

URL: <https://www.learnpyqt.com/courses/packaging-and-distribution/packaging-pyqt5-pyside2-applications-windows-pyinstaller/>



PyQt

Packaging PyQt5 & PySide2 applications for Windows, with PyInstaller



Martin Fitzpatrick [Packaging and distribution](#)
Read time 23:57

Turn your Qt5 application into a distributable installer for Windows

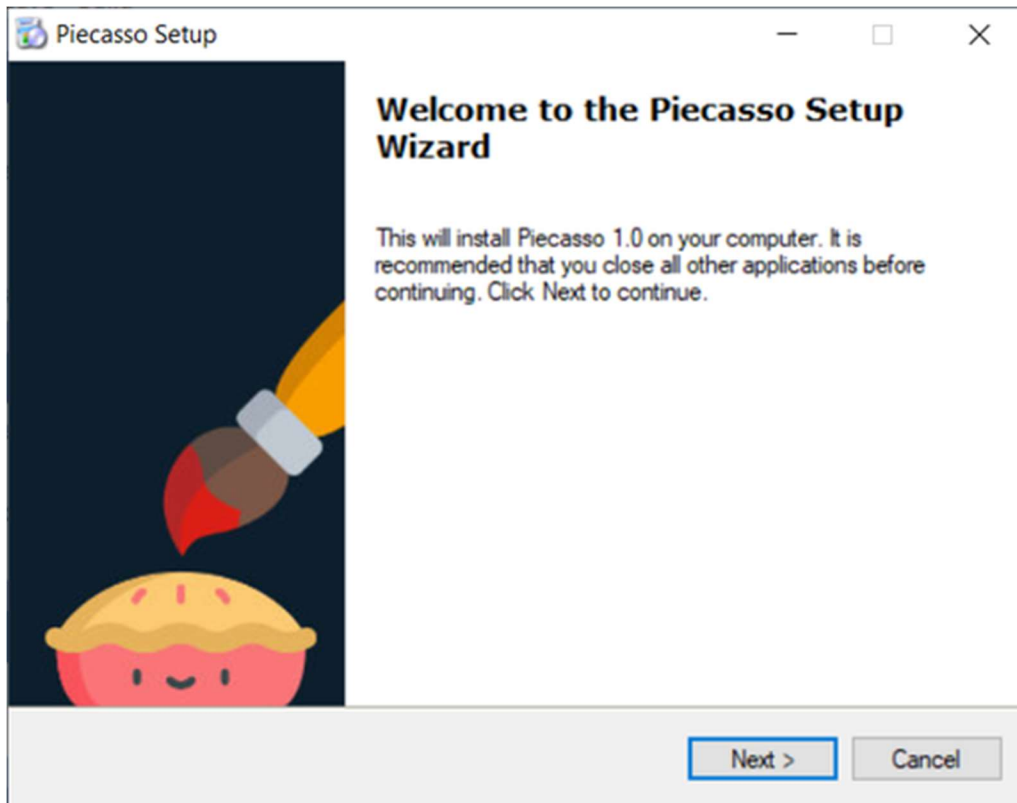
There is not much fun in creating your own desktop applications if you can't share them with other people — whether than means publishing it commercially, sharing it online or just giving it to someone you know. Sharing your apps allows other people to benefit from your hard work!

The good news is there are tools available to help you do just that with your Python applications which work well with apps built using Qt5. In this tutorial we'll look at the most popular tool for packaging Python applications: *PyInstaller*.

This tutorial is broken down into a series of steps, using *PyInstaller* to build first simple, and then increasingly complex PyQt5 applications into distributable EXE files on Windows. You can choose to follow it through completely, or skip ahead to the examples that are most relevant to your own project.

We finish off by using *InstallForge* to create a distributable Windows installer for *Piecaso* — a completely functional Paint clone made with Python 3 & Qt5

You always need to compile your app on your target system. So, if you want to create a Mac .app you need to do this on a Mac, for an EXE you need to use Windows.



Picasso Installer for Windows

If you're impatient, you can download the [Picasso Installer for Windows](#) right away! *Picasso* is one of our [15 Minute Apps](#), a collection of *minute* (small) apps built with Python & Qt5 and including all source code.

Requirements

PyInstaller works out of the box with both PyQt `PyQt5` and Qt for Python `PySide2` and, as of writing, *PyInstaller* is compatible up to 3.7. Whatever project you're working on, you should be able to package your apps.

You can install *PyInstaller* using `pip`.

```
pip3 install PyInstaller
```

Install in virtual environment (optional)

You can also opt to install PyQt5 and *PyInstaller* in a virtual environment (or your applications virtual environment) to keep your environment clean.

```
python3 -m venv packenv
```

Once created, activate the virtual environment by running from the command line —

```
call packenv\scripts\activate.bat
```

Finally, install the required libraries. For PyQt5 you would use —

```
pip3 install PyQt5 PyInstaller
```

Or for Qt for Python (PySide2) —

```
pip3 install PySide2 PyInstaller
```

Getting Started

It's a good idea to start packaging your application *from the very beginning* so you can confirm that packaging is still working as you develop it. This is particularly important if you add additional dependencies. If you only think about packaging at the end, it can be difficult to debug exactly *where* the problems are.

T> If you've already got an application you've created and want to know how to package *that* you may want to skip ahead to the advanced examples.

For this example we're going to start with a simple skeleton app, which doesn't do anything interesting. Once we've got the basic packaging process working, we'll extend the application to include icons and data files. We'll confirm the build as we go along.

To start with, create a new folder for your application and then add the following skeleton app in a file named `app.py`. You can also [download the source code and associated files](#)

- ~~PyQt5~~
- PySide2

```
from PySide2 import QtWidgets

import sys

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

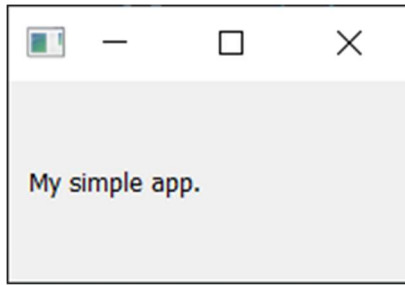
        self.setWindowTitle("Hello World")
        l = QtWidgets.QLabel("My simple app.")
        l.setMargin(10)
        self.setCentralWidget(l)
        self.show()

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    w = MainWindow()
    app.exec_()
```

This is a basic bare-bones application which creates a custom `QMainWindow` and adds a simple widget `QLabel` to it. You can run this app as follows.

```
python app.py
```

This should produce the following window (on Windows 10).



Simple skeleton app PyQt5/PySide2

Build #1, a basic app

Now we have our simple application skeleton in place, we can run our first build test to make sure everything is working.

Open your terminal (command prompt) and navigate to the folder containing your project. You can now run the following command to run the *PyInstaller* build.

```
pyinstaller app.py
```

You'll see a number of messages output, giving debug information about what *PyInstaller* is doing. These are useful for debugging issues in your build, but can otherwise be ignored. The output that I get for running the command on Windows 10 is shown below.









```
U:\home\martin\helloworld>pyinstaller app.py
INFO: PyInstaller: 3.6
INFO: Python: 3.7.6
INFO: Platform: Windows-10-10.0.18362-SP0
INFO: wrote U:\home\martin\helloworld\app.spec
INFO: UPX is not available.
INFO: Extending PYTHONPATH with paths
['U:\\home\\martin\\helloworld', 'U:\\home\\martin\\helloworld']
INFO: checking Analysis
INFO: Building Analysis because Analysis-00.toc is non existent
INFO: Initializing module dependency graph...
INFO: Caching module graph hooks...
INFO: Analyzing base_library.zip ...
INFO: Caching module dependency graph...
INFO: running Analysis Analysis-00.toc
INFO: Adding Microsoft.Windows.Common-Controls to dependent assemblies of
final executable
    required by
c:\users\gebruiker\appdata\local\programs\python\python37\python.exe
INFO: Analyzing U:\home\martin\helloworld\app.py
INFO: Processing module hooks...
INFO: Loading module hook "hook-encodings.py"...
INFO: Loading module hook "hook-pydoc.py"...
INFO: Loading module hook "hook-PyQt5.py"...
WARNING: Hidden import "sip" not found!
INFO: Loading module hook "hook-PyQt5.QtWidgets.py"...
INFO: Loading module hook "hook-xml.py"...
INFO: Loading module hook "hook-PyQt5.QtCore.py"...
INFO: Loading module hook "hook-PyQt5.QtGui.py"...
INFO: Looking for ctypes DLLs
INFO: Analyzing run-time hooks ...
INFO: Including run-time hook 'pyi_rth_pyqt5.py'
INFO: Looking for dynamic libraries
INFO: Looking for eggs
INFO: Using Python library
c:\users\gebruiker\appdata\local\programs\python\python37\python37.dll
INFO: Found binding redirects:
[]
INFO: Warnings written to U:\home\martin\helloworld\build\app\warn-app.txt
```

```

INFO: Graph cross-reference written to
U:\home\martin\helloworld\build\app\xref-app.html
INFO: checking PYZ
INFO: Building PYZ because PYZ-00.toc is non existent
INFO: Building PYZ (ZlibArchive) U:\home\martin\helloworld\build\app\PYZ-
00.pyz
INFO: Building PYZ (ZlibArchive) U:\home\martin\helloworld\build\app\PYZ-
00.pyz completed successfully.
INFO: checking PKG
INFO: Building PKG because PKG-00.toc is non existent
INFO: Building PKG (CArchive) PKG-00.pkg
INFO: Building PKG (CArchive) PKG-00.pkg completed successfully.
INFO: Bootloader
c:\users\gebruiker\appdata\local\programs\python\python37\lib\site-
packages\PyInstaller\bootloader\Windows-64bit\run.exe
INFO: checking EXE
INFO: Building EXE because EXE-00.toc is non existent
INFO: Building EXE from EXE-00.toc
INFO: Appending archive to EXE U:\home\martin\helloworld\build\app\app.exe
INFO: Building EXE from EXE-00.toc completed successfully.
INFO: checking COLLECT
INFO: Building COLLECT because COLLECT-00.toc is non existent
INFO: Building COLLECT COLLECT-00.toc
INFO: Building COLLECT COLLECT-00.toc completed successfully.

```

If you look in your folder you'll notice you now have two new folders `dist` and `build`.

Name	Date modified	Type	Size
 <code>_pycache_</code>	06/04/2020 17:49	File folder	
 <code>build</code>	07/04/2020 13:49	File folder	
 <code>dist</code>	08/04/2020 14:15	File folder	
 <code>app</code>	03/04/2020 15:05	Python Source File	1 KB
 <code>app.spec</code>	08/04/2020 14:15	SPEC File	1 KB
 <code>hand_icon</code>	06/04/2020 16:16	Icon	23 KB
 <code>resources</code>	06/04/2020 17:49	Python Source File	8 KB
 <code>resources.qrc</code>	06/04/2020 17:49	QRC File	1 KB

`build` & `dist` folders created by PyInstaller

Below is a truncated listing of the folder content, showing the `build` and `dist` folders.

```

.
├── app.py
├── app.spec
├── build
│   └── app
│       ├── Analysis-00.toc
│       ├── COLLECT-00.toc
│       ├── EXE-00.toc
│       ├── PKG-00.pkg
│       ├── PKG-00.toc
│       ├── PYZ-00.pyz
│       ├── PYZ-00.toc
│       ├── app.exe
│       ├── app.exe.manifest
│       ├── base_library.zip
│       ├── warn-app.txt
│       └── xref-app.html
└── dist
    └── app
        ├── MSVCP140.dll
        ├── PyQt5
        ├── app.exe
        ├── app.exe.manifest
        └── Qt5Core.dll
    ...

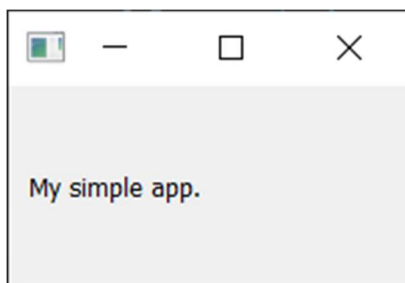
```

The `build` folder is used by *PyInstaller* to collect and prepare the files for bundling, it contains the results of analysis and some additional logs. For the most part, you can ignore the contents of this folder, unless you're trying to debug issues.

