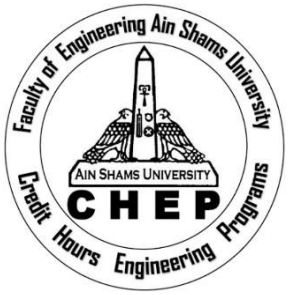


Faculty of Engineering Ain Shams University
Credit Hours Engineering Programs

Computer Engineering and Software Systems

CSE 426 – Software Maintenance and Evolution

Course Project

Delivered to:

Prof. Ayman Bahaa

Made by:

Ashraf Muhammed Sabry

(12p2018)

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 Ashraf Sabry. The source code could be found on the repo your/repo/here.

The goal of this tool was to provide a simple environment where the users can write, edit, compile, and run micro-python codes.

The tool was developed using Python and PYQT5 which is a library that allows using the Qt GUI framework from python and the Qt is written with C++ therefore benefitting from the high speed of the C++.

In a 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 discuss the changes made to this system in order to give it the support of an additional language which is C sharp so that the editor can support its format.

2. Requirements

2.1. Functional requirements

F1: Opening existing files (any extension).

F2: Editing the open file.

F3: Displaying files list next to the coding tab.

F4: Browsing the files in the different directories.

F5: Saving the files after editing their contents.

F6: Allowing the user to select a port from the available ports on the PC.

F7: Syntax highlighting in the code tab according to the language of the code.

2.2. New functional requirements

F8: Automatic detection and display of the language based on file extensions.

F9: Compiling and running python/C# functions.

F10: Allowing the user to enter parameters for python/C# functions.

F11: Exception handling and hints display.

2.3. Non-functional requirement

1. The system must be written in python.
2. The system should be delivered in 22nd of June 2022.
3. The system must support different OSs.
4. The system 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 it can detect the language of a file from its extension weather it is python or C#. The editor now also supports 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 for these keywords and operations were add as well.

The user can save the edits made on a file weather it was python code or C# (they are saved in the same directory of the IDE under the name saved_file with the proper extension chosen by the user).

3.2. Use case diagram

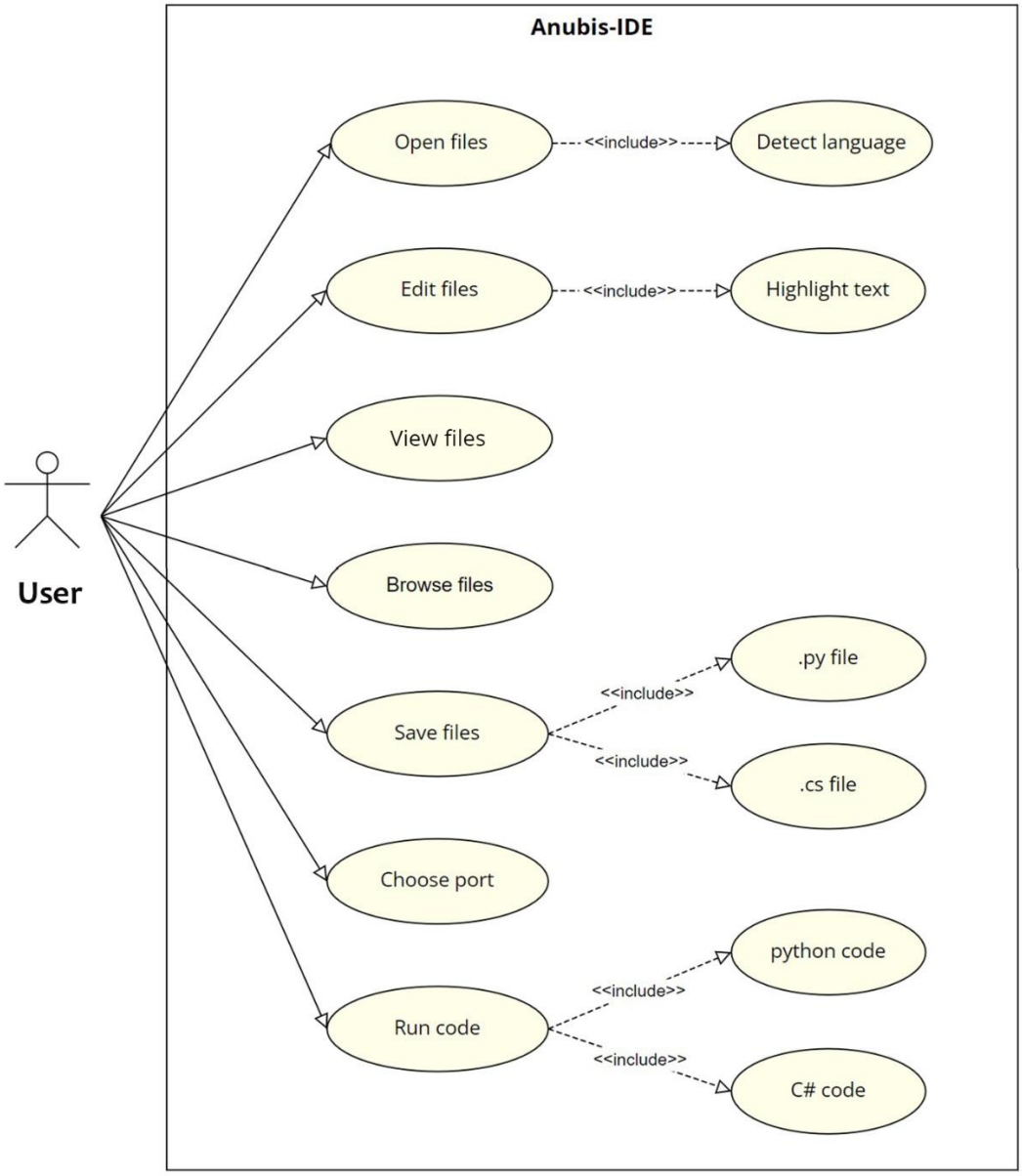


Figure 1: Use case diagram

3.3. Narrative Description of the use cases

- **Use Case #1 (Open files - U1)**

Author	Ashraf Sabry.
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 uses the shortcut “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. 4. The user clicks on the open button.

- **Use Case #2 (Edit file – U2)**

Author	Ashraf Sabry.
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 or an existing file. 2. The user edits the file by writing to it.

- **Use Case #3 (View files – U3)**

Author	Ashraf Sabry.
Use case	View files.
Goal in context	View files at the display 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 files of the same directory. 3. The files are shown to the user in the display tab.

- ## Use Case #4 (Browse files – U4)

Author	Ashraf Sabry.
Use case	Browse files.
Goal in context	Browsing the files of the system.
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.
Secondary actor	IDE.
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 code tab.

- ## Use Case #5 (Save files – U5)

Author	Ashraf Sabry.
Use case	Save files
Goal in context	Save files that were 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 uses the shortcut “Ctrl + s”. 4. The IDE saves the file under the name “saved_file” with the proper extension (.py for python and .cs for C#).
Exceptions	1.1. The user loaded contents from a file of an unsupported extension. 1.2. The system displays an error message.

- ## Use Case #6 (Choose port – U6)

Author	Ashraf Sabry.
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 chooses a port.

- **Use Case #7 (Run code – U7)**

Author	Ashraf Sabry.
Use case	Run code.
Goal in context	Compile and run snippets of python/C# code.
Precondition	A python/C# function definition is present in the code tab.
Successful end condition	The IDE compiles and run the code and the output is displayed in the output field.
Failed end condition	The code does not compile or run.
Primary actor	IDE.
Secondary actor	User.
Trigger	The user presses on the run button or uses shortcut 'Ctrl+r'.
Main flow	1. The user loads a file containing the function definition. 2. The IDE detects the language based on the file extension. 3. The user may edit the function definition in code tab. 4. For functions with parameters, the user provides parameters in the ***'parameters'*** tab and for function without, the user unchecks the ***'has parameters?'*** checkbox which is checked by default. 5. The user presses on the run button or uses shortcut 'Ctrl+r'. 6. For python functions, the IDE runs the function right away. 7. For C# functions, the IDE wraps the function inside a ***'main.cs'*** template and saves the combination in a temp file named ***'temp_code_runner.cs'*** then compiles and runs it. 8. The IDE displays the output/hints in the output field.
Exceptions	1.1 The user didn't load a file of the required language causing failure of language auto detection. 1.2 The IDE displays a hint. 3.1. Syntax error. 3.2. The IDE displays an error message. 4.1. The user didn't provide required parameters for the function or didn't uncheck the ***'has parameters?'*** checkbox for a functions with parameters. 4.2. The IDE displays a hint. 7.1. The user doesn't have the required dependencies for running C# code. 7.2. The system displays an error.

- ## Use Case #8 (Detect language – U8)

Author	Ashraf Sabry.
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 uses the appropriate highlighter.
Failed end condition	The system does not detect the file extension.
Primary actor	IDE.
Trigger	The user opens an existing file or clicks on a file in the files tab.
Main flow	1. The IDE reads the file name. 2. The IDE get the file extension. 3. The IDE detects the language and displays it. 4. The IDE selects the highlighter according to this extension.

Exceptions	2.1. The extension is not supported by the IDE. 2.2. The IDE will display ***'Language: Unsupported'*** and will choose a default highlighter.

- ## Use Case #9 (Syntax highlighting– U9)

Author	Ashraf Sabry.
Use case	Highlight text.
Goal in context	The code is highlighted according to its syntax of the programming language.
Precondition	Code tab is loaded with content.
Successful end condition	The code is highlighted according to its syntax.
Failed end condition	The code is not highlighted according to its syntax.
Primary actor	IDE.
Trigger	Code is written in the code tab or The user opens a code file.
Main flow	1. The IDE captures the text in the code snippets written in the tab of coding. 2. The IDE 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 based on the syntax and are displayed accordingly.

Exceptions	2.1. There file does not have a supported extension. 2.2. The system will choose a default highlighter.

