PyQt5 Tutorial: A Window Application with File IO

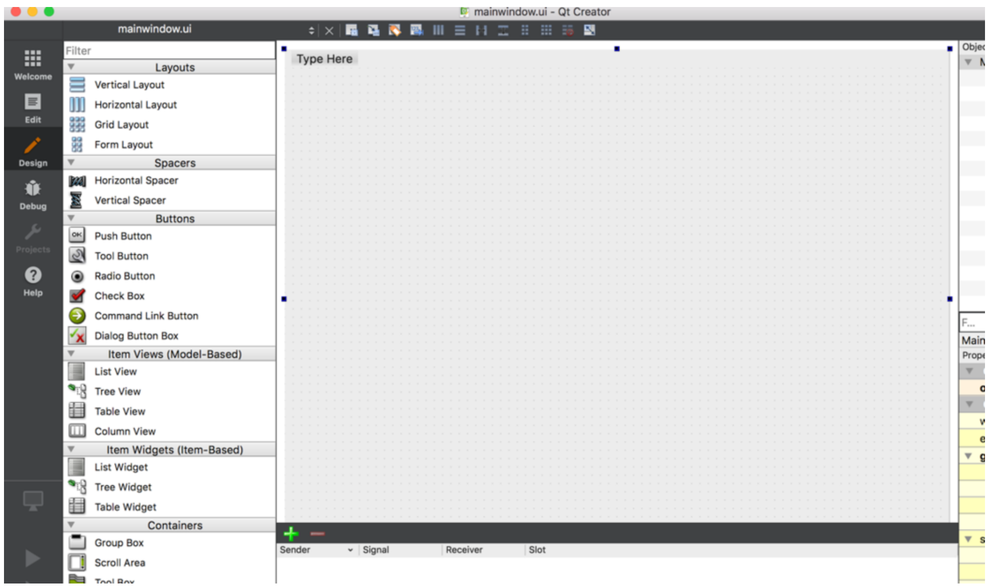
D. Thiebaut (talk) 09:04, 31 July 2018 (EDT)

In this tutorial we create a simple application with a Graphical User Interface (GUI) implemented with Qt5's **Qt Creator**, and code the whole application as a simple Model-View-Controller example (MVC) implemented in Python Version 3.5. The application uses a dialog to browse for a file, opens it, displays it contents in a text-edit widget, allows the user to edit the contents in the same text edit, and finally gives the user the option of saving the file to a new copy with a ".bak" suffix. We use PyInstaller to create an app that is self-contained and can be distributed to others for execution.

Contents

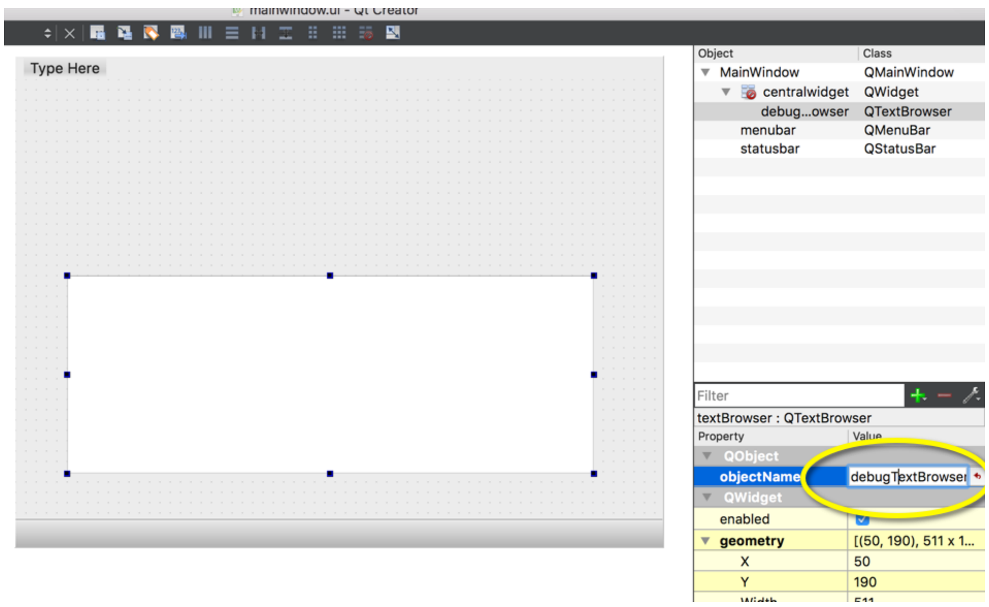
Create A Window UI

- Qt Creator
- New File or Project
- Qt
- Qt Designer Form
- Main Window
- mainwindow.ui
- Pick a directory to host the project
- Accept defaults
- Start Editing the UI