The `dist` (for "distribution") folder contains the files to be distributed. This includes your application, bundled as an executable file, together with any associated libraries (for example PyQt5) and binary `.dll` files.

Everything necessary to run your application will be in this folder, meaning you can take this folder and "distribute" it to someone else to run your app.

You can try running your app yourself now, by running the executable file, named `app.exe` from the `dist` folder. After a short delay you'll see the familiar window of your application pop up as shown below.



Simple app, running after being packaged

You may also notice a console/terminal window pop up as your application runs. We'll cover how to stop that happening shortly.

In the same folder as your Python file, alongside the `build` and `dist` folders *PyInstaller* will have also created a `.spec` file. In the next section we'll take a look at this file, what it is and what it does.

The Spec file

The `.spec` file contains the build configuration and instructions that *PyInstaller* uses to package up your application. Every *PyInstaller* project has a `.spec` file, which is generated based on the command line options you pass when running `pyinstaller`.

When we ran `pyinstaller` with our script, we didn't pass in anything other than the name of our Python application file. This means our spec file currently contains only the default configuration. If you open it, you'll see something similar to what we have below.

```
# -*- mode: python ; coding: utf-8 -*-

block_cipher = None

a = Analysis(['app.py'],
             pathex=['U:\\home\\martin\\helloworld'],
             binaries=[],
             datas=[],
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
pyz = PYZ(a.pure, a.zipped_data,
          cipher=block_cipher)
exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='app',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=True )
coll = COLLECT(exe,
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=False,
```

```
upx=True,  
upx_exclude=[],  
name='app')
```

The first thing to notice is that this is a Python file, meaning you can edit it and use Python code to calculate values for the settings. This is mostly useful for complex builds, for example when you are targeting different platforms and want to conditionally define additional libraries or dependencies to bundle.

Once a `.spec` file has been generated, you can pass this to `pyinstaller` instead of your script to repeat the previous build process. Run this now to rebuild your executable.

```
pyinstaller app.spec
```

The resulting build will be identical to the build used to generate the `.spec` file (assuming you have made no changes). For many *PyInstaller* configuration changes you have the option of passing command-line arguments, or modifying your existing `.spec` file. Which you choose is up to you.

Tweaking the build

So far we've created a simple first build of a very basic application. Now we'll look at a few of the most useful options that *PyInstaller* provides to tweak our build. Then we'll go on to look at building more complex applications.

Naming your app

One of the simplest changes you can make is to provide a proper "name" for your application. By default the app takes the name of your source file (minus the extension), for example `main` or `app`. This isn't usually what you want.











You can provide a nicer name for *PyInstaller* to use for the executable (and `dist` folder) either by editing the `.spec` file to add a `name=` under the `app` block.

```
exe = EXE(pyz,  
          a.scripts,  
          [],  
          exclude_binaries=True,  
          name='app',  
          name='Hello World',  
          debug=False,  
          bootloader_ignore_signals=False,  
          strip=False,  
          upx=True,  
          console=False # False = do not show console.  
          )
```

Alternatively, you can re-run the `pyinstaller` command and pass the `-n` or `--name` configuration flag along with your `app.py` script.

```
pyinstaller -n "Hello World" app.py
# or
pyinstaller --name "Hello World" app.py
```

The resulting EXE file will be given the name `Hello World.exe` and placed in the folder `dist\Hello World\`.

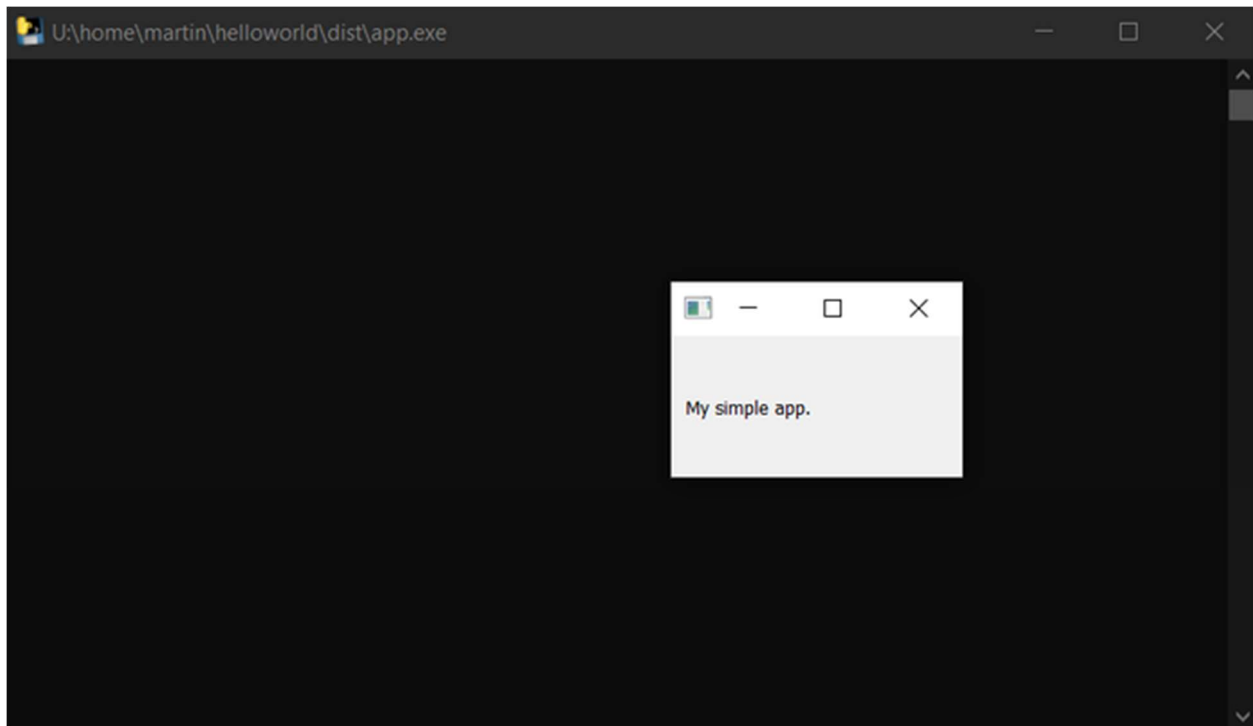
Name	Date modified	Type	Size
 PyQt5	08/04/2020 16:03	File folder	
 _bz2	08/04/2020 16:03	Python Extension ...	88 KB
 _hashlib	08/04/2020 16:03	Python Extension ...	39 KB
 _lzma	08/04/2020 16:03	Python Extension ...	252 KB
 _socket	08/04/2020 16:03	Python Extension ...	75 KB
 _ssl	08/04/2020 16:03	Python Extension ...	122 KB
 base_library	08/04/2020 16:03	Compressed (zipp...	769 KB
 d3dcompiler_47.dll	08/04/2020 16:03	Application extens...	4,077 KB
 Hello World	08/04/2020 16:03	Application	1,422 KB
 Hello World.exe.manifest	08/04/2020 16:03	MANIFEST File	2 KB

Application with custom name "Hello World"

The name of the `.spec` file is taken from the name passed in on the command line, so this will *also* create a new spec file for you, called `Hello World.spec` in your root folder.

Hiding the console window

When you run your packaged application you will notice that a console window runs in the background. If you try and close this console window your application will also close. You almost never want this window in a GUI application and *PyInstaller* provides a simple way to turn this off.



Application running with terminal in background

You can fix this in one of two ways. Firstly, you can edit the previously created `.spec` file setting `console=False` under the EXE block as shown below.

```
exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='app',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=False # False = do not show console.
        )
```

Alternatively, you can re-run the `pyinstaller` command and pass the `-w`, `--noconsole` or `--windowed` configuration flag along with your `app.py` script.

```
pyinstaller -w app.py
# or
pyinstaller --windowed app.py
# or
pyinstaller --noconsole app.py
```

There is no difference between any of the options.


Re-running `pyinstaller` will re-generate the `.spec` file. If you've made any other changes to this file these will be lost.

One File Build

On Windows *PyInstaller* has the ability to create a one-file build, that is, a single EXE file which contains all your code, libraries and data files in one. This can be a convenient way to share simple applications, as you don't need to provide an installer or zip up a folder of files.

To specify a one-file build provide the `--onefile` flag at the command line.

```
pyinstaller --onefile app.py
```

Name	Date modified	Type	Size
 app	08/04/2020 14:17	Application	33,824 KB

Result of a one-file build

Note that while the one-file build is easier to distribute, it is slower to execute than a normally built application. This is because every time the application is run it must create a temporary folder to unpack the contents of the executable. Whether this trade-off is worth the convenience for your app is up to you!

Using the `--onefile` option makes quite a few changes to the `.spec` file. You *can* make these changes manually, but it's much simpler to use the command line switch when first creating your `.spec`

Since debugging a one file app is much harder, you should make sure everything is working with a normal build before you create a one-file package.

Setting an application Icon

By default *PyInstaller* EXE files come with the following icon in place.

Name	Date modified	Type	Size
PyQt5	03/04/2020 13:54	File folder	
_bz2	03/04/2020 13:54	Python Extension ...	88 KB
_hashlib	03/04/2020 13:54	Python Extension ...	39 KB
_lzma	03/04/2020 13:54	Python Extension ...	252 KB
_socket	03/04/2020 13:54	Python Extension ...	75 KB
_ssl	03/04/2020 13:54	Python Extension ...	122 KB
app	03/04/2020 13:54	Application	1,422 KB
app.exe.manifest	03/04/2020 13:54	MANIFEST File	2 KB
base_library	03/04/2020 13:54	Compressed (zipp...	769 KB

Default PyInstaller application icon, on app.exe

You will probably want to customize this to make your application more recognisable. This can be done easily using the `--icon=<filename>` command-line switch to *PyInstaller*. On Windows the icon should be provided as an `.ico` file.

```
pyinstaller --windowed --icon=hand_icon.ico app.py
```

The [portable version of IcoFx](#) is a good free tool to create icons on Windows.

Or, by adding the `icon=` parameter to your `.spec` file.

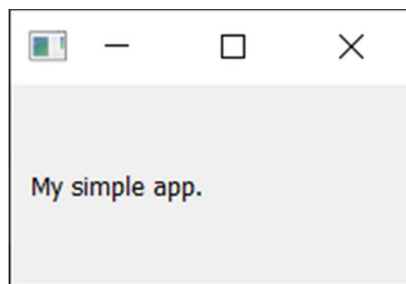
```
exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='blarh',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=False,
          icon='hand_icon.ico')
```

If you now re-run the build (by using the command line arguments, or running with your modified `.spec` file) you'll see the specified icon file is now set on your application's EXE file.

Name	Date modified	Type	Size
PyQt5	06/04/2020 17:49	File folder	
_bz2	06/04/2020 17:49	Python Extension ...	88 KB
_hashlib	06/04/2020 17:49	Python Extension ...	39 KB
_lzma	06/04/2020 17:49	Python Extension ...	252 KB
_socket	06/04/2020 17:49	Python Extension ...	75 KB
_ssl	06/04/2020 17:49	Python Extension ...	122 KB
app	06/04/2020 17:49	Application	1,401 KB
app.exe.manifest	06/04/2020 17:49	MANIFEST File	2 KB
base_library	06/04/2020 17:49	Compressed (zipp...	769 KB

Custom application icon (a hand) on app.exe

However, if you run your application, you're going to be disappointed.



The custom EXE icon is not applied to the window

The specified icon is not showing up on the window, and it will also not appear on your taskbar.

Why not? Because the icon used for the window isn't determined by the icons in the executable file, but by the application itself. To show an icon on our window we need to modify our simple application a little bit, to add a call to `.setWindowIcon()`.

