FREE eBook

LEARNING

pyqt5

Free unaffiliated eBook created from
Stack Overflow contributors.

#pyqt5

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: pyqt5

It is an unofficial and free pyqt5 ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official pyqt5.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with pyqt5

Remarks

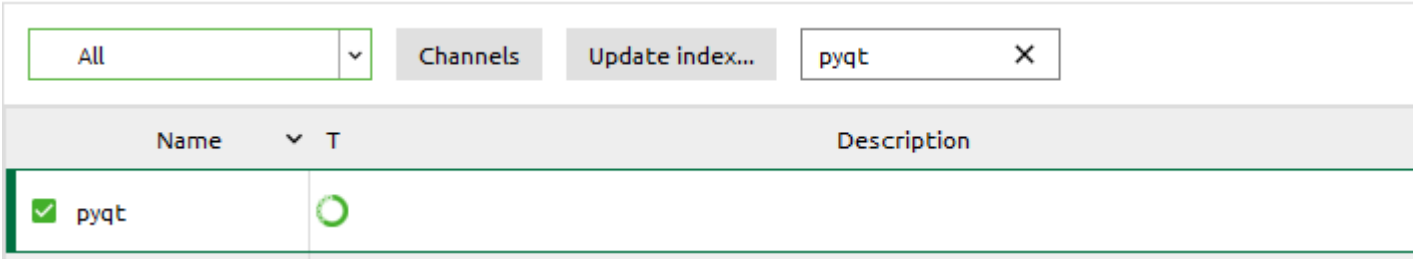
This section provides an overview of what pyqt5 is, and why a developer might want to use it.

It should also mention any large subjects within pyqt5, and link out to the related topics. Since the Documentation for pyqt5 is new, you may need to create initial versions of those related topics.