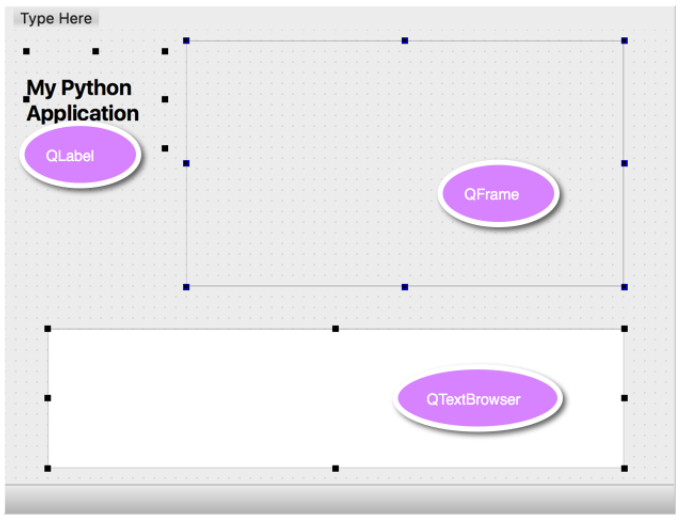


Add Widgets

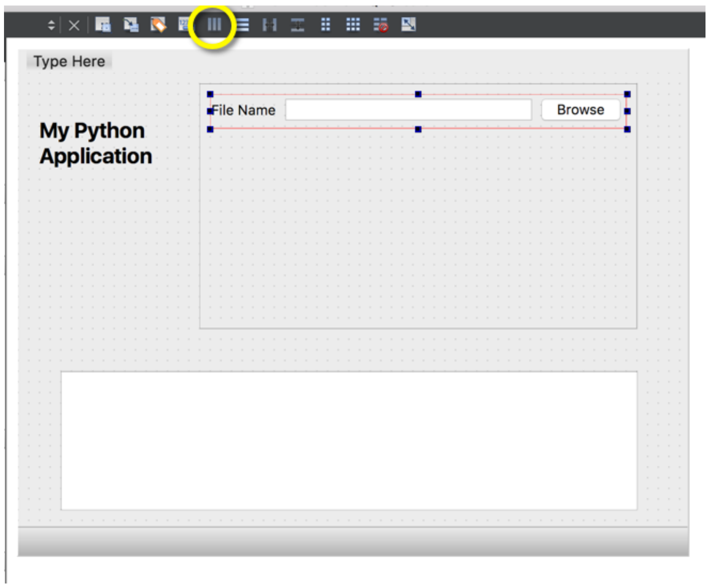
- Add a QTextBrowser and put it at the bottom of the window. Rename it debugTextBrowser. We will use it to display debugging information.