- PySide2

```
from PySide2 import QtWidgets, QtGui
import sys

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.setWindowTitle("Hello World")
        l = QtWidgets.QLabel("My simple app.")
        l.setMargin(10)
        self.setCentralWidget(l)
```

```

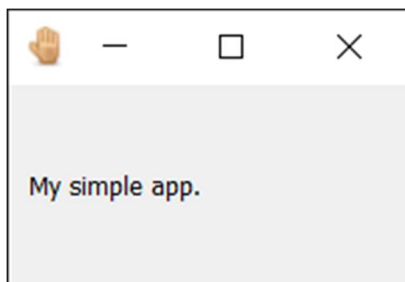
        self.show()

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    app.setWindowIcon(QtGui.QIcon('hand_icon.ico'))
    w = MainWindow()
    app.exec_()

```

Here we've added the `.setWindowIcon` call to the `app` instance. This defines a *default* icon to be used for all windows of our application. You *can* override this on a per-window basis if you like, by calling `.setWindowIcon` on the window itself.

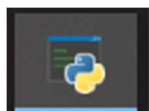
If you run the above application you should now see the icon appears on the window.



Window showing the custom hand icon

But, unfortunately, it may still not show on the taskbar.

If it does for you, great! But it may not work when you distribute your application, so it's probably a good idea to follow the next steps anyway.



Custom icon is not shown on the toolbar

The final tweak we need to make to get the icon showing on the taskbar is to add some cryptic incantations to the top of our Python file.

When you run your application, Windows looks at the executable and tries to guess what "application group" it belongs to. By default, any Python scripts (including your application) are grouped under the same "Python" group, and so will show the Python icon. To stop this happening, we need to provide Windows with a different application identifier.

The code below does this, by calling `QtWin.setCurrentProcessExplicitAppUserModelID()` with a custom application id.

- PySide2

```

from PySide2 import QtWidgets, QtGui

```



```

try:
    # Include in try/except block if you're also targeting Mac/Linux
    from PySide2.QtWinExtras import QtWin
    myappid = 'mycompany.myproduct.subproduct.version'
    QtWin.setCurrentProcessExplicitAppUserModelID(myappid)
except ImportError:
    pass

# ..or..
# import ctypes
# myappid = 'mycompany.myproduct.subproduct.version'
# ctypes.windll.shell32.SetCurrentProcessExplicitAppUserModelID(myappid)

import sys

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.setWindowTitle("Hello World")
        l = QtWidgets.QLabel("My simple app.")
        l.setMargin(10)
        self.setCentralWidget(l)

        self.show()

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    app.setWindowIcon(QtGui.QIcon('hand_icon.ico'))
    w = MainWindow()
    app.exec_()

```

You can, alternatively, use the Python call

`ctypes.windll.shell32.SetCurrentProcessExplicitAppUserModelID`. There is no difference.

The listing above shows a generic `mycompany.myproduct.subproduct.version` string, but you *should* change this to reflect your actual application. It doesn't really matter what you put for this purpose, but the convention is to use reverse-domain notation, `com.mycompany` for the company identifier.

With this added to your script, running it should now show the icon on your window and taskbar. The final step is to ensure that this icon is correctly packaged with your application and continues to be shown when run from the `dist` folder.

Try it, it wont.

The issue is that our application now has a dependency on a *external data file* (the icon file) that's not part of our source. For our application to work, we now need to distribute this data file

along with it. *PyInstaller* can do this for us, but we need to tell it what we want to include, and where to put it in the output.

In the next section we'll look at the options available to you for managing data files associated with your app.

Data files and Resources

So far we successfully built a simple app which had no external dependencies. However, once we needed to load an external file (in this case an icon) we hit upon a problem. The file wasn't copied into our `dist` folder and so could not be loaded.

In this section we'll look at the options we have to be able to bundle external resources, such as icons or Qt Designer `.ui` files, with our applications.

Bundling data files with PyInstaller

The simplest way to get these data files into the `dist` folder is to just tell *PyInstaller* to copy them over. *PyInstaller* accepts a list of individual file paths to copy over, together with a folder path relative to the `dist` folder where it should copy them *to*.

As with other options, this can be specified by command line arguments, `--add-data`

```
pyinstaller --windowed --icon=hand_icon.ico --add-data="hand_icon.ico;."
app.py
```

You can provide `--add-data` multiple times. Note that the path separator is platform-specific, on Windows use ``;` while on Linux or Mac use ``:`

Or via the `datas` list in the Analysis section of the spec file, shown below.

```
a = Analysis(['app.py'],
             pathex=['U:\\home\\martin\\helloworld'],
             binaries=[],
             datas=[('hand_icon.ico', '.')],
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
```

And then execute the `.spec` file with

```
pyinstaller app.spec
```

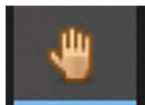
In both cases we are telling *PyInstaller* to copy the specified file `hand_icon.ico` to the location `.` which means the output folder `dist`. We could specify other locations here if we wanted. On the command line the source and destination are separated by the path separator `;`, whereas in the `.spec` file, the values are provided as a 2-tuple of strings.

If you run the build, you should see your `.ico` file now in the output folder `dist` ready to be distributed with your application.

Name	Date modified	Type	Size
PyQt5	03/04/2020 15:20	File folder	
_bz2	03/04/2020 15:20	Python Extension ...	88 KB
_hashlib	03/04/2020 15:20	Python Extension ...	39 KB
_lzma	03/04/2020 15:20	Python Extension ...	252 KB
_socket	03/04/2020 15:20	Python Extension ...	75 KB
_ssl	03/04/2020 15:20	Python Extension ...	122 KB
app	03/04/2020 15:20	Application	1,387 KB
app.exe.manifest	03/04/2020 15:20	MANIFEST File	2 KB
base_library	03/04/2020 15:20	Compressed (zipp...	769 KB
d3dcompiler_47.dll	03/04/2020 15:20	Application extens...	4,077 KB
hand_icon	03/04/2020 15:20	Icon	19 KB
libcrypto-1_1.dll	03/04/2020 15:20	Application extens...	3,303 KB
libEGL.dll	03/04/2020 15:20	Application extens...	24 KB

The icon file copied to the `dist` folder

If you run your app from `dist` you should now see the icon on the window, and on the taskbar as expected.



The hand icon showing on the toolbar

The file must be loaded in Qt using a *relative path*, and be in the same relative location to the EXE as it was to the `.py` file for this to work.

If your icon looks blurry it means you don't have large-enough icon variations in your `.ico` file. An `.ico` file can contain multiple different sized icons in the same file. Ideally you want to have 16x16, 32x32, 48x48 and 256x256 pixel sizes included, although fewer will still work.

The main advantage of using *PyInstaller* to bundle your files in this way is you can use Python in your `.spec` file to search and add the files to bundle. So for example, you could get a list of all files in a folder named `icons` and add them to the `datas=` parameter. Then, as you add more icons to that folder they would be bundled automatically.

The trade-off is that you can sometimes hit problems, particularly when bundling your applications cross-platform. In the next section we'll look at an alternative, often more reliable, method using the Qt Resources system.

Qt Resources

Bundling data files with *PyInstaller* usually works quite well. However, there can be issues with using relative paths, particularly when bundling cross-platform applications, as different systems have different standards for dealing with data files. If you hit these problems they can unfortunately be quite difficult to debug.

Thankfully, Qt comes to the rescue with its *resource system*. Since we're using Qt for our GUI we can make use of Qt Resources to bundle, identify and load resources in our application. The resulting bundled data is included in your application as Python code, so *PyInstaller* will automatically pick it up and we can be sure it will end up in the right place.

In this section we'll look at how to bundle files with our application using the Qt Resources system.

The QRC file

The core of the Qt Resources system is the *resource file* or QRC. The `.qrc` file is a simple XML file, which can be opened and edited with any text editor.

You can also create QRC files and add and remove resources using Qt Designer, which we'll cover later.

Simple QRC example

A very simple resource file is shown below, containing a single resource (our application icon).

```
<!DOCTYPE RCC>
<RCC version="1.0">
  <qresource prefix="icons">
    <file alias="hand_icon.ico">hand_icon.ico</file>
  </qresource>
</RCC>
```

The name between the `<file>` `</file>` tags is the path to the file, relative to the resource file. The `alias` is the name which this resource will be known by from within your application. You can use this *rename* icons to something more logical or simpler in your app, while keeping the original name externally.

For example, if we wanted to use the name `application_icon.ico` internally, we could change this line to.

```
<file alias="application_icon.ico">hand_icon.ico</file>
```

This only changes the name used *inside* your application, the filename remains unchanged.

Outside this tag is the `qresource` tag which specifies a `prefix`. This is a *namespace* which can be used to group resources together. This is effectively a virtual folder, under which nested resources can all be found.

Using a QRC file

To use a `.qrc` file in your application you first need to compile it to Python. PyQt5 & PySide2 come with a command line tool to do this, which takes a `.qrc` file as input and outputs a Python file containing the compiled data. This can then be imported into your app as for any other Python file or module.

To compile our `resources.qrc` file to a Python file named `resources.py` we can use

- PyQt5
- PySide2

bash

```
pyside2-rcc resources.qrc -o resources.py
```

To use the resource file in our application we need to make a few small changes. Firstly, we need to `import resources` at the top of our app, to load the resources into the Qt resource system, and then secondly we need to update the path to the icon file to use the resource path format as follows:

```
app.setWindowIcon(QtGui.QIcon('/:icons/hand_icon.ico'))
```

The prefix `:/` indicates that this is a *resource path*. The first name "icons" is the *prefix* namespace and the filename is taken from the file *alias*, both as defined in our `resources.qrc` file.

The updated application is shown below.

- PyQt5
- PySide2

python

```
from PySide2 import QtWidgets, QtGui
```

```
try:
```

```
    # Include in try/except block if you're also targeting Mac/Linux
    from PySide2.QtWinExtras import QtWin
    myappid = 'com.learnpyqt.examples.helloworld'
    QtWin.setCurrentProcessExplicitAppUserModelID(myappid)
```

```
except ImportError:
    pass
```

```
import sys
```

```
import resources # Import the compiled resource file.
```

```

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.setWindowTitle("Hello World")
        l = QtWidgets.QLabel("My simple app.")
        l.setMargin(10)
        self.setCentralWidget(l)

        self.show()

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    app.setWindowIcon(QtGui.QIcon('/:icons/hand_icon.ico'))
    w = MainWindow()
    app.exec_()

```

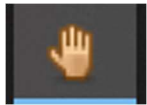
You can run the build as follows,

```
pyinstaller --windowed --icon=hand_icon.ico app.py
```

or re-run it using your existing `.spec` file.

```
pyinstaller app.spec
```

If you run the resulting application in `dist` you should see the icon is working as intended.



The hand icon showing on the toolbar

The advantage of this method is that your data files are guaranteed to be bundled as they are treated as code — *PyInstaller* finds them through the imports in your source. You also don't need to worry about platform-specific locations for data files. You only need to take care to rebuild the `resources.py` file any time you add or remove data files from your project.

Of course, this approach isn't appropriate for any files you want to be readable or editable by end-users. However, there is nothing stopping you from combining this approach with the previous one as needed.

Build #2, bundling Qt Designer UIs and Icons

We've now managed to build a simple app with a single external icon file as a dependency. Now for something a bit more realistic!

In complex Qt applications it's common to use Qt Designer to define the the UI, including icons on buttons and menus. How can we distribute UI files with our applications and ensure the linked icons continue to work as expected?

Below is the UI for a demo application we'll use to demonstrate this. The app is a simple counter, which allows you to increase, decrease or reset the counter by clicking the respective buttons. You can also [download the source code and associated files](#).



The counter UI created in Qt Designer

The UI consists of a `QMainWindow` with a vertical layout containing a single `QLabel` and 3 `QPushButton` widgets. The buttons have Increment, Decrement and Reset labels, along with icons from [the Fugue set by p.yusukekamiyamane](#). The application icon is a free icon from [Freepik](#).

The UI was created in Qt Designer as described in this [tutorial](#).

