Student Name: Kirolos Raouf Atteya Saad

Computer Engineering and Software systems

Project Study

UEL Module Title: Software Engineering 2

Module Code: EG7642

ASU Course Title: Software Maintenance and Evolution
Course Code: CSE 426

Lecturer : Prof: Ayman Bahaa

UELID:U1927300
ASUID:16P8045

University of East London

GitHub Link:

https://github.com/kirolos997/AnubisVersion2

Part 1

- Study About the Project
- Issues Solved
- Trial Run and Screenshots

Note: New Updates or modified Parts in the reports are being wrapped inside a blue rectangle showing difference from old PDF version

1- A study about the project

- **Concept and benefits**: this is a part of graduation project introduced by a team of Ain shams university written in Python language for creating an open-source editor that enable the user to write, edit, compile and run micro python codes (micro python is a code that runs on small embedded development boards mainly microcontrollers). Being open-source code is an advantage for allowing different developers to use this code, make some evolutions and maintain besides the provided functionality by this team.

- **New added features in this version**:
  1- **A fast executed for python code**: in this feature, the editor user will enter a code for a single function that would be automatically wrapped inside a program that has a main function that will call the function. The user would be asked to provide a list of parameters to be passed from the main function to the called function.
  2- **Support for C# format**: the editor automatically recognizes which format to use based on the file extension selected/opened and display it.
  3- **Fixing Issues**: all old issues discussed before in last version are fixed in this new version and we will big into these details later in this report (comment, restructure, naming,…)
  4- **Save As and Save:** fixing issues in save button and introduce new feature as saveAs to support different ways (format) to save a file

- **Repository files:**
  - Anubis.py: the file in which the main core project code exists having:
    - 4 classes namely (signal, widget, UI, TextWidget)
    - 5 functions defined in the global scope namely (serial_ports, reading, opening, automaticFileExtensionSetter, main).
  - Python_coloring.py: the file in which the developer used to define his style, colorings, fonts, formats and syntax highlighter for the python language. It consists of:
    - 1 class namely (PythonHighlighter)
    - a function in global scope named (format)

- 2 styles namely (STYLES, STYLES2)

- Requirements.txt: the needed dependencies and libraries for the environment to work correctly.
- README.md: it consists of:
  - objectives: Goal and brief description.
  - Requirements: the needed requirements written in the requirements.txt file as for example PYQT5.
  - A guide: how to clone and run the project.
- Anubis.img: a simple image for the project.
- CSharpFormat.py: The file in which the first new added feature of supporting the C# format is being implemented. It consists of:
  - 1 class namely (CSharpHighlighter)
  - 1 function namely (highLightBlock)
- CSharpRegex.py: The file in which the regular expression for the language C# have been implemented to support the language and used in CSharpFormat.py and it consists of:
  - A set of regex Rules defined based on C# language
- CSharpStyleAndColors.py: The file in which the colors, styles, format for the C# regular expression. It consists of:
  - 1 function namely (Format)
  - A set of style and format for each regex used by C# language
- ReservedCSharpIdentifiers.py: The file in which the reserved keywords, operators, braces used by C# language. It consists of:
  - A set of Keywords of C#
  - A set of operators of C#
  - A set of braces of C#

- FastExecution.py: The Template in which the user will enter his code for being wrapped and tested and it includes the second new added feature

- **Brief description about each main function and main class:**
  - Anubis.py file:
    1- Signal class: creating a new signal for communication
    2- UI class: holds the main functionality of the project as for example creating signal connection, creating the menu items, shortcuts, creating a widget (main window for the application) instance, run the program, save, save As, Fast Execution Mode and open files buttons. Also the algorithm for implementing the fast execution method. Functions used (InitUI, RunMenuButton, PortClickedMenuButton, saveMenuButton, saveAsMenuButton, openMenuButton, fastExecutionAlgorithm)
    3- TextWidget class: some usage from the Python_coloring.py file to color the code written in the editor tab of the program in the widget class. Function used (itUI)
    4- Widget class: the main application window implementation that appears, opening and saving functionality for a file. Functions used (initUI, saving, open, onClickedFromLeftTab)
    5- SerialPorts () function: listing available Com Ports with python
    6- Reading () and opening functions: save and open, defining a new slot.
    7- automaticFileExtensionSetter function: This new added feature checks the file extension based on the chosen file and set the suitable format for the language used by the file. Also, it prints on the screen the type of file the user chooses
    8- main() function: The wrapper or the Main section for the fast execution mode .the fast execution calls given function here and asks the user to specify the parameters.

- Python_coloring.py file:
    1- Format () function: define the format used in the text editor
    2- 2 styles: syntax styles that can be shared by all languages
    3- PythonHighlighter class: main implementation of the coloring styles used in the main editor window. Functions used (highlightBlocks, match_multiline, ___init___).

- CSharpFormat.py:
    1- CSharpHighligher class: Accept the input passed from Anubis.py file, uses the defined regex, detect them in passed data, color and style them. Function used highlightBlock.
    2- Note: this file I divide its functionalities on separates files and then I import them. The goal is to reduce dependencies and better maintenance and evolution processes
        - File Used: CSharpRegex.py, CSharpStyleAndColors.py, ReservedCSharpIdentifiers.py.

- **Program Metrics**:

I am going to one metrics (Lines of code "LOC") to evaluate the performance of the code mainly for the files showing functionality which is Anubis.py and Python_Coloring.py.

1- I used a free tool named CLOC to count the number of lines **in CSharpFormat.py** file.

```
[Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$ cloc CSharpFormat.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.86  T=0.01 s (77.1 files/s, 2930.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1              7             13             18
-------------------------------------------------------------------------------
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$
```

2- I used a free tool named CLOC to count the number of lines in **Python_Coloring.py** file.

```
github.com/AlDanial/cloc v 1.86  T=0.01 s (70.2 files/s, 13341.8 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1             29             48            113
-------------------------------------------------------------------------------
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$
```

3- I used a free tool named CLOC to count the number of lines in **CSharpRegex.py**

```
[Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$ cloc CSharpRegex.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.86  T=0.01 s (76.4 files/s, 4125.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1             14             16             24
-------------------------------------------------------------------------------
```

4- I used a free tool named CLOC to count the number of lines in **CSharpStyleAndColors.py**

```
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$ cloc CSharpStyleAndColors.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.86  T=0.01 s (87.8 files/s, 5880.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1             15             27             25
-------------------------------------------------------------------------------
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$
```

5- I used a free tool named CLOC to count the number of lines in **FastExecution.py.**

```
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$ cloc FastExecution.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.86  T=0.01 s (97.4 files/s, 1558.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1              6              8              2
-------------------------------------------------------------------------------
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$
```

6- I used a free tool named CLOC to count the number of lines in **ReservedCSharpIdentifiers.py.**

```
-------------------------------------------------------------------------------
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$ cloc ReservedCSharpIdentifiers.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.86  T=0.01 s (86.7 files/s, 5898.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1              3             27             38
-------------------------------------------------------------------------------
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$
```

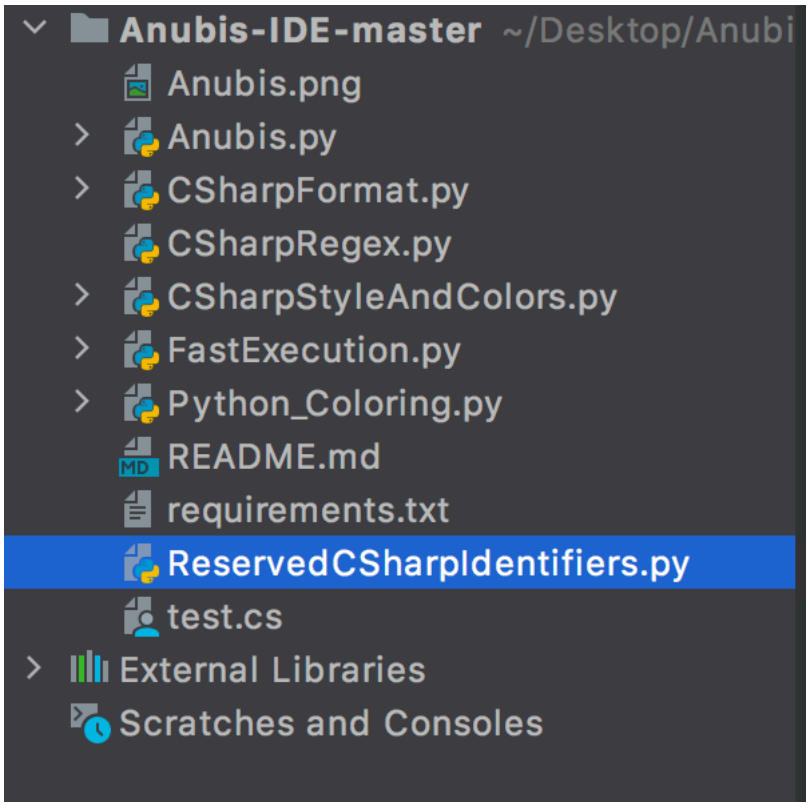
7- I used a free tool named CLOC to count the number of lines in **Anbuis.py.**

```
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$ cloc Anubis.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.86  T=0.02 s (55.5 files/s, 24603.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1             75            135            233
-------------------------------------------------------------------------------
Kirolos-MacBook-Pro:Anubis-IDE-master KIROLOSSAAD$
```

2- Issues in the project That were Solved in this Version

1- Writing the **whole project into one single file** named Anubis.py **without dividing the project into separate small files** where each file addressing a single class and **regroup them into a single starting point file**.

- Solution: The project new added features were divided among separates files to facilitate the maintenance and the execution of the process



2- **Bad naming convention** for **variables name, imports and class Names** leading to a very bad condition in case another teams needs to maintain this code or adding new features.

- Solution: The whole project files were scanned and I change the name of same variables, functions to give a better insight about the meaning

```
# It focus on the file extension in orde
def onClickedFromLeftTab(self, index):
```

```
@QtCore.pyqtSlot()
def PortClickedMenuButton(self):
    action = self.sender()
    self.portNo = action.text()
    self.portFlag = 0

# This function to saveMenuButton the code into a file
def saveMenuButton(self):
    self.newSignalConnection.reading.emit("name")

# New added functionality for saveAs files to support both python and CSharp
def saveAsMenuButton(self):
    name = QFileDialog.getSaveFileName(self, 'Save File')
    if name[0] !='' and name[0] !='FastExecution.py':
        with open(name[0], 'w') as file:
            TEXTData = text.toPlainText()
            file.write(TEXTData)
            file.close()

# This function to openMenuButton a file and exhibits it to the user in a text editor
def openMenuButton(self):
    fileName = QFileDialog.getOpenFileName(self,'Open File','/home')
    automaticFileExtensionSetter(fileName[0])
```

3- - **Very poor commenting** in order to explain the functionality for a certain class or a function in the system and even bad English in some parts. This below image shows a block of code with only a single line of comment (poor).