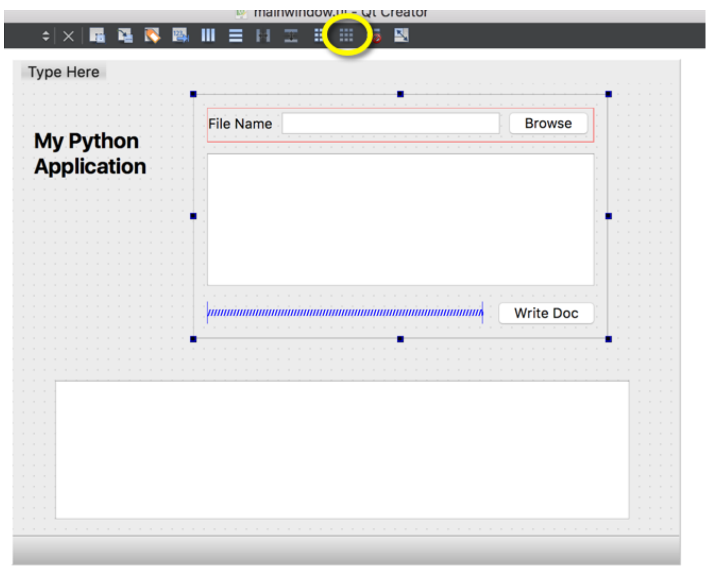
- Add a QLabel in the top right corner of the window. Double click on it to edit its caption, and enter "My Python Application" (or another name of your choice).
  - In the lower right corner of the Qt-Creator window, in the QLabel properties, locate the **Text Wrap** option and click it. This will make the text of your label wrap in the space allocated.
  - In the **font** area of the QLabel properties, locate the **Point Size** and set it to 20. Locate the **Bold** button and clic it.
  - In the same properties area, locate the **maximumSize** area, and set the **Width** to 130 (or some width that will allow all the words of your caption to appear without truncation.
- Add a **QFrame** in the space to the right of the label you just created.



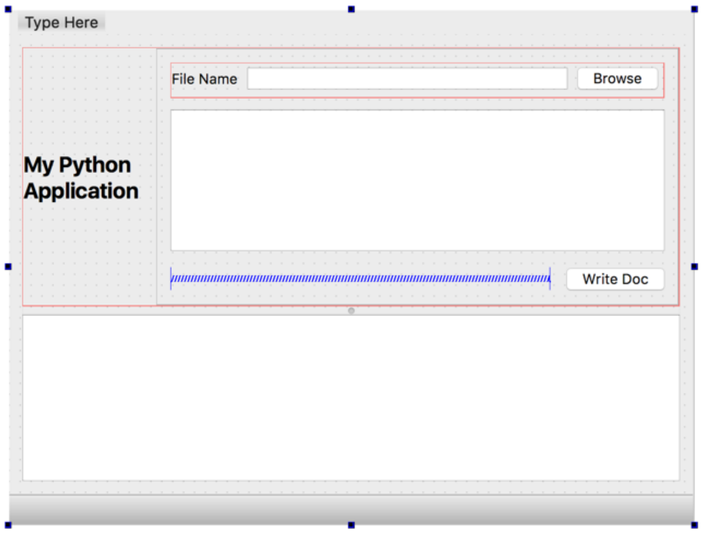
- Add a QLabel, a QLineEdit, and a QPushButton inside the QFrame you just added. Click on the label to set its caption to "File Name." Click on the push-button to set its caption to "Browse."
- Select the label, lineEdit, and pushButton and click on the Horizontal Layout button (3 vertical blue bars) to force them to stay aligned horizontally, next to each other. A red layout box will encapsulate the three widget.



- Add a QTextEdit inside the frame, below the horizontal layout just created.
- Add a QPushButton at the bottom of the frame. Place a horizontal spacer next to it, on the left.
- Click on the QFrame that contains the 6 widgets just added, and click on the grid layout icon, at the top of Qt Creator.