Resources

The icons in this project were added to the buttons from within Qt Designer. When doing this you have two options —

1. add the icons as files, and ensure that the relative path locations of icons are maintained after installation (not always possible, or fun)
2. add the icons using the Qt Resource system

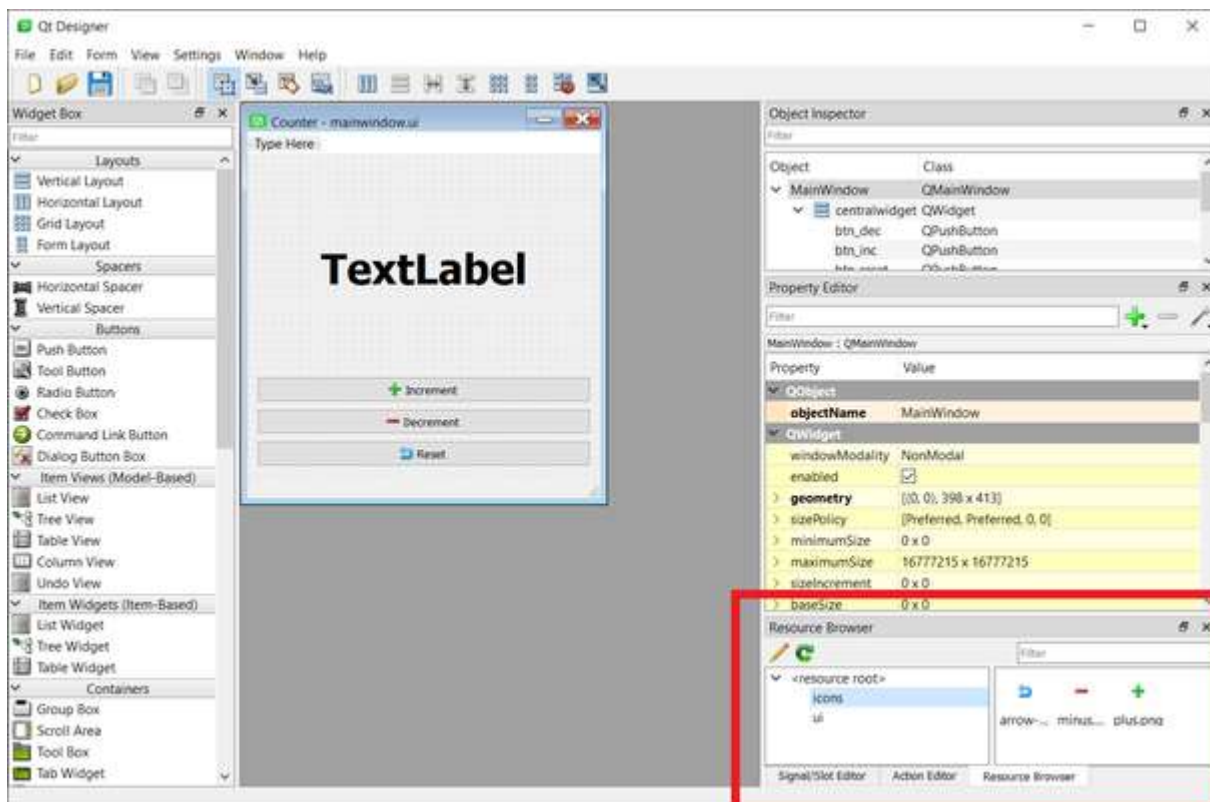
Here we're using approach (2) because it's less prone to errors.

The method for Qt Resources in your UI differs depending on whether you're using Qt Creator or Qt Designer standalone. The steps are described below.

Adding Resources in Qt Designer (Preferred)

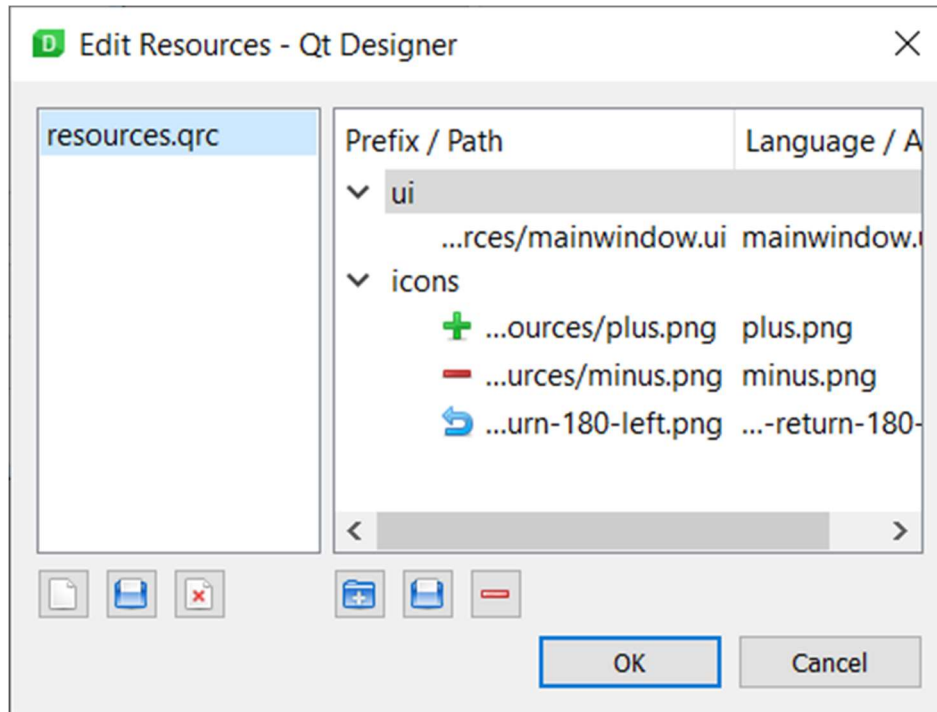
If you're using the standalone Qt Designer, the resource browser is available as a dockable widget, visible in the bottom right by default. If the Resource Browser is hidden you can show it through the "View" menu on the toolbar.

To add, edit and remove resource files click on the pencil icon in the Resource browser panel. This will open the resource editor.



Qt Designer resource browser

In the resource editor view you can open an existing resource file by clicking on the document folder icon (middle icon) on the bottom left.



Qt Designer resource editor

On the left hand panel you can also create and delete resource files from your UI. While on the right you can create new prefixes, add files to the prefix and delete items. Changes to the resource file are saved automatically.

Adding Resources in Qt Creator

In order to be able to add icons using the Qt Resource system from within Qt Creator you need to have an active Qt Project, and add both your UI and resource files to it.

If you don't have a Qt Creator project set up you can create one in your existing source folder. Qt Creator will prompt before overwriting any of your files. Click on "+ New", choose "Qt for Python - Empty" for project type. Select the folder *above* your source folder for "Create in", and provide the name of your source folder as the project name. You can delete any files created, except the `.pyproject` which holds the project settings.

×

Qt for Python - Empty

Location

Summary

Project Location

Creates a Qt for Python application that contains only the main code for a QApplication.

Name: counter

Create in: U:\home\martin

Browse...

☐ Use as default project location

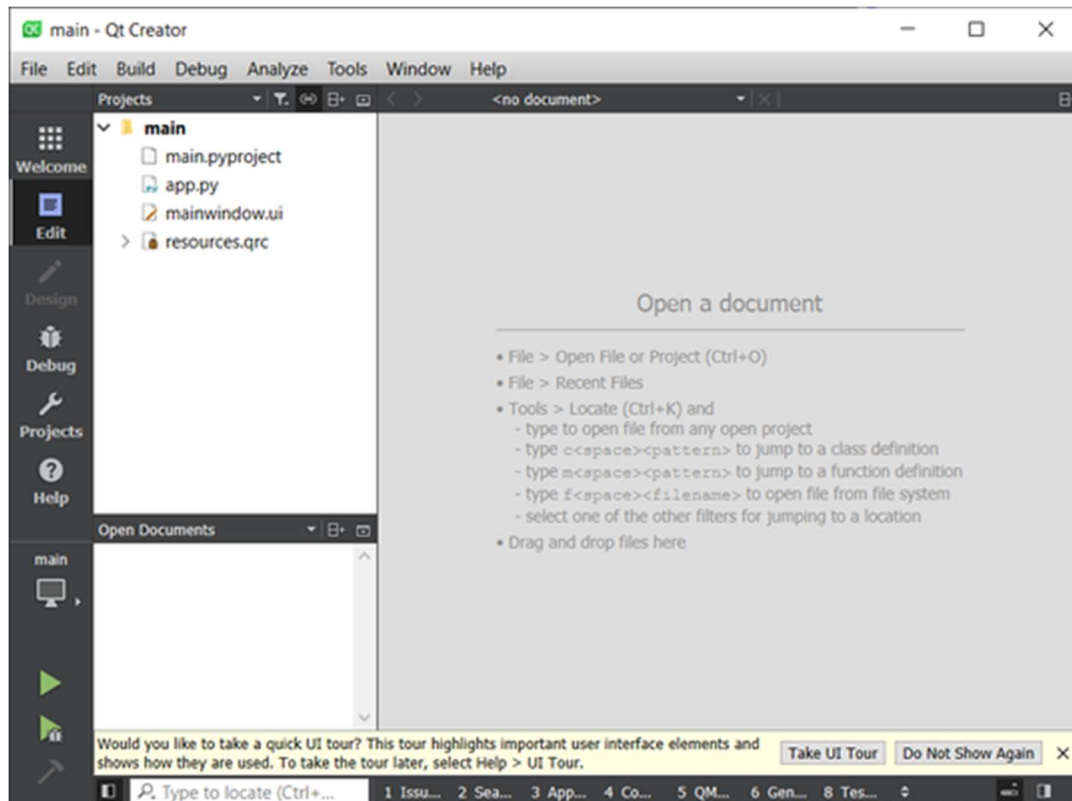
The project already exists.

Next

Cancel

This message will be shown when creating a new Qt Creator project in an existing folder

To add resources to your existing project, select the "Edit" view on the left hand panel. You will see a file tree browser in the left hand panel. Right-click on the folder and choose "Add existing files..." and add your existing .qrc file to the project.



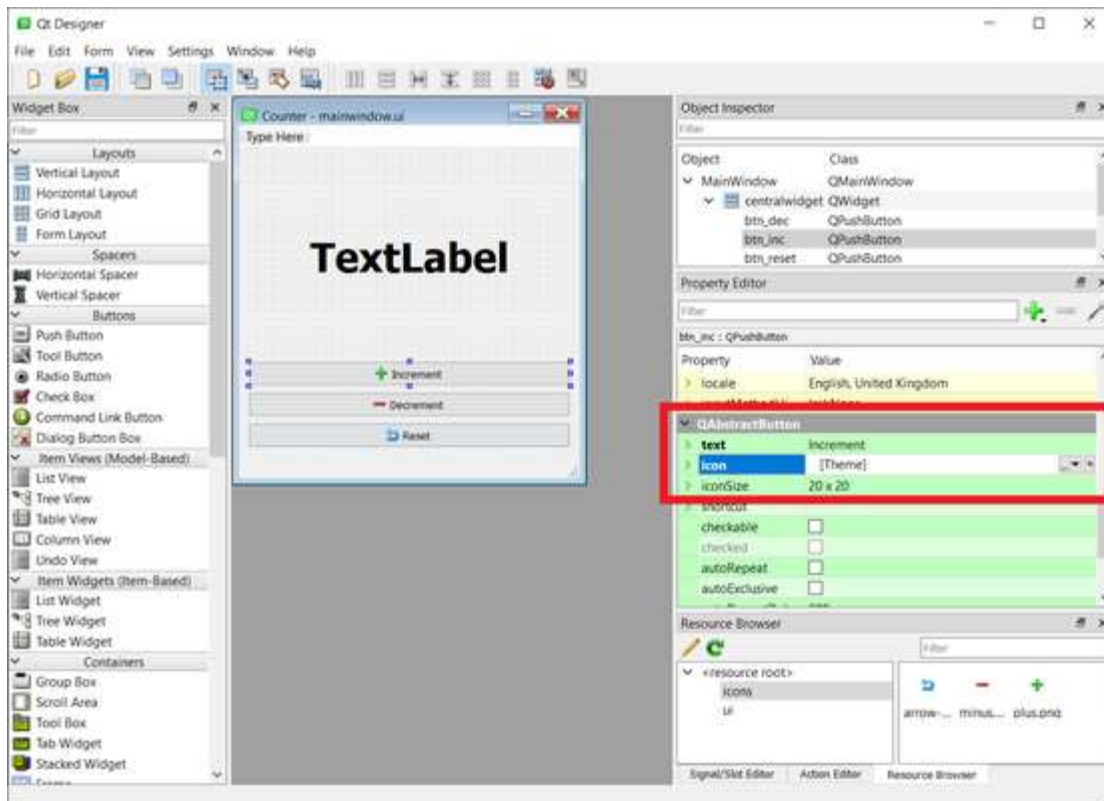
Qt Creator "Edit" view, showing a list of files in the project

The UI doesn't update when you add/remove things here, this seems to be a bug in Qt Creator. If you close and re-open Qt Creator the files will be there.

