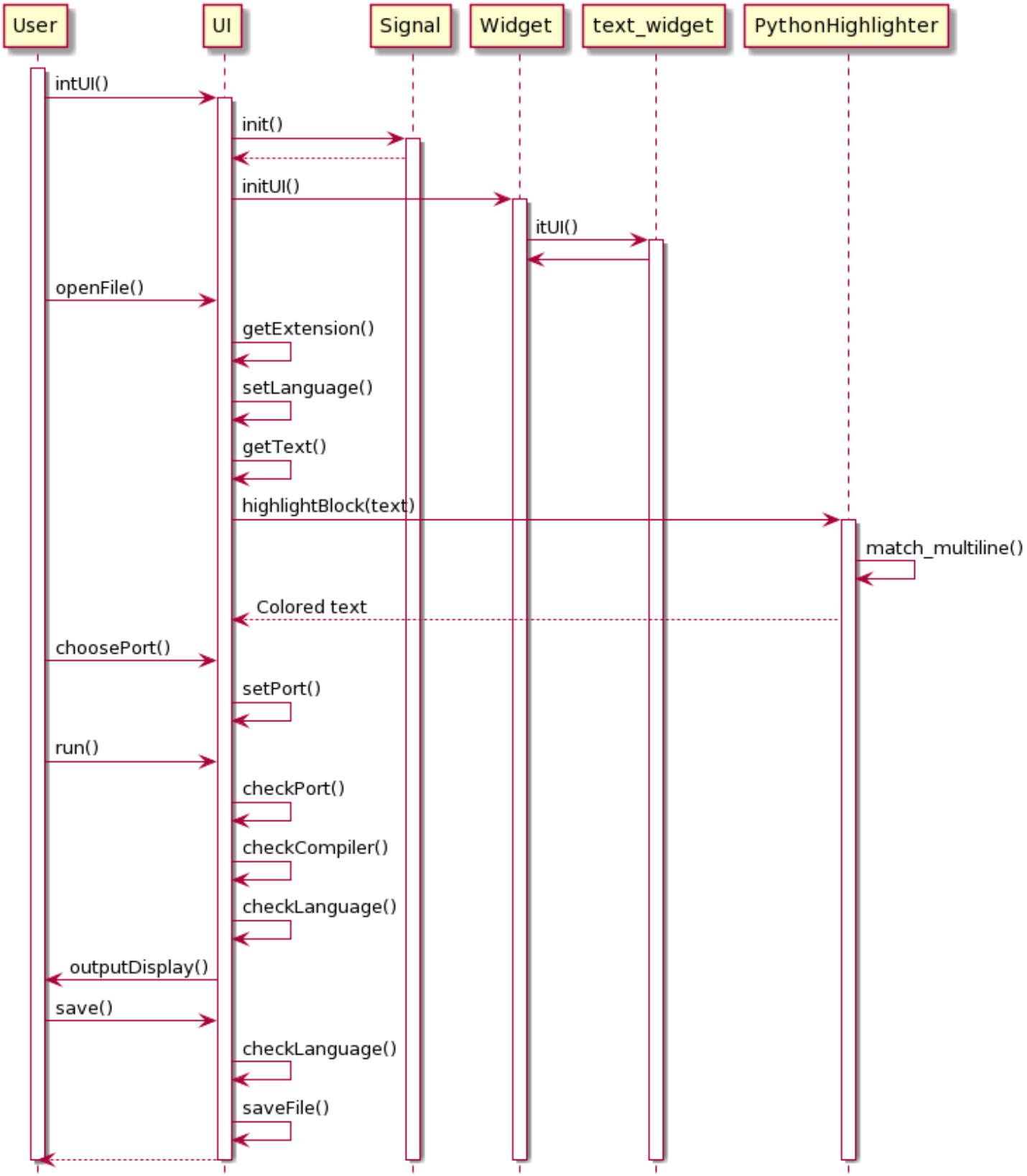


Figure 4: Open python file

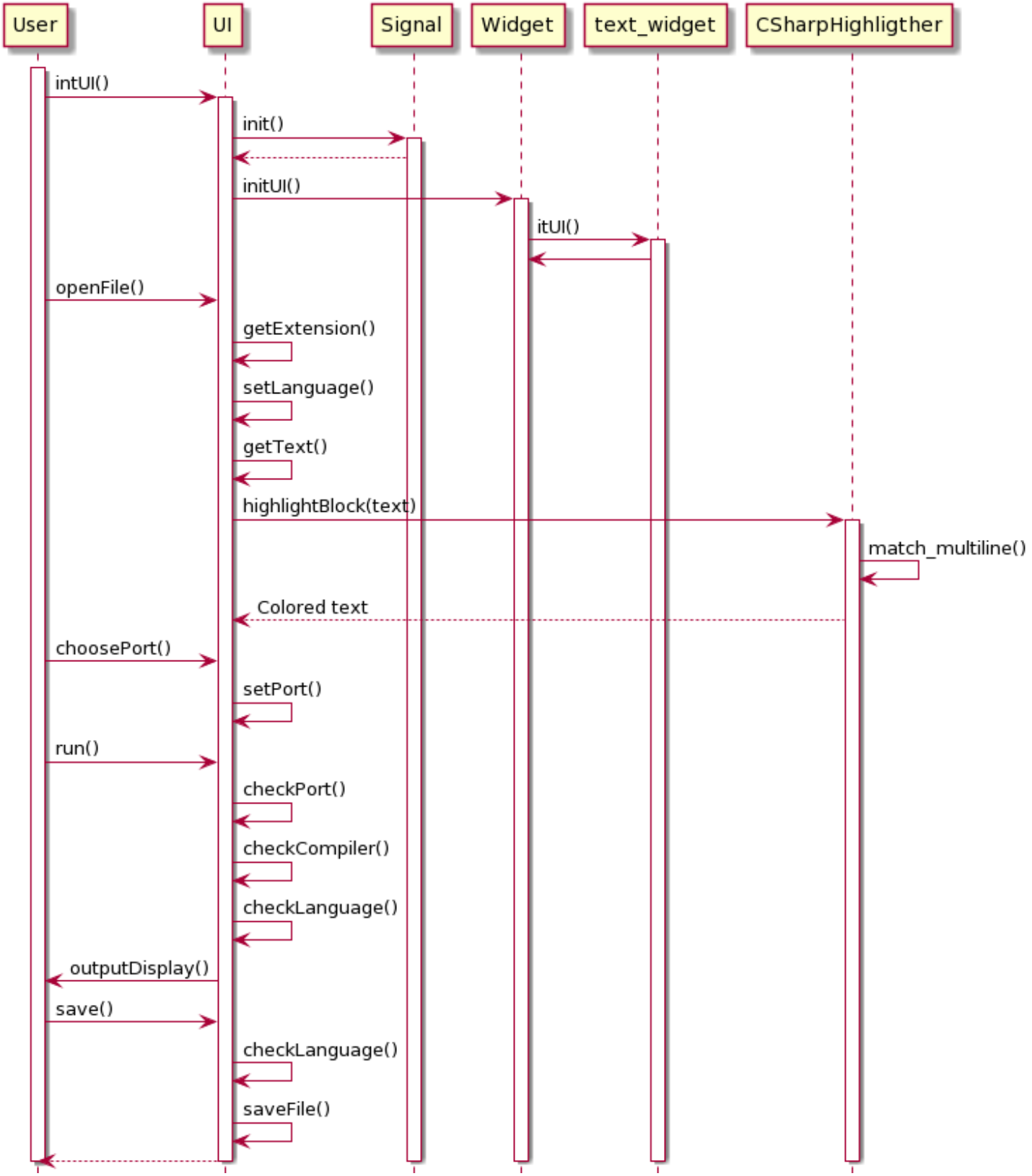
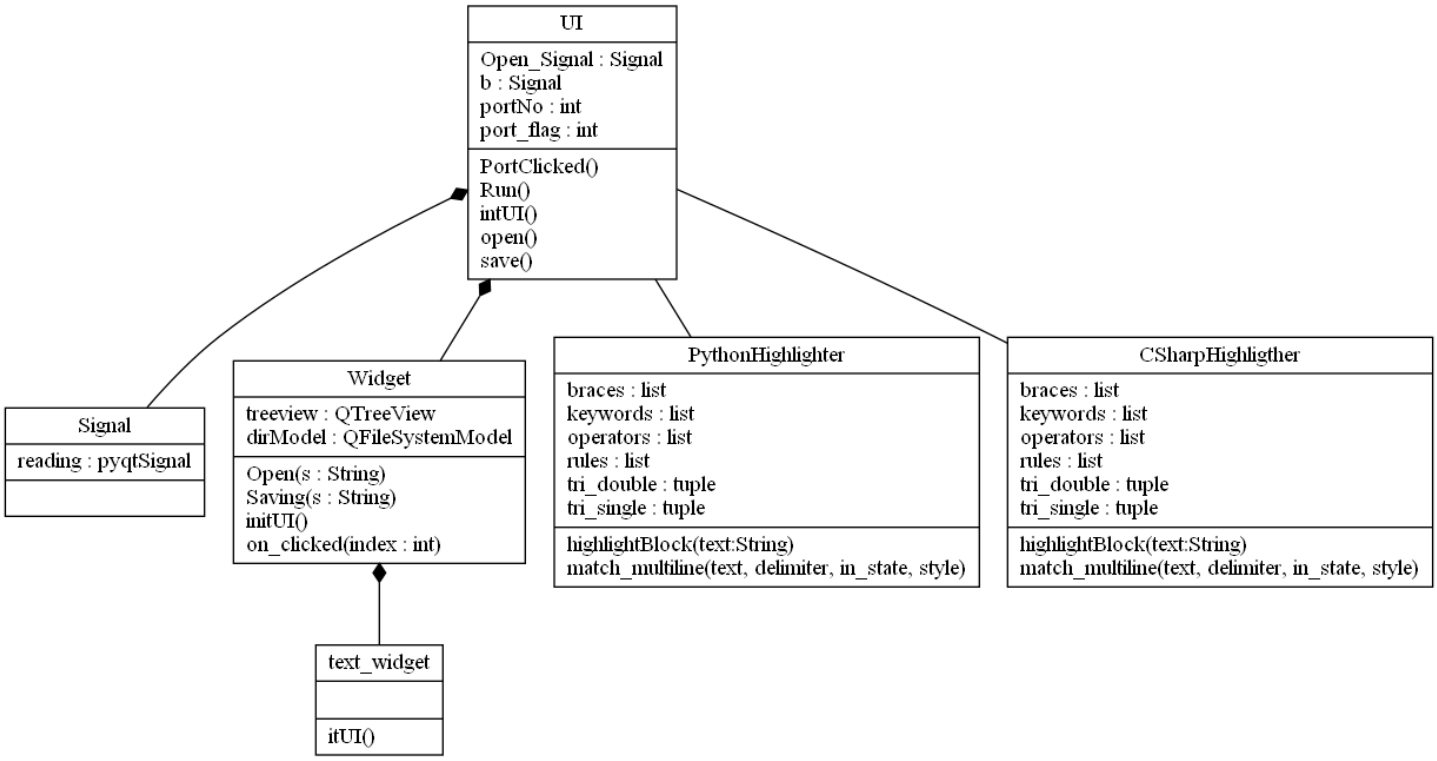


Figure 5: Open C# code

3.5. Class diagram



UI class:

This class is considered to be the main window class where the UI is initiated in the function intUI()
The save() is used to save code into a file
The open() is used to open a file and exhibits it to the user in the text editor
The PortClick() is used to get which port that the user selected
The Run() is used to make the functionality in the code by compiling and running it.