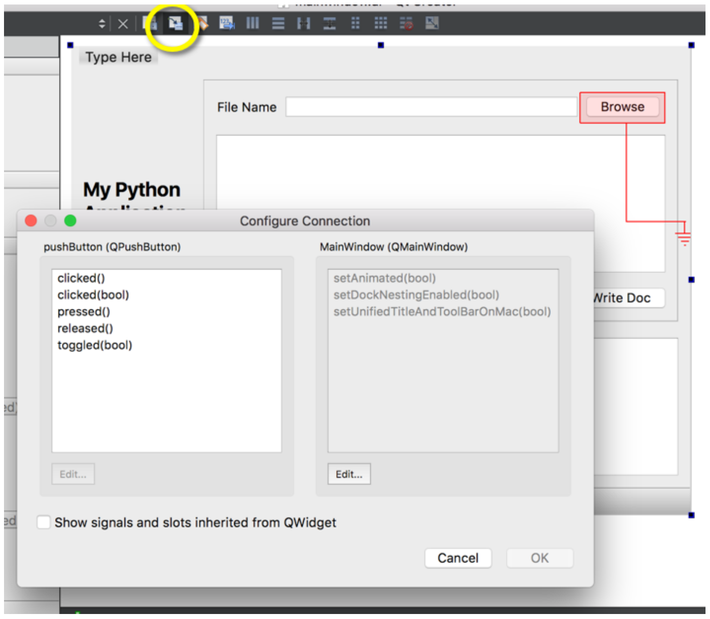


- Select the frame and the label to its left and click on the **Horizontal Layout** icon. This packages them in a horizontal red rectangle.
- Click on this red rectangle to select it, and click on the bottom text-browser to select it as well. Click on the **Layout Vertical with Splitter** icon to join the two selected rectangular areas.
- Click on the main window, the largest block in the design window, and click on the **Grid Layout** icon. The main window is now ready with its geometry.

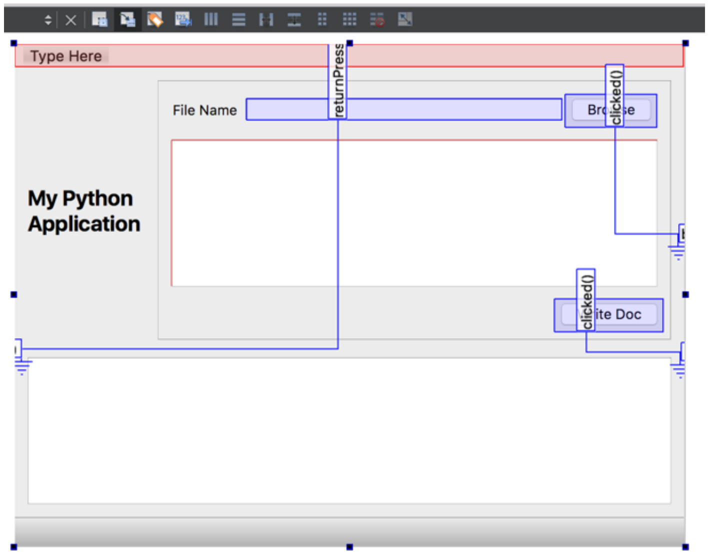


Connect Signals to Slots

- Switch to the Signals & Slots editing mode by clicking on the **Edit Signals/Slots** icon at the top of the Qt Creator window.
- Click on the **Browse** push-button and drag the red line to the main-window background. A "ground" symbol should appear. Release the mouse. In the **Configure Connection** window that appears, click on **clicked()** in the left pane.



- Click the "Edit" button under the right pane to create a new slot.
- In the new window that appears, under the slot pane, click on the + button to add a slot, and replace the default **slot1()** by **browseSlot()**. Click OK. Select the **clicked** signal and the **browseSlot()** slot, and click **Ok**. You should now have a red line linking the Browse button to the main window.
- Similarly, connect the **clicked** signal of the "Write Doc" button to a new slot called **writeDocSlot()**.
- Similarly, connect the **return Pressed** signal of the QLineEdit widget to the main window, to a slot you will create and name **returnedPressedSlot()**. This slot will be triggered when you enter a new file name in the line edit, and press Enter to validate it.



Conversion of the UI file into a Python Class

- PyQt5 provides a utility to convert the ui file (mainwindow.ui) into a Python script that users PyQt5. We will use it to generate the script, and make a few edits.
- Open a terminal window, travel to the directory where your UI file is saved, and type the following command:

```
pyuic5 -x mainwindow.ui -o mainwindow.py
```

- Now use your favorite editor, open the mainwindow.py file, and make the changes highlighted in the code below (only the sections that need editing are shown). Your code will likely be slightly different from the one shown here as the order in which you organized your UI might be different from the one adopted here.