- Solution: The whole project files were scanned and fully documentation in a meaningful way, providing details for each classes, functions, objects and variables.

```
"""                                                                                                    ⚠ 17  ⚠ 85
Introduction: this file includes the main project functionalities including reading, opening, compiling & running
(Not yet implemented), saving both python code and CSharp.
This is considered as Version 2 from original repos on https://github.com/a1h2med/Anubis-IDE that includes many new
features.
This project is part of their graduation project and it intends to make a fully functioned IDE from scratch
################################################################################################################################################
"""
"""
################################################################################################################################################
   The imports needed for the project including:
    1-PYQT5,I/O library
    2-Python_Coloring including the format used for python code
    3-CSharpFormat including the format used for CSharp code
    4-FastExecution including the template in which the user enters his code
################################################################################################################################################
"""
import sys
import glob
import os
import serial
import Python_Coloring
import CSharpFormat
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import FastExecution
from io import StringIO

"""
```

4- **Leaving unrelated lines of code** in the project in a commented form, he needs to clean the code.

- Solution: all unrelated or non-functionally parts were removed from the project

5- **Bad structuring for the code**, declaring global variables but between classes declarations which degrades **the process of reverse engineering** (say program slicing and tracking variables)

- Solution: The files were restructured in a better way (sections) and before each section there is a brief description about it.

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

# This includes the path for the file the user opens/selects
filePathForSelectedFile = ''
# This include the list of parameters passed by the user in Fast Execution Mode
parametersList=[]
# A boolean for fast Execution Button mode( if flag%2==0 then open window for user to enter his code,
# otherwise it executes it if user press again)
flag=1

"""
#########################################################################################################################
 Global function section used in various places
#########################################################################################################################
"""
"""
Lists serial port names
:raises EnvironmentError: On unsupported or unknown platforms
:returns: A list of the serial ports available on the system
 """
```

6- **very bad documentations for the project** including:

- No Class diagrams.
- No Control / data Flow chart.
- No Sequence diagrams.
- No System description

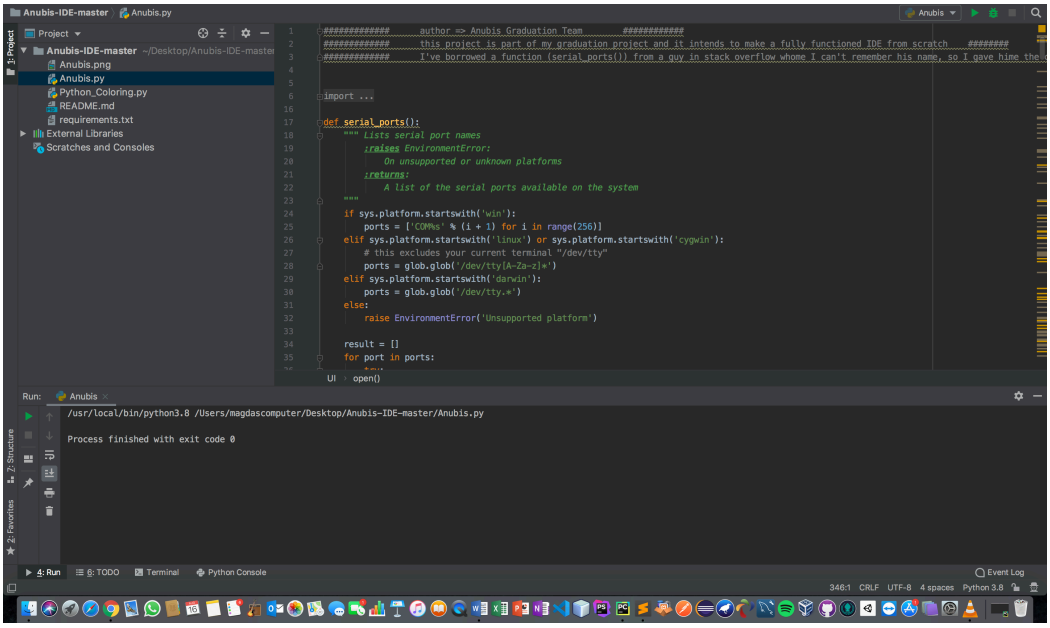
- Solution: This information was provided in this report

7- **The error messages don't remove** after I re-press the run button instead they all accumulate.

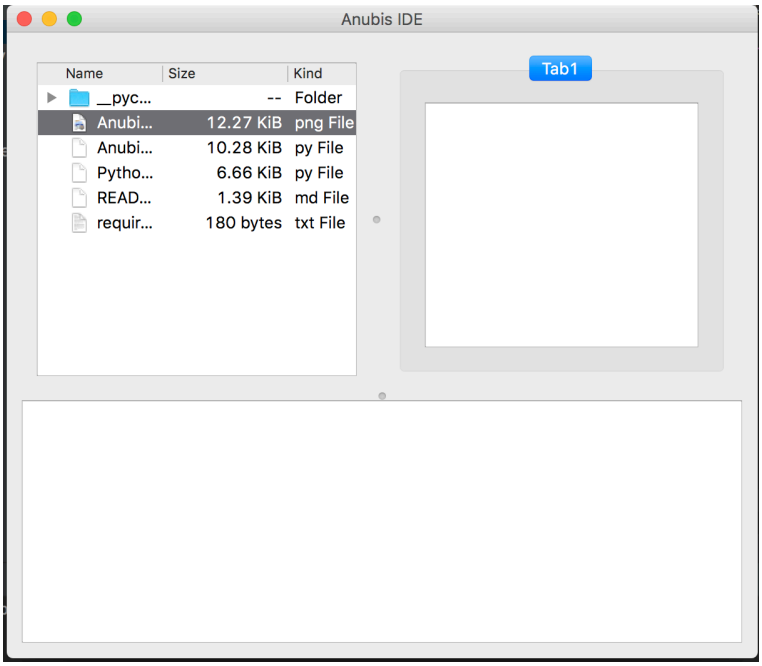
- Solution: This part of the GUI was fixed during the implementation of this version

3- Trial runs analysis Old

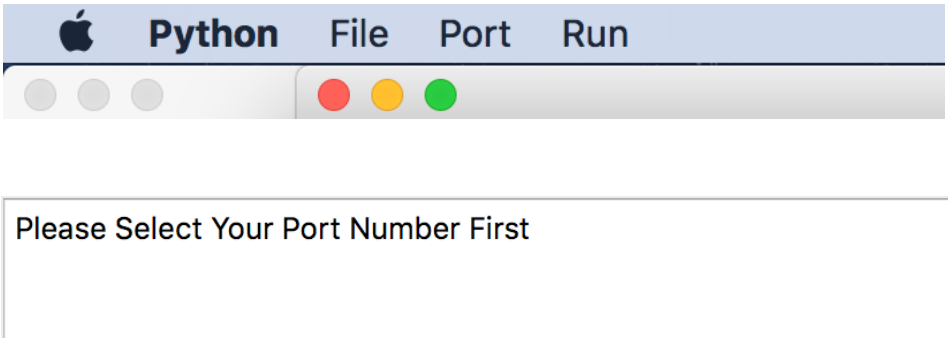
1- Open the project using PyCharm program.



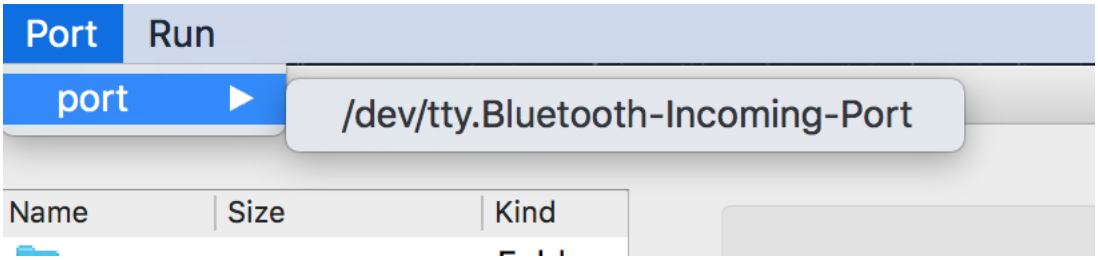
2- I press the run button and this GUI appears.



3- I press the run button and this message appears.



4- I select a port number. Restart the run process



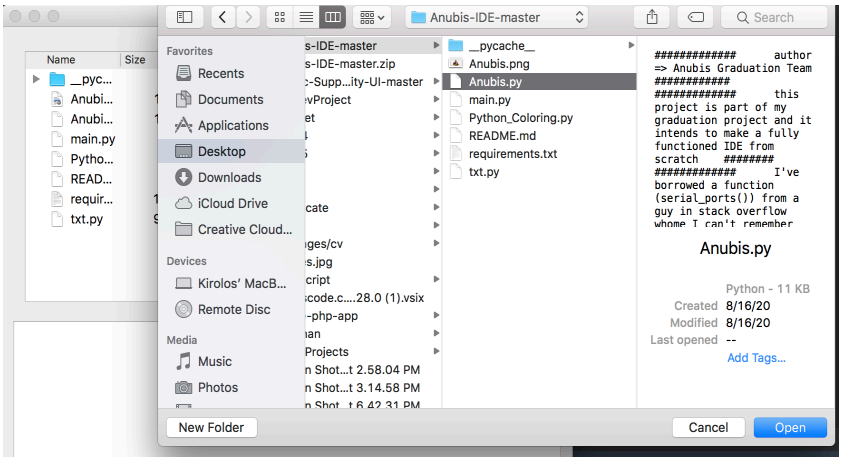
5- The given function "Run" as implemented have 2 options:

- Option1: the if statement is true, read the file as plain text then print error message shown before. So, the run function doesn't make any significant operation.
- Option2: the first error message shown before