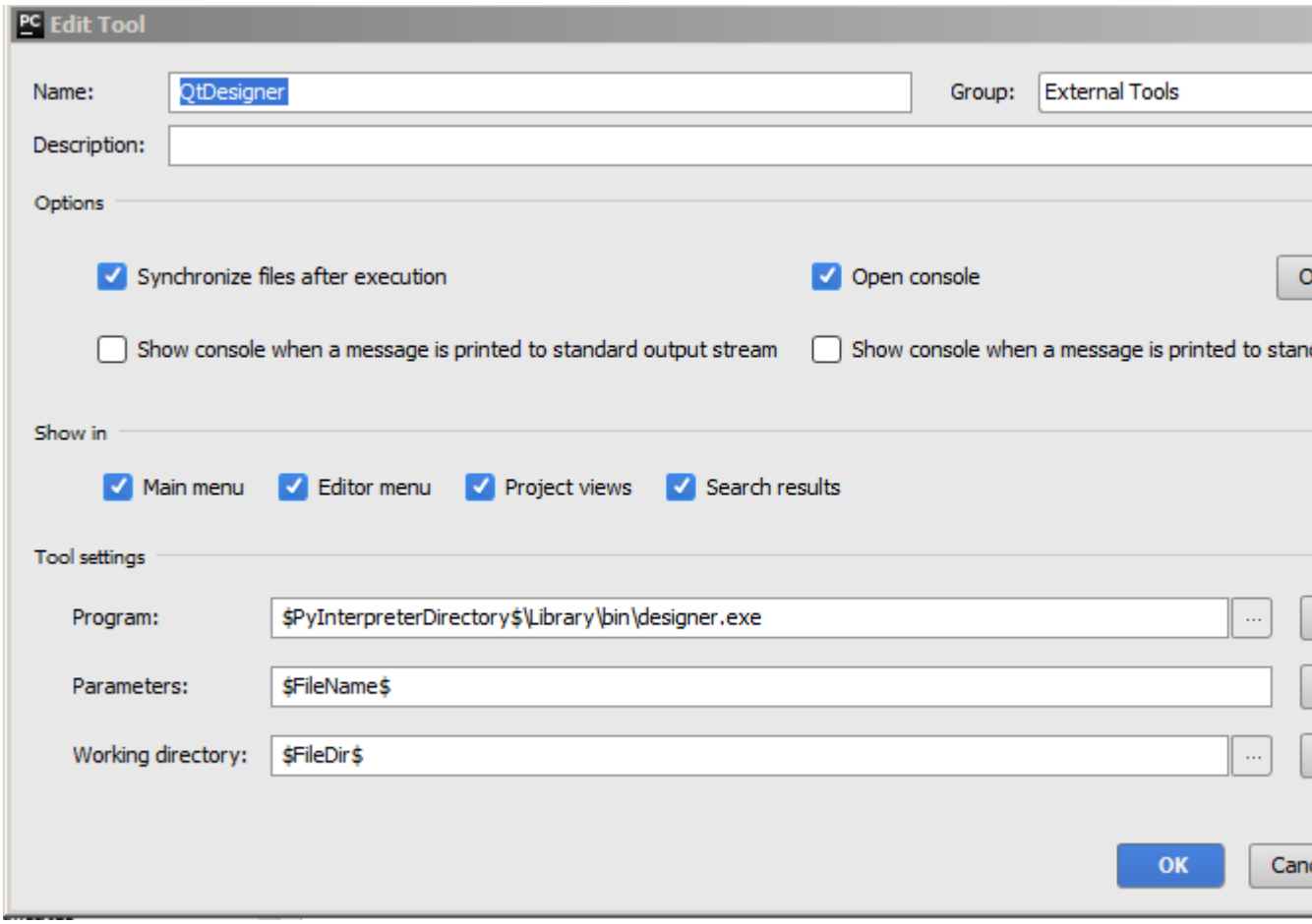
Examples

Installation or Setup

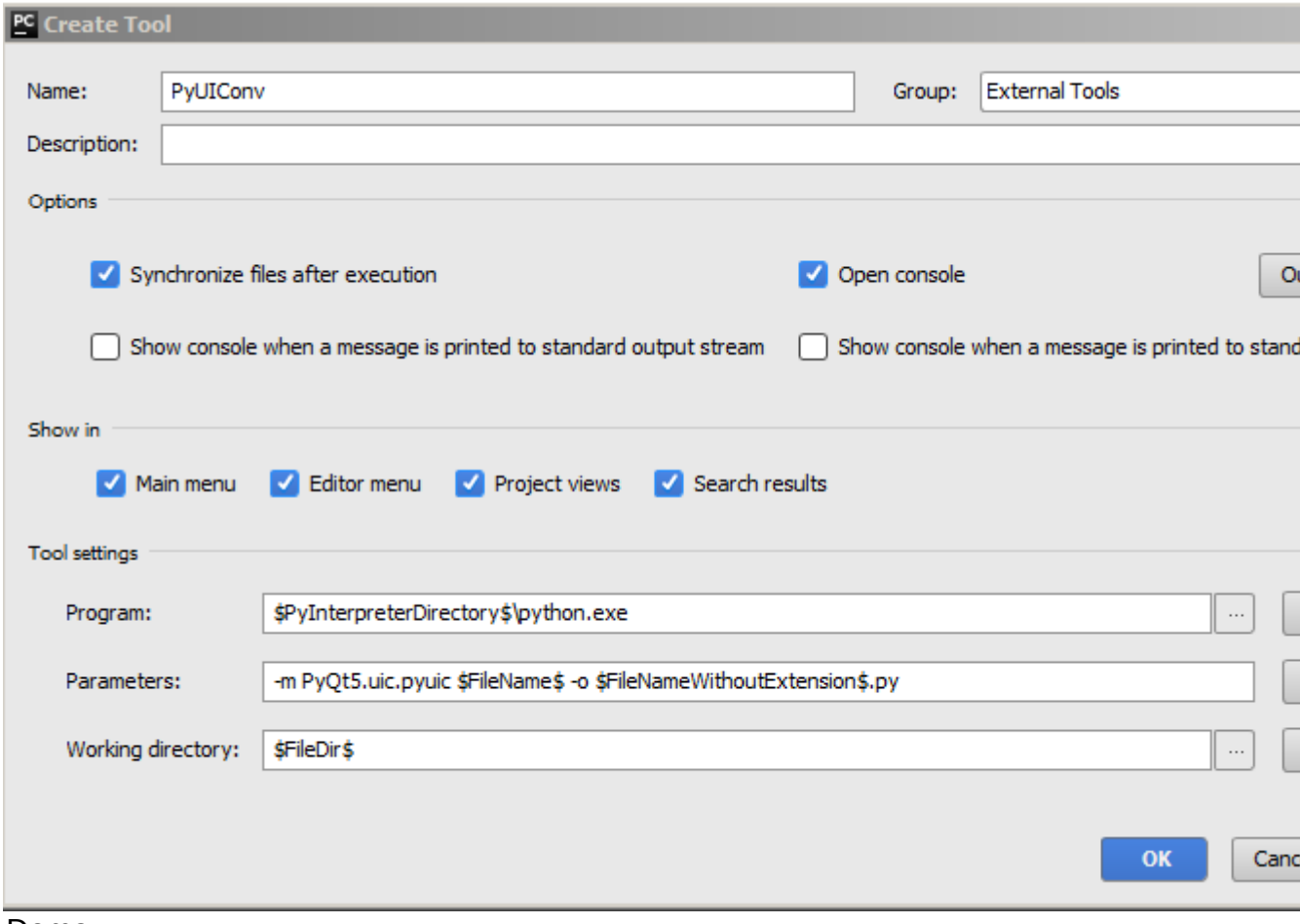
1. Install Anaconda(PyQt5 is build-in), especially for windows user.