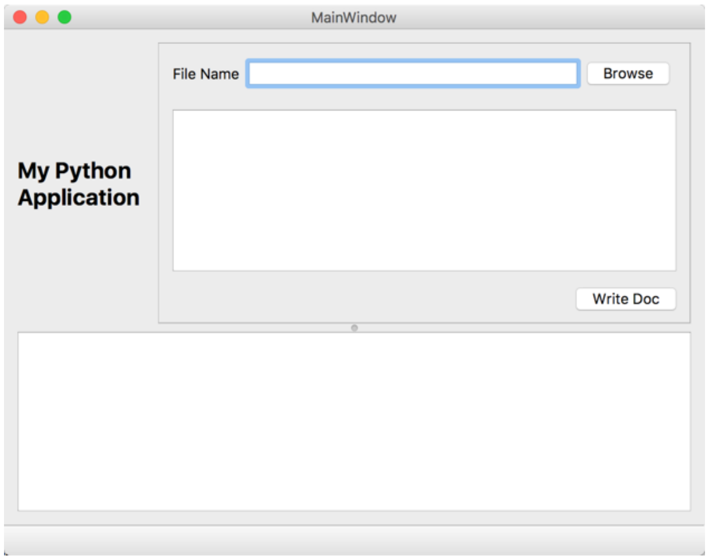
```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'mainwindow.ui'
#
# Created by: PyQt5 UI code generator 5.10.1
#
# WARNING! All changes made in this file will be lost!

from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import QObject, pyqtSlot

class Ui_MainWindow( QObject ):

        .
        .
        .

        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)

        self.retranslateUi(MainWindow)
        self.pushButton.clicked.connect( self.browseSlot)
        self.pushButton_2.clicked.connect( self.writeDocSlot)
        self.lineEdit.returnPressed.connect( self.returnPressedSlot)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
        self.label.setText(_translate("MainWindow", "My Python Application"))
        self.label_2.setText(_translate("MainWindow", "File Name"))
        self.pushButton.setText(_translate("MainWindow", "Browse"))
        self.pushButton_2.setText(_translate("MainWindow", "Write Doc"))


    @pyqtSlot( )
    def returnPressedSlot( self ):
        pass

    @pyqtSlot( )
    def writeDocSlot( self ):
        pass

    @pyqtSlot( )
    def browseSlot( self ):
        pass


if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

Testing the PyQt script

We can now test our Python script. Use the Python interpreter for which you have installed PyQt5 (in our case Python 3.5), and launch the script from the command line, or from Idle 3.5 in which you will have loaded the mainwindow.py program:

```
python3.5 mainwindow.py
```

If everything was done following the steps above, you should be able to see your app:



Adding Python Functionality with a Model-View-Controller (MVC) paradigm

The Model-View-Controller

The idea behind the Model-View-Controller paradigm is that the graphical user interface (UI) should be separate from the logic of the application, with very well defined links between the two. This makes it much easier to modify the logic of the model without having to change any code relating to the UI. Similarly, editing code relating to how the UI operates in response to user input should not influence the code controlling the logic of the model.

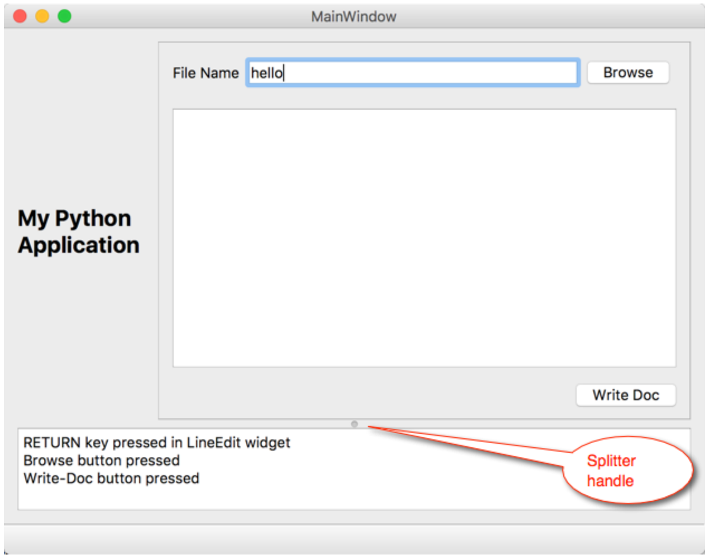
A Skeleton View Module

We are going to subclass the mainwindow.py file we have just created and add code to the subclass. The reason we will do this is simple. If at some point you need to modify the UI and either add new signals/slots connections, or add options to the menu system, or particular features, you will need to edit the UI file with Qt Creator, and convert the ui to python using the **pyuic5** command. Any additions you would have made to **mainwindow.py** would be lost with the new conversion.

So we will create a new python script with an editor or with Idle, and create a subclass of **Ui_MainWindow** which is the class defined in mainwindow.py. We will call our new script **MyApp.py**. This file will contain the class **MainWindowUIClass** which will inherit from **Ui_MainWindow**.

