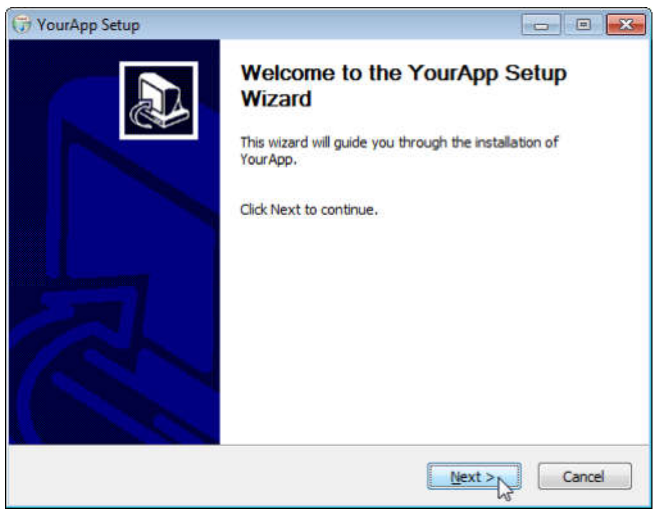

YourApp Setup
Welcome to the YourApp Setup Wizard
This wizard will guide you through the installation of YourApp.
Click Next to continue.
Next >
Cancel

PyQt5 tutorial

Learn how you can create a Python GUI in 2019.

This PyQt5 tutorial shows how to use Python 3 and Qt to create a GUI on Windows, Mac or Linux. It even covers creating an installer for your app.

What is PyQt5?

PyQt is a library that lets you use the Qt GUI framework from Python. Qt itself is written in C++. By using it from Python, you can build applications much more quickly while not sacrificing much of the speed of C++.

PyQt5 refers to the most recent version 5 of Qt. You may still find the occasional mention of (Py)Qt4 on the web, but it is old and no longer supported.

An interesting new competitor to PyQt is Qt for Python. Its API is virtually identical. Unlike PyQt, it is licensed under the LGPL and can thus be used for free in commercial projects. It's backed by the Qt company, and thus likely the future. We use PyQt here because it is more mature. Since the APIs are so similar, you can easily switch your apps to Qt for Python later.

Install PyQt

The best way to manage dependencies in Python is via a virtual environment. A virtual environment is simply a local directory that contains the libraries for a specific project. This is unlike a system-wide installation of those libraries, which would affect all of your other projects as well.

To create a virtual environment in the current directory, execute the following command:

```
python3 -m venv venv
```

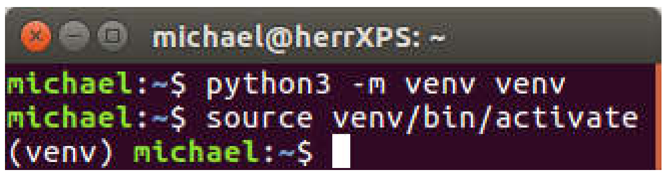
This creates the `venv/` folder. To activate the virtual environment on Windows, run:

```
call venv/scripts/activate.bat
```

On Mac and Linux, use:

```
source venv/bin/activate
```

You can see that the virtual environment is active by the `(venv)` prefix in your shell:


```
michael@herrXPS: ~
michael:~$ python3 -m venv venv
michael:~$ source venv/bin/activate
(venv) michael:~$
```