3.4. Sequence diagram

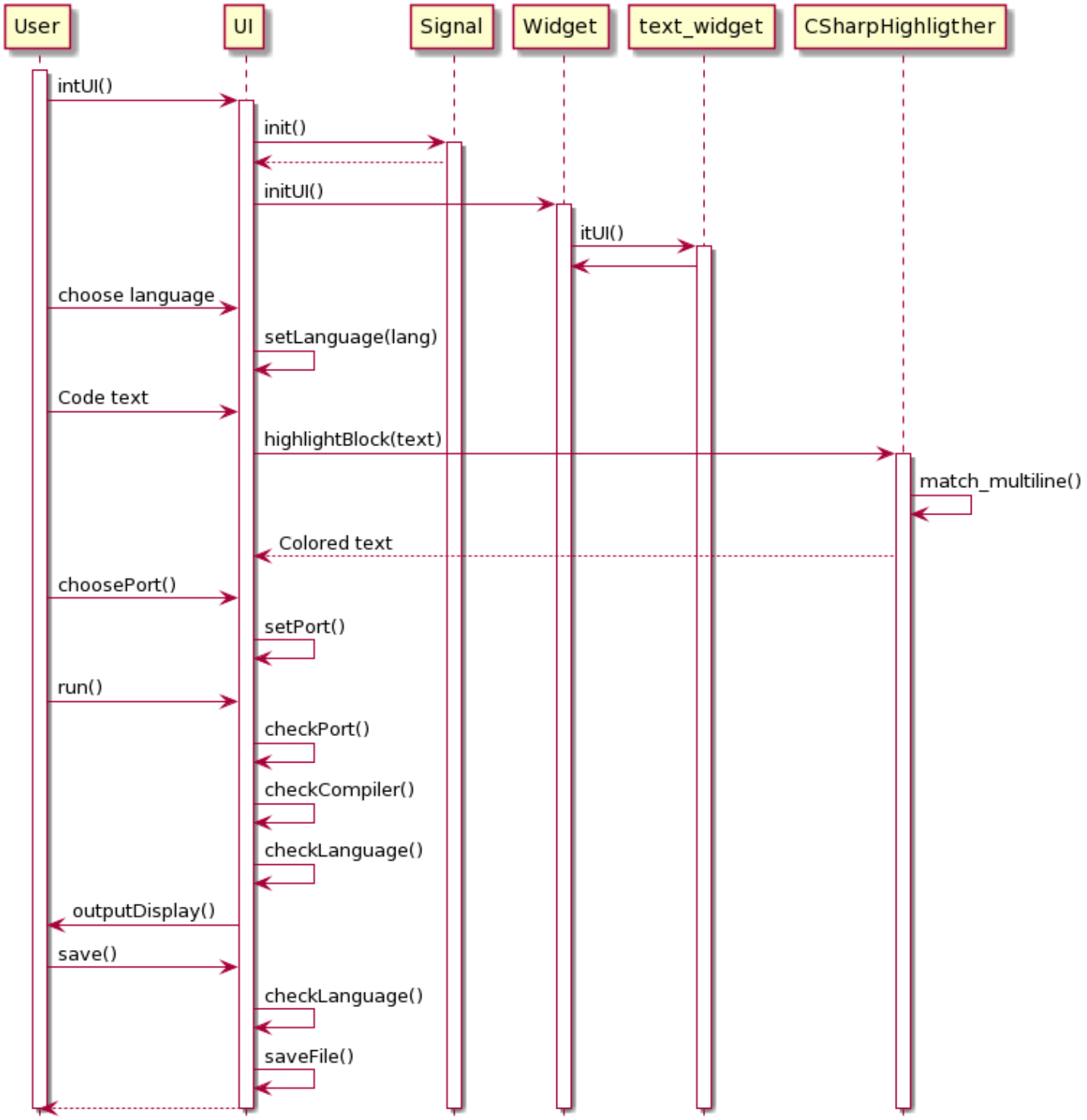


Figure 2: Sequence diagram for writing C# code

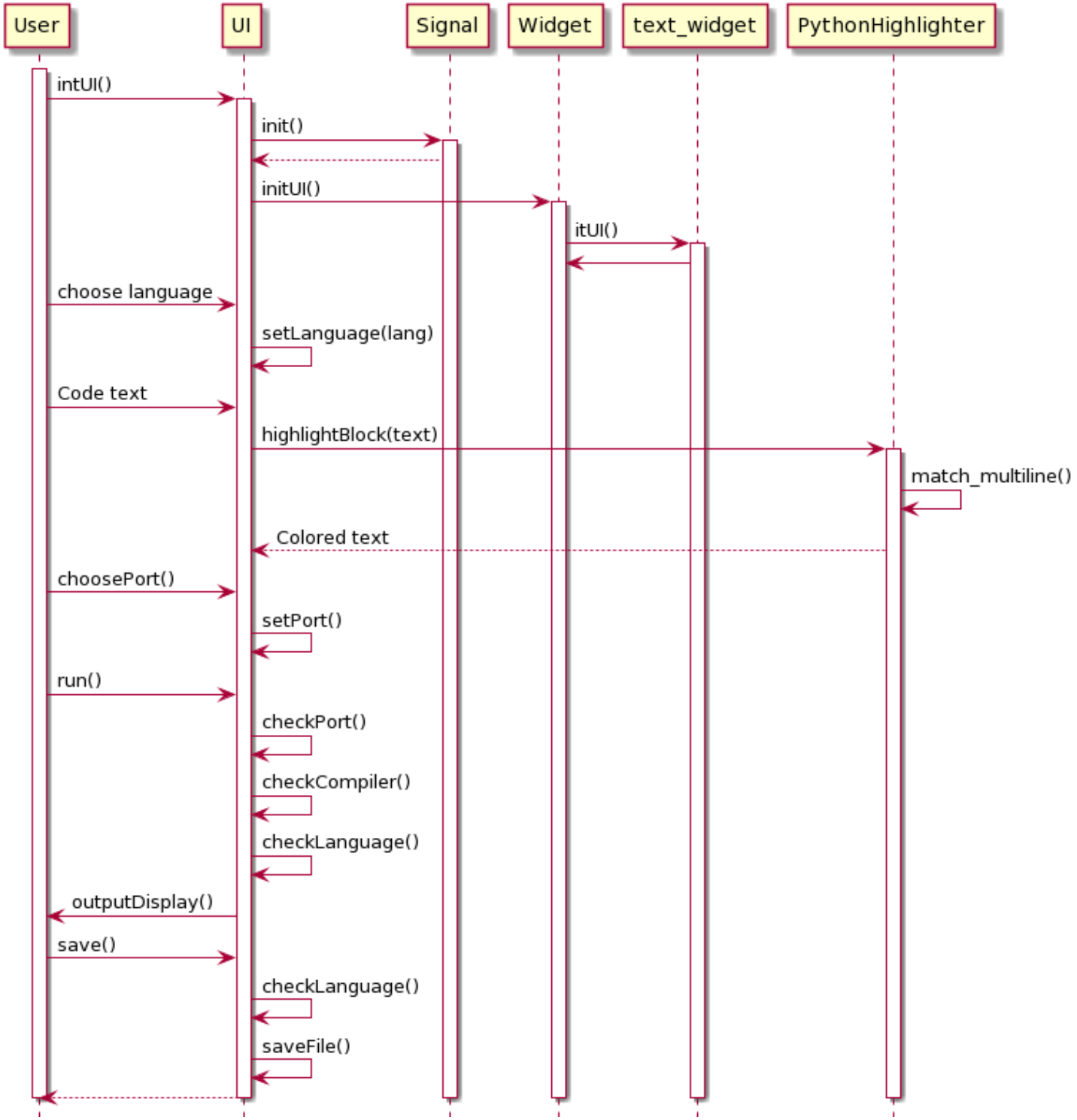


Figure 3: Sequence diagram for writing python code

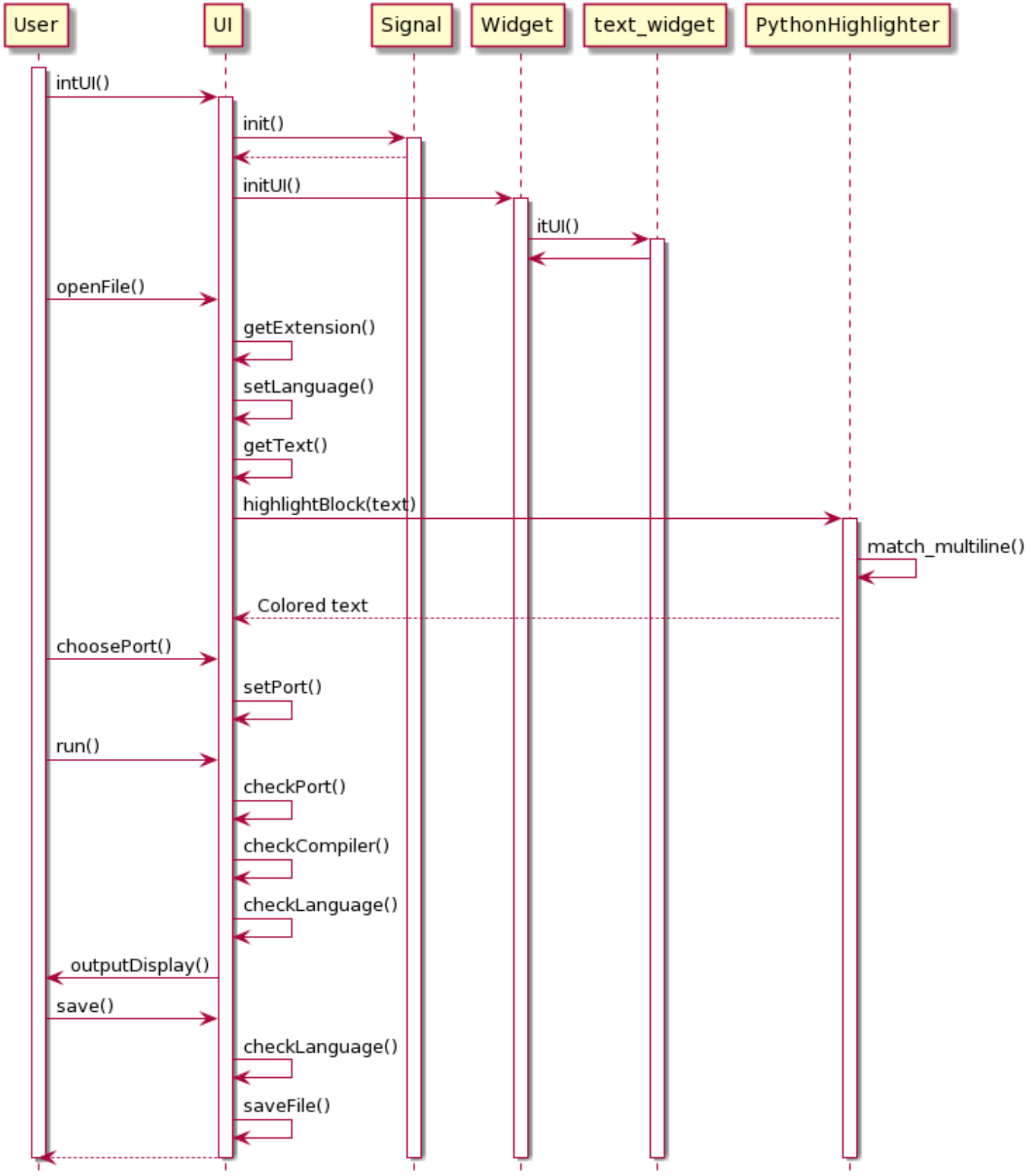


Figure 4: Open python file

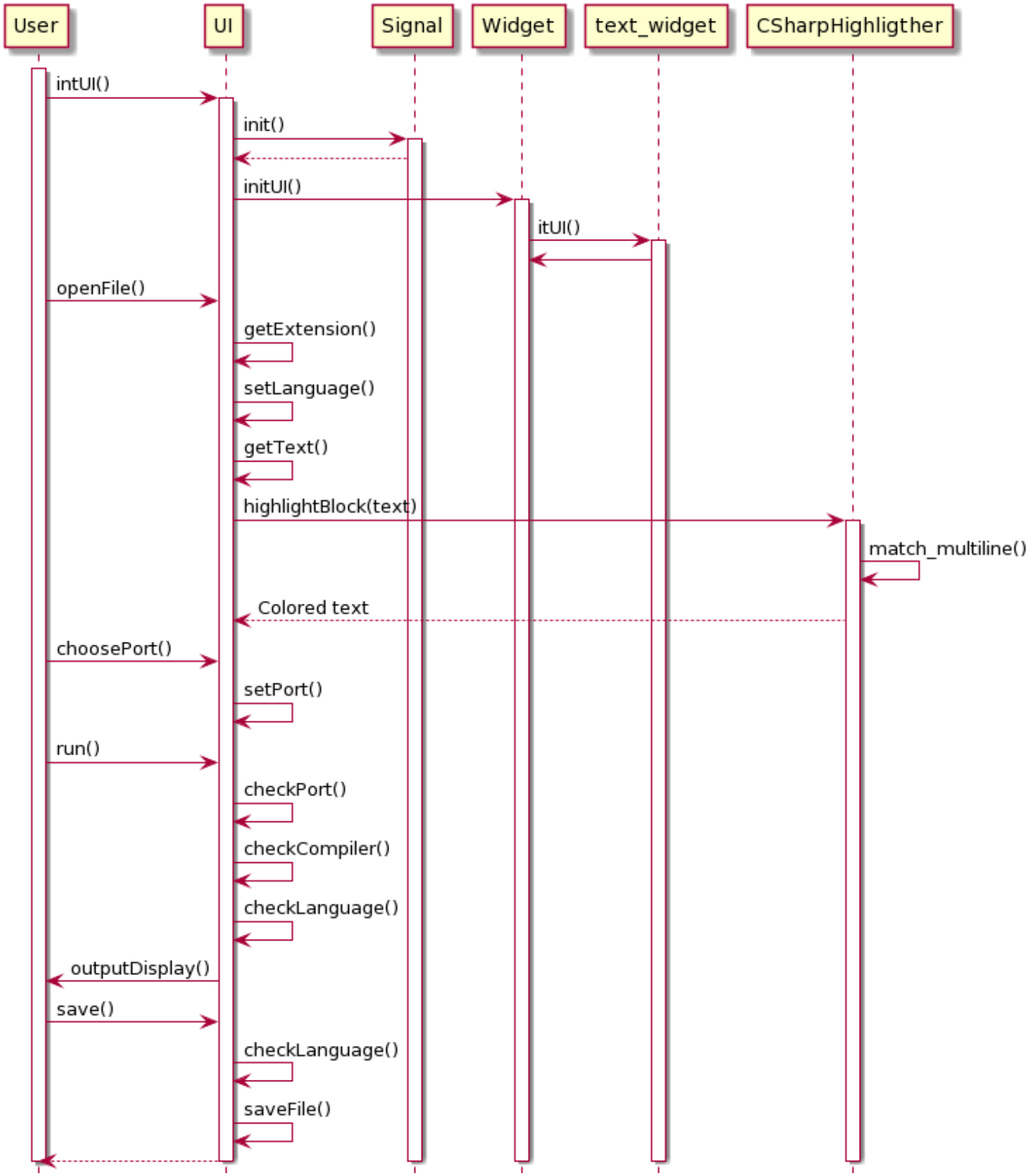


Figure 5: Open C# code

3.5. Class diagram

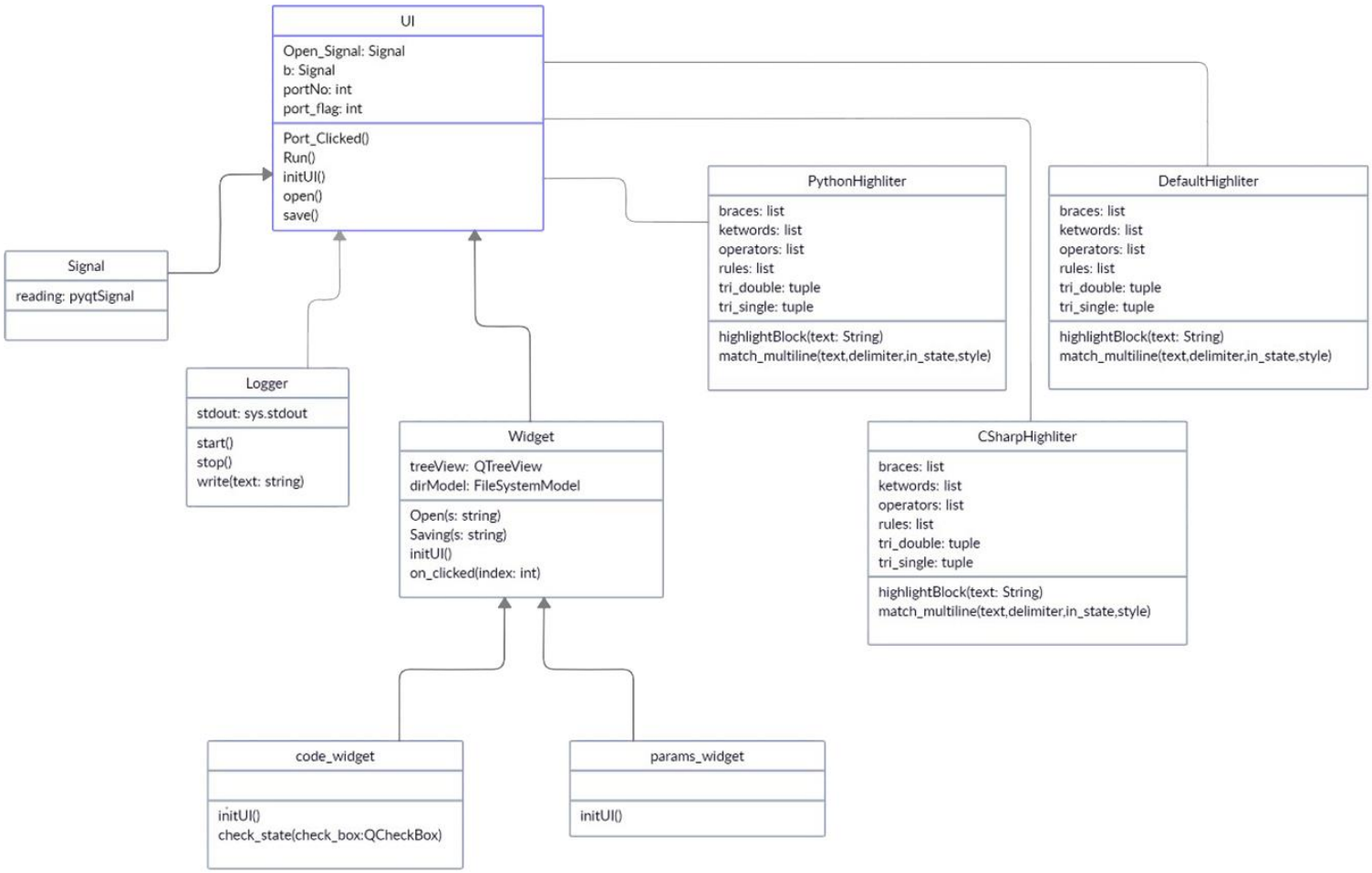


Figure 6: Class diagram

UI class:

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

Signal class:

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

Logger class:

- This class is responsible for storing the standard output of the function being run so it can be displayed later in the output window.
- start() is used to start the logger.
- stop() is used to stop the logger.
- write() is used to write stdout into the output window.

Widget class:

- This class is acts as a base class for code_widget and params_widget classes.
- initUI() The Open() and on_clicked() are used to define a new slot and take string to populate the text editor with.
- Saving() is used to define a new slot to save the text in the text editor.
- treeview is used to show directory files and dirModel is used to make a file system variable.

code_widget class:

- This class is responsible for the code editor tab.
- initUI() is used to define a new slot and take string to populate the text editor with.
- check_state() is used to check the state of the 'Has parameters' checkbox which is used to run functions with/without parameters.

params_widget class:

- This class is responsible for the parameters tab.
- initUI() is used to define a new slot and take string to add the parameters text editor.

PythonHighlighter , CSharpHighligther and DefaultHighlighter classes:

- These classes are responsible for highlighting the code according to the syntax.
- keywords list has the keywords of python in **PythonHighlighter** and keywords of C# in **CSharpHighligther**. Unsupported extensions follow **DefaultHighlighter**.
- operators list has the operators as (+, -, *, /)
- highlightBlock() is used to apply the syntax highlighting to the given block of text.
- match_multiline() is used for 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.

4. Code & examples

4.1. Source code

Anubis.py

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

# pyqt imports
import Python_Coloring
import CSharp_Coloring
import Default_coloring
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

# other necessary imports
import sys
import os
import glob
import serial
import time


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):
    """The Signal class provides a way to declare and connect Qt signals in a pythonic way."""
    # 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)


# Logger Class
class Logger():
    """The Logger class is used to capture stdout so it can be displayed later in the output window."""
    stdout = sys.stdout
    messages = []

    def start(self):
        sys.stdout = self

    def stop(self):
        sys.stdout = self.stdout

    def write(self, text):
        self.messages.append(text)


# Logger object
log = Logger()


# code editor and output fields
```

```
editor_code = QTextEdit
text_output = QTextEdit

# initializing filename and language (should be assigned a value when a file is opened or clicked)
filename = None
language = 'Not specified'


# code_widget class
class code_widget(QWidget):
    """The code_widget class is responsible for the main code text editor."""