```
############################        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):
```

6- The close button works well.

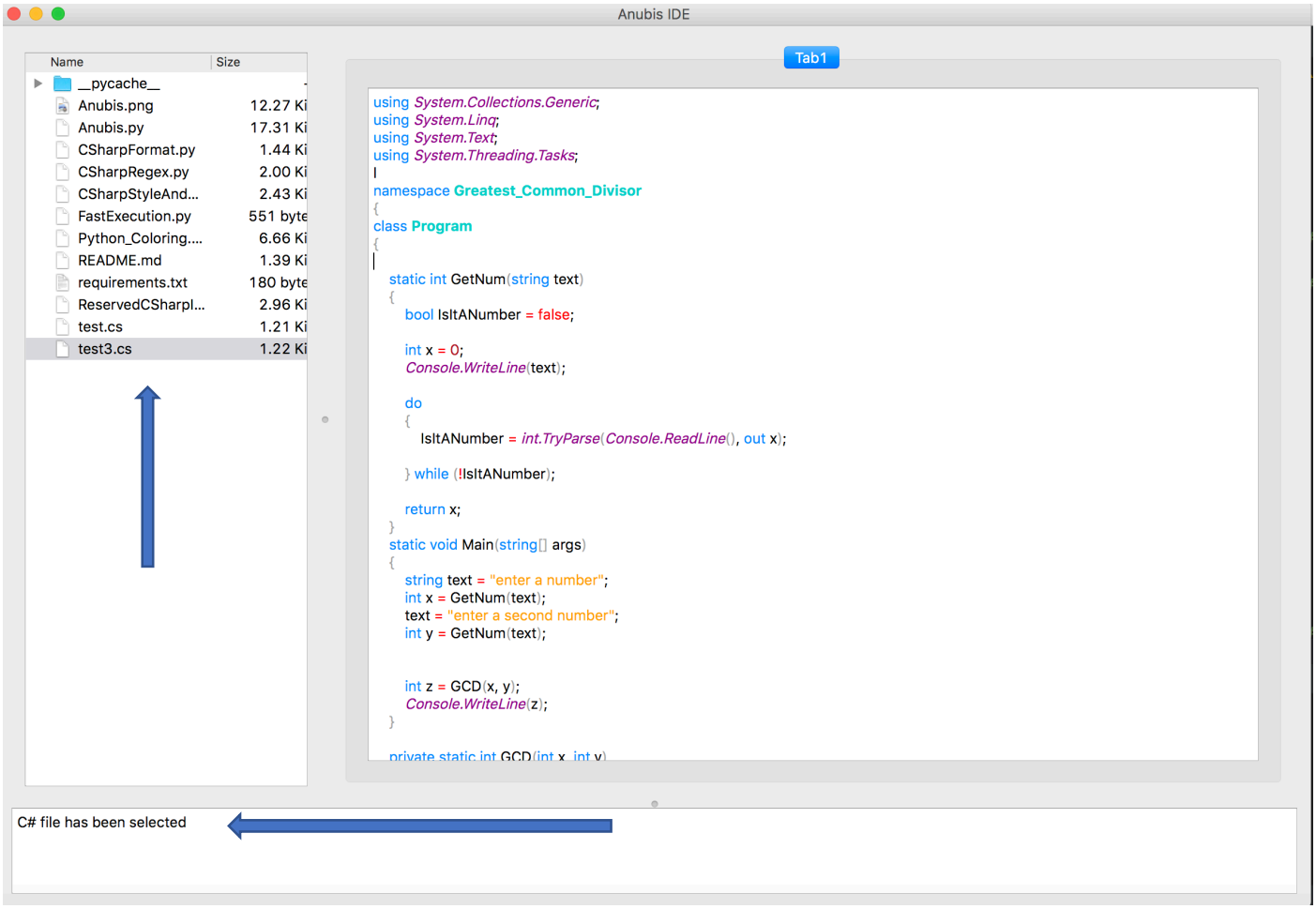
7- The open button seems to not work well as an error shows in the PyCharm compiler when I select to open a file.



4- Trial runs For New Features

1- Automatically detection for the file format and based on the language, the program displays it in different view. Also, it prints a line declaring the type of file selected whether python or C#.

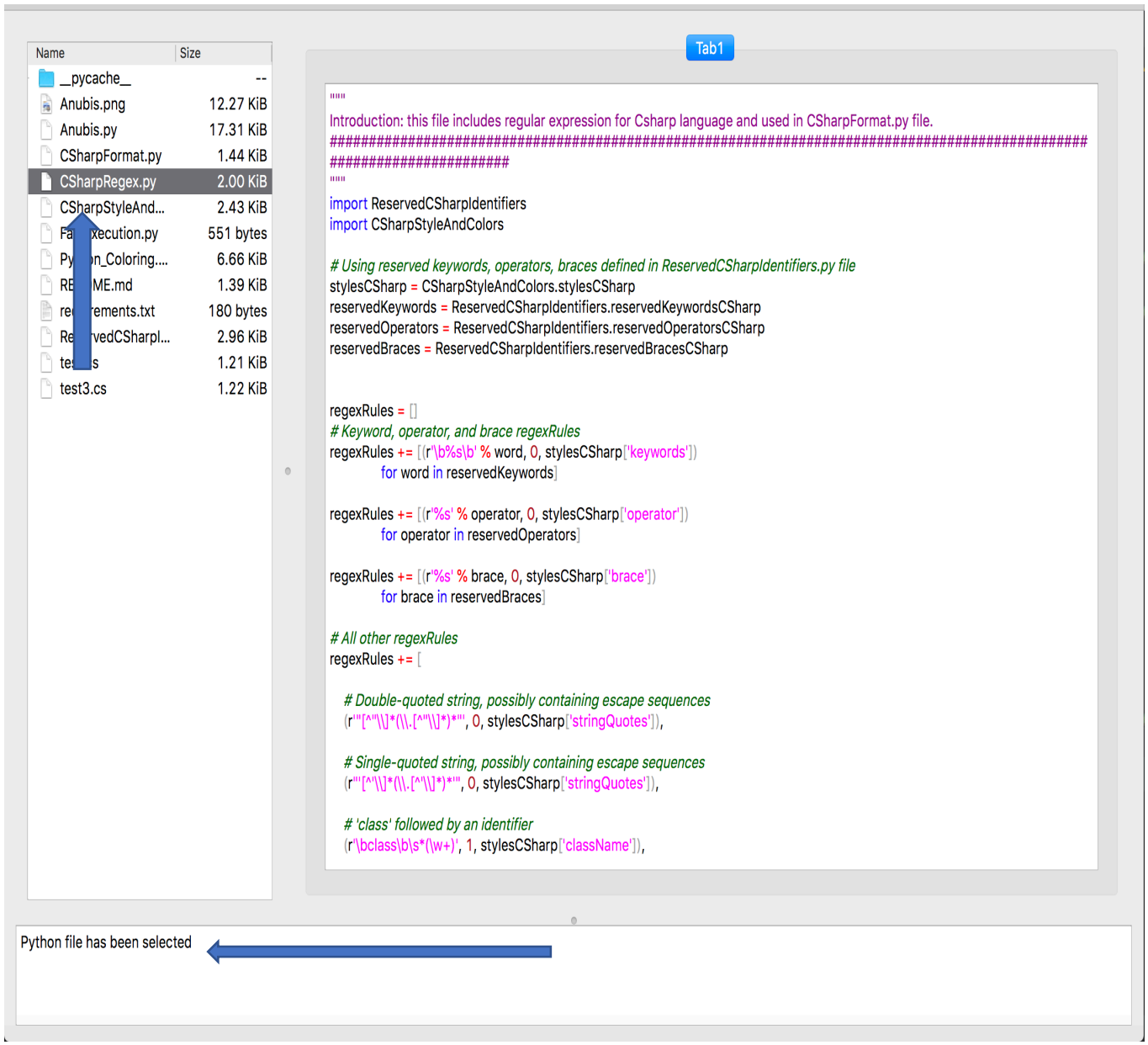
- Testing C# file For a GCD code taken from the internet by selecting it from the left-hand tab directly:



```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Greatest_Common_Divisor
{
class Program
{

  static int GetNum(string text)
  {
    bool IsItANumber = false;

    int x = 0;
    Console.WriteLine(text);

    do
    {
      IsItANumber = int.TryParse(Console.ReadLine(), out x);

    } while (!IsItANumber);

    return x;
  }
  static void Main(string[] args)
  {
    string text = "enter a number";
    int x = GetNum(text);
    text = "enter a second number";
    int y = GetNum(text);


    int z = GCD(x, y);
    Console.WriteLine(z);
  }

  private static int GCD(int x, int y)
```

```
    int z = GCD(x, y);
    Console.WriteLine(z);
  }

  private static int GCD(int x, int y)
  {
    int v = 0;
    int n = 0;

    v = GetGreatestDivisor(x, y);


    return v;

  }

  static int GetGreatestDivisor(int m, int h)
    {

      do
      {
        for (int i = m; i <= 1; i--)



          if (m%i == 0 && h%i == 0)
          {
            int x = 0;
            x = i;

            return x;
          }
      } while (true);
      return m;
    }

  }
}
```

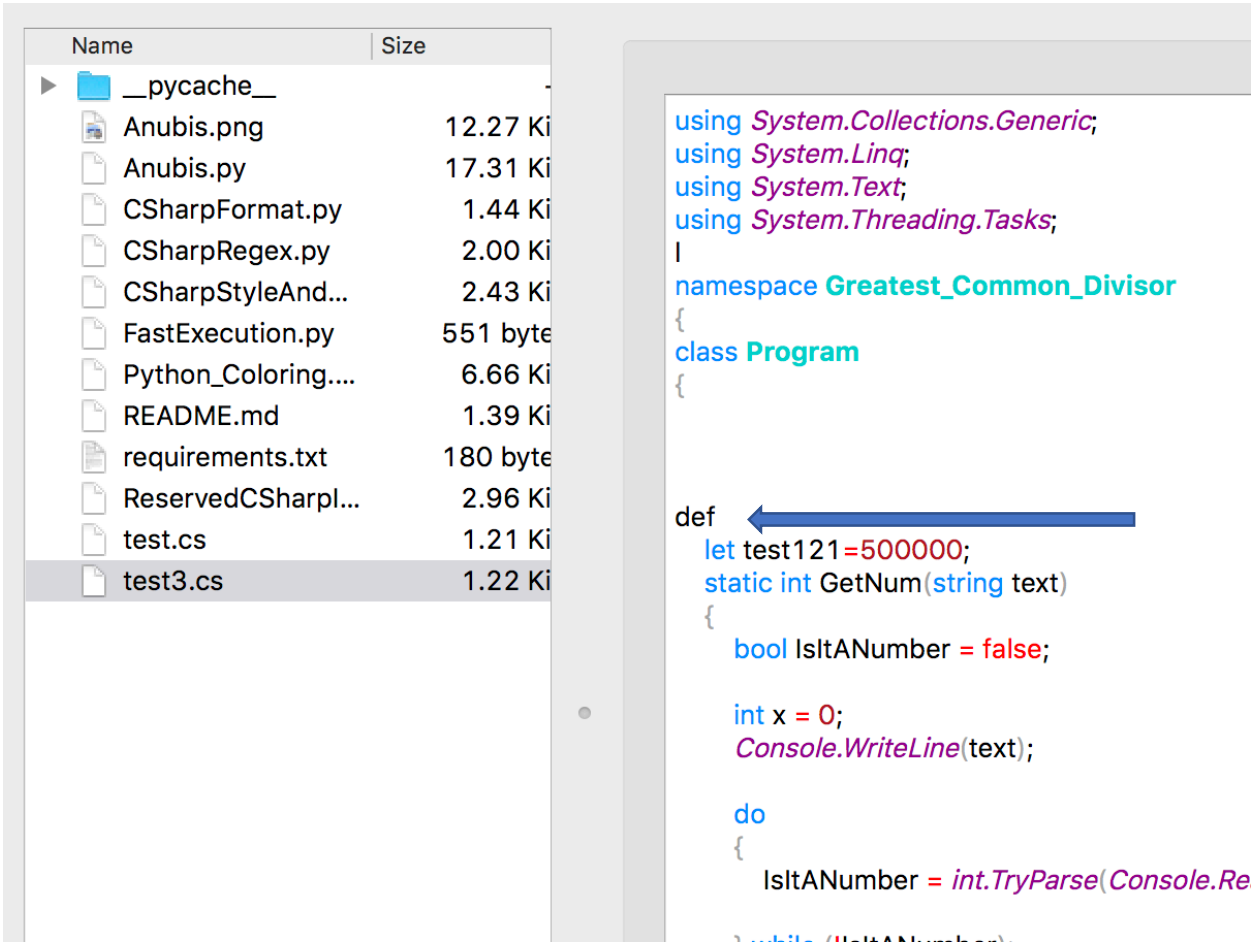
- Testing Python file: From the directory by selecting it from the left-hand tab:



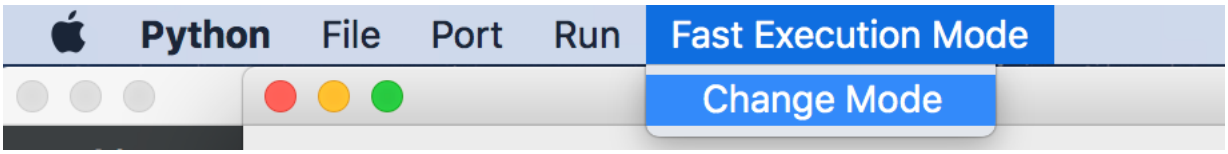
```
"""
Introduction: this file includes styles used to color & format c# file and used in CSharpFormat.py file.
2 sections :format, stylesCSharp
#########################################################################################################
#########################
  A function that returns a QTextCharFormat with the given attributes passed.
#########################################################################################################
#########################
"""
from PyQt5.QtGui import QColor, QTextCharFormat, QFont


def format (color, style=''):