Once you have added the QRC file to the file listing you should be able to expand the file as if it were a folder, and browser the resources within. You can also add and remove resources using this interface.

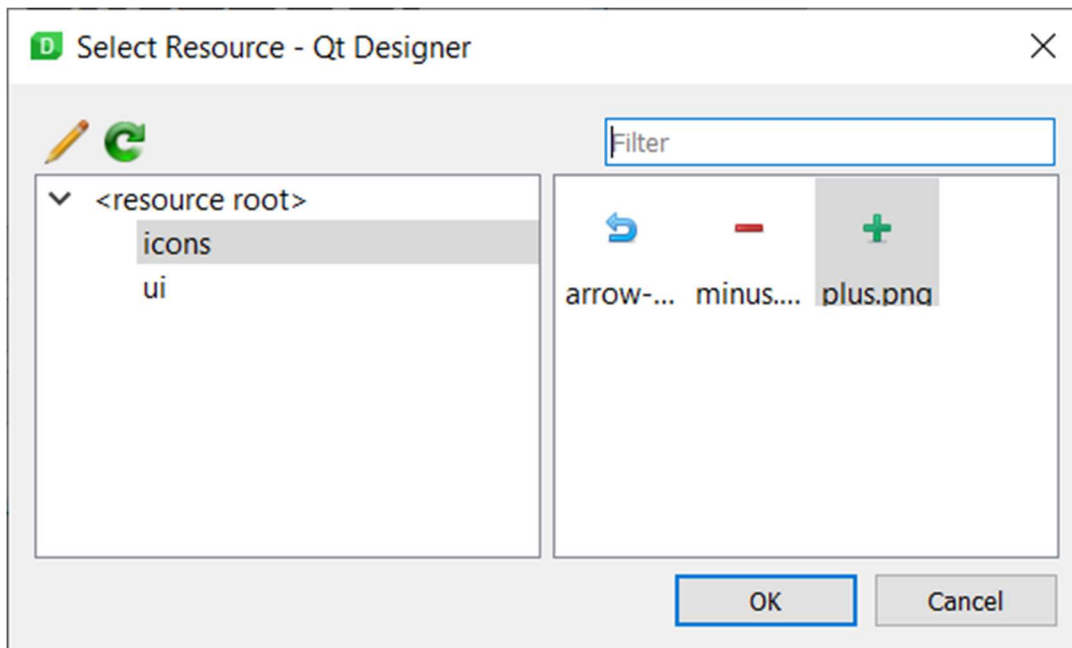
Using resources in Qt Creator and Qt Designer

Once the Resource file is loaded you will be able to access it from the designer properties. The screenshot below shows the Designer with our counter app open, and the *increment* button selected. The icon for the button can be chosen by clicking the small black down arrow and selecting "Choose Resource..."



Setting the icon for a button in Qt Designer (or Qt Creator)

The Resource chooser window that appears allows you to pick icons from the resource file(s) in the project to use in your UI.



Selecting a resource in the Qt Designer resource dialog

Selecting the icons from the resource file in this way ensures that they will always work, as long as you compile and bundle the compiled resource file with your app.

Loading UIs from the Resource file (PyQt5 only)

The last change required is in the loading of the UI file. In PyQt5 the `.loadUi` handler does not understand resource paths, and so we cannot load our `mainwindow.ui` file directly with it. Instead we need to load the file manually, and pass the result into `.loadUi` ourselves. The code to do that is shown below.

```
# Load the UI
fileh = QtCore.QFile('/:ui/mainwindow.ui')
fileh.open(QtCore.QFile.ReadOnly)
uic.loadUi(fileh, self)
fileh.close()
```

You may want to wrap this in a function if you're using it a lot. The PySide2 handler understands resource paths natively, so we don't need this workaround.

Alternatively, you can also compile your UI files to Python. This is covered later.

The finished app

Below is our updated `app.py` which loads the `mainwindow.ui` file and defines 3 custom slots to increment, decrement and reset the number. These are connected to signals of the widgets defined in the UI (`btn_inc`, `btn_dec` and `btn_reset` for the 3 buttons respectively) along with a method to update the displayed number (`label` for the `QLabel`).

- PyQt5
- PySide2

python

```
from PySide2 import QtWidgets, QtGui, QtCore
from PySide2.QtUiTools import QUiLoader
```

```
import sys
import resources
```

```
loader = QUiLoader()
```

```
try:
```

```
    # Include in try/except block if you're also targeting Mac/Linux
    from PyQt5.QtWinExtras import QtWin
    myappid = 'com.learnpyqt.examples.counter'
    QtWin.setCurrentProcessExplicitAppUserModelID(myappid)
```

```
except ImportError:
    pass
```

```
class WindowWrapper(QtCore.QObject):
```

```

def __init__(self, *args, **kwargs):
    super(WindowWrapper, self).__init__()

    self.ui = loader.load('/:ui/mainwindow.ui', None)
    self.ui.show()

    # Set value of counter
    self.counter = 0
    self.update_counter()

    # Bind
    self.ui.btn_inc.clicked.connect(self.inc)
    self.ui.btn_dec.clicked.connect(self.dec)
    self.ui.btn_reset.clicked.connect(self.reset)

def update_counter(self):
    self.ui.label.setText(str(self.counter))

def inc(self):
    self.counter += 1
    self.update_counter()

def dec(self):
    self.counter -= 1
    self.update_counter()

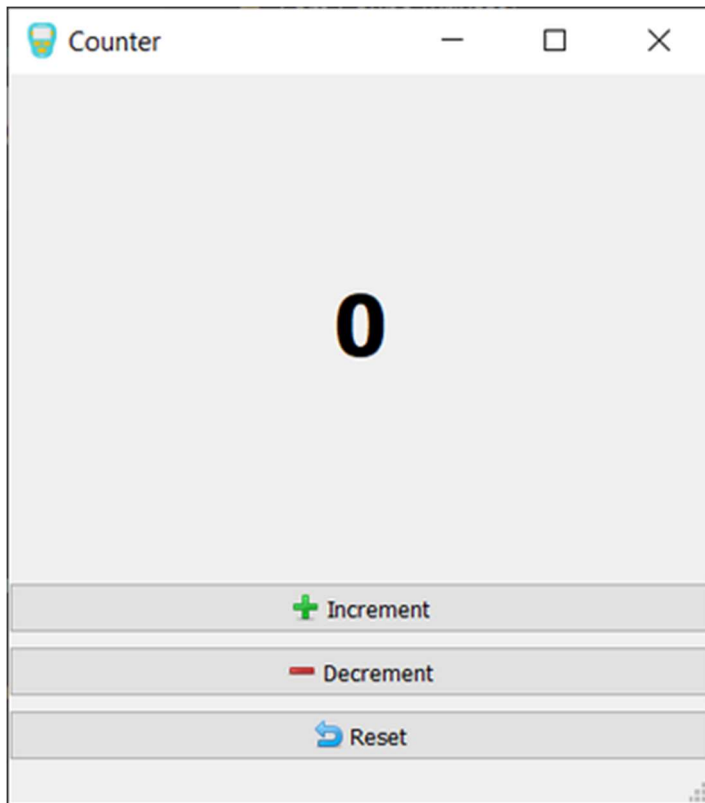
def reset(self):
    self.counter = 0
    self.update_counter()

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    app.setWindowIcon(QtGui.QIcon('/:icons/counter.ico'))
    main = WindowWrapper()
    sys.exit(app.exec_())

```

If you have made any changes to the `resources.qrc` file, or haven't compiled it yet do so now using `pyrcc5 resources.qrc -o resources.py` for **PyQt5** or `pyside2-rcc resources.qrc -o resources.py` for **PySide2**.

If you run this application you should see the following window.



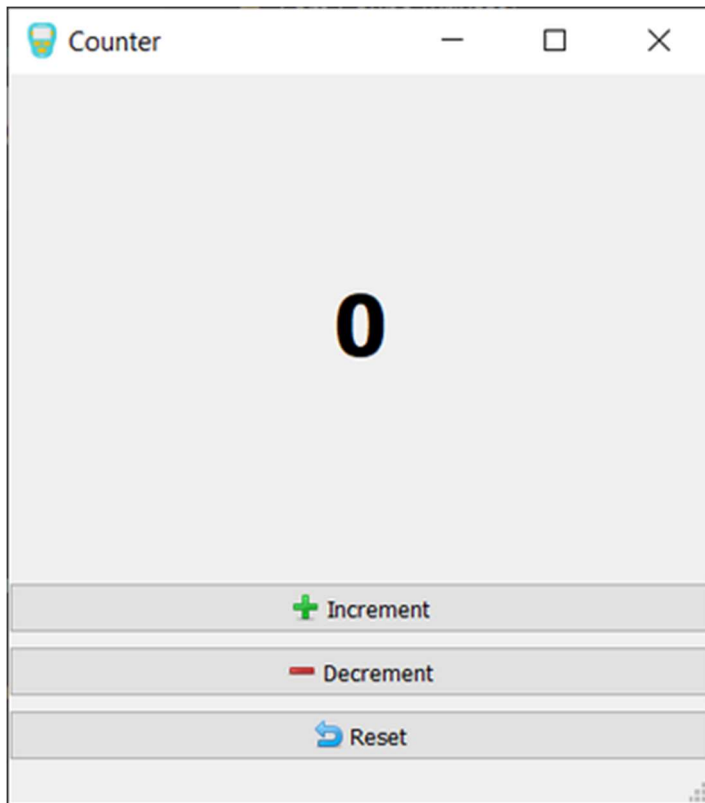
Counter app, with all icons showing

We'll build our app as before using the command line to perform an initial build and generate a `.spec` file for us. We can use that `.spec` file in future to repeat the build.

```
pyinstaller --windowed --icon=resources/counter.ico app.py
```

PyInstaller will analyse our `app.py` file, bundling all the necessary dependencies, including our compiled `resources.py` into the `dist` folder.

Once the build process is complete, open the `dist` folder and run the application. You should find it works, with all icons — from the application itself, through to the icons embedded in our UI file — working as expected.



Counter app, with all icons showing

This shows the advantage of using this approach — if your application works before bundling, you can be pretty sure it will continue to work after.

Build #3, bundling a complete app

The applications so far haven't done very much. Next we'll look at something more complete — our custom *Piecasso* Paint application. The source code is [available to download here](#) or in the [15 minute apps repository](#).

The source code is not covered in depth here, only the steps required to package the application. The source and more info is available in the [15 minute apps](#) repository of example Qt applications. The custom application icons were created using icon art by [Freepik](#).

Prepared for packaging the project has the following structure (truncated for clarity).

```
.
├── paint.py
├── Piecasso.spec
├── mainwindow.ui
├── MainWindow.py
├── README.md
├── requirements.txt
├── resources.qrc
├── resources_rc.py
├── screenshot-paint1.jpg
├── screenshot-paint2.jpg
├── icons
│   ├── blue-folder-open-image.png
│   ├── border-weight.png
│   ├── cake.png
│   ├── disk.png
│   ├── document-image.png
│   ├── edit-bold.png
│   └── ...
├── stamps
│   ├── pie-apple.png
│   ├── pie-cherry.png
│   ├── pie-cherry2.png
│   ├── pie-lemon.png
│   ├── pie-moon.png
│   ├── pie-pork.png
│   ├── pie-pumpkin.png
│   └── pie-walnut.png
```

The main source of the application is in the `paint.py` file.