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

    def initUI(self):
        global editor_code
        editor_code = QTextEdit()
        vbox = QVBoxLayout()
        hbox = QHBoxLayout()
        vbox.addLayout(hbox)

        # checkbox for using parameters or not
        has_params = QCheckBox("has parameters?")
        has_params.setChecked(True)
        self.has_params = True
        has_params.stateChanged.connect(lambda: self.check_state(has_params))

        # displaying selected language
        self.lang_label = QLabel()
        self.lang_label.setText(f"Language: {language}")
        self.lang_label.setAlignment(
            QtCore.Qt.AlignRight | QtCore.Qt.AlignVCenter)

        # highlighting
        if language == 'Python':
            Python_Coloring.PythonHighlighter(editor_code)
            self.lang_label.setStyleSheet("color: #0a8216")
        elif language == 'C#':
            CSharp_Coloring.CSharpHighlighter(editor_code)
            self.lang_label.setStyleSheet("color: #0a8216")
        else:
            Default_coloring.DefaultHighlighter(editor_code)
            self.lang_label.setStyleSheet("color: #a32a0b")
```

```
        # highlighting
        if language == 'Python':
            Python_Coloring.PythonHighlighter(editor_code)
            self.lang_label.setStyleSheet("color: #0a8216")
        elif language == 'C#':
            CSharp_Coloring.CSharpHighlighter(editor_code)
            self.lang_label.setStyleSheet("color: #0a8216")
        else:
            Default_coloring.DefaultHighlighter(editor_code)
            self.lang_label.setStyleSheet("color: #a32a0b")

        # adding to the hbox
        hbox.addWidget(has_params)
        hbox.addWidget(self.lang_label)

        # editor
        vbox.addWidget(editor_code)
        self.setLayout(vbox)

    # the function associated with the 'has parameters?' checkbox
    def check_state(self, check_box):
        """this function enables for tracking the state of the 'has parameters?' checkbox
        to be used later while running functions whith/without parameters."""
        self.has_params = check_box.isChecked()


# params_widget class
class params_widget(QWidget):
    """The params_widget class is responsible for the parameters text editor."""

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

    def initUI(self):
        global editor_params
        editor_params = QTextEdit()
        hbox = QHBoxLayout()
        hbox.addWidget(editor_params)
        self.setLayout(hbox)


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

```
    """The Widget class is responsible for combining different widgets to form the final layout of the
applications."""

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

    def initUI(self):
        # code editor and parameters tabs
        tabs = QTabWidget()
        self.code_editor = code_widget()
        params_editor = params_widget()
        tabs.addTab(self.code_editor, "Code")
        tabs.addTab(params_editor, "Parameters")

        # output text field (for displaying output/errors/hints)
        global text_output
        text_output = QTextEdit()
        text_output.setReadOnly(True)

        # treeview for showing the directory included files
        self.treeview = QTreeView()

        # path for current directory
        path = QDir.currentPath()

        # making a Filesystem variable, setting its root path and applying somefilters (which I need) on it
        self.dirModel = QFileSystemModel()
        self.dirModel.setRootPath(QDir.rootPath())

        # NoDotAndDotDot => Do not list the special entries "." and "..".
        # AllDirs =>List all directories; i.e. don't apply the filters to directory names.
        # Files => List files.
        self.dirModel.setFilter(QDir.NoDotAndDotDot |
                                QDir.AllDirs | QDir.Files)
        self.treeview.setModel(self.dirModel)
        self.treeview.setRootIndex(self.dirModel.index(path))
        self.treeview.clicked.connect(self.on_clicked)

        # left and right boxes
        Left_hbox = QHBoxLayout()
        Right_hbox = QHBoxLayout()

        # Adding treeview to the left box and code editor to the right one
```

```
    Left_hbox.addWidget(self.treeview)
    Right_hbox.addWidget(tabs)

    # arranging left/right layouts
    Left_hbox_Layout = QWidget()
    Right_hbox_Layout = QWidget()
    Left_hbox_Layout.setLayout(Left_hbox)
    Right_hbox_Layout.setLayout(Right_hbox)

    # splitter for seperating left and right layouts and make it easier to change the spacing between them
    H_splitter = QSplitter(Qt.Horizontal)
    H_splitter.addWidget(Left_hbox_Layout)
    H_splitter.addWidget(Right_hbox_Layout)
    H_splitter.setStretchFactor(1, 1)

    # splitter for seperating the upper and lower sides of the window
    V_splitter = QSplitter(Qt.Vertical)
    V_splitter.addWidget(H_splitter)
    V_splitter.addWidget(text_output)

    # final layout
    Final_Layout = QHBoxLayout(self)
    Final_Layout.addWidget(V_splitter)
    self.setLayout(Final_Layout)

# slot for saving the text inside the code editor into main.py file
@pyqtSlot(str)
def Saving(s):
    if language == 'Python':
        dest_filename = 'saved_file.py'
        with open(dest_filename, 'w') as f:
            TEXT = editor_code.toPlainText()
            f.write(TEXT)
            text_output.append(f'saved python file to {dest_filename}')
    elif language == 'C#':
        dest_filename = 'saved_file.cs'
        with open(dest_filename, 'w') as f:
            TEXT = editor_code.toPlainText()
            f.write(TEXT)
            text_output.append(f'saved C# file to {dest_filename}')
    else:
        text_output.append('Unsupported file extension!')

# slot for opening a file and loading its contents to the code editor
```

```
    @pyqtSlot(str)
    def Open(s):
        global editor_code
        editor_code.setText(s)

    # function for handling clicking on a file in the treeview
    def on_clicked(self, index):
        """this function handles clicking on a file in the treeview
        it is responsible for two tasks:

        - loading and displaying the file content to the code editor.
        - updating the global variables filename and language."""

        global language, filename
        filepath = self.sender().model().filePath(index)
        filepath = tuple([filepath])

        if filepath[0]:
            try:
                f = open(filepath[0], 'r')
                filename = filepath[0].split('/')[-1]
                language = 'Python' if '.py' in filename else 'C#' if '.cs' in filename else 'Unsupported'
                self.code_editor.lang_label.setText(f"Language: {language}")
                with f:
                    data = f.read()
                    editor_code.setText(data)

                # highlighting
                if language == 'Python':
                    Python_Coloring.PythonHighlighter(editor_code)
                    self.code_editor.lang_label.setStyleSheet("color: #0a8216")
                elif language == 'C#':
                    CSharp_Coloring.CSharpHighlighter(editor_code)
                    self.code_editor.lang_label.setStyleSheet("color: #0a8216")
                else:
                    Default_coloring.DefaultHighlighter(editor_code)
                    self.code_editor.lang_label.setStyleSheet("color: #a32a0b")
            except Exception as e:
                pass


# slot that is connected to the Widget class (reading)
# it takes the (input string) and establishes a connection with the widget class, sends the string to it
@pyqtSlot(str)
def reading(s):
```

```
    b = Signal()
    b.reading.connect(Widget.Saving)
    b.reading.emit(s)


# slot that is connected to the Widget class (opening)
# it takes the (input string) and establishes a connection with the widget class, sends the string to it
@pyqtSlot(str)
def opening(s):
    b = Signal()
    b.reading.connect(Widget.Open)
    b.reading.emit(s)


# UI Class
class UI(QMainWindow):
    """The UI class is responsible for the main window
    and contains the functionality for running python/C# code."""

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

    def intUI(self):
        self.port_flag = 1
        self.read_signal = Signal()
        self.open_signal = Signal()

        # connecting (self.open_signal) with opening function
        self.open_signal.reading.connect(opening)

        # connecting (self.read_signal) with reading function
        self.read_signal.reading.connect(reading)

        # creating menu items
        menu = self.menuBar()
        filemenu = menu.addMenu('File')
        Port = menu.addMenu('Port')
        Run = menu.addMenu('Run')

        # displaying ports 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)

# creating run actions
RunAction = QAction("Run", self)
RunAction.triggered.connect(self.Run)
RunAction.setShortcut("Ctrl+R")
# adding action
Run.addAction(RunAction)

# creating file handling actions
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)
# adding actions
filemenu.addAction(Save_Action)
filemenu.addAction(Close_Action)
filemenu.addAction(Open_Action)

# Seting the window Geometry
self.setGeometry(200, 150, 800, 500)

# centering the window
qtRectangle = self.frameGeometry()
centerPoint = QDesktopWidget().availableGeometry().center()
qtRectangle.moveCenter(centerPoint)
self.move(qtRectangle.topLeft())

# window title and icon
self.setWindowTitle('Anubis IDE')
self.setWindowIcon(QtGui.QIcon('Anubis.png'))

# Widget object
self.widget = Widget()
```

```
        self.setCentralWidget(self.widget)
        self.show()

    # Run function definition
    def Run(self):
        """Run function is responsible for running python/C# functions.

        ### procedure is as following:

        - the user opens a python/C# file that contains a function difinition.
        - the content of the file is loaded to the code editor.
        - the `has prameters?` checkbox is checked by default assuming the function requires parameters.
        - if the function requires parameters, the user shoud add them in the `parameters` tab.
            - parameters should be added such that there is one parameter per line.
            - for python functions, keyword parameters can be used (one per line).
            - for more instructions, please follow the screenshots in the `documentation` pdf.
        - if the function does not require parameters, the user should simply uncheck the `has prameters?`
checkbox.
        - the user can then run the code by clicking Run in the menu or using the shortcut `Ctrl+R`.
        - output, errors or hints should be displayed in the output field.
        - some exception handling was done, eg. displaying a hint when the user forgets to enter parameters.

        ### for the code to run, these steps are followed:
        #### python functions:
        - the code is excecuted right away by means of `exec` function.

        #### C# functions:
        - the function is wrapped inside a main function from the template `main.cs`.
        - the combination of the two files is saved to a temporary file `temp_code_runner.cs`
        - the code is excecuted by means of the `os` module as follows:
            - `csc temp_code_runner.cs` command is excecuted followed by `temp_code_runner`"""

        if self.port_flag == 0:
            func_text = editor_code.toPlainText()
            """Compiler Part"""
            text_output.append("Sorry, there is no attached compiler.")

        else:
            # executing the function
            # starting the logger to capture output
            log.start()
            try:
                # python file
                if language == 'Python':
```

```
        try:
            # function  input text by user
            func_text = editor_code.toPlainText()
            if func_text.strip() == '':
                raise Exception("function definition missing!")
            elif 'def' not in func_text:
                raise Exception("Bad function definition!")

            # parsing function definition
            funcname_2_end = func_text.replace('def', '').strip()
            funcname = funcname_2_end[:funcname_2_end.index(
                '(')].strip()

            # executing function definition
            exec(func_text)

            # executing function call
            if self.widget.code_editor.has_params:
                # parameters input text by user
                params_text = editor_params.toPlainText()
                joined_params = ','.join(line.strip()
                                         for line in params_text.split('\n') if line)
                if joined_params == '':
                    raise Exception(
                        "parameters missing while has parameters is checked!")

                # executing function with given parameters
                exec(f'{funcname}({joined_params})')
            else:
                # executing parameterless function
                exec(f'{funcname}()')
        except ValueError:
            print('Bad function definition!')

    # csharp file
    elif language == 'C#':
        try:
            # reading main.cs (a fixed template that's only usable combined with a function)
            with open('main.cs', 'r') as f:
                main_content = f.read()

            # deleting old temp_code_runner.cs/temp_code_runner.exe files if exist
            if 'temp_code_runner.cs' in ''.join(glob.glob("./*")):
                os.remove("temp_code_runner.cs")
```

```
        if 'temp_code_runner.exe' in ''.join(glob.glob("./*")):
            os.remove("temp_code_runner.exe")

        # saving code to temp file
        with open('temp_code_runner.cs', 'w') as f:
            TEXT = editor_code.toPlainText()
            # adding function definition
            main_content = main_content.replace(
                'function_definition_placeholder', TEXT)
            # getting function name
            funcname = TEXT[:TEXT.index('(')].split()[-1]
            # adding function call
            if self.widget.code_editor.has_params:
                # parameters input text by user
                params_text = editor_params.toPlainText()
                joined_params = ','.join(line.strip()
                                         for line in params_text.split('\n') if line)
                if joined_params == '':
                    raise Exception(
                        "parameters missing while has parameters is checked!")
                main_content = main_content.replace(
                    'function_call_placeholder', f'{funcname}({joined_params})')
            else:
                main_content = main_content.replace(
                    'function_call_placeholder', f'{funcname}()')
            f.write(main_content)
            time.sleep(0.1)

        # compiling the .cs file
        os.popen(f'csc temp_code_runner.cs')

        # waiting a bit for the combiler to finish
        time.sleep(0.4)

        # running the .cs file and capture the stdout
        if 'temp_code_runner.exe' in ''.join(glob.glob("./*")):
            cs_output = os.popen('temp_code_runner').read()
        else:
            cs_output = 'C# compile error\n'

        # adding output to the logger
        log.messages.append(cs_output)
    except Exception as e:
        print(e)
```

```
            # no file selected
            elif filename == None:
                print('no file selected!')
            # unsupported language
            else:
                print('Unsupported language!')
        except Exception as e:
            print(e)

        # stopping the logger
        log.stop()

        # adding logs to the output window text
        text_output.append(''.join(log.messages))
        log.messages = []

# 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

# function for saving the code into a file
def save(self):
    self.read_signal.reading.emit("name")

# function for opening a file and exhibits it to the user in a text editor
def open(self):
    global language, filename
    filepath = QFileDialog.getOpenFileName(
        self, 'Open File', '.')

    if filepath[0]:
        try:
            f = open(filepath[0], 'r')
            filename = filepath[0].split('/')[-1]
            language = 'Python' if '.py' in filename else 'C#' if '.cs' in filename else 'Unsupported'
            self.widget.code_editor.lang_label.setText(
                f"Language: {language}")
            with f:
                data = f.read()
            self.open_signal.reading.emit(data)
        except Exception as e:
```

```
                pass


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

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
```