    # define 2 variables, one Constructs a color that is a copy of color.
    # The other focuses on the style which can be bold, italic or normal
    syntaxColor = QColor()
    syntaxFormat= QTextCharFormat()

    # check if a color is being passed and assign it as a foreground
    if type(color) is not str:
        syntaxColor.setRgb(color[0], color[1], color[2])
    else:
        syntaxColor.setNamedColor(color)
    syntaxFormat.setForeground(syntaxColor)

    # check if a style is being passed
    if 'bold' in style:
        syntaxFormat.setFontWeight(QFont.Bold)
    if 'italic' in style:
        syntaxFormat.setFontItalic(True)

    # return the format based on the given implementation and passed style
    return syntaxFormat
"""
#########################################################################################################
##
    This section includes the style used by CSharp for formating the file.
```

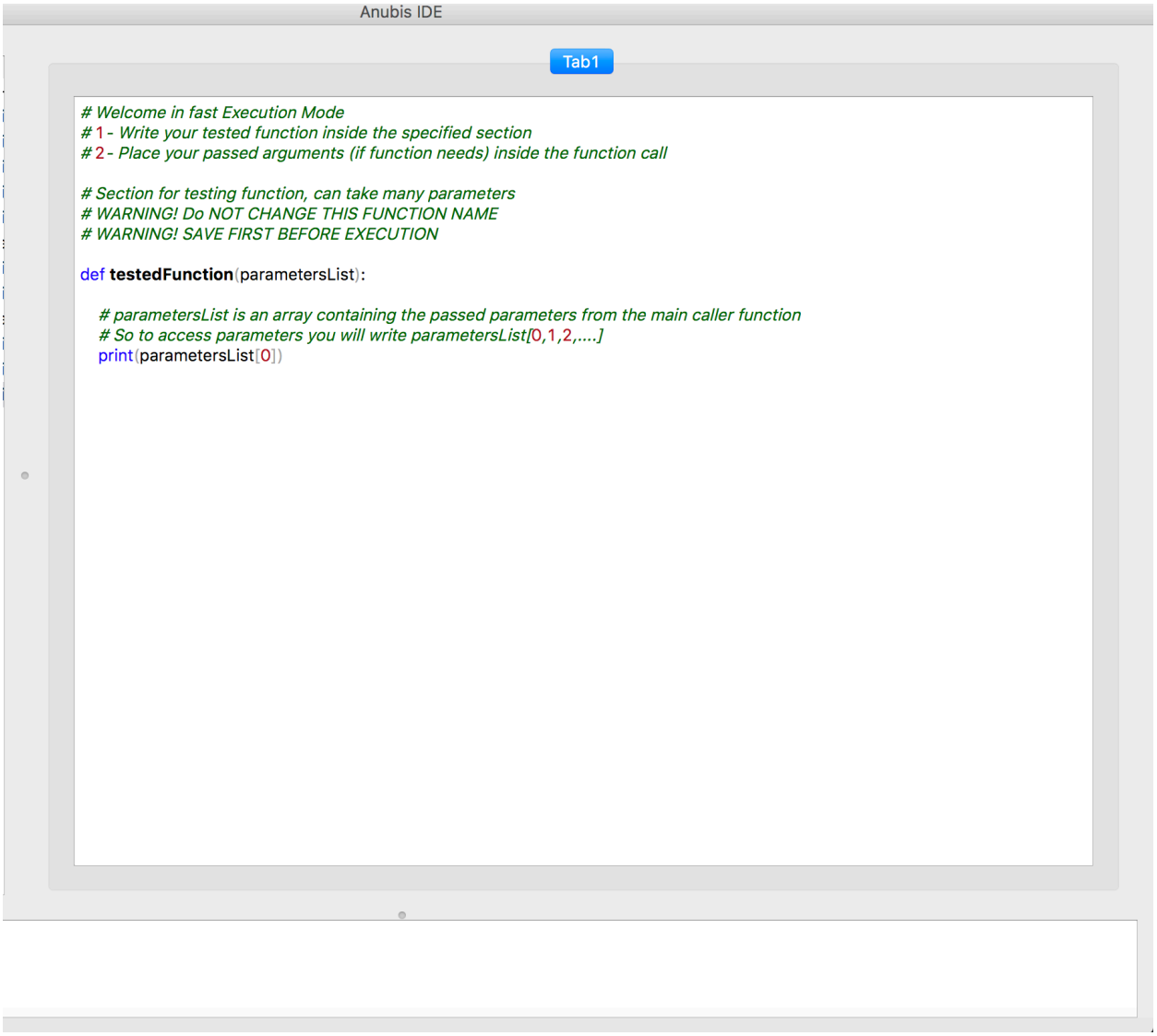
```
"""
################################################################################################################
##
   This section includes the style used by CSharp for formating the file.
   The chosen colors are based on the ones used in VSCode editor
################################################################################################################
##
"""
stylesCSharp = {
      # Reserved keywords coloring in C#
     'keywords': format('dodgerBlue'),

      # Operators coloring in C#
     'operator': format('red'),

      # Braces coloring in C#
     'brace': format('darkGray'),

      # Numbers coloring in C#
     'numbers': format('brown'),

      # Strings including: variablesName, functions  coloring in C#
     'string': format('black', 'italic'),

      # comment coloring in C#
     'comment': format('darkGreen', 'italic'),

      # className coloring in C#
     'className': format('mediumTurquoise', 'bold'),

      # String included in quotes coloring in C#
     'stringQuotes': format('orange'),

      # Functionalities built in like console. or imports inside in quotes coloring in C#
     'functionalities': format('darkMagenta','italic'),

  }
```

- Testing a python code inside C# code, so the expected output is not recognizing the keywords as not being defined as a regex for the C# language.



2- Fast Execution Mode, if the user selects this mode from the menu bar then a new window opens telling him where to place his code and giving some guidelines for how to use this functionality

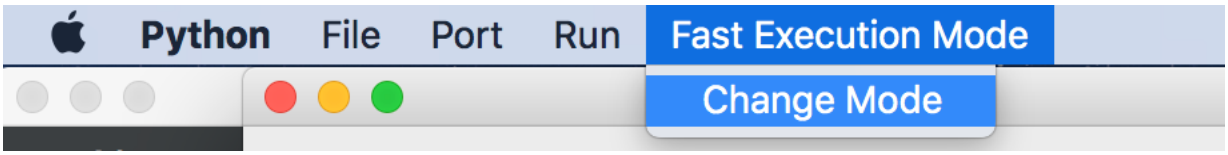




- Trying code that finds the maximum between two given numbers written by the user

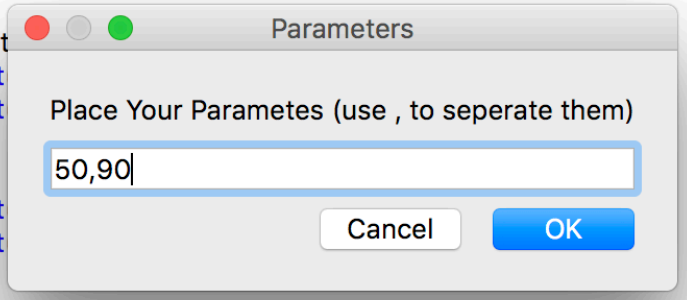
```
# Welcome in fast Execution Mode
# 1 - Write your tested function inside the specified section
# 2 - Place your passed arguments (if function needs) inside the function call

# Section for testing function, can take many parameters
# WARNING! Do NOT CHANGE THIS FUNCTION NAME
# WARNING! SAVE FIRST BEFORE EXECUTION

def testedFunction(parametersList):

   # parametersList is an array containing the passed parameters from the main caller function
   # So to access parameters you will write parametersList[0,1,2,....]

   if(parametersList[0]>parametersList[1]):
         print('The maximum number is the first = ')
         print(parametersList[0])