Packaging Resources

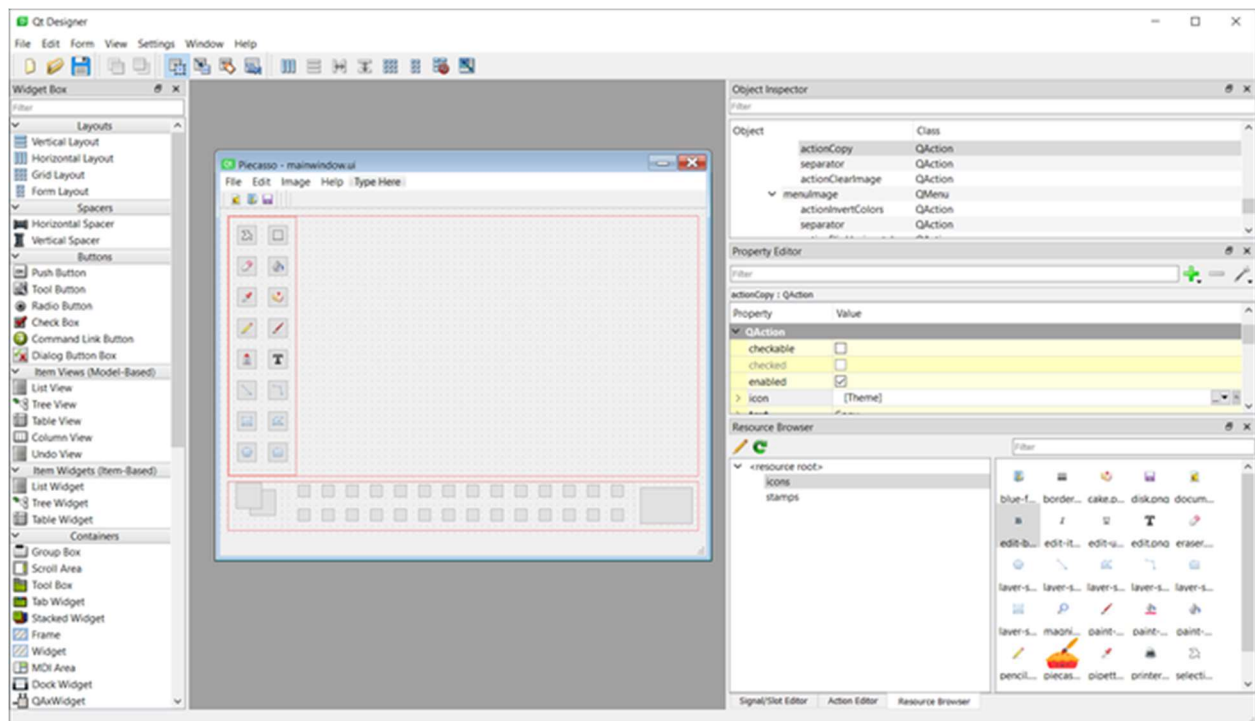
The resources for *Piecaso* are bundled using the Qt Resource system, referenced from the `resources.qrc` file in the root folder. There are two folders of images, `icons` which contains the icons for the interface and `stamps` which contains the pictures of pie for "stamping" the image when the application is running.

The `icons` were added to the UI in Qt Designer, while the stamps are loaded in the application itself using Qt Resource paths, e.g. `:/stamps/<image name>`.

The `icons` folder also contains the application icon, in `.ico` format.

The UI in Qt Designer

The UI for *Piecaso* was designed using Qt Designer. Icons on the buttons and actions were set from the Qt resources file already described.



Piecaso UI, created in Qt Designer

The resulting UI file was saved in the root folder of the project as `mainwindow.ui` and then compiled using the UI compiler to produce an importable `.py` file, as follows.

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

T> For more on building UIs with Qt Designer see the [introductory tutorial](#).

This build process also adds imports to `MainWindow.py` for the compiled version of the resources using in the UI, in our case `resources.qrc`. This means we do not need to import the resources separately into our app. However, we still need to build them, and use the specific name that is used for the import in `MainWindow.py`, here `resources_rc`.

```
pyrcc5 resources.qrc -o resources_rc.py
```

`pyuic5` follows the pattern `<resource name>_rc.py` when adding imports for the resource file, so you will need to follow this when compiling resources yourself. You can check your compiled UI file (e.g. `MainWindow.py`) to double check the name of the import if you have problems.

Building the app

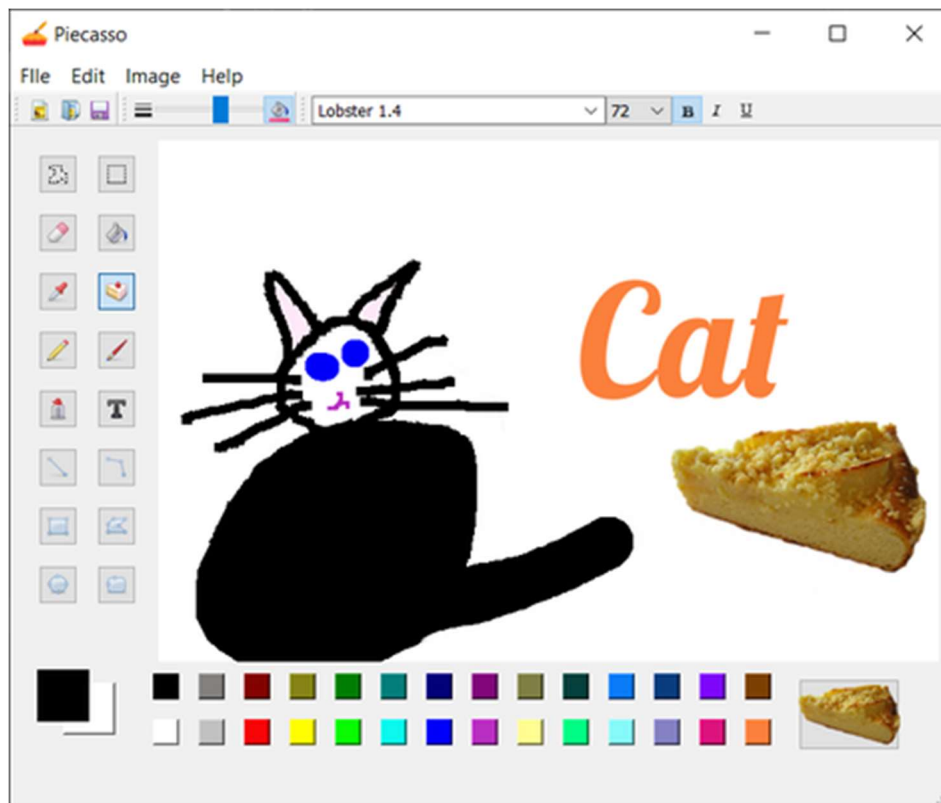
With all the above setup, we can build *Piecasso* as follows from the source folder.

```
pyinstaller --windowed --icon=icons/piecasso.ico --name Piecasso paint.py
```

If you download the source code, you will also be able to run the same build using the provided `.spec` file.

```
pyinstaller Piecasso.spec
```

This packages everything up ready to distribute in the `dist/Piecasso` folder. We can run the executable to ensure everything is bundled correctly, and see the following window, minus the terrible drawing.



Picasso Screenshot, with a poorly drawn cat

Creating an installer

So far we've used *PyInstaller* to bundle applications for distribution. The output of this bundling process is a folder, named `dist` which contains all the files our application needs to run.

While you *could* share this folder with your users as a ZIP file it's not the best user experience. Desktop applications are normally distributed with *installers* which handle the process of putting the executable (and any other files) in the correct place, adding *Start Menu* shortcuts and the like.

Now we've successfully bundled our complex application, we'll next look at how we can take our `dist` folder and use it to create a functioning Windows installer.

To create our installer we'll be using a tool called [InstallForge](#). InstallForge is free and you can download the installer from [this page](#).

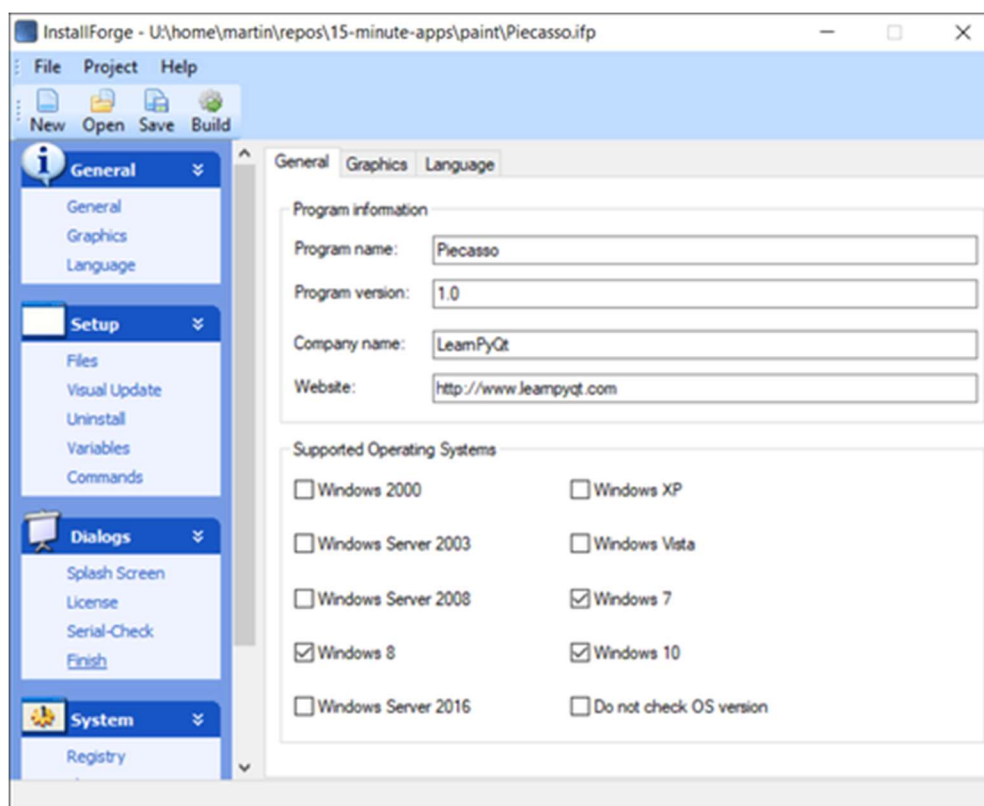
The InstallForge configuration is also in the Picasso source folder, `Picasso.ifp` however bear in mind that the source paths will need to be updated for your system.

Another popular tool is [NSIS](#) which is a *scriptable* installer, meaning you configure it's behaviour by writing custom scripts. If you're going to be building your application frequently and want to automate the process, it's definitely worth a look.

We'll now walk through the basic steps of creating an installer with *InstallForge*. If you're impatient, you can download the [Piecasso Installer for Windows here](#).

General

When you first run *InstallForge* you'll be presented with this General tab. Here you can enter the basic information about your application, including the name, program version, company and website.



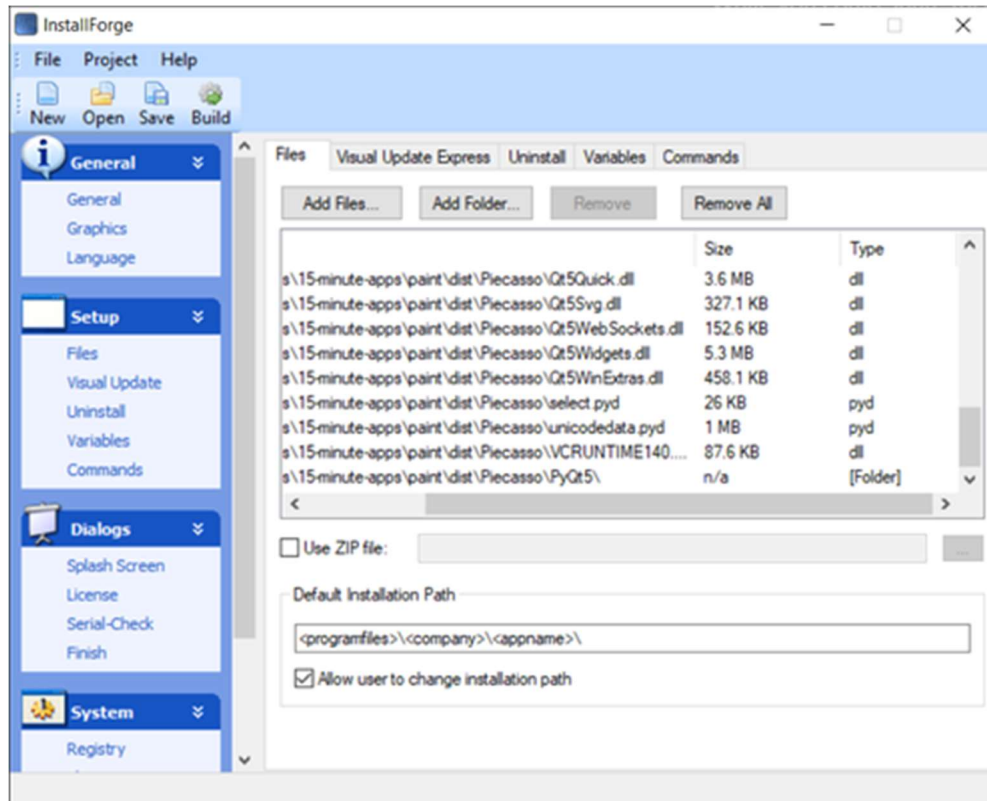
InstallForge initial view, showing General settings

You can also select the target platforms for the installer, from various versions of Windows that are available. For desktop applications you currently *probably* only want to target Windows 7, 8 and 10.