```
# MyApp.py
# D. Thiebaut
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import QObject, pyqtSlot
from mainwindow import Ui_MainWindow
import sys

class MainWindowUIClass( Ui_MainWindow ):
    def __init__( self ):
        '''Initialize the super class
        '''
        super().__init__()

    def setupUi( self, MW ):
        ''' Setup the UI of the super class, and add here code
        that relates to the way we want our UI to operate.
        '''
        super().setupUi( MW )

        # close the lower part of the splitter to hide the
        # debug window under normal operations
        self.splitter.setSizes([300, 0])

    def debugPrint( self, msg ):
        '''Print the message in the text edit at the bottom of the
        horizontal splitter.
        '''
        self.debugTextBrowser.append( msg )

    # slot
    def returnPressedSlot( self ):
        ''' Called when the user enters a string in the line edit and
        presses the ENTER key.
        '''
        self.debugPrint( "RETURN key pressed in LineEdit widget" )

    # slot
    def writeDocSlot( self ):
        ''' Called when the user presses the Write-Doc button.
        '''
        self.debugPrint( "Write-Doc button pressed" )

    # slot
    def browseSlot( self ):
        ''' Called when the user presses the Browse button
        '''
        self.debugPrint( "Browse button pressed" )



def main():
    """
    This is the MAIN ENTRY POINT of our application.  The code at the end
    of the mainwindow.py script will not be executed, since this script is now
    our main program.   We have simply copied the code from mainwindow.py here
    since it was automatically generated by '''pyuic5'''.

    """
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = MainWindowUIClass()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())

main()
```

- Run the program you just created, and click on the two buttons a few times. Enter some string in the line edit, and press ENTER.
- Locate the tiny circle at the bottom of the window, which represents the splitter, and lift it to reveal the text-browser. You should see some debug messages that were generated when you clicked the buttons. Click some of the buttons or press ENTER in the line-edit again to see the debug messages appear.



RETURN key pressed in LineEdit widget
Browse button pressed
Write-Doc button pressed

- This tests the functionality of our View component. We will need to add some code to finish the app, but for right now we have a good start for our app, and need to define the logic of our model, implement it, and connect model and view together.
- In the next section, we present our model, and what functionality we want to implement, and show the final code for the View and Model scripts.

The Model

We create a simple model with the following logic:

1. Let the user pick a file name, either while browsing the file system, or by specifying a file path.
2. Verify that the file is valid and can be opened. The application assumes (but not test) that the file is a text file.
3. Display the contents of the file in the TextEdit widget.
4. Allow the user to make changes to the contents of the TextEdit.
5. Allow the user to write the changes to a new file which has the same name as the original file, and suffixed with ".bak."

```
# model.py
# D. Thiebaut
# This is the model part of the Model-View-Controller
# The class holds the name of a text file and its contents.
# Both the name and the contents can be modified in the GUI
# and updated through methods of this model.
#

class Model:
    def __init__( self ):
        '''
        Initializes the two members the class holds:
        the file name and its contents.
        '''
        self.fileName = None
        self.fileContent = ""

    def isValid( self, fileName ):
        '''
        returns True if the file exists and can be
        opened.  Returns False otherwise.
        '''
        try:
            file = open( fileName, 'r' )
            file.close()
            return True
        except:
            return False

    def setFileName( self, fileName ):
        '''
        sets the member fileName to the value of the argument
        if the file exists.  Otherwise resets both the filename
        and file contents members.
        '''
        if self.isValid( fileName ):
            self.fileName = fileName
            self.fileContents = open( fileName, 'r' ).read()
        else:
            self.fileContents = ""
            self.fileName = ""