  else:
         print('The maximum number is the second = ')
         print(parametersList[1])
```

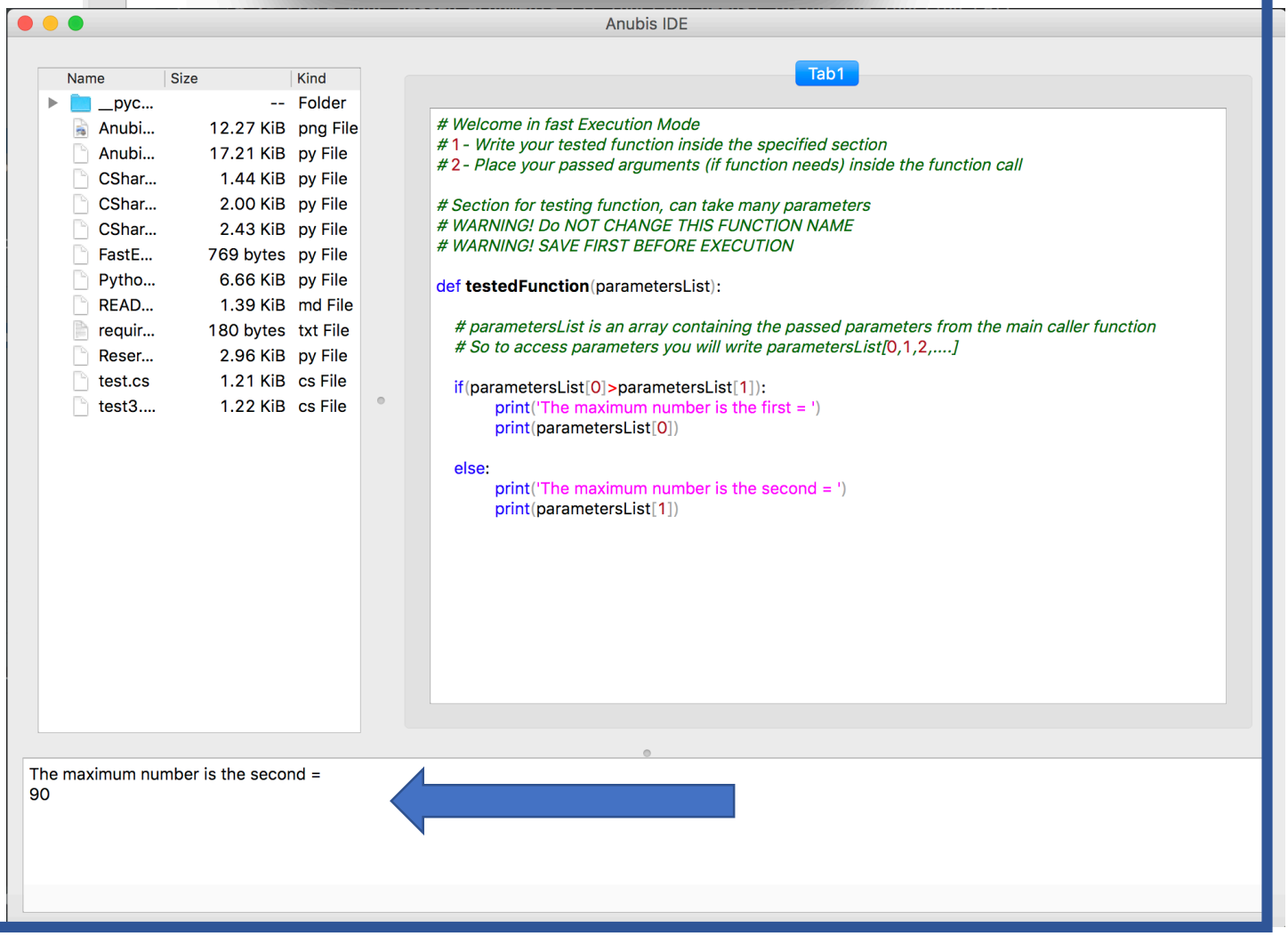
- It's a must to save the code before proceeding.
- Press again the fast Execution mode



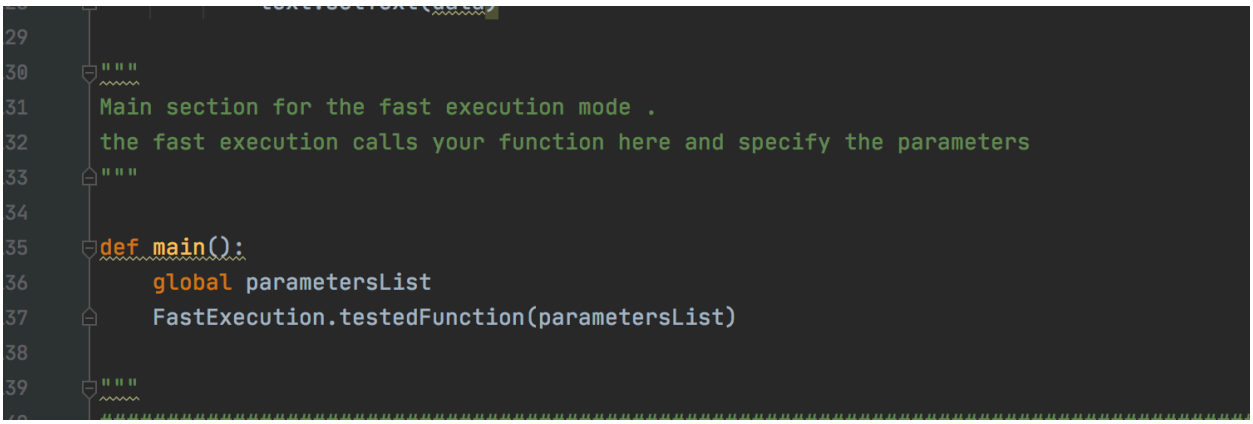
- As we can see, another window appears asking the user to specify the parameters
- Note:
  - User could pass no parameters in other condition
  - User could pass any number of parameters as needed

```
def testedFunction(parametersList):

    # parametersList is an array containing the passed parameters from the main caller function
    # So to access parameters you will write parametersList[0,1,2,....]

    if(paramet
        print
        print

    else:
        print
        print
```





- It’s important to note that this function is being called by a main () function that we discussed before in earlier sections.



```
29
30  """
31  Main section for the fast execution mode .
32  the fast execution calls your function here and specify the parameters
33  """
34
35  def main():
36      global parametersList
37      FastExecution.testedFunction(parametersList)
38
39  """
```

3- Trying **the save button,** we can see that the same file is updated and saved permanently.

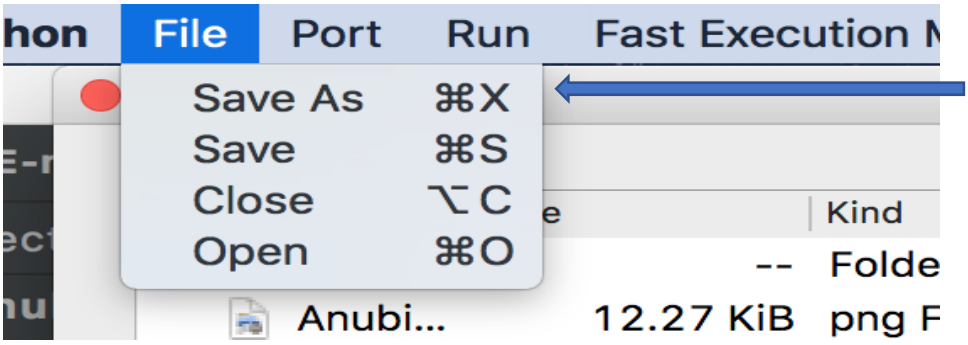
```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Greatest_Common_Divisor
{
class Program
{

   static int GetNum(string text)
   {
      bool IsItANumber = false;
      int x = 0;
      Console.WriteLine(text);

      do
      {
```

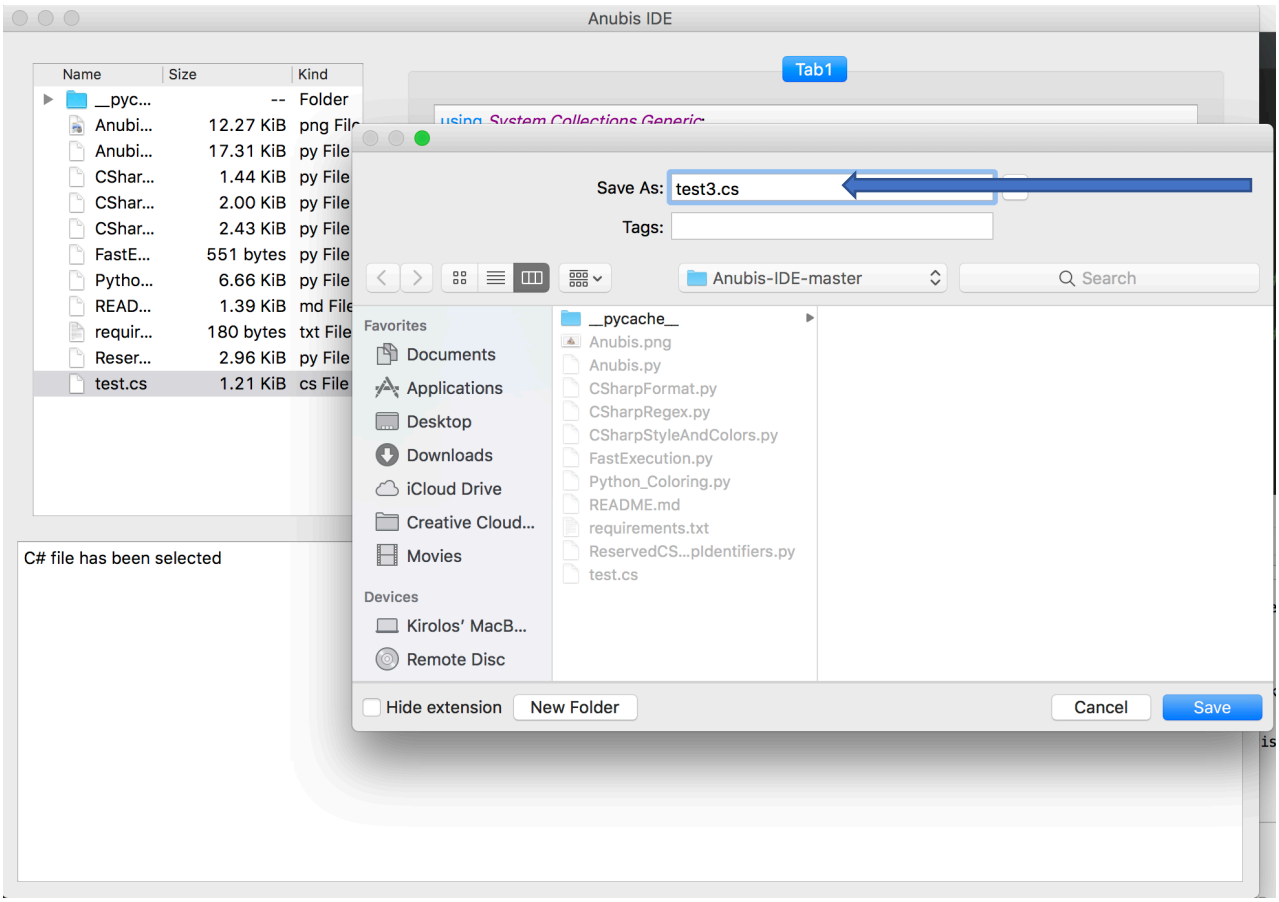
```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
l
namespace Greatest_Common_Divisor
{
class Program
{

   static int GetNum(string text)
   {
      let var xyz=10;

      bool IsItANumber = false;