2. Integrate QtDesigner and QtUIConvert in PyCharm(External Tools)
    - Open PyCharm `Settings` > `Tools` > `External Tools`
    - Create Tool(QtDesigner) - used to edit *.ui files

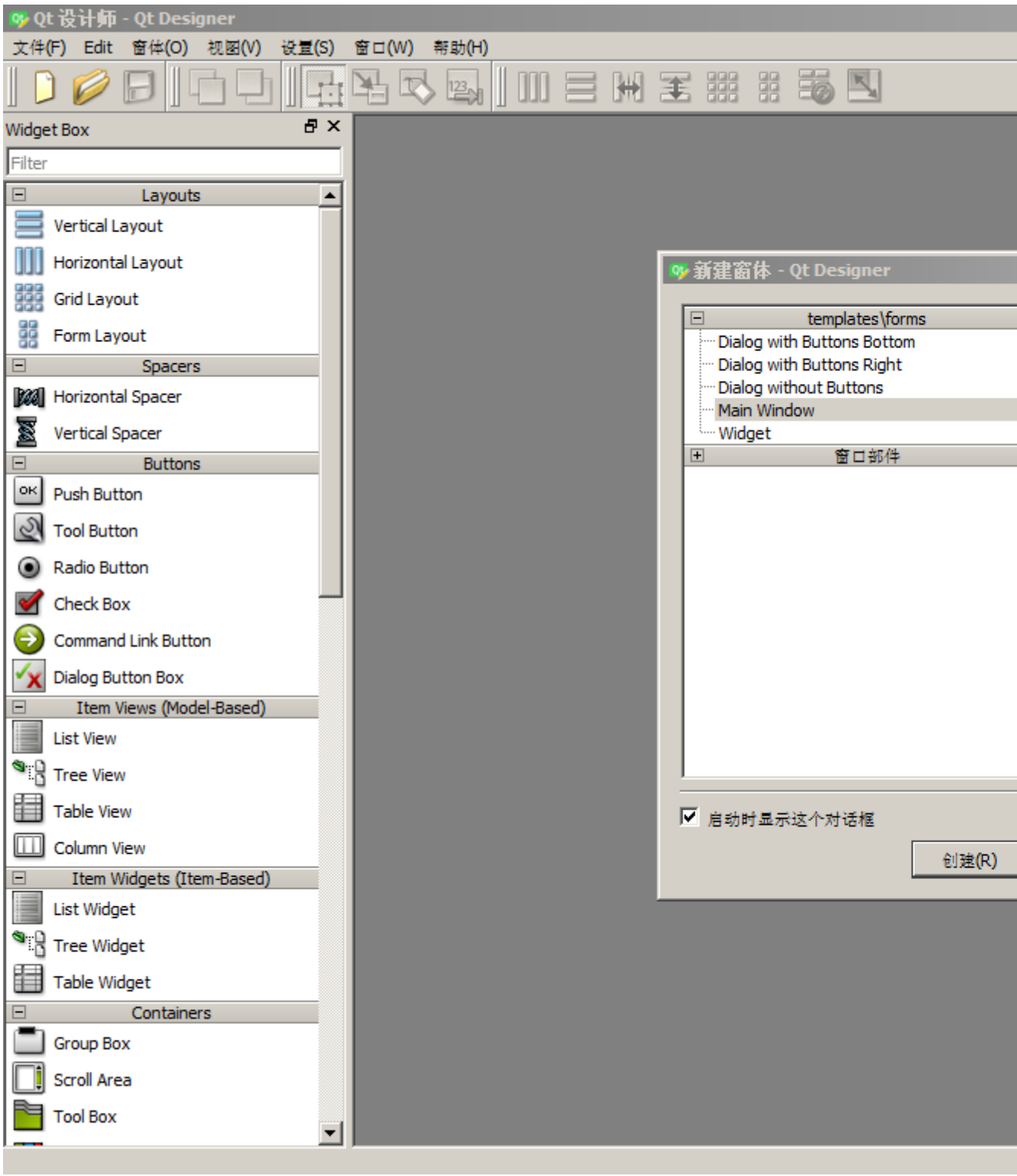


- Create Tool(PyUIConv) - used to convert *.ui to *.py

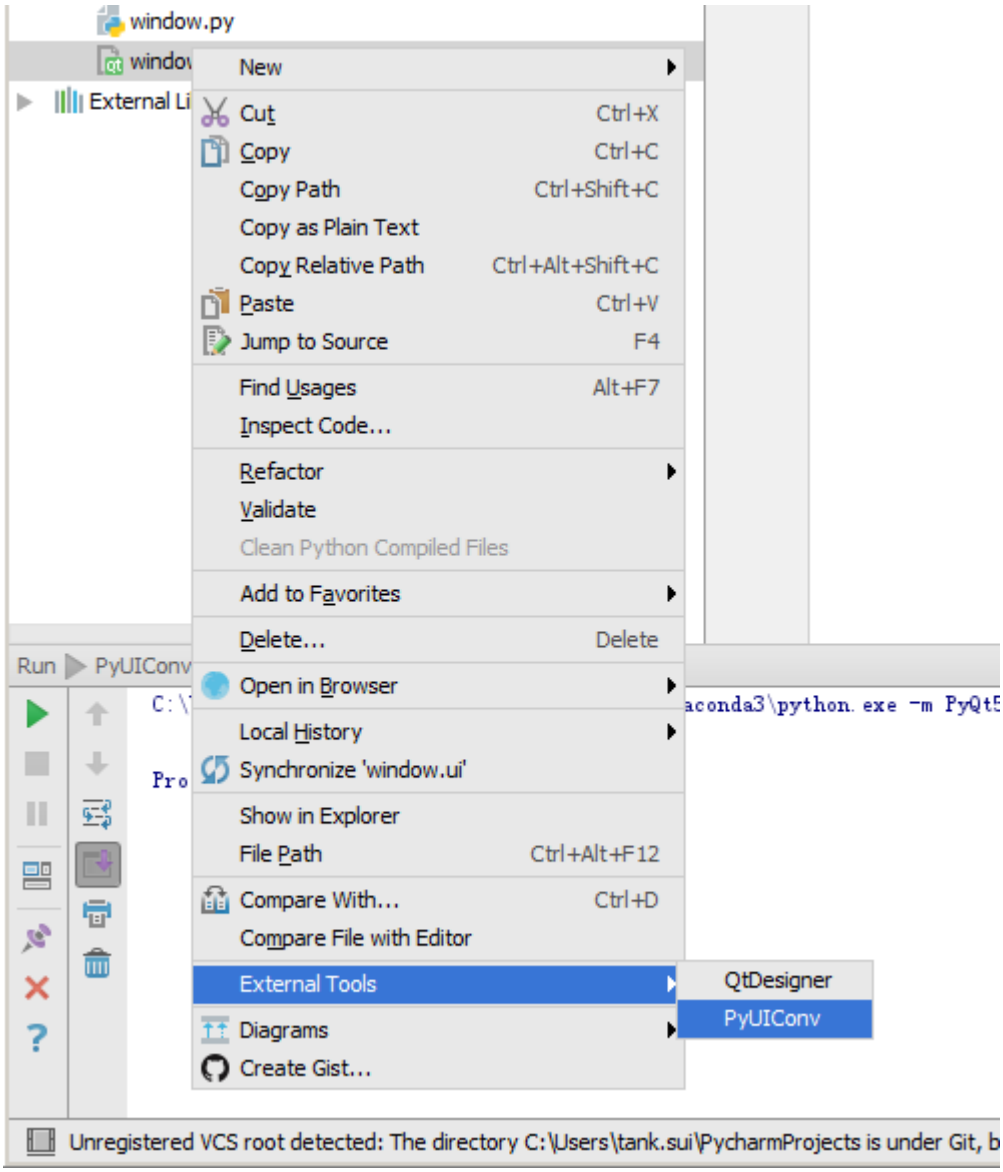


3. Write Demo

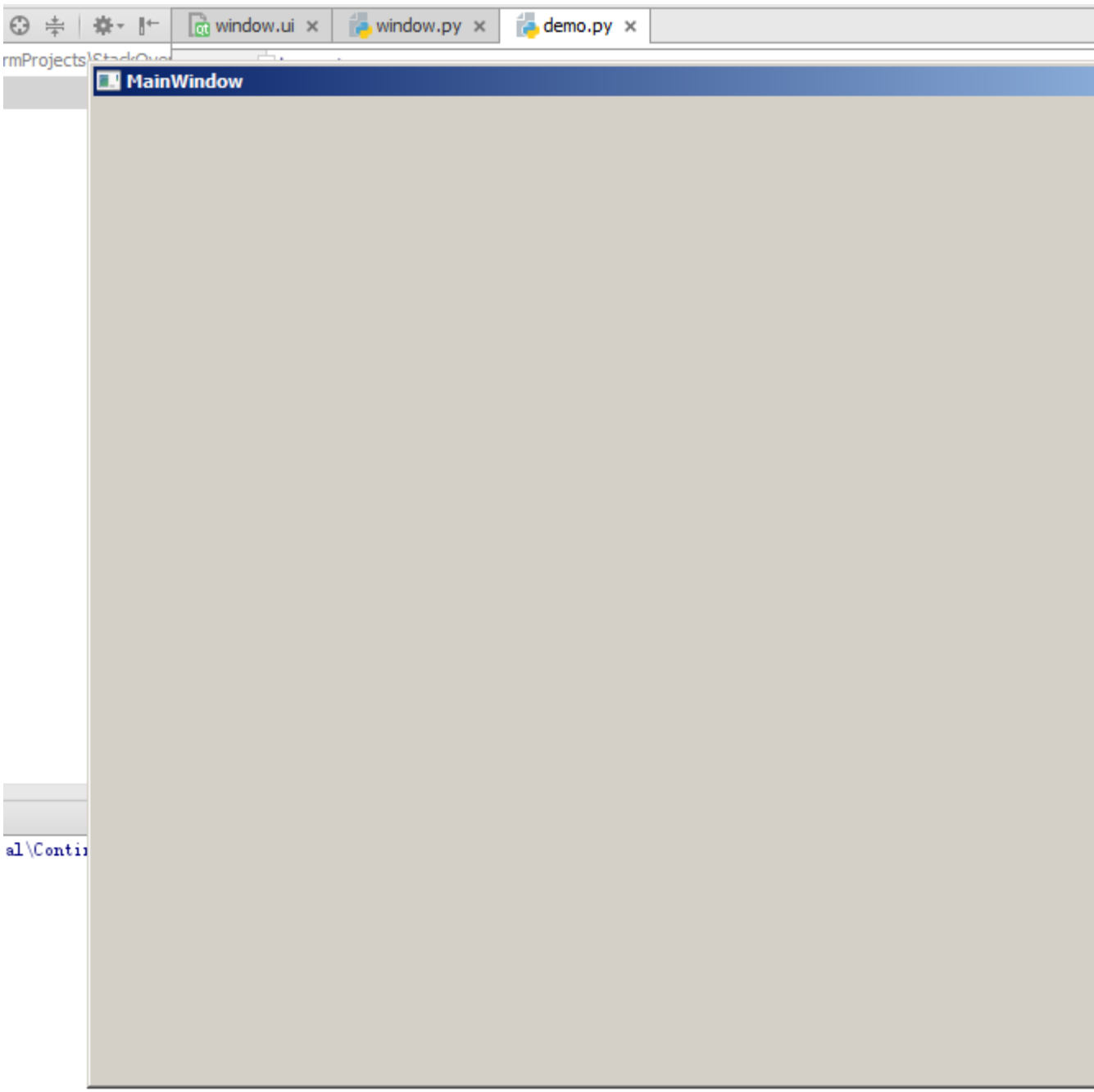
- new window.ui by external tool(QtDesigner)



- convert to window.py by external tool(PyUIConv)


window.py
External Li
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy as Plain Text
Copy Relative Path Ctrl+Alt+Shift+C
Paste Ctrl+V
Jump to Source F4
Find Usages Alt+F7
Inspect Code...
Refactor
Validate
Clean Python Compiled Files
Add to Favorites
Delete... Delete
Open in Browser
Local History
Synchronize 'window.ui'
Show in Explorer
File Path Ctrl+Alt+F12
Compare With... Ctrl+D
Compare File with Editor
External Tools
Diagrams
Create Gist...
QtDesigner
PyUIConv
Run
PyUIConv
Unregistered VCS root detected: The directory C:\Users\tank.sui\PycharmProjects is under Git, b

- demo




```
import sys
from PyQt5.QtWidgets import QApplication,QMainWindow
from window import Ui_MainWindow

if __name__ == '__main__':
    app = QApplication(sys.argv)
    w = QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(w)
    w.show()
    sys.exit(app.exec_())
```

Hello World Example

This example creates a simple window with a button and a line-edit in a layout. It also shows how to connect a signal to a slot, so that clicking the button adds some text to the line-edit.