    def getFileName( self ):
        '''
        Returns the name of the file name member.
        '''
        return self.fileName

    def getFileContents( self ):
        '''
        Returns the contents of the file if it exists, otherwise
        returns an empty string.
        '''
        return self.fileContents

    def writeDoc( self, text ):
        '''
        Writes the string that is passed as argument to a
        a text file with name equal to the name of the file
        that was read, plus the suffix ".bak"
        '''
        if self.isValid( self.fileName ):
            fileName = self.fileName + ".bak"
            file = open( fileName, 'w' )
            file.write( text )
            file.close()
```

Revised View Module

We update the **MyApp.py** module with interactions with the model.

A File-Browsing dialog is added, as well as a Warning dialog if the file specified in the Line-Edit is invalid. A member variable is added to hold a reference to the model, which is implemented by the class Model, in **model.py**.

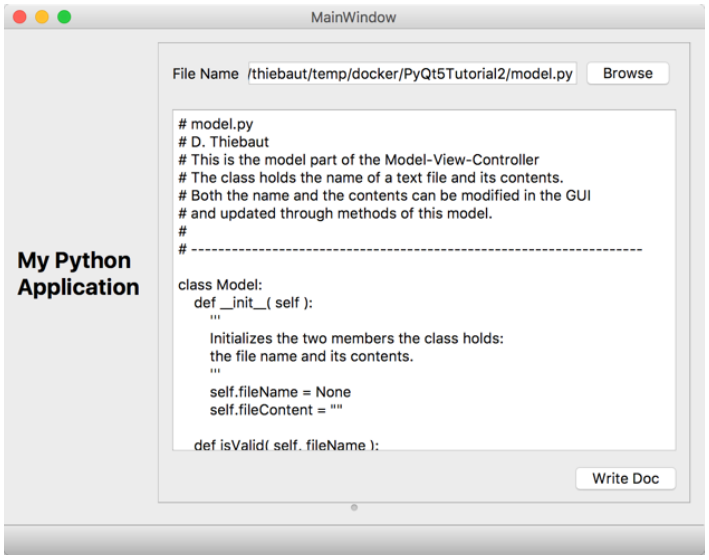
```
# MyApp.py
# D. Thiebaut
# PyQt5 Application
# Editable UI version of the MVC application.
# Inherits from the Ui_MainWindow class defined in mainwindow.py.
# Provides functionality to the 3 interactive widgets (2 push-buttons,
# and 1 line-edit).
# The class maintains a reference to the model that implements the logic
# of the app.  The model is defined in class Model, in model.py.

from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import QObject, pyqtSlot
from mainwindow import Ui_MainWindow
import sys
from model import Model

class MainWindowUIClass( Ui_MainWindow ):
    def __init__( self ):
        '''Initialize the super class
        '''
        super().__init__()
        self.model = Model()

    def setupUi( self, MW ):
        ''' Setup the UI of the super class, and add here code
        that relates to the way we want our UI to operate.
        '''
        super().setupUi( MW )

        # close the lower part of the splitter to hide the
        # debug window under normal operations
        self.splitter.setSizes([300, 0])

    def debugPrint( self, msg ):
        '''Print the message in the text edit at the bottom of the
        horizontal splitter.
        '''
        self.debugTextBrowser.append( msg )

    def refreshAll( self ):
        '''
        Updates the widgets whenever an interaction happens.
        Typically some interaction takes place, the UI responds,
        and informs the model of the change.  Then this method
        is called, pulling from the model information that is
        updated in the GUI.
        '''
        self.lineEdit.setText( self.model.getFileName() )
        self.textEdit.setText( self.model.getFileContents() )

    # slot
    def returnPressedSlot( self ):
        ''' Called when the user enters a string in the line edit and
        presses the ENTER key.
        '''
        fileName =  self.lineEdit.text()
        if self.model.isValid( fileName ):
            self.model.setFileName( self.lineEdit.text() )
            self.refreshAll()
        else:
            m = QtWidgets.QMessageBox()
            m.setText("Invalid file name!\n" + fileName )
            m.setIcon(QtWidgets.QMessageBox.Warning)
            m.setStandardButtons(QtWidgets.QMessageBox.Ok
                                 | QtWidgets.QMessageBox.Cancel)
            m.setDefaultButton(QtWidgets.QMessageBox.Cancel)
            ret = m.exec_()
            self.lineEdit.setText( "" )
            self.refreshAll()
            self.debugPrint( "Invalid file specified: " + fileName  )