      int x = 0;
      Console.WriteLine(text);
```

- If we open the file in another text editor we will see the file has been modified.

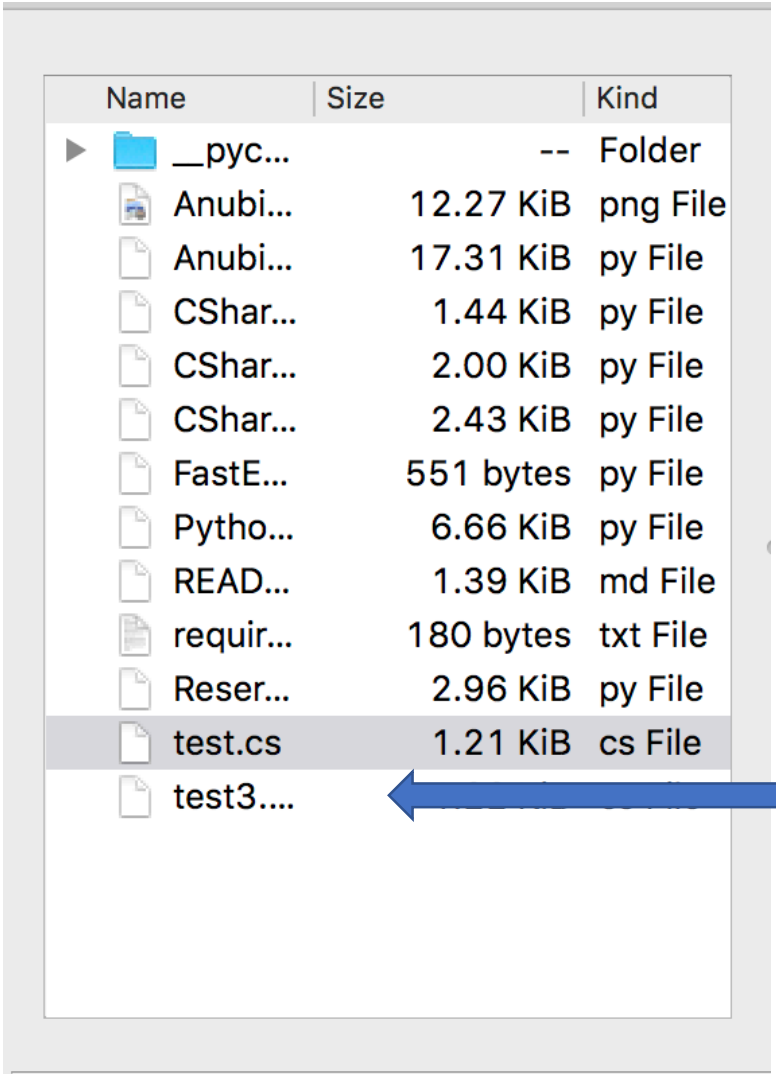
```
test.cs
1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Text;
4  using System.Threading.Tasks;
5  l
6  namespace Greatest_Common_Divisor
7  {
8  class Program
9  {
10
11 let var xyz=10;
12
13
14     static int GetNum(string text)
15     {
16         bool IsItANumber = false;
17         int x = 0;
18         Console.WriteLine(text);
19
20         do
21         {
22             IsItANumber = int.TryParse(Console.ReadLine(), out x);
```

4- **Trying the new saveAs button**, we can see it opens another window to select the file name and extension





- As we can see the new file has been saved and created

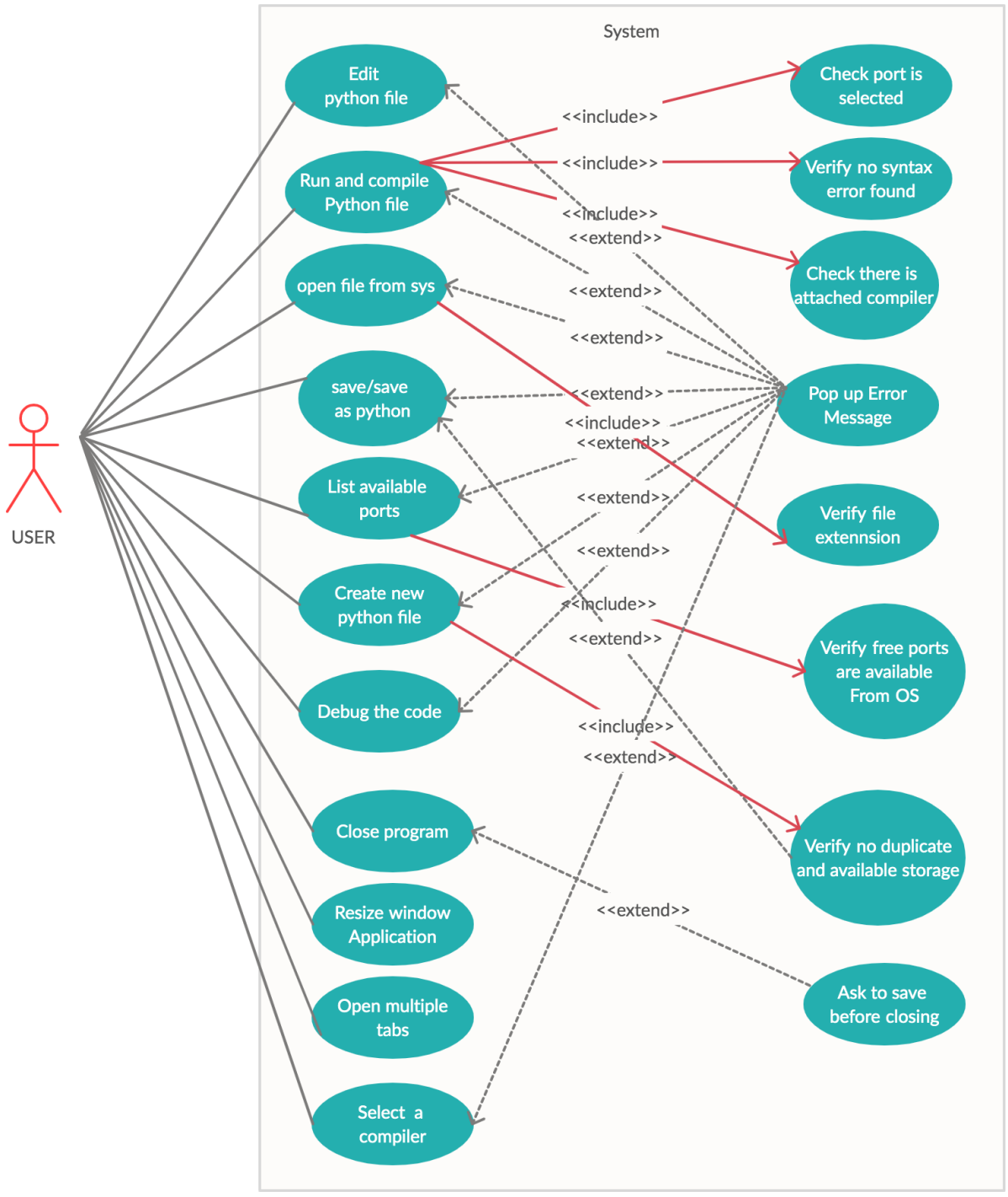


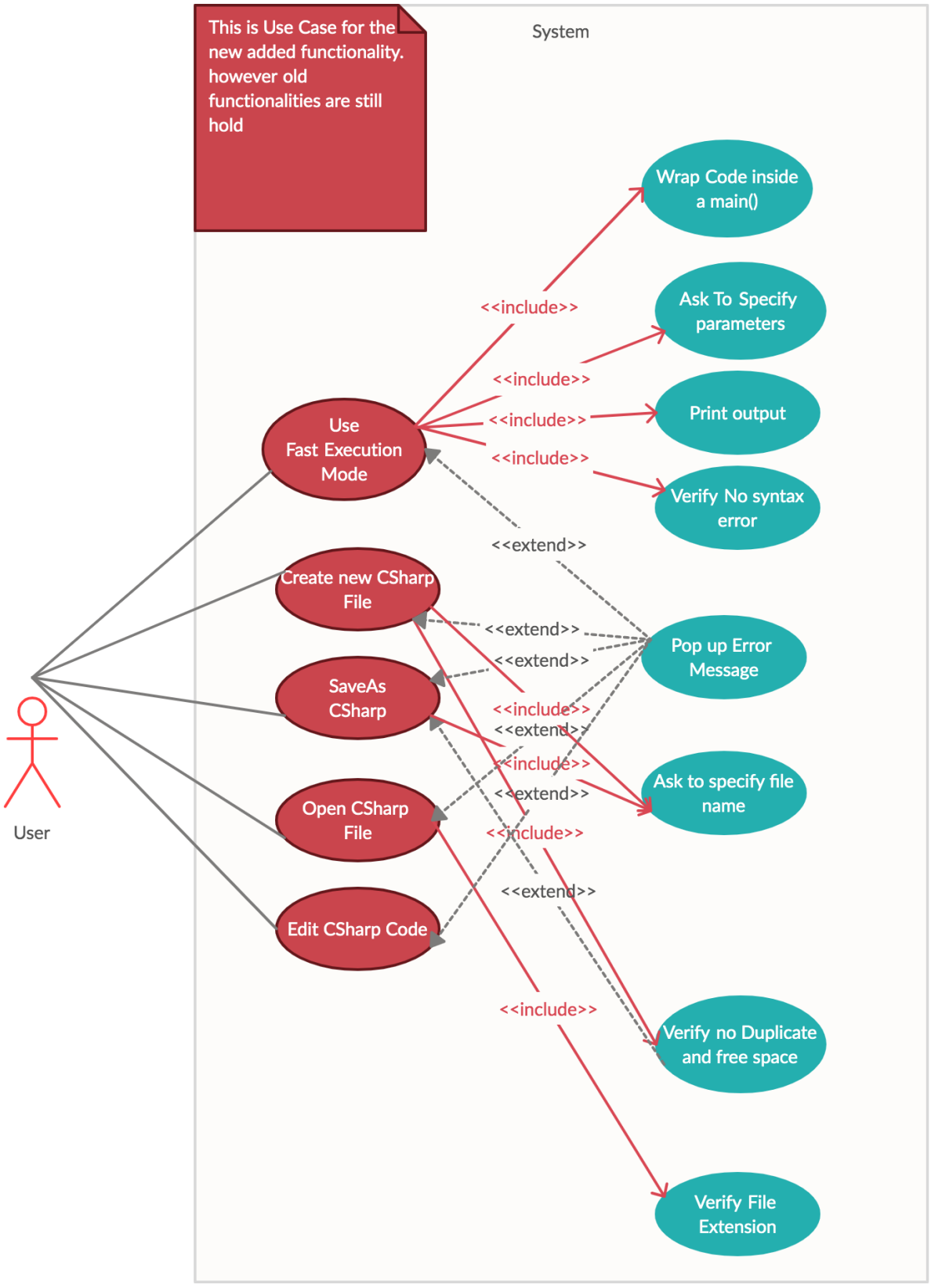
Part 2

- SRS
- Use Case

Note: New Updates or modified Parts in the reports are being wrapped inside a blue rectangle showing difference from old PDF version

1- Use Case Diagram old system




This is Use Case for the new added functionality. however old functionalities are still hold
System
Wrap Code inside a main()
Ask To Specify parameters
Print output
Verify No syntax error
Use Fast Execution Mode
Create new CSharp File
SaveAs CSharp
Open CSharp File
Edit CSharp Code
Pop up Error Message
Ask to specify file name
Verify no Duplicate and free space
Verify File Extension
User
<<include>>
<<include>>
<<include>>
<<include>>
<<extend>>
<<extend>>
<<extend>>
<<include>>
<<extend>>
<<include>>
<<extend>>
<<include>>
<<extend>>
<<include>>

2- New Added SRS

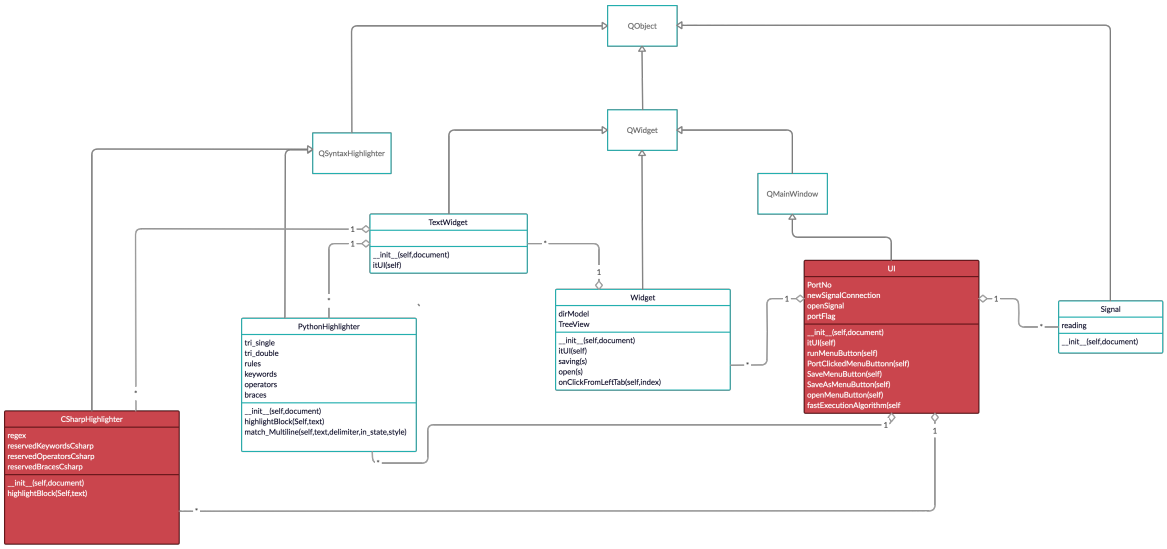
Functional Requirements
Fast Execution mode for is being introduced in the system allowing the user to rapidly execute a function by providing the needed parameters and display the output.
C# editor format, so the user can select a file and the program will detect the extension (Python or C#) and use the suitable formats
SaveAs button, enable the user to save new file with new name, chose format and select location to save.

Part 3 Design

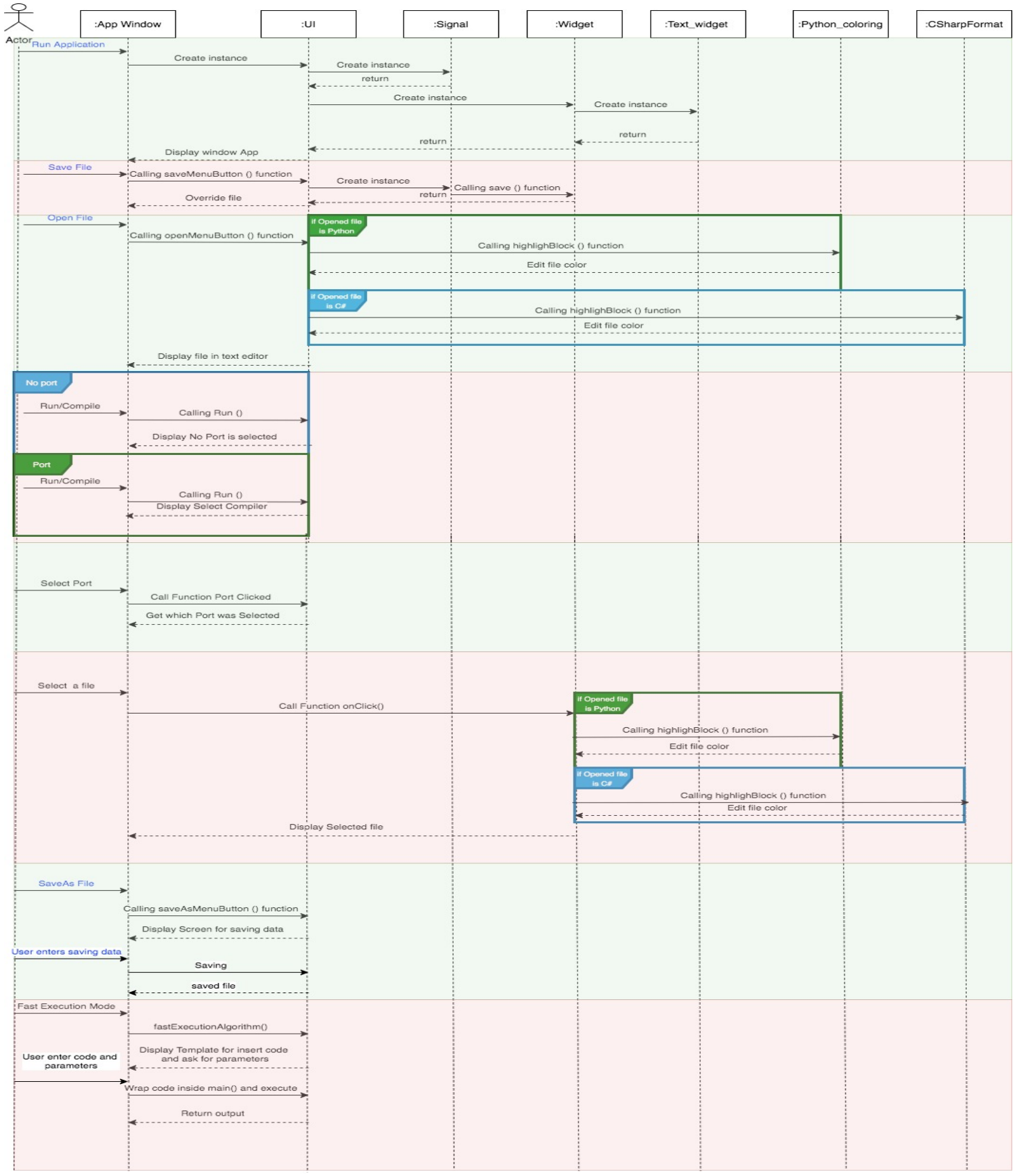
- Class Diagram
- Sequence Diagram

Note: New Updates or modified Parts in the reports are being wrapped inside a blue rectangle showing difference from old PDF version

1- Class Diagram



2- Sequence Diagram



Part 4

- Code

A- Anubis.py