Widget class:

This class is responsible to make the tab in the ide where the text editor is found and it is done in the initUI()
The Open() and on_clicked() are used to define a new slot and take string to set the text editor with this string.
The Saving() is to define a new slot to save the text in the text editor.
The treeview is used to show directory included files and dirModel is used to make a file system variable

Signal class:

This class is responsible for initiating a signal which will take string as the input in the attribute called reading.

text_widget class:

This class is responsible to connect the Qtab with necessary layouts.

PythonHighlighter and CSharpHighligther

These classes are responsible for highlighting the code according to the syntax, the keywords list has the keywords of python in **PythonHighlighter** and keywords of C# in **CSharpHighligther**, the operator list has the operators as (+, -, *, /)
The highlightBlock() is used to apply the syntax highlighting to the given block of text.
The match_multiline() do the highlighting of multi-line strings and the delimiter is for triple-single-quotes or triple-double-quotes, and the in state is a unique integer to represent the state changing inside the strings.

2. Code

Anubis.py

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


import sys
import glob
import serial

import CSharp_Coloring
import Python_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 w
idget class)
text = QTextEdit
text2 = QTextEdit
# There is no language by default
global language
language = ""
#
#
#
#
############ 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 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 (whi
ch 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 whic
h 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 win
dow
        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):
        print(language)
        if language == "python":
           with open('main.py', 'w') as f:
               textToSave = text.toPlainText()
               f.write(textToSave)
        elif language == "C#":
           with open('main.cs', 'w') as f:
               textToSave = text.toPlainText()
               f.write(textToSave)
        else:
           text2.append("Please, Choose a language.")


    # 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.pythonEditor(self.ui)
        elif fileExtension == "cs":
            UI.csharpEditor(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 m
ade this function in between for futuer use
```

```
# All what it do is to take the (input string) and establish a connection with the widget c
lass, 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')
        languageMenu = 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)

        # 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)


        pythonChoice = QAction('Python', self)
        pythonChoice.triggered.connect(self.pythonEditor)
        csharpChoice = QAction('C#', self)
        csharpChoice.triggered.connect(self.csharpEditor)
        languageMenu.addAction(pythonChoice)
        languageMenu.addAction(csharpChoice)

        # 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 language == "":
            text2.append("Choose a language")

        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")

    def pythonEditor(self):
        global language
        language = "python"
        Python_Coloring.PythonHighlighter(text)

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

    # 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')
        file_extension = file_name[0].split(".")[1]

        if file_name[0]:
            f = open(file_name[0],'r')
            with f:
                if file_extension == "py":
                    data = f.read()
                    self.Open_Signal.reading.emit(data)
                    self.pythonEditor();
                elif file_extension == "cs":
                    data = f.read()
                    self.Open_Signal.reading.emit(data)
                    self.csharpEditor()

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

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

CSharp_Coloring.py

```
import sys
from typing import Literal
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('blue'),
      'operator': format('red'),
       'brace': format('darkGray'),
       'class': format('blue'),
       'classID': format('black', 'bold italic'),
       'string': format('magenta'),
       'string2': format('darkMagenta'),
       'comment': format('darkGreen', 'italic'),
       'numbers': format('purple'),
        'logicalOperators': format('green'),
        'literalKeywords':format('lightBlue'),
        'accessKeywords': format('lightBlue'),
        'typeKeywords':format('blue')


   }

class CSharpHighlighter(QSyntaxHighlighter):
    """Syntax highlighter for the C Sharp language.
    """
```

```
    #CSharp keywords
    keywords =[

    #Modifier Keywords
        'abstract', 'async', 'const', 'event',
        'extern', 'new', 'override', 'partial',
        'readonly', 'sealed', 'static', 'unsafe',
        'virtual', 'volatile',

    #Access Modifier Keywords
        'public', 'private', 'protected', 'internal',

    #Statement Keywords
        'if', 'else', 'switch', 'case', 'do', 'for',
        'foreach', 'in', 'while', 'break', 'continue',
        'default', 'goto', 'return', 'yield', 'throw',
        'try', 'catch', 'finally', 'checked', 'unchecked',
        'fixed', 'lock',

    #Namespace Keywords
        'using', '. operator' , ':: operator', 'extern alias',

    #Operator Keywords
        'as', 'await', 'is', 'new', 'sizeof', 'typeof',
        'stackalloc', 'checked', 'unchecked',

    #Contextual Keywords
        'add', 'var', 'dynamic', 'global', 'set', 'value',

    #Query Keywords
        'from', 'where', 'select', 'group', 'into', 'orderby',
        'join', 'let', 'in', 'on', 'equals', 'by', 'ascending', 'descending'
    ]

    literalKeywords = [
        'null', 'false', 'true', 'value', 'void'
    ]

    typeKeywords = [
        'bool', 'byte', 'char', 'class', 'decimal',
         'double', 'enum', 'float', 'int', 'long',
         'sbyte', 'short', 'string', 'struct', 'uint',
         'ulong', 'ushort'
    ]

    accessKeywords = [
        'base','this'
    ]

    # CSharp operators
```

```
    operators = [
        '=',
        # logical
        '!', '?', ':',
        # Comparison
        '==', '!=', '<', '<=', '>', '>=',
        # Arithmetic
        '\+', '-', '\*', '/', '\%', '\+\+', '--',
        # Assignment
        '\+=', '-=', '\*=', '/=', '\%=', '<<=', '>>=', '\&=', '\^=', '\|=',
        # Bitwise
        '\^', '\|', '\&', '\~', '>>', '<<',
    ]

    # Logical Operators
    logicalOperators = [
        '&&','\|\|', '!','<<','>>'
    ]

    # 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]
        rules += [(r'%s' % b, 0, STYLES['logicalOperators'])
                  for b in CSharpHighlighter.logicalOperators]

        rules += [(r'\b%s\b' % w, 0, STYLES['literalKeywords'])
                  for w in CSharpHighlighter.literalKeywords]
        rules += [(r'\b%s\b' % w, 0, STYLES['accessKeywords'])
                  for w in CSharpHighlighter.accessKeywords]
        rules += [(r'\b%s\b' % w, 0, STYLES['typeKeywords'])
```

```
                for w in CSharpHighlighter.typeKeywords]


        # 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']),

            # Class
            (r'\bClass\b',0,STYLES['class']),

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

        # 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
```