    # slot
    def writeDocSlot( self ):
        ''' Called when the user presses the Write-Doc button.
        '''
        self.model.writeDoc( self.textEdit.toPlainText() )
        self.debugPrint( "WriteDoc button pressed!" )

    # slot
    def browseSlot( self ):
        ''' Called when the user presses the Browse button
        '''
        #self.debugPrint( "Browse button pressed" )
        options = QtWidgets.QFileDialog.Options()
        options |= QtWidgets.QFileDialog.DontUseNativeDialog
        fileName, _ = QtWidgets.QFileDialog.getOpenFileName(
                        None,
                        "QFileDialog.getOpenFileName()",
                        "",
                        "All Files (*);;Python Files (*.py)",
                        options=options)
        if fileName:
            self.debugPrint( "setting file name: " + fileName )
            self.model.setFileName( fileName )
            self.refreshAll()

def main():
    """
    This is the MAIN ENTRY POINT of our application.  The code at the end
    of the mainwindow.py script will not be executed, since this script is now
    our main program.   We have simply copied the code from mainwindow.py here
    since it was automatically generated by '''pyuic5'''.

    """
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = MainWindowUIClass()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())

main()
```

Testing

- We are ready to test our application, either by running MyApp.py directly from Idle, or by launching it from the prompt:

```
python3.5 MyApp.py
```

- Browse for a text file of your choice, verify that its contents appear in the text-edit, make a small edit, and click on the **Write Doc** button. Verify that a new file has appeared next to your original file, the new one with a **.bak** extension.



```
# model.py
# D. Thiebaut
# This is the model part of the Model-View-Controller
# The class holds the name of a text file and its contents.
# Both the name and the contents can be modified in the GUI
# and updated through methods of this model.
#
# --------------------------------------------------------------------

class Model:
    def __init__( self ):
        '''
        Initializes the two members the class holds:
        the file name and its contents.
        '''
        self.fileName = None
        self.fileContent = ""

    def isValid( self, fileName ):
```

Deploying

- We use PyInstaller (https://pyinstaller.readthedocs.io/en/v3.3.1/usage.html) to generate a binary executable. PyInstaller works well with PyQt5 and will package all the required libraries in a single executable.

- Install PyInstaller for your platform.
- At the command prompt, find your way to the folder that contains our may python files:

```
MyApp.py
mainwindow.py
model.py
```

and type:

```
pyinstaller --onefile MyApp.py

445 INFO: PyInstaller: 3.3.1
445 INFO: Python: 3.5.4
460 INFO: Platform: Darwin-16.7.0-x86_64-i386-64bit
... (several output lines removed for the sake of simplicity)
15080 INFO: Building EXE because out00-EXE.toc is non existent
15080 INFO: Building EXE from out00-EXE.toc
15080 INFO: Appending archive to EXE /Users/thiebaut/temp/PyQt5Tutorial2/dist/MyApp
15118 INFO: Fixing EXE for code signing /Users/thiebaut/temp/PyQt5Tutorial2/dist/MyApp
15123 INFO: Building EXE from out00-EXE.toc completed successfully.
```

- Look in the **dist/** folder that **pyInstaller** created in your original folder. There should be an executable file called **MyApp**. Run it from the command line, or from a Window browser, and verify that you get the same functionality you experienced in the Testing section.

- If you want to share the app you just built with others, you only need to give them the **MyApp** file. It is fairly large (183 MB in our case), and contains everything needed to run the code.

- Congrats! You now have a simple skeleton for implementing a simple MVC application in PyQt5, coding in Python, and using Qt for the GUI.

Retrieved from "http://www.science.smith.edu/dftwiki/index.php?title=PyQt5_Tutorial:_A_Window_Application_with_File_IO&oldid=33937"