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget


if __name__ == '__main__':

    app = QApplication(sys.argv)

    w = QWidget()
    w.resize(250, 150)
    w.move(300, 300)
    w.setWindowTitle('Hello World')
    w.show()

    sys.exit(app.exec_())
```

Analysis

```
app = QtWidgets.QApplication(sys.argv)
```

Every PyQt5 application must create an application object. The sys.argv parameter is a list of arguments from a command line. Python scripts can be run from the shell.

```
w = QWidget()
```

The `QWidget` widget is the base class of all user interface objects in PyQt5. We provide the default constructor for `QWidget`. The default constructor has no parent. A widget with no parent is called a window.

```
w.resize(250, 150)
```

The `resize()` method resizes the widget. It is 250px wide and 150px high.

```
w.move(300, 300)
```

The `move()` method moves the widget to a position on the screen at x=300, y=300 coordinates.

```
w.setWindowTitle('Hello World')
```

Here we set the title for our window. The title is shown in the titlebar.

```
w.show()
```

The `show()` method displays the widget on the screen. A widget is first created in memory and later shown on the screen.

```
sys.exit(app.exec_())
```

Finally, we enter the mainloop of the application. The event handling starts from this point. The mainloop receives events from the window system and dispatches them to the application widgets.

The mainloop ends if we call the `exit()` method or the main widget is destroyed. The `sys.exit()` method ensures a clean exit. The environment will be informed how the application ended.

The `exec_()` method has an underscore. It is because the exec is a Python keyword. And thus, `exec_()` was used instead.

Adding an application icon

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
from PyQt5.QtGui import QIcon


class Example(QWidget):

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

        self.initUI()


    def initUI(self):

        self.setGeometry(300, 300, 300, 220)
        self.setWindowTitle('Icon')
        self.setWindowIcon(QIcon('web.png'))

        self.show()


if __name__ == '__main__':

    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

Analysis

Function arguments in Python

In Python, user-defined functions can take four different types of arguments.

1. *Default arguments:*

- Function definition

  ```
  def defaultArg( name, msg = "Hello!"):
  ```

- Function call

  ```
  defaultArg( name)
  ```

2. *Required arguments:*

- Function definition

```
def requiredArg (str,num):
```

- Function call:

```
requiredArg ("Hello",12)
```

3. *Keyword arguments:*

- Function definition

```
def keywordArg( name, role ):
```

- Function call

```
keywordArg( name = "Tom", role = "Manager")
```

or

```
keywordArg( role = "Manager", name = "Tom")
```

4. *Variable number of arguments:*

- Function definition

```
def varlengthArgs(*varargs):
```

- Function call

```
varlengthArgs(30,40,50,60)
```

```
class Example(QWidget):

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

Three important things in object oriented programming are classes, data, and methods. Here we create a new class called `Example`. The `Example` class inherits from the `QWidget` class. This means that we call two constructors: the first one for the `Example` class and the second one for the inherited class. The `super()` method returns the parent object of the `Example` class and we call its constructor. The `self` variable refers to the object itself.

Why have we used `__init__`?

Check this out:

```
class A(object):
    def __init__(self):
        self.lst = []

class B(object):
    lst = []
```

and now try:

```
>>> x = B()
>>> y = B()
>>> x.lst.append(1)
>>> y.lst.append(2)
>>> x.lst
[1, 2]
>>> x.lst is y.lst
True
```

and this:

```
>>> x = A()
>>> y = A()
>>> x.lst.append(1)
>>> y.lst.append(2)
>>> x.lst
[1]
>>> x.lst is y.lst
False
```

> Does this mean that x in class B is established before instantiation?