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
```

main.cs template

```
using System;


public class MainClass
{
    public static void Main()
    {
        // function call
        function_call_placeholder;
    }


    function_definition_placeholder
}
```

4.2. Function file examples

Case 1: python function with parameters
func.py

```
def print_sum(x, y):
    print(x+y)
```

Case 2: python function without parameters
func.py

```
def say_hellp():
   print('hello')
```

Case 3: C# function with parameters
func.cs

```
static void Square(int i)
{
    // calculating the square of the input
    int input = i;
    int product = input * input;

    // printing the result
    Console.WriteLine("Result:");
    Console.WriteLine(product);
}
```

Case 4: C# function without parameters
func.cs

```
static void Say_hello()
{
    Console.WriteLine("Hello");
}
```

5. Screenshots

1. Main screen of the app

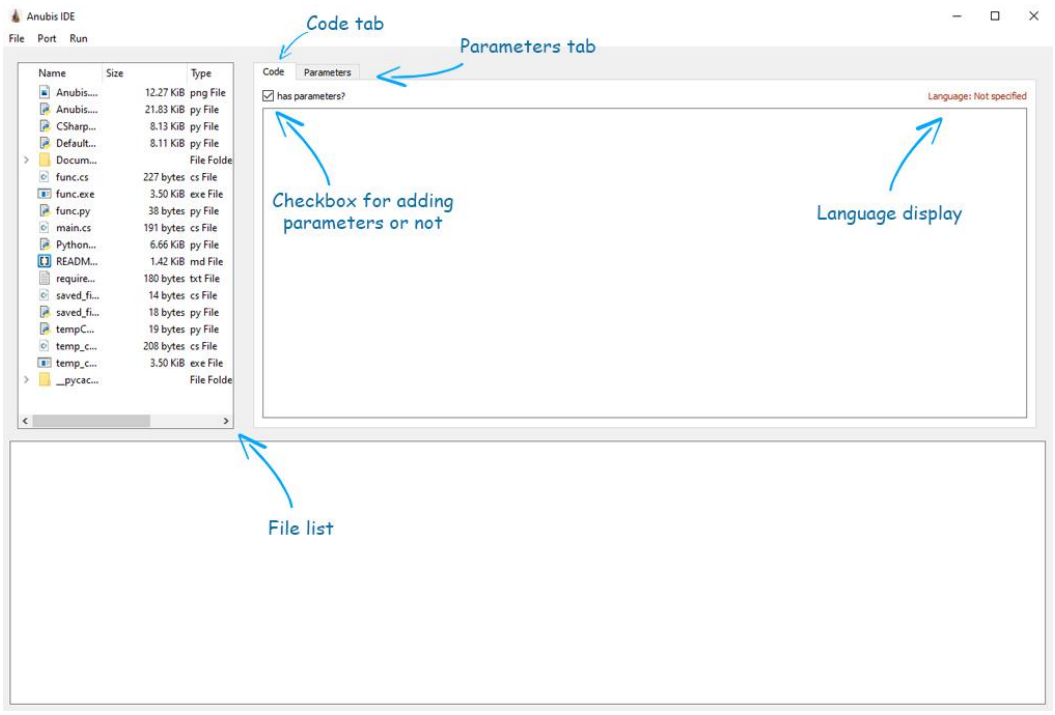


Figure 7: Main screen

2. Opening a file with unsupported extension

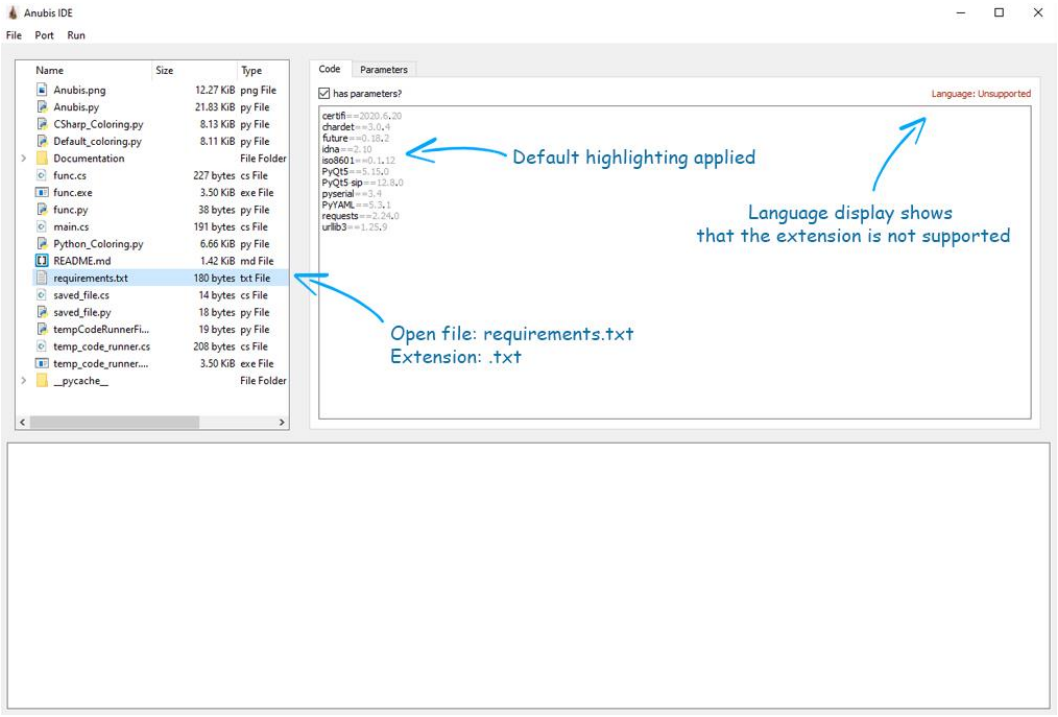


Figure 8: Opening a file with unsupported extension

3. Opening a file with supported extension (.py)

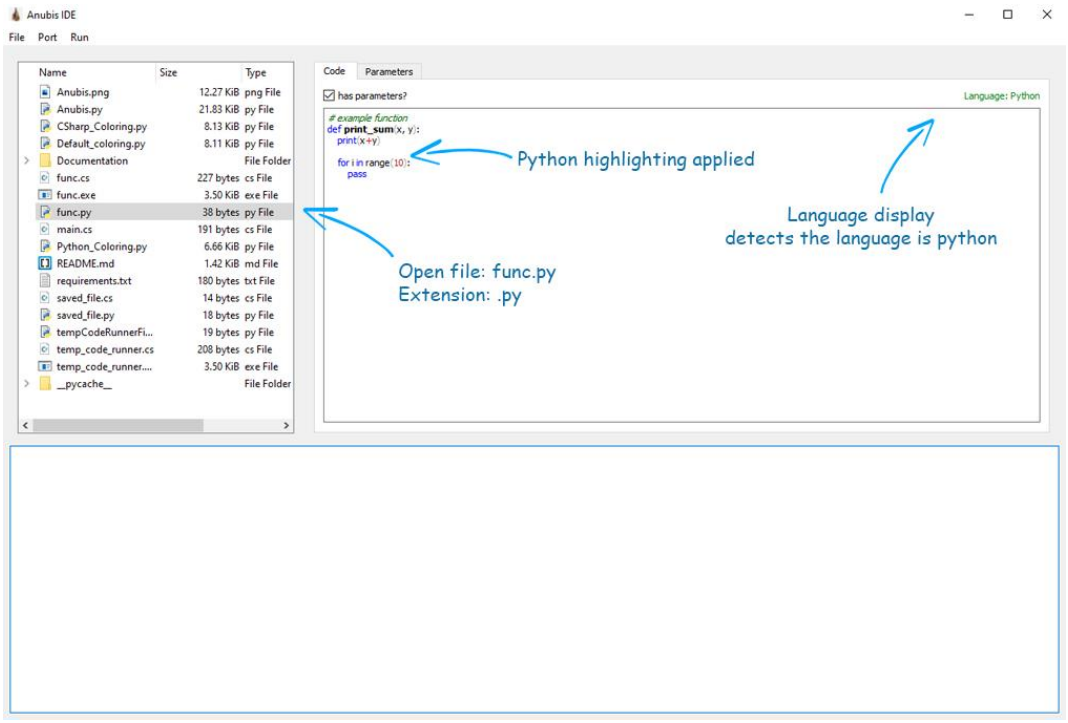


Figure 9: Opening a file with supported extension (.py)

4. Opening a file with supported extension (.cs)

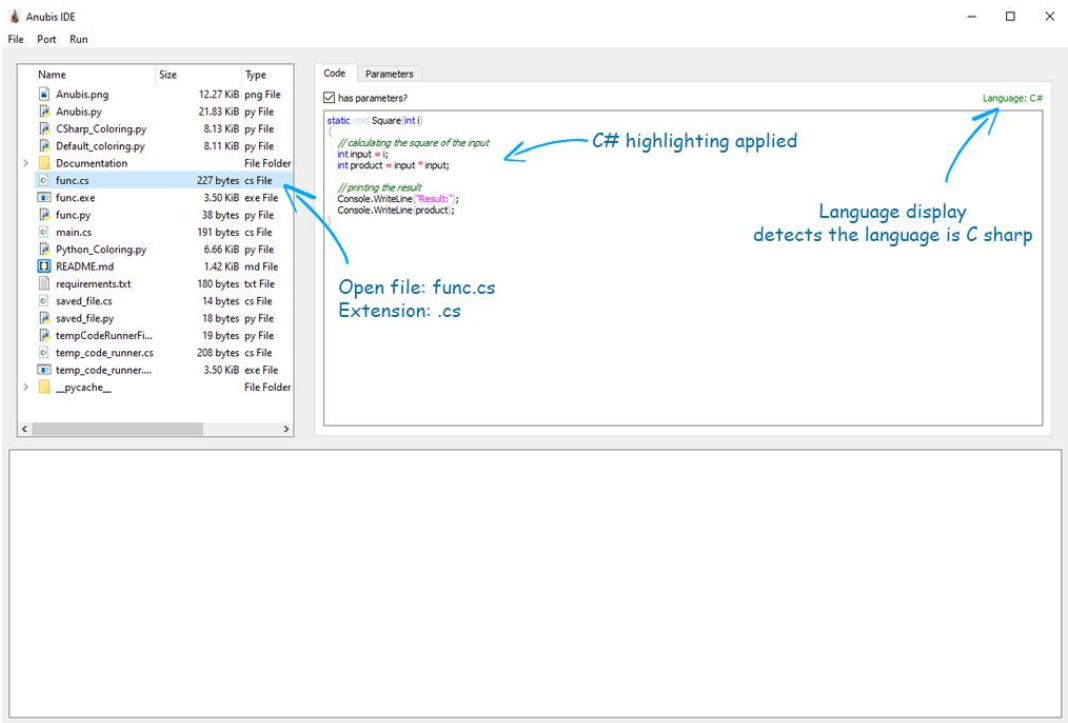


Figure 10: Opening a file with supported extension (.cs)

5. Opening a file

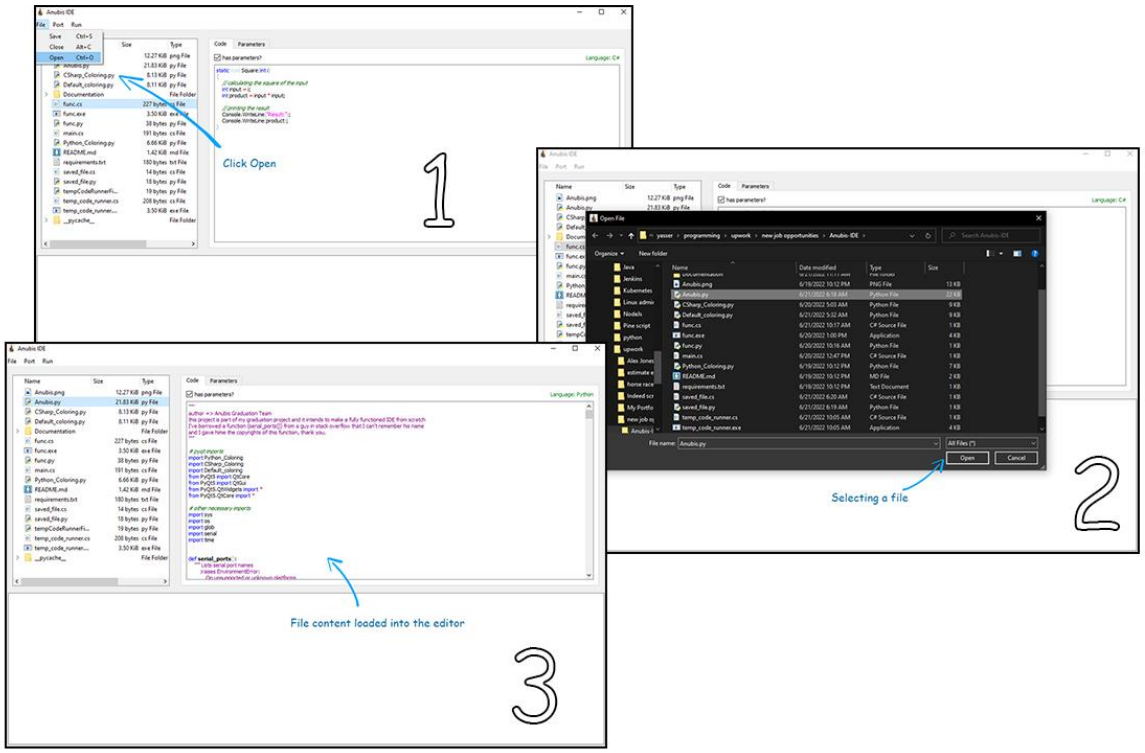


Figure 11: Opening a file

6. Saving a python file to saved_file.py