```
"""
Introduction: this file includes the main project functionalities including
reading, opening, compiling & running
(Not yet implemented), saving both python code and CSharp.
This is considered as Version 2 from original repos on
https://github.com/a1h2med/Anubis-IDE that includes many new
features.
This project is part of their graduation project and it intends to make a
fully functioned IDE from scratch
##########################################################################
########################################
"""
"""
##########################################################################
########################################
   The imports needed for the project including:
    1-PYQT5,I/O library
    2-Python_Coloring including the format used for python code
    3-CSharpFormat including the format used for CSharp code
    4-FastExecution including the template in which the user enters his code
##########################################################################
########################################
"""
import sys
import glob
import os
import serial
import Python_Coloring
import CSharpFormat
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import FastExecution
from io import StringIO

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

# This includes the path for the file the user opens/selects
filePathForSelectedFile = ''
# This include the list of parameters passed by the user in Fast Execution
Mode
parametersList=[]
```

```
# A boolean for fast Execution Button mode( if flag%2==0 then open window for
user to enter his code,
# otherwise it executes it if user press again)
flag=1

"""
#############################################################################
##########################################
 Global function section used in various places
#############################################################################
##########################################
"""
"""
Lists serial port names
:raises EnvironmentError: On unsupported or unknown platforms
:returns: A list of the serial ports available on the system
 """

def serialPorts():

    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

"""
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):
    newSignalConnection = Signal()
    newSignalConnection.reading.connect(Widget.Saving)
    newSignalConnection.reading.emit(s)

# same as reading Function
@pyqtSlot(str)
def Openning(s):
```

```
    newSignalConnection = Signal()
    newSignalConnection.reading.connect(Widget.Open)
    newSignalConnection.reading.emit(s)

"""
This new added feature check the file extension based on user event and set
the suitable format for the language
used by the file.
Also it prints on the screen the type of file the user chooses
"""

def automaticFileExtensionSetter(fileName):

    global filePathForSelectedFile

    # First case for C# file: set format and print a message on the editor
    if fileName and fileName.lower().endswith(('.cs')):
        CSharpFormat.CSharpHighlighter(text)
        text2.clear()
        text2.append('C# file has been selected')

    # second case for Python file: set format and print a message on the
editor
    if fileName and fileName.lower().endswith(('.py')):
        Python_Coloring.PythonHighlighter(text)
        text2.clear()
        text2.append('Python file has been selected')

    # Open file and display it in the editor tab
    if fileName:
        filePathForSelectedFile = fileName
        file = open(fileName, 'r')
        with file:
            data = file.read()
            text.setText(data)

"""
Main section for the fast execution mode .
the fast execution calls your function here and specify the parameters
"""

def main():
    global parametersList
    FastExecution.testedFunction(parametersList)

"""
##########################################################################
########################################
Signal class that initializing a Signal which will take (string) as an input
##########################################################################
########################################
"""
class Signal(QObject):

    reading = pyqtSignal(str)
    # init Function for the Signal class
    def __init__(self):
        QObject.__init__(self)
```

```
"""
#############################################################################
########################################
TextWidget class which is made to connect the QTab with the necessary
layouts.
Class which is made to implement some usage from the Python_coloring.py file
to color the code written in
the editor tab of the program in the widget class. Function used (itUI).
#############################################################################
########################################
"""

class TextWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.itUI()
    def itUI(self):
        global text
        text = QTextEdit()
        hbox = QHBoxLayout()
        hbox.addWidget(text)
        self.setLayout(hbox)

"""
#############################################################################
########################################
Widget class the main application window implementation that appears, opening
and saving functionality for a file.
Functions used (initUI, saving, openMenuButton, onClickedFromLeftTab).
#############################################################################
########################################
"""
class Widget(QWidget):

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

    def initUI(self):

        # This widget is responsible of making Tab in IDE which makes the
Text editor looks nice
        tab = QTabWidget()
        tx = TextWidget()
        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.onClickedFromLeftTab)

        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 saveMenuButton the text inside
the first text editor
    @pyqtSlot(str)
    def Saving(s):
        name = os.path.basename(filePathForSelectedFile)
        if name != '':
            with open(name, 'w') as file:
                TEXT = text.toPlainText()
                file.write(TEXT)
                file.flush()
                os.fsync(file.fileno())
                file.close()

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

    # Function gets called when the user click on a file from the left hand
tab.
    # It focus on the file extension in order to select which format to use
for displaying the code(PYTHON/CSHARP).
    def onClickedFromLeftTab(self, index):
        filePath = self.sender().model().filePath(index)
        filePath = tuple([filePath])
        automaticFileExtensionSetter(filePath[0])


"""
################################################################################
########################################
UI class holds the main functionality of the project as for example creating
signal connection, creating the menu items, shortcuts, creating a widget
(main window for the application) instance, run
the program, saveMenuButton and openMenuButton files buttons. Functions used
(InitUI, RunMenuButton, PortClickedMenuButton, saveMenuButton,
openMenuButton)
################################################################################
########################################
"""
class UI(QMainWindow):
    def __init__(self):
        super().__init__()
        self.intUI()

    def intUI(self):
        self.portFlag = 1
        self.newSignalConnection = Signal()

        self.Open_Signal = Signal()

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

        # connecting (self.newSignalConnection) with reading function
        self.newSignalConnection.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')
        fastExecutionMode = menu.addMenu('Fast Execution Mode')

        # 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 (serialPorts())
function
        # copyrights of serialPorts() 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 = serialPorts()

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

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

        # Making and adding RunMenuButton Actions
        FastExecutionAction = QAction("Change Mode", self)
        FastExecutionAction.triggered.connect(self.fastExecutionAlgorithm)
        fastExecutionMode.addAction(FastExecutionAction)

        RunAction = QAction("RunMenuButton", self)
        RunAction.triggered.connect(self.RunMenuButton)
        Run.addAction(RunAction)

        # Making and adding File Features
        Save_Action = QAction("Save", self)
        Save_Action.triggered.connect(self.saveMenuButton)
        Save_Action.setShortcut("Ctrl+S")

        Save_As_Action = QAction("Save As", self)
        Save_As_Action.triggered.connect(self.saveAsMenuButton)
        Save_As_Action.setShortcut("Ctrl+x")

        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.openMenuButton)

        fileMenu.addAction(Save_As_Action)
        fileMenu.addAction(Save_Action)
        fileMenu.addAction(Close_Action)
        fileMenu.addAction(Open_Action)