Yes, it's a class attribute (it is shared between instances). While in class A it's an instance attribute.

```
self.initUI()
```

The creation of the GUI is delegated to the `initUI()` method.

```
self.setGeometry(300, 300, 300, 220)
self.setWindowTitle('Icon')
self.setWindowIcon(QIcon('web.png'))
```

All three methods have been inherited from the `QWidget` class. The `setGeometry()` does two things: it locates the window on the screen and sets it size. The first two parameters are the x and y positions of the window. The third is the width and the fourth is the height of the window. In fact, it combines the `resize()` and `move()` methods in one method. The last method sets the application icon. To do this, we have created a `QIcon` object. The `QIcon` receives the path to our icon to be displayed.

```
if __name__ == '__main__':

    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

The application and example objects are created. The main loop is started.

Showing a tooltip

```
import sys
```

```
from PyQt5.QtWidgets import (QWidget, QToolTip,
    QPushButton, QApplication)
from PyQt5.QtGui import QFont


class Example(QWidget):

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

        self.initUI()


    def initUI(self):

        QToolTip.setFont(QFont('SansSerif', 10))

        self.setToolTip('This is a <b>QWidget</b> widget')

        btn = QPushButton('Button', self)
        btn.setToolTip('This is a <b>QPushButton</b> widget')
        btn.resize(btn.sizeHint())
        btn.move(50, 50)

        self.setGeometry(300, 300, 300, 200)
        self.setWindowTitle('Tooltips')
        self.show()


if __name__ == '__main__':

    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

Analysis

```
QToolTip.setFont(QFont('SansSerif', 10))
```

This static method sets a font used to render tooltips. We use a 10px SansSerif font.

```
self.setToolTip('This is a <b>QWidget</b> widget')
```

To create a tooltip, we call the `setTooltip()` method. We can use rich text formatting.

```
btn = QPushButton('Button', self)
btn.setToolTip('This is a <b>QPushButton</b> widget')
```

We create a push button widget and set a tooltip for it.

```
btn.resize(btn.sizeHint())
btn.move(50, 50)
```

The button is being resized and moved on the window. The `sizeHint()` method gives a recommended size for the button.

Package your project into excutable/installer

cx_Freeze - a tool can package your project to excutable/installer

- after install it by pip, to package `demo.py`, we need `setup.py` below.

```
import sys
from cx_Freeze import setup, Executable

# Dependencies are automatically detected, but it might need fine tuning.
build_exe_options = {
    "excludes": ["tkinter"],
    "include_files":[('./platforms','./platforms')] # need qwindows.dll for qt5 application
}

# GUI applications require a different base on Windows (the default is for a
# console application).
base = None
if sys.platform == "win32":
    base = "Win32GUI"

setup(  name = "demo",
        version = "0.1",
        description = "demo",
        options = {"build_exe": build_exe_options},
        executables = [Executable("demo.py", base=base)])
```

- then build

```
python .\setup.py build
```

- then dist

```
python .\setup.py bdist_msi
```

Read Getting started with pyqt5 online: https://riptutorial.com/pyqt5/topic/7403/getting-started-with-pyqt5

Chapter 2: Introduction to Progress Bars

Introduction

Progress Bars are an integral part of user experience and helps users get an idea on the time left for a given process that runs on the GUI. This topic will go over the basics of implementing a progress bar in your own application.

This topic will touch lightly on QThread and the new signals/slots mechanism. Some basic knowledge of PyQt5 widgets is also expected of readers.

When adding examples only use PyQt5 and Python built-ins to demonstrate functionality.

PyQt5 Only

Remarks

Experimenting with these examples is the best way to get started learning.

Examples

Basic PyQt Progress Bar

This is a very basic progress bar that only uses what is needed at the bare minimum.

It would be wise to read this whole example to the end.

```
import sys
import time

from PyQt5.QtWidgets import (QApplication, QDialog,
                             QProgressBar, QPushButton)

TIME_LIMIT = 100

class Actions(QDialog):
    """
    Simple dialog that consists of a Progress Bar and a Button.
    Clicking on the button results in the start of a timer and
    updates the progress bar.
    """
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setWindowTitle('Progress Bar')
        self.progress = QProgressBar(self)
        self.progress.setGeometry(0, 0, 300, 25)
        self.progress.setMaximum(100)
```

```
        self.button = QPushButton('Start', self)
        self.button.move(0, 30)
        self.show()

        self.button.clicked.connect(self.onButtonClick)

    def onButtonClick(self):
        count = 0
        while count < TIME_LIMIT:
            count += 1
            time.sleep(1)
            self.progress.setValue(count)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = Actions()
    sys.exit(app.exec_())
```

The progress bar is first imported like so `from PyQt5.QtWidgets import QProgressBar`

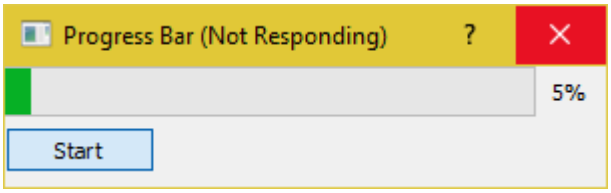
Then it is initialized like any other widget in `QtWidgets`

The line `self.progress.setGeometry(0, 0, 300, 25)` method defines the `x,y` positions on the dialog and width and height of the progress bar.