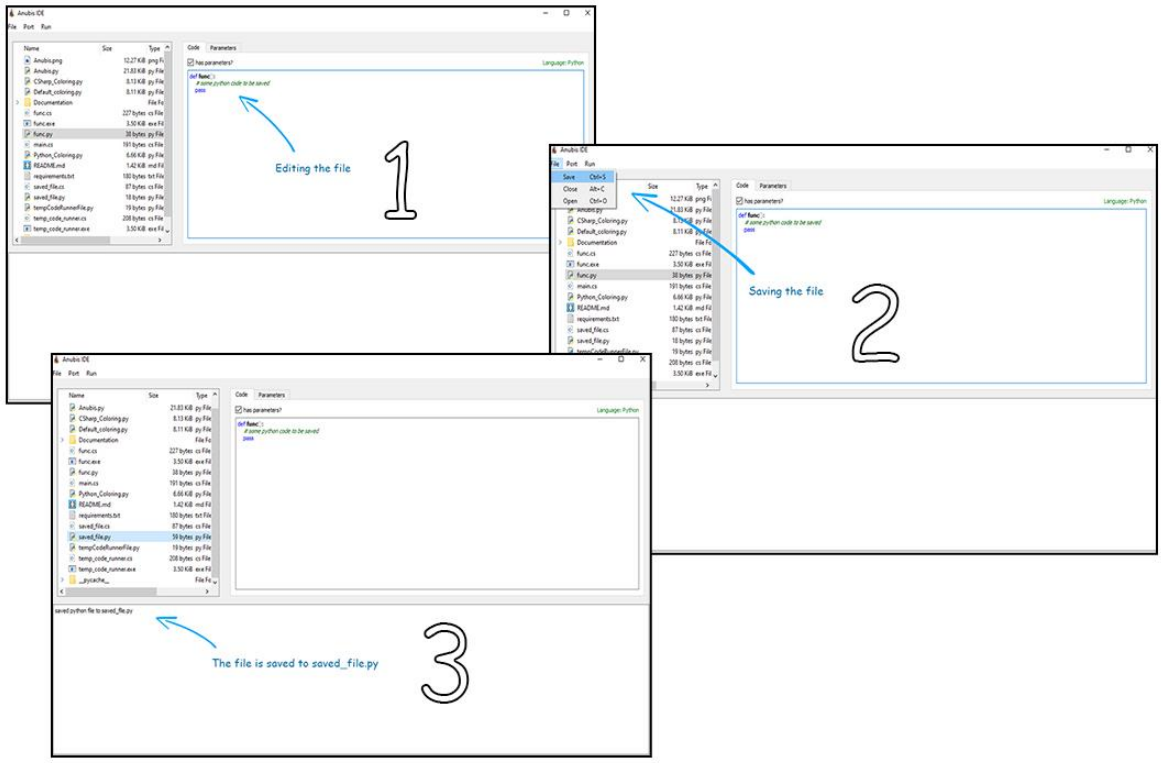


Figure 12: Saving a python file to saved_file.py

7. Saving a C# file to saved_file.cs

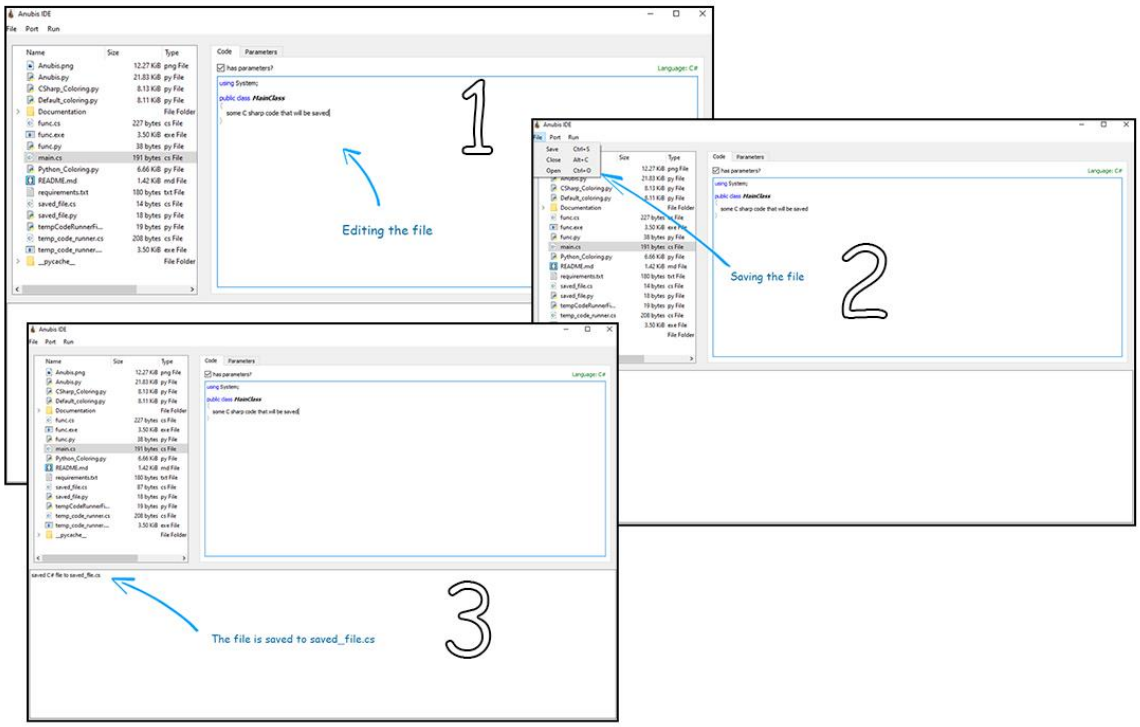


Figure 13: Saving a C# file to saved_file.cs

8. Choosing a port

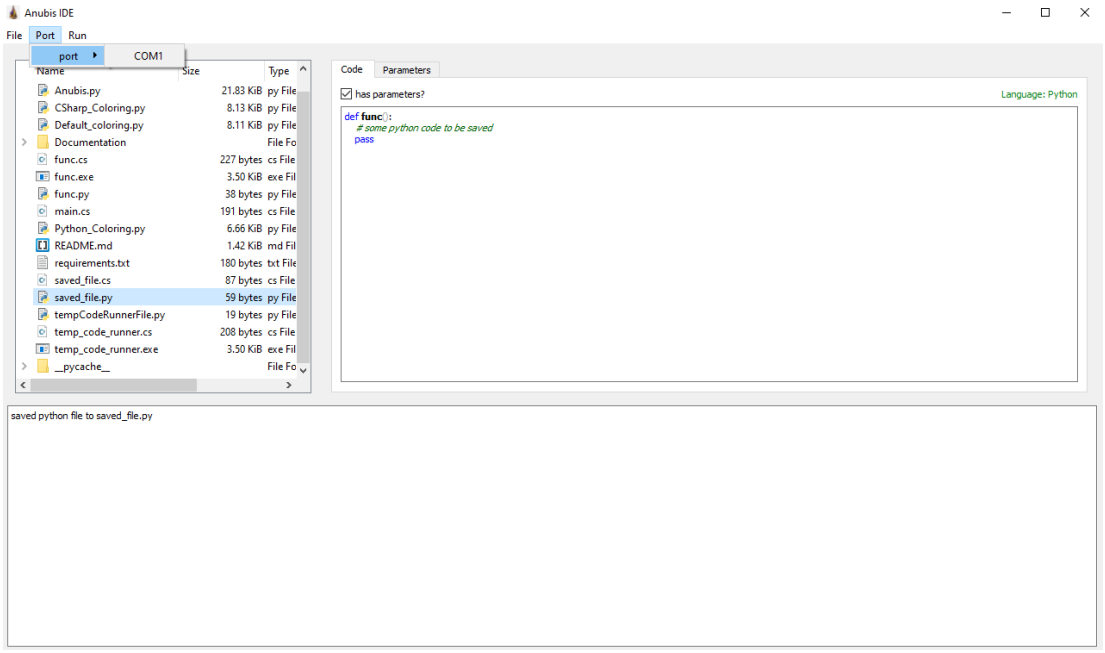


Figure 14: Choosing a port

9. Running a python function (without parameters)

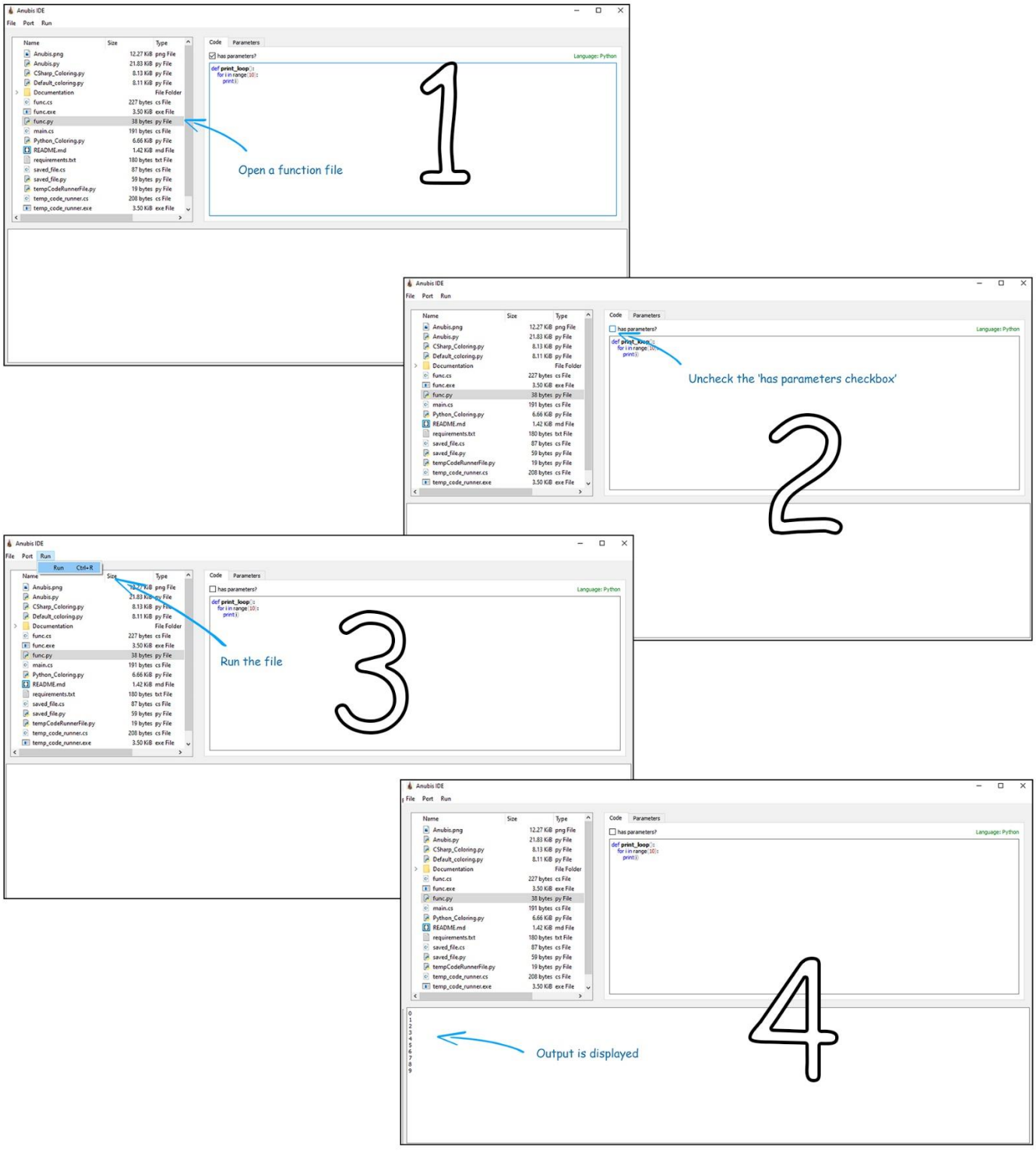


Figure 15: Running a python function (without parameters)

10. Running a python function (with parameters)

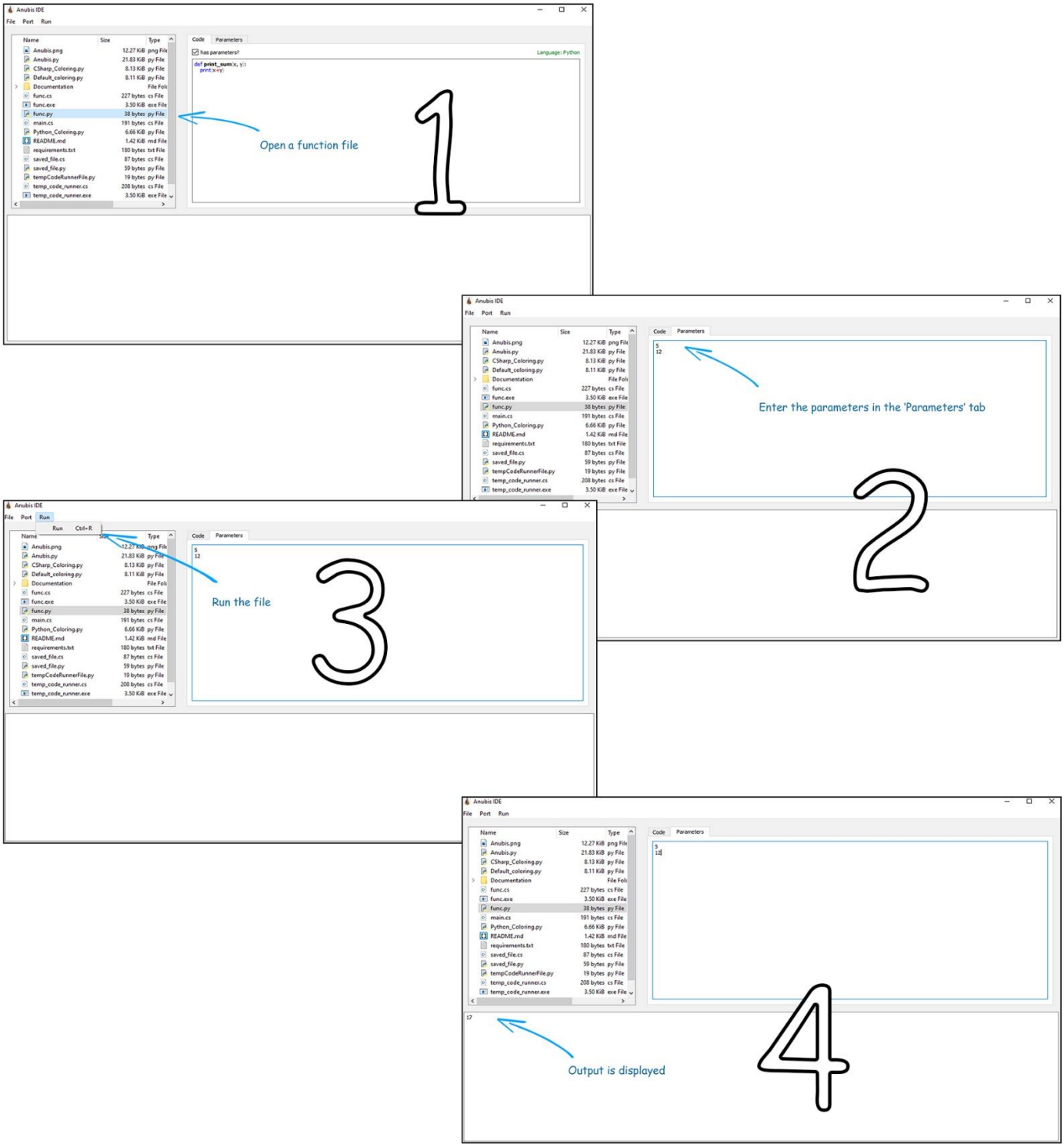


Figure 16: Running a python function (with parameters)

C# dependencies:

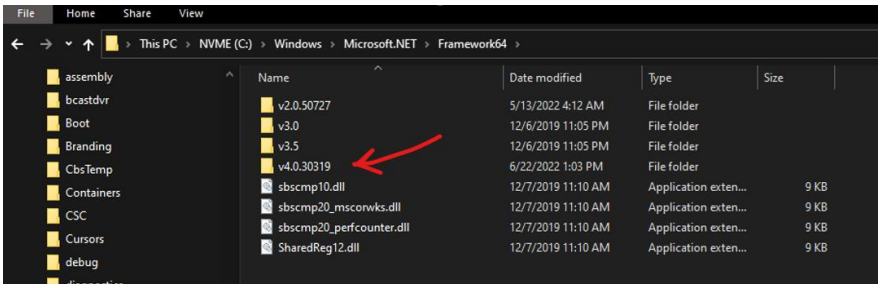
To be able to run C# functions, the following dependencies must be satisfied:

- Microsoft .NET SDK installed on the system.