Setup

Click on the left sidebar to open the "Files" page under "Setup". Here you can specify the files to be bundled in the installer.

Use "Add Files..." and select *all the files* in the `dist/Picasso` folder produced by *PyInstaller*. The file browser that pops up allows multiple file selections, so you can add them all in a single go, however you need to add folders separately. Click "Add Folder..." and add any folders under `dist/Picasso` such as the `PyQt5` folder.

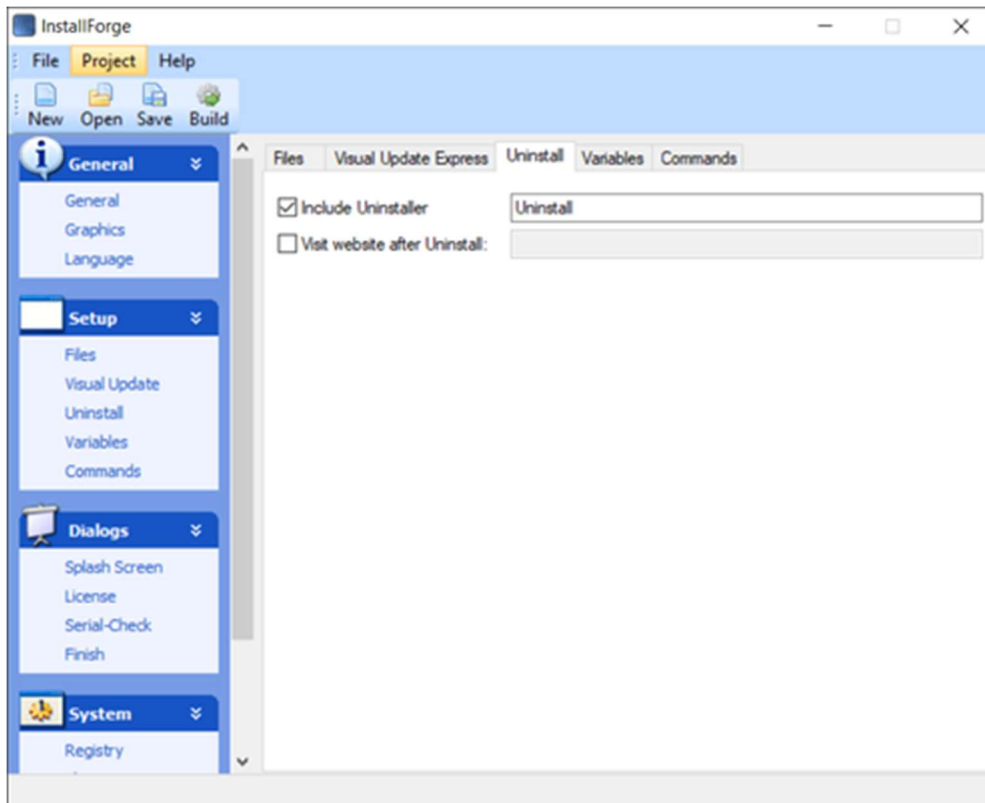


InstallForge Files view, add all files & folders to be packaged

Once you're finished scroll through the list to the bottom and ensure that the folders are listed to be included. You want all files and folders *under* `dist/Picasso` to be present. But the folder `dist/Picasso` itself *should not* be listed.

The default install path can be left as-is. The values between angled brackets, e.g. `<company>`, are variables and will be filled automatically.

Next, it's nice to allow your users to uninstall your application. Even though it's undoubtedly awesome, they may want to remove it at some time in the future. You can do this under the "Uninstall" tab, simply by ticking the box. This will also make the application appear in "Add or Remove Programs".

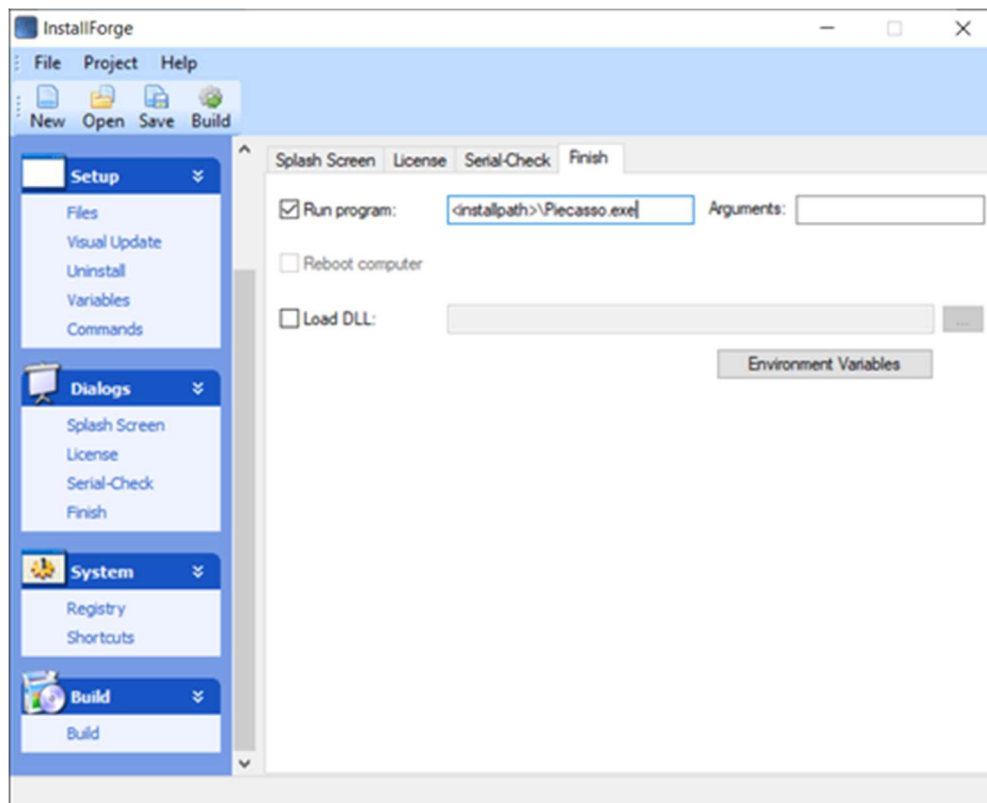


InstallForge add Uninstaller for your app

Dialogs

The "Dialogs" section can be used to show custom messages, splash screens or license information to the user. The "Finish" tab lets you control what happens once the installer is complete, and it's helpful here to give the user the *option* to run your program.

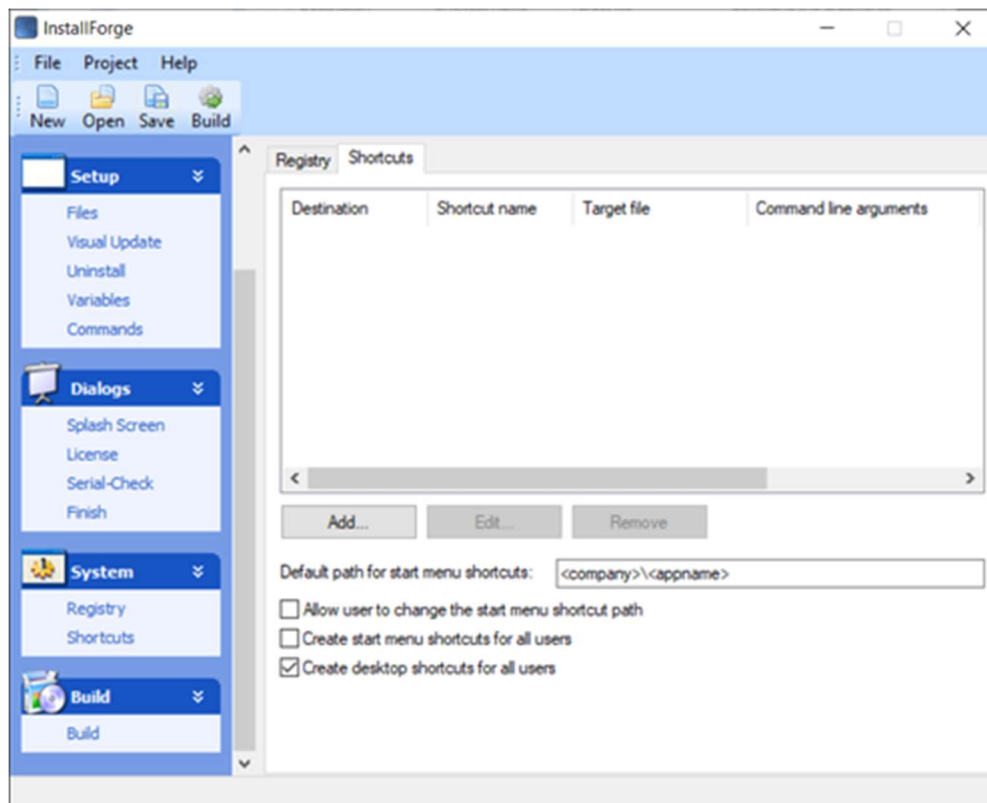
To do this you need to tick the box next to "Run program" and add your own application EXE into the box. Since `<installpath>\` is already specified, we can just add `Picasso.exe`.



InstallForge configure optional run program on finish install

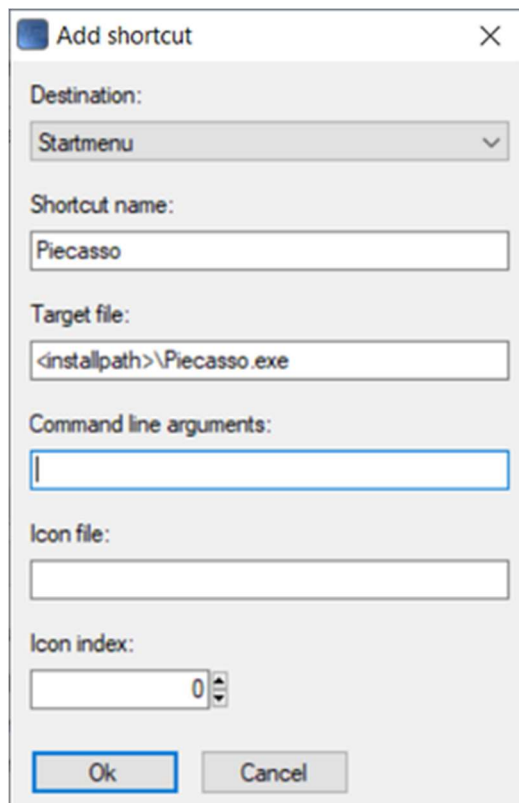
System

Under "System" select "Shortcuts" to open the shortcut editor. Here you can specify shortcuts for both the Start Menu and Desktop if you like.