We then move the button using `.move()` by `30px` downwards so that there will be a gap of `5px` between the two widgets.

Here `self.progress.setValue(count)` is used to update the progress. Setting a maximum value using `.setMaximum()` will also automatically calculated the values for you. For example, if the maximum value is set as 50 then since `TIME_LIMIT` is 100 it will hop from 0 to 2 to 4 percent instead of 0 to 1 to 2 every second. You can also set a minimum value using `.setMinimum()` forcing the progress bar to start from a given value.

Executing this program will produce a GUI similar to this.



As you can see, the GUI will most definitely freeze and be unresponsive until the counter meets the `TIME_LIMIT` condition. This is because `time.sleep` causes the OS to believe that program has become stuck in an infinite loop.

QThread

So how do we overcome this issue ? We can use the threading class that PyQt5 provides.

```
import sys
import time

from PyQt5.QtCore import QThread, pyqtSignal
```

```
from PyQt5.QtWidgets import (QApplication, QDialog,
                             QProgressBar, QPushButton)

TIME_LIMIT = 100

class External(QThread):
    """
    Runs a counter thread.
    """
    countChanged = pyqtSignal(int)

    def run(self):
        count = 0
        while count < TIME_LIMIT:
            count +=1
            time.sleep(1)
            self.countChanged.emit(count)

class Actions(QDialog):
    """
    Simple dialog that consists of a Progress Bar and a Button.
    Clicking on the button results in the start of a timer and
    updates the progress bar.
    """
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setWindowTitle('Progress Bar')
        self.progress = QProgressBar(self)
        self.progress.setGeometry(0, 0, 300, 25)
        self.progress.setMaximum(100)
        self.button = QPushButton('Start', self)
        self.button.move(0, 30)
        self.show()

        self.button.clicked.connect(self.onButtonClick)

    def onButtonClick(self):
        self.calc = External()
        self.calc.countChanged.connect(self.onCountChanged)
        self.calc.start()

    def onCountChanged(self, value):
        self.progress.setValue(value)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = Actions()
    sys.exit(app.exec_())
```

Let's break down these modifications.

```
from PyQt5.QtCore import QThread, pyqtSignal
```

This line imports `Qthread` which is a `PyQt5` implementation to divide and run some parts(eg: functions, classes) of a program in the background(also know as multi-threading). These parts are also called threads. All `PyQt5` programs by default have a main thread and the others(worker

threads) are used to offload extra time consuming and process intensive tasks into the background while still keeping the main program functioning.

The second import `pyqtSignal` is used to send data(signals) between worker and main threads. In this instance we will be using it to tell the main thread to update the progress bar.

Now we have moved the while loop for the counter into a separate class called `External`.

```
class External(QThread):
    """
    Runs a counter thread.
    """
    countChanged = pyqtSignal(int)

    def run(self):
        count = 0
        while count < TIME_LIMIT:
            count +=1
            time.sleep(1)
            self.countChanged.emit(count)
```

By sub-classing `QThread` we are essentially converting `External` into a class that can be run in a separate thread. Threads can also be started or stopped at any time adding to it's benefits.

Here `countChanged` is the current progress and `pyqtSignal(int)` tells the worker thread that signal being sent is of type `int`. While, `self.countChanged.emit(count)` simply sends the signal to any connections in the main thread(normally it can used to communicate with other worker threads as well).

```
def onButtonClick(self):
        self.calc = External()
        self.calc.countChanged.connect(self.onCountChanged)
        self.calc.start()

def onCountChanged(self, value):
    self.progress.setValue(value)
```

When the button is clicked the `self.onButtonClick` will run and also start the thread. The thread is started with `.start()`. It should also be noted that we connected the signal `self.calc.countChanged` we created earlier to the method used to update the progress bar value. Every time `External::run::count` is updated the `int` value is also sent to `onCountChanged`.

This is how the GUI could look after making these changes.

It should also feel much more responsive and will not freeze.

Read Introduction to Progress Bars online: https://riptutorial.com/pyqt5/topic/9544/introduction-to-

progress-bars

Credits

S. No	Chapters	Contributors
1	Getting started with pyqt5	Ansh Kumar, Community, ekhumoro, suiwenfeng
2	Introduction to Progress Bars	daegontaven