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
```

3. Screenshots

1. Main screen of the app and add to it the “Language” in the tool bar to choose the language from it.

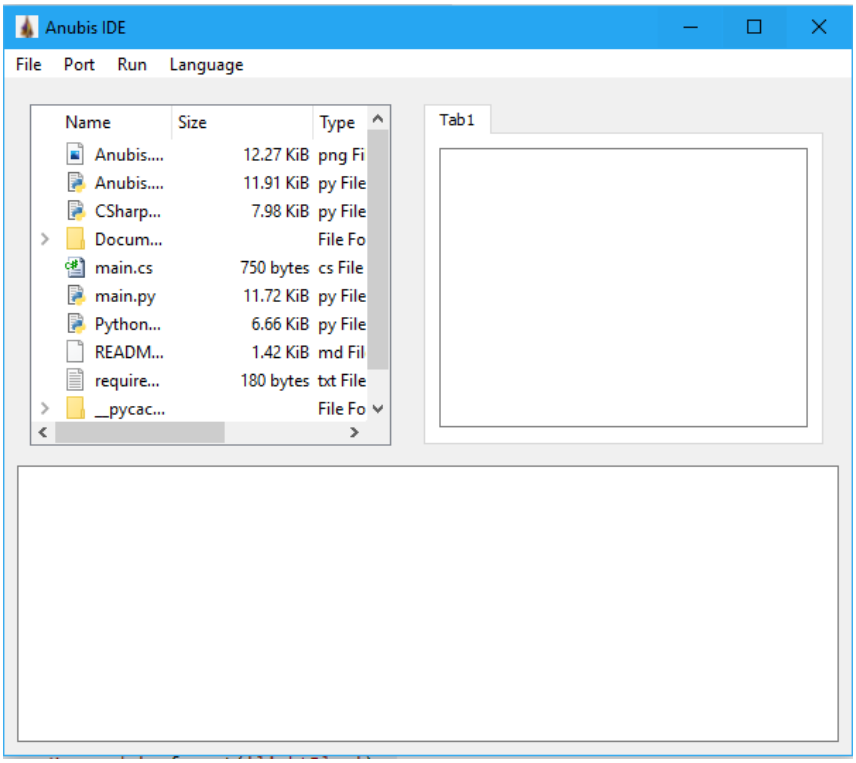


Figure 6: Main screen

2. Write C# code in the tab window without specifying the language.

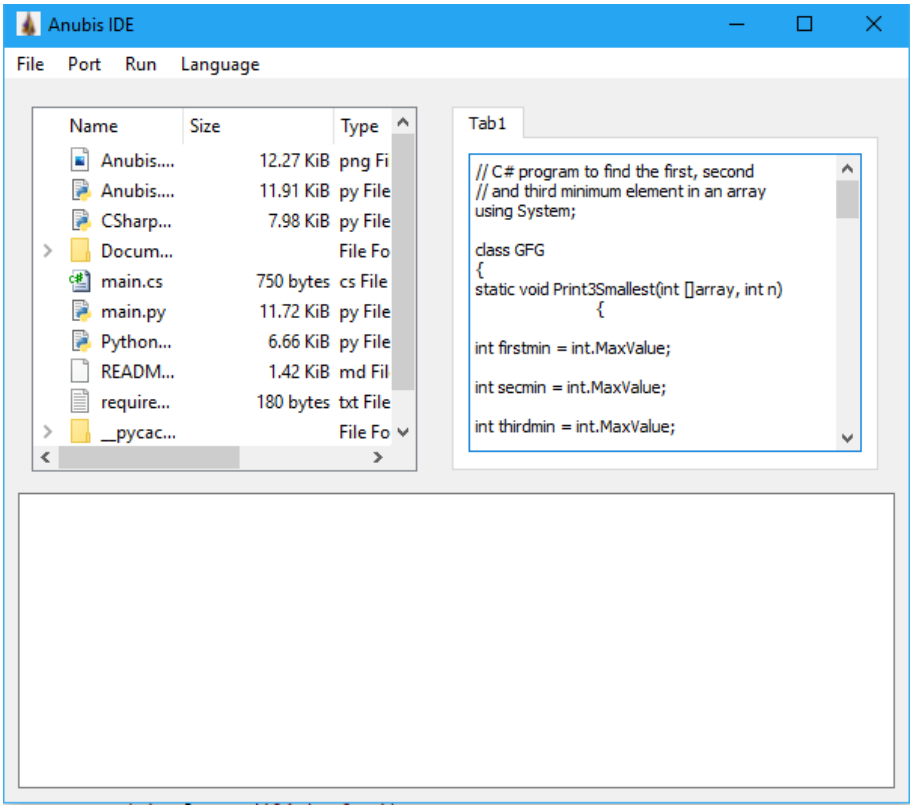


Figure 7: Writing C# code

3. Specifying the language to C# and the editor will automatically color the code according to the syntax.

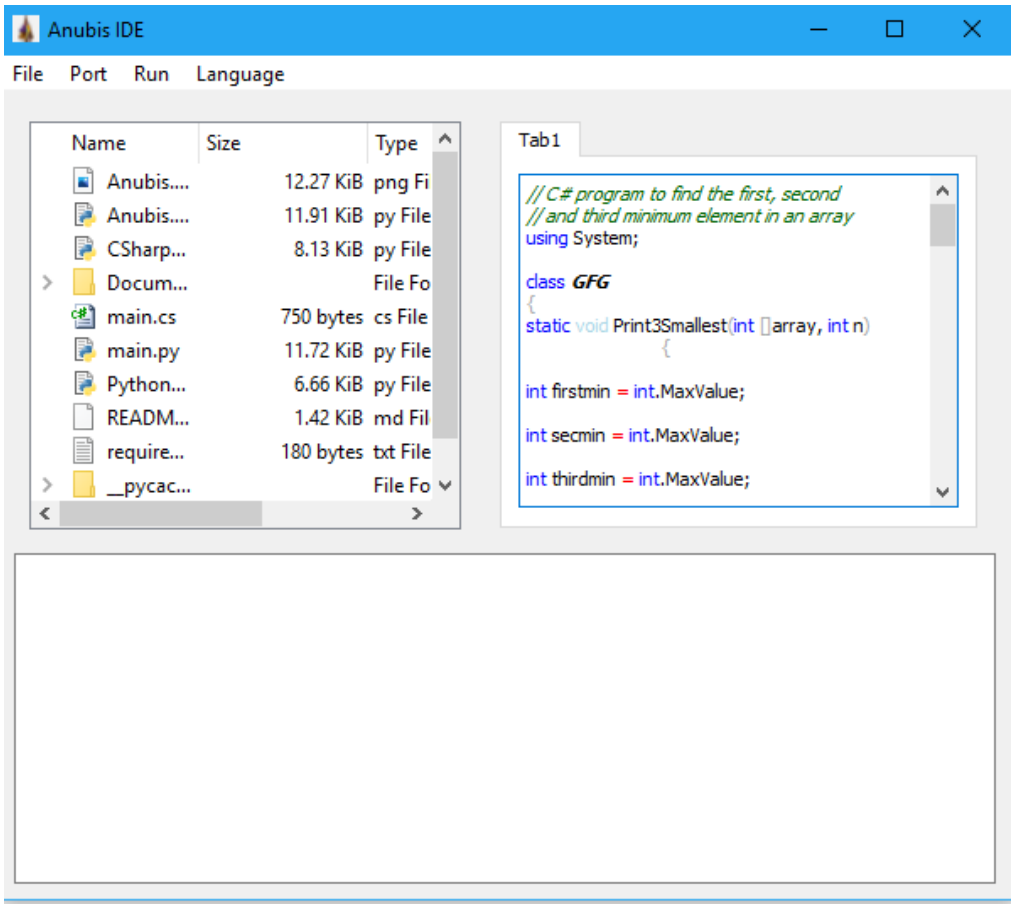


Figure 8: Choose C# language

4. Choosing a file with .cs as an extension

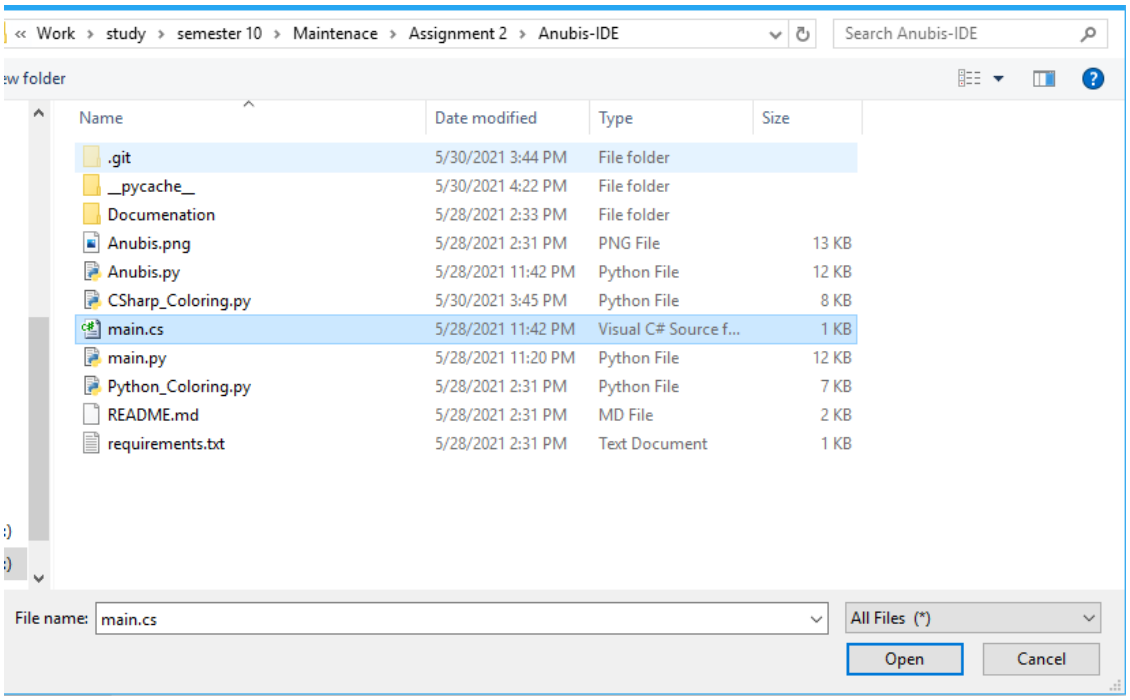


Figure 9: Choose CSharp file

5. Display the C# file with the coloring according to syntax and saving the file will generate a main.cs file.

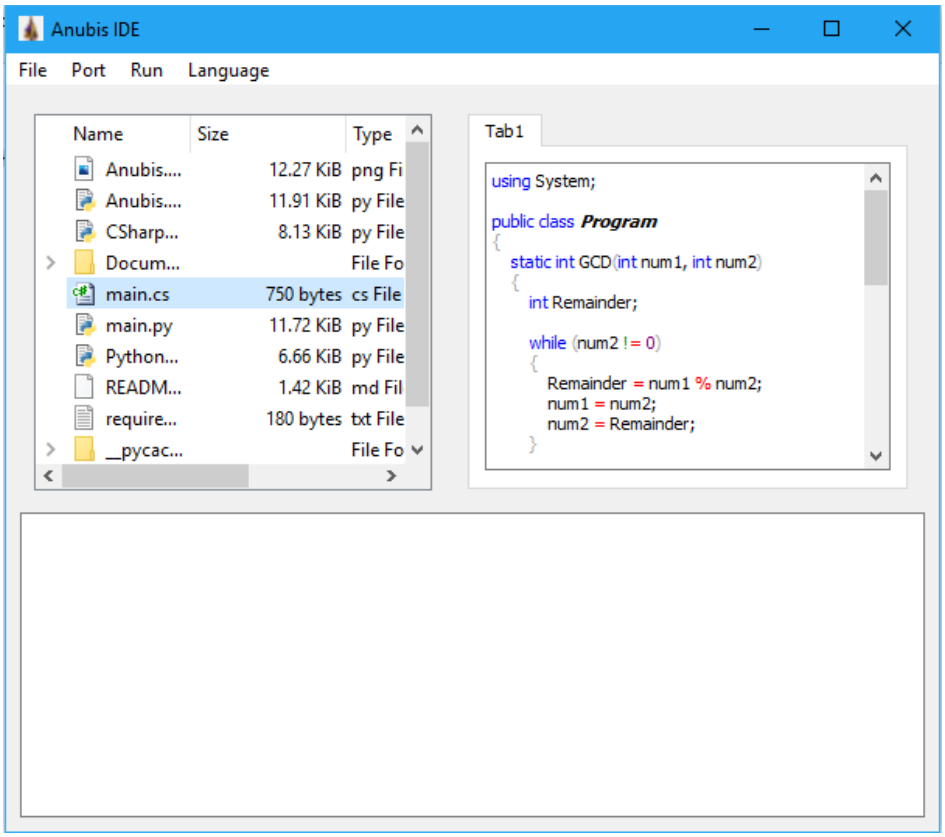


Figure 10: Display C# code

6. Trying to run code without specifying the language will display an error

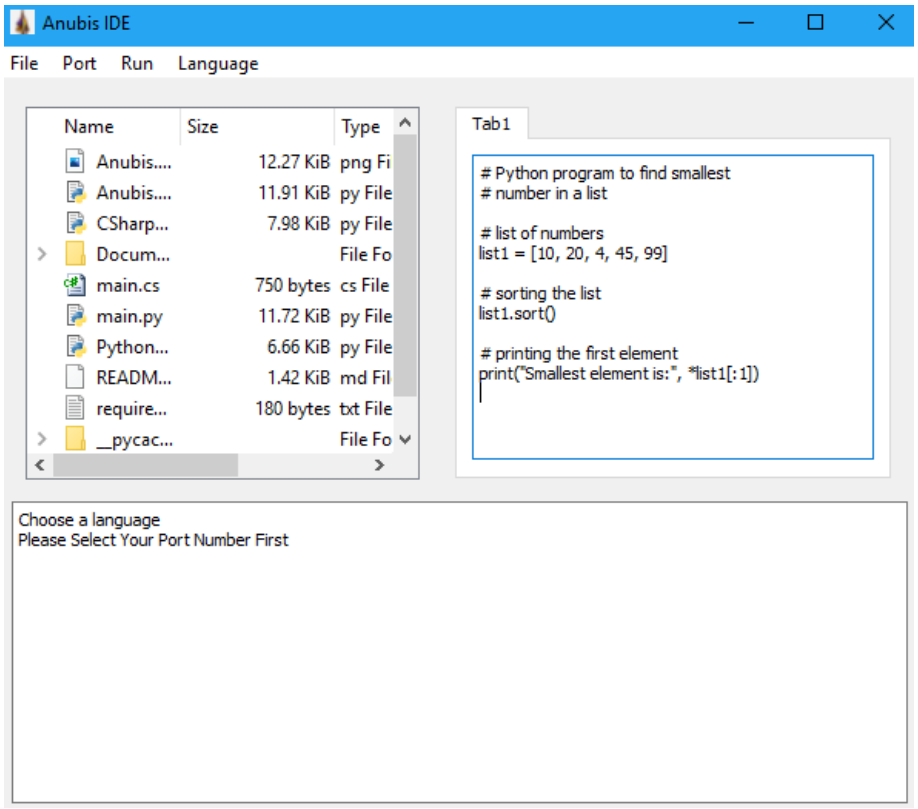


Figure 11: Display an error

7. Write python code in the tab window without specifying the language.

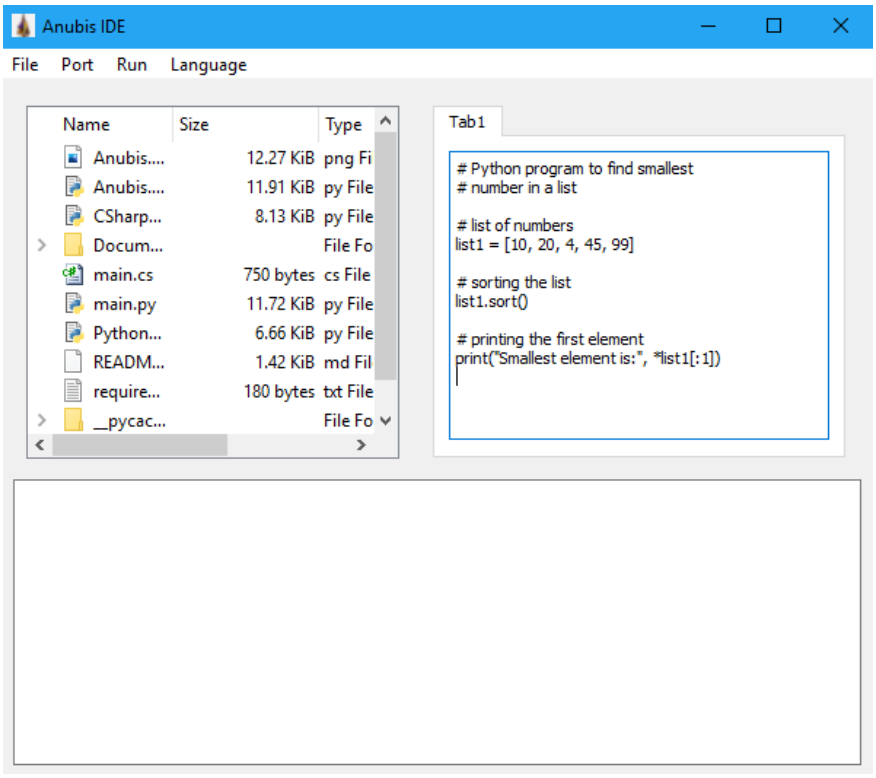


Figure 12: Writing python code

8. Specifying the language to python and the editor will automatically color the code according to the syntax.

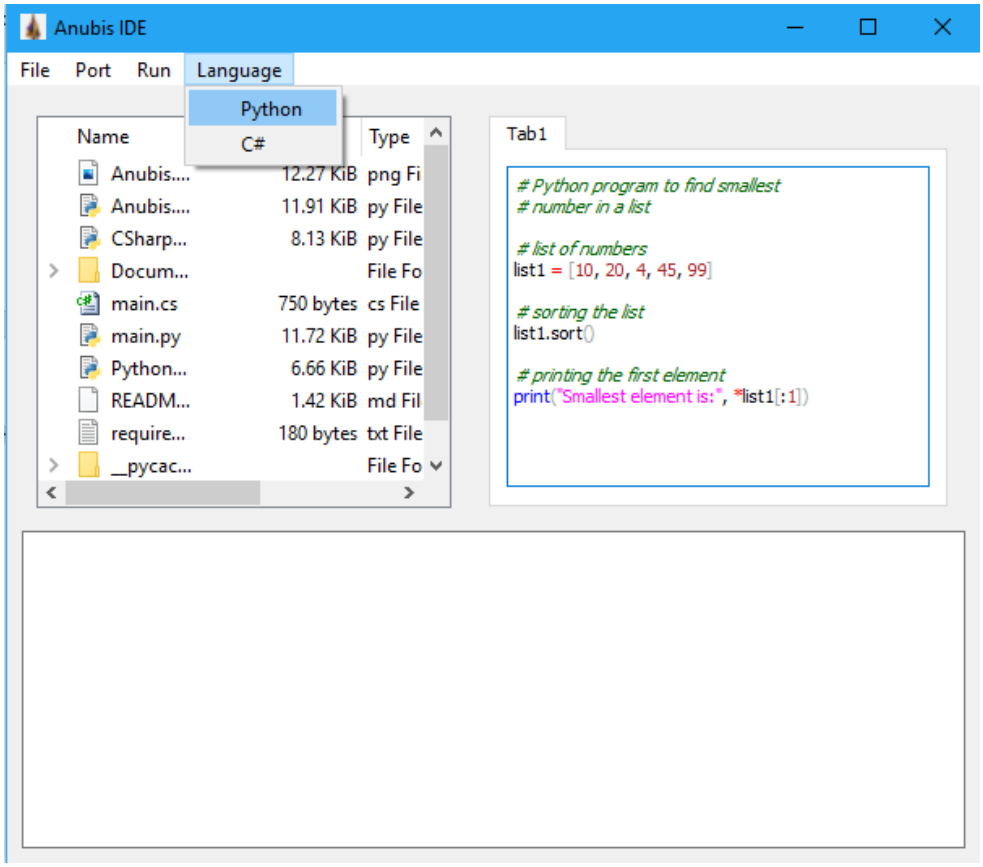


Figure 13: Choose python language

9. Choosing a file with .py as an extension

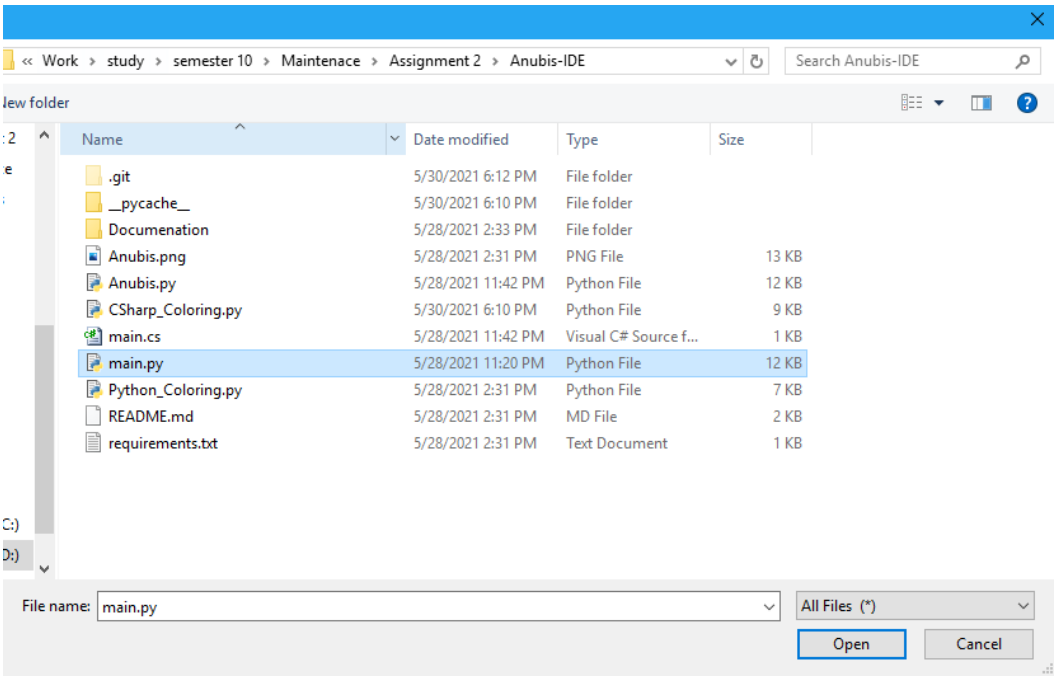


Figure 14: Choosing .py file

10. Display the python file with the coloring according to syntax and saving the file will generate a main.py file.

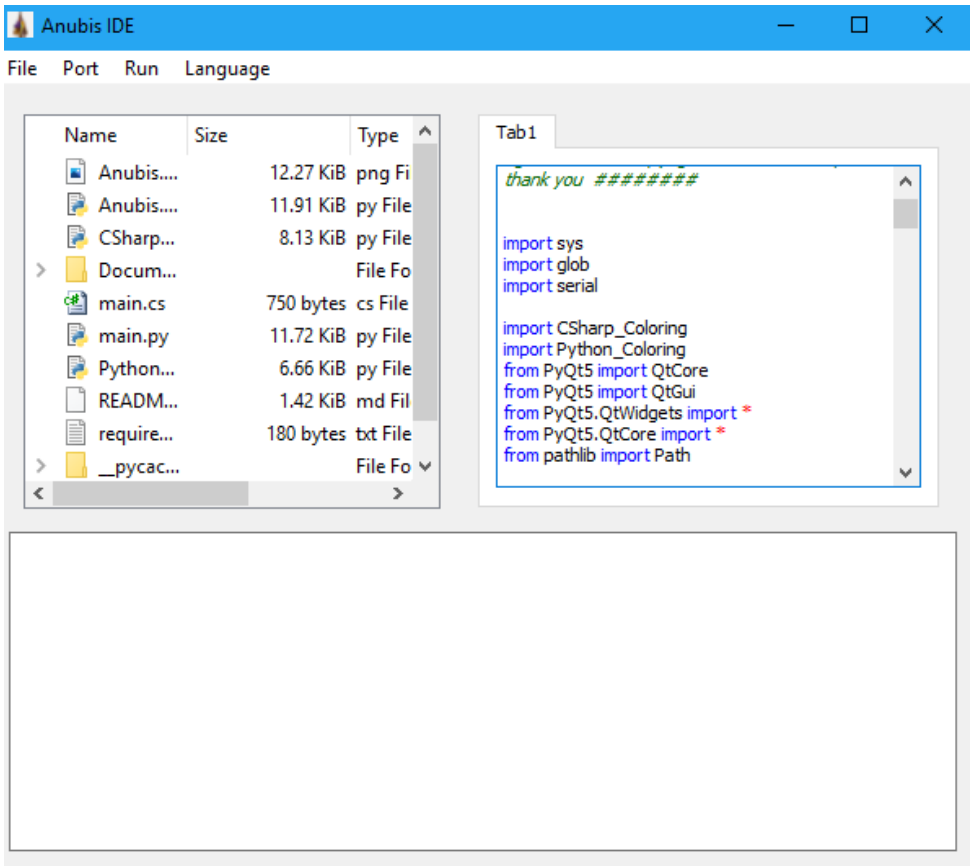


Figure 15: Displaying python saved file