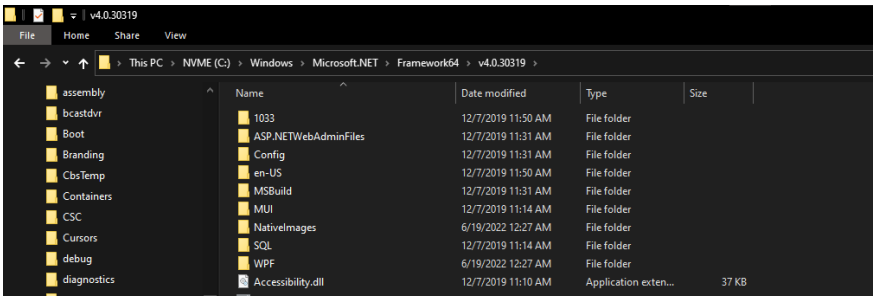
  Can be downloaded on https://dotnet.microsoft.com/en-us/download

- Microsoft csc compiler directory added to your path:

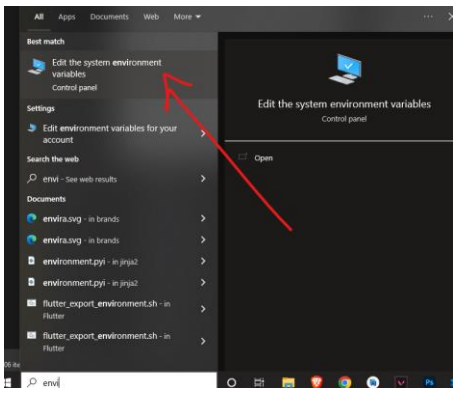
  1. Go to C:\Windows\Microsoft.NET\Framework64\

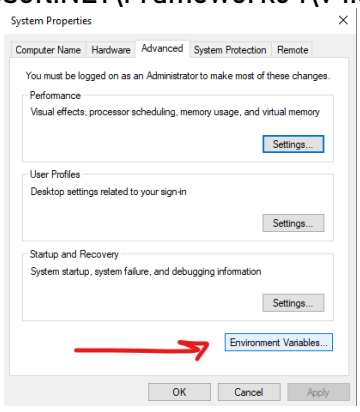


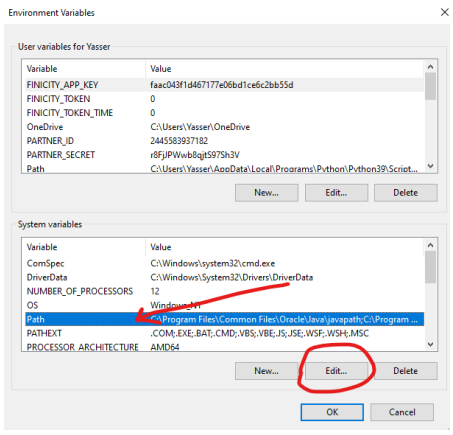
  2. Open the version you desire (e.g. v4.0.30319)

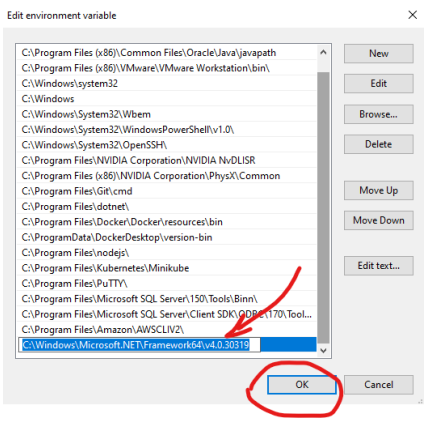


  3. Add 'C:\Windows\Microsoft.NET\Framework64\v4.0.30319' to your Path variable









11. Running a C# function (without parameters)

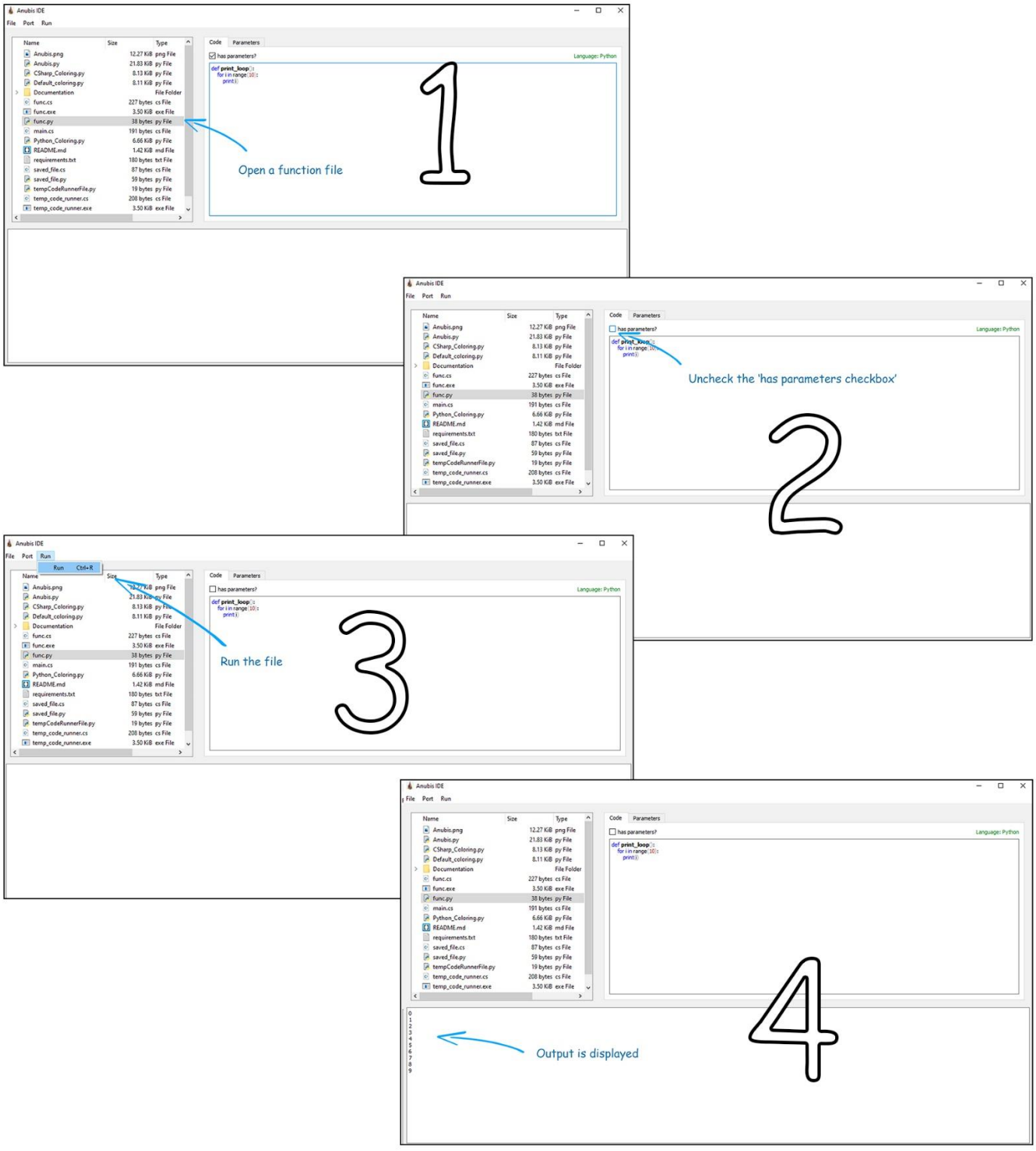


Figure 17: Running a C# function (without parameters)

12. Running a C# function (with parameters)

Figure 18: Running a C# function (with parameters)