        # Seting the window Geometry
```

```
        self.setGeometry(200, 150, 600, 500)
        self.setWindowTitle('Anubis IDE')
        self.setWindowIcon(QtGui.QIcon('Anubis.png'))

        widget = Widget()

        self.setCentralWidget(widget)
        self.show()

    ##########################        Start Of the Functions
#################
    def RunMenuButton(self):
        if self.portFlag == 0:
        #
        ##### Compiler Part which is still missied
        #
            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 PortClickedMenuButton(self):
        action = self.sender()
        self.portNo = action.text()
        self.portFlag = 0

    # This function to saveMenuButton the code into a file
    def saveMenuButton(self):
        self.newSignalConnection.reading.emit("name")

    # New added functionality for saveAs files to support both python and
CSharp
    def saveAsMenuButton(self):
        name = QFileDialog.getSaveFileName(self, 'Save File')
        if name[0] !='' and name[0] !='FastExecution.py':
            with open(name[0], 'w') as file:
                TEXTData = text.toPlainText()
                file.write(TEXTData)
                file.close()

    # This function to openMenuButton a file and exhibits it to the user in a
text editor
    def openMenuButton(self):
        fileName = QFileDialog.getOpenFileName(self,'Open File','/home')
        automaticFileExtensionSetter(fileName[0])


    """
    New added feature to the system, it include the fast execution mode.
    A fast executed for python code: in this feature the editor user will
enter a code for a single function that would
    be automatically wrapped inside a program that has a main function that
will call the the function.
    The user would be asked to also to provide a list of parameters to be
passed from the main to the called function
    """
```

```
    def fastExecutionAlgorithm(self):
        global flag
        flag=flag+1

        # first of all it opens a new window with the location to place the
user code for testing
        if flag % 2 == 0 :
            text2.clear()
            text2.append("Fast Execution mode Applied")
            automaticFileExtensionSetter('FastExecution.py')
            text2.clear()

        # the user execute the function he wrote, the program will ass the
user to specify the list of parameters.
        # the output for the execution will be printed on the editor GUI
        else :
            text, okPressed = QInputDialog.getText(self, "Parameters", "Place
Your Parametes (use , to seperate them)", QLineEdit.Normal, "")
            if okPressed and text != '':
                text2.clear()
                global parametersList
                parametersList = text.split(",")
                codeOut = StringIO()
                codeErr = StringIO()
                sys.stdout = codeOut
                sys.stderr = codeErr

                # this line will execute the needed code the user made
                eval(compile('main()', '', 'eval'))
                sys.stdout = sys.__stdout__
                sys.stderr = sys.__stderr__
                text2.append(codeOut.getvalue())
                text2.append(codeErr.getvalue())
                codeOut.close()
                codeErr.close()


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

B- CsharpFormat.py

```
"""
#############################################################################
#####################
    Importing the needed functionalities from PYQT5.
    PyQt is a Python binding of the cross-platform GUI toolkit Qt,
implemented as a Python plug-in.
#############################################################################
#####################
"""
from PyQt5.QtCore import QRegExp
from PyQt5.QtGui import QSyntaxHighlighter
import CSharpRegex

"""Syntax highlighter for the Python language."""
class CSharpHighlighter(QSyntaxHighlighter):

    def __init__(self, document):
        QSyntaxHighlighter.__init__(self, document)

        # Creating Regular expression for CSharp code
        rules = CSharpRegex.regexRules

        # 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)
"""
#############################################################################
#####################
    Importing the needed functionalities from PYQT5.
    PyQt is a Python binding of the cross-platform GUI toolkit Qt,
implemented as a Python plug-in.
#############################################################################
#####################
"""
from PyQt5.QtCore import QRegExp
from PyQt5.QtGui import QSyntaxHighlighter
import CSharpRegex
```

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

    def __init__(self, document):
        QSyntaxHighlighter.__init__(self, document)

        # Creating Regular expression for CSharp code
        rules = CSharpRegex.regexRules

        # 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)
```

C-CsharpRegex.py

```
"""
Introduction: this file includes regular expression for Csharp language and
used in CSharpFormat.py file.
##############################################################################
########################################
"""
import ReservedCSharpIdentifiers
import CSharpStyleAndColors

# Using reserved keywords, operators, braces defined in
ReservedCSharpIdentifiers.py file
stylesCSharp = CSharpStyleAndColors.stylesCSharp
reservedKeywords = ReservedCSharpIdentifiers.reservedKeywordsCSharp
reservedOperators = ReservedCSharpIdentifiers.reservedOperatorsCSharp
reservedBraces = ReservedCSharpIdentifiers.reservedBracesCSharp


regexRules = []
# Keyword, operator, and brace regexRules
regexRules += [(r'\b%s\b' % word, 0, stylesCSharp['keywords'])
               for word in reservedKeywords]

regexRules += [(r'%s' % operator, 0, stylesCSharp['operator'])
               for operator in reservedOperators]

regexRules += [(r'%s' % brace, 0, stylesCSharp['brace'])
               for brace in reservedBraces]

# All other regexRules
regexRules += [

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

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

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

    # 'namespace' followed by an identifier
    (r'\bnamespace\b\s*(\w+)', 1, stylesCSharp['className']),

    # a word followed by a dot followed by another word
    (r'\w*\.\w*', 0, stylesCSharp['functionalities']),

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

    # MultiLine comment
    (r'/\*(?:(?!\*/).)*\*/', 0, stylesCSharp['comment']),
```

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

D- CSharpStyleAndColor.Py

```
"""
Introduction: this file includes styles used to color & format c# file and
used in CSharpFormat.py file.
2 sections :format, stylesCSharp
#############################################################################
########################################
   A function that returns a QTextCharFormat with the given attributes
passed.
#############################################################################
########################################
"""
from PyQt5.QtGui import QColor, QTextCharFormat, QFont


def format (color, style=''):

    # define 2 variables, one Constructs a color that is a copy of color.
    # The other focuses on the style which can be bold, italic or normal
    syntaxColor = QColor()
    syntaxFormat= QTextCharFormat()

    # check if a color is being passed and assign it as a foreground
    if type(color) is not str:
        syntaxColor.setRgb(color[0], color[1], color[2])
    else:
        syntaxColor.setNamedColor(color)
    syntaxFormat.setForeground(syntaxColor)

    # check if a style is being passed
    if 'bold' in style:
        syntaxFormat.setFontWeight(QFont.Bold)
    if 'italic' in style:
        syntaxFormat.setFontItalic(True)

    # return the format based on the given implementation and passed style
    return syntaxFormat
"""
#############################################################################
#####################
    This section includes the style used by CSharp for formating the file.
    The chosen colors are based on the ones used in VSCode editor
#############################################################################
#####################
"""
stylesCSharp = {
        # Reserved keywords coloring in C#
       'keywords': format('dodgerBlue'),

        # Operators coloring in C#
       'operator': format('red'),

        # Braces coloring in C#
       'brace': format('darkGray'),
```

```
        # Numbers coloring in C#
       'numbers': format('brown'),

        # Strings including: variablesName, functions  coloring in C#
       'string': format('black', 'italic'),

        # comment coloring in C#
       'comment': format('darkGreen', 'italic'),

        # className coloring in C#
       'className': format('mediumTurquoise', 'bold'),

        # String included in quotes coloring in C#
       'stringQuotes': format('orange'),

        # Functionalities built in like console. or imports inside in quotes
coloring in C#
       'functionalities': format('darkMagenta','italic'),

   }
```

E- FastExecution.Py

```
F- # Welcome in fast Execution Mode
   # 1- Write your tested function inside the specified section
   # 2- Place your passed arguments (if function needs) inside the
   function call

   # Section for testing function, can take many parameters
   # WARNING! Do NOT CHANGE THIS FUNCTION NAME
   # WARNING! SAVE FIRST BEFORE EXECUTION

   def testedFunction(parametersList):

       # parametersList is an array containing the passed parameters from
   the main caller function
       # So to access parameters you will write parametersList[0,1,2,....]

       if(parametersList[0]==parametersList[1]):
                print('The maximum number is the first = ')
                print(parametersList[0])

       else:
                print('The maximum number is the second = ')
                print(parametersList[1])
```

F- Python_coloring.Py

```
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 regexRules
        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 regexRules
        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
```

G-ReservedCSharpIdentifiers.Py

```
"""
Introduction: this file includes all reserved words used in CSharp language
and used in CSharpFormat.py file.
3 sections :reservedKeywordsCSharp, reservedOperatorsCSharp,
reservedBracesCSharp
##########################################################################
########################################

Reserved identifiers that have special meanings to the compiler in C#.
They cannot be used as identifiers in your program unless they include @ as a
prefix.
Link Microsoft doc: https://docs.microsoft.com/en-us/dotnet/csharp/language-
reference/keywords/
"""
reservedKeywordsCSharp = [
    'abstract', 'as', 'base', 'bool', 'break', 'byte',
    'case', 'catch', 'char', 'checked', 'class', 'const',
    'continue', 'decimal', 'default', 'delegate',
    'do', 'double', 'else', 'enum',
    'event', 'explicit', 'extern', 'false',
    'finally', 'fixed', 'float', 'for',
    'foreach', 'goto', 'if', 'implicit',
    'in', 'int', 'interface', 'internal',
    'is', 'lock', 'long', 'namespace',
    'new', 'null', 'object', 'operator',
    'out', 'override', 'params', 'private',
    'protected', 'public', 'readonly', 'ref',
    'return', 'sbyte', 'sealed', 'short',
    'sizeof', 'stackalloc', 'static', 'string',
    'struct', 'switch', 'this', 'throw',
    'true', 'try', 'typeof', 'uint',
    'ulong', 'unchecked', 'unsafe', 'ushort',
    'using', 'virtual', 'void', 'volatile',
    'while', 'add', 'alias', 'ascending',
    'async', 'await', 'by', 'descending', 'dynamic',
    'equals', 'from', 'get', 'global',
    'group', 'into', 'join', 'let', 'nameof', 'notnull',
    'on', 'orderby', 'partial', 'remove', 'select',
    'set', 'unmanaged', 'value', 'var', 'when', 'where', 'yield'
]

"""
##########################################################################
########################################
 Operators that have special meanings to the compiler in C#.
 Documentation link: https://www.programiz.com/csharp-programming/operators
```

```
##########################################################################
########################################
 """
reservedOperatorsCSharp = [
    # Assignment Operators
    '=',
    # Arithmetic Operators
    '\+', '-', '\*', '/', '\%',
    # Relational Operators
    '==', '!=', '<', '<=', '>', '>=',
    # Logical Operators
    'true', 'false',
    # Bitwise Operators
    '\^', '\|', '\&', '\~', '>>', '<<',
    # Compound Assignment Operators
    '\+=', '\-=', '\*=', '\%=', '\&=', '\|=', '^=', '<<=', '>>=', '=>',
    # Unary operation
    '++', "--", '!',
]

"""
##########################################################################
########################################
Reserved braces used in programing and special meanings to the compiler in
C#.
Documentation link: https://docs.microsoft.com/en-us/dotnet/csharp/language-
reference/
##########################################################################
########################################
"""
reservedBracesCSharp = [
    '\{', '\}', '\(', '\)', '\[', '\]',
]
```