InstallForge configure Shortcuts, for Start Menu and Desktop

Click "Add..." to add new shortcuts for your application. Choose between Start menu and Desktop shortcuts, and fill in the name and target file. This is the path your application EXE will end up at once installed. Since `<installpath>\` is already specified, you simply need to add your application's EXE name onto the end, here `Picasso.exe`



Add shortcut [X]

Destination:
Startmenu [v]

Shortcut name:
Picasso

Target file:
<installpath>\Picasso.exe

Command line arguments:
[]

Icon file:
[]

Icon index:
0 [up/down arrows]

[Ok] [Cancel]

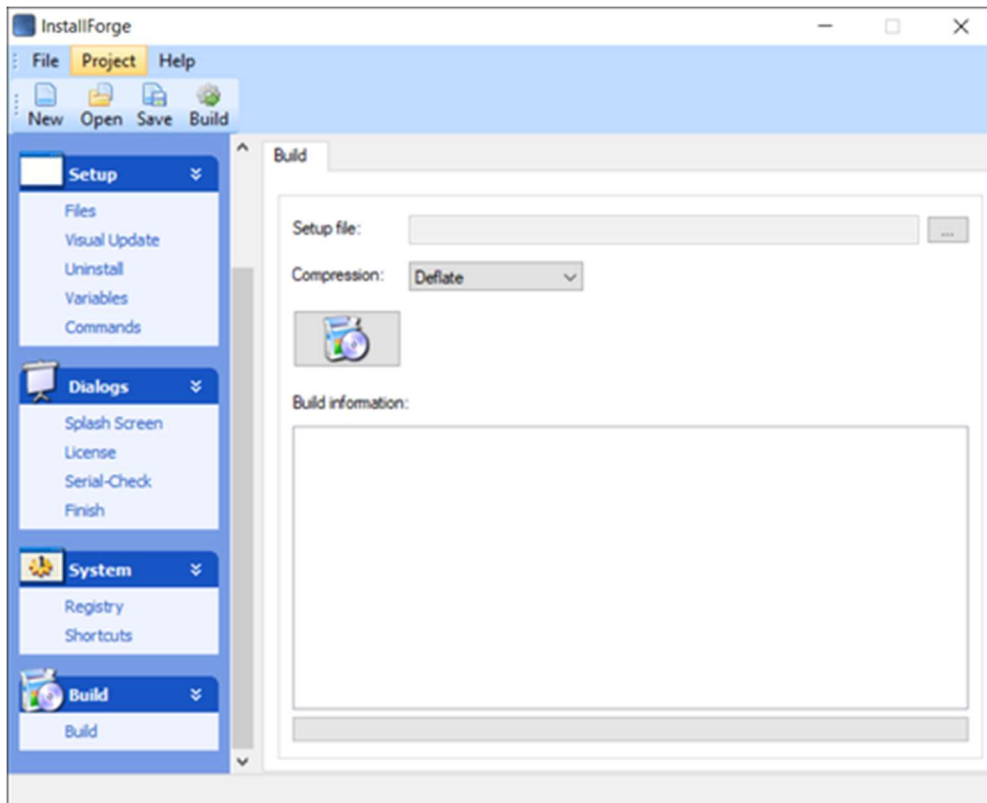
InstallForge, adding a Shortcut

Build

With the basic settings in place, you can now build your installer.

At this point you can save your *InstallForge* project so you can re-build the installer from the same settings in future.

Click on the "Build" section at the bottom to open the build panel.

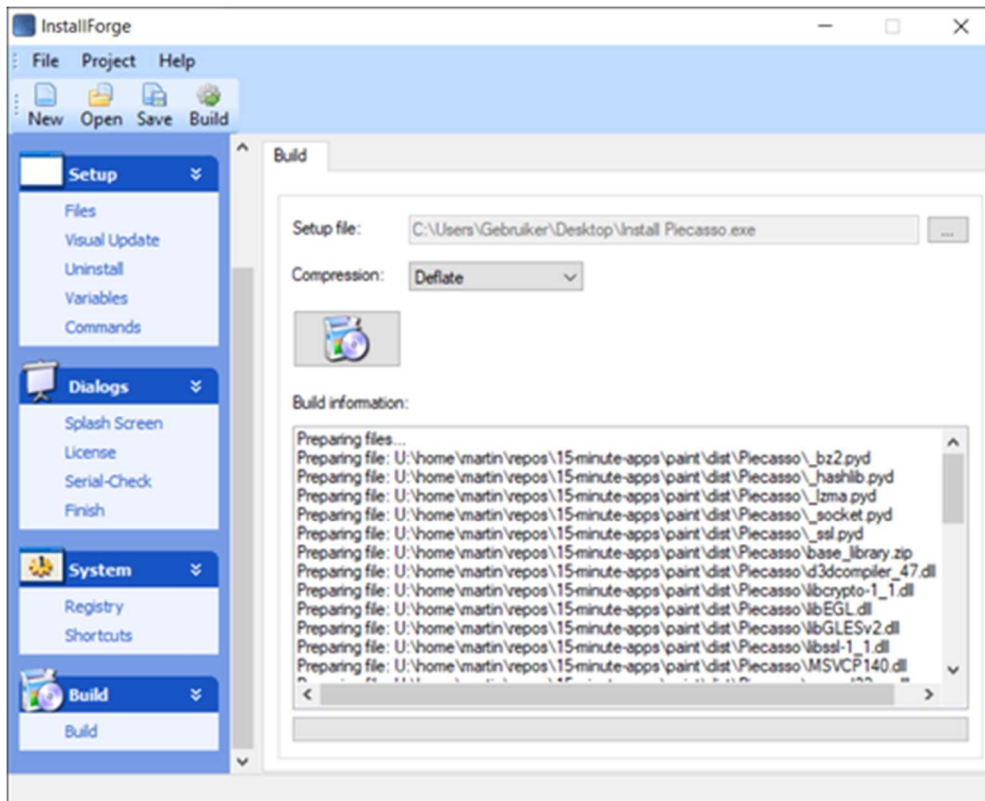


InstallForge, ready to build

Click on the large icon button to start the build process. If you haven't already specified a setup file location you will be prompted for one. This is the location where you want the *completed installer* to be saved.

Don't save it in your `dist` folder.

The build process will begin, collecting and compressing the files into the installer.

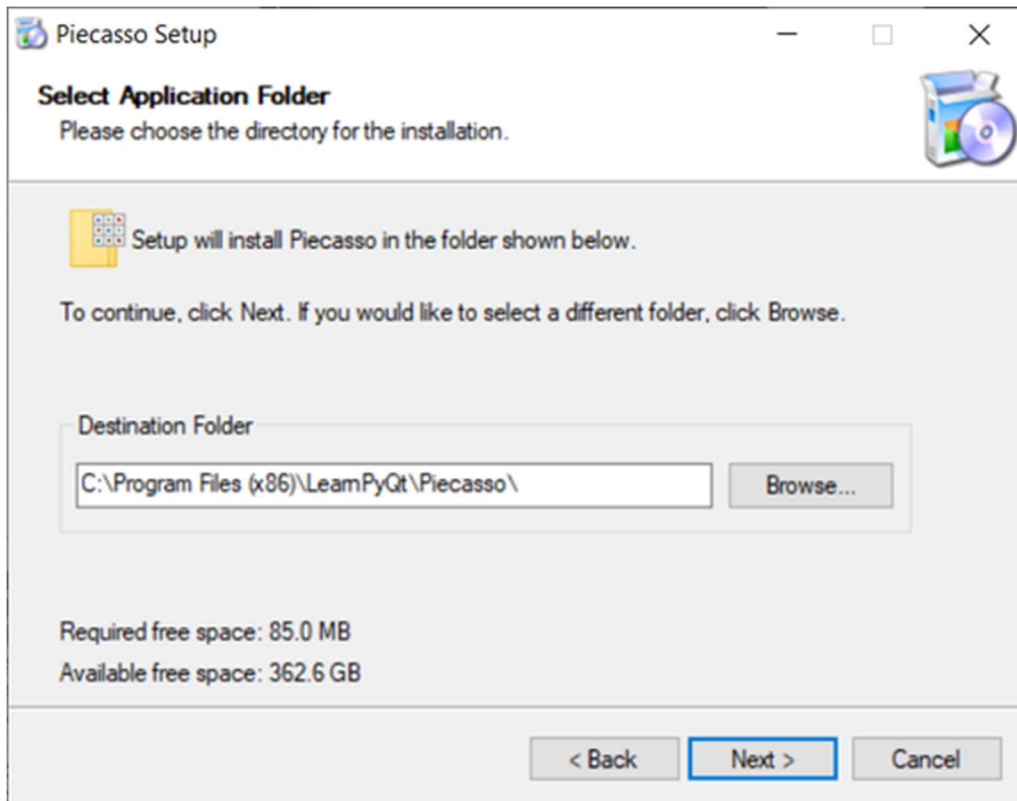


InstallForge, build complete

Once complete you will be prompted to run the installer. This is entirely optional, but a handy way to find out if it works.

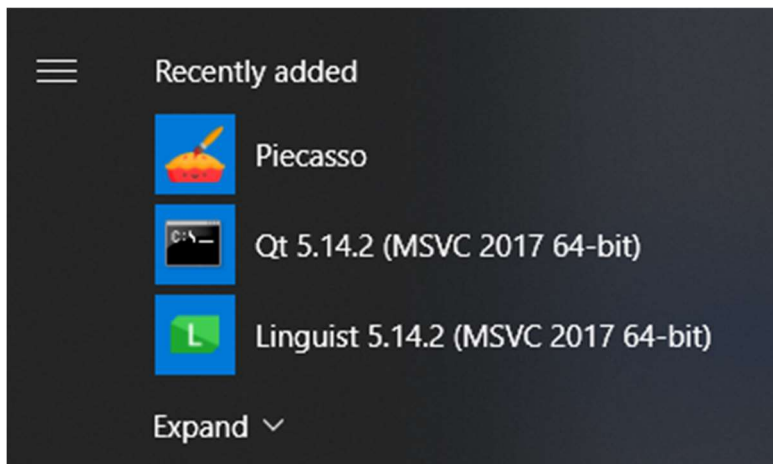
Running the installer

The installer itself shouldn't have any surprises, working as expected. Depending on the options selected in *InstallForge* you may have extra panels or options.



InstallForge, running the resulting installer

Step through the installer until it is complete. You can optionally run the application from the last page of the installer, or you can find it in your start menu.



Picasso in the Start Menu on Windows 10

Wrapping up

In this tutorial we've covered how to build your PyQt5 or PySide2 applications into a distributable EXE using *PyInstaller*. Following this we walked through the steps of using

InstallForge to build [an installer for the app](#). Following these steps you should be able to package up your own applications and make them available to other people.

For a complete view of all *PyInstaller* bundling options take a look at the [PyInstaller usage documentation](#).

Enjoyed this?

I've also written a book.

Create Simple GUI Applications is my hands-on guide to making desktop apps with Python. Learn everything you need to build *real* apps.



Share on [Twitter](#) [Facebook](#) [Reddit](#) and help support this site!

1/2

[Sign up](#) to track your progress!

Curriculum

- [Packaging PyQt5 & PySide2 applications for Windows, with PyInstaller](#) (23:57)
- [Packaging PyQt5 apps with fbs](#) (10:38)

Table of contents

- [Requirements](#)
- [Getting Started](#)
- [Build #1, a basic app](#)
- [The Spec file](#)
- [Tweaking the build](#)
- [Data files and Resources](#)
- [Build #2, bundling Qt Designer UIs and Icons](#)
- [Build #3, bundling a complete app](#)
- [Creating an installer](#)
- [Wrapping up](#)

Share

- [Twitter](#)
- [Facebook](#)
- [Reddit](#)

Discussion

- [PyQt5 Installation](#)
- [PyQt5 Courses](#)
- [PyQt5 Example Apps](#)
- [Widget Library](#)
- [PyQt5 tutorial](#)
- [Latest articles](#)
- [Write with me](#)
- [Contact us](#)
- [Affiliate program](#)
- [Licensing, Privacy & Legal](#)

Learn PyQt — Copyright ©2019-2020 [Martin Fitzpatrick](#) Tutorials CC-BY-NC-SA Public code [BSD](#) & [MIT](#)

Registered in the Netherlands with the KvK [66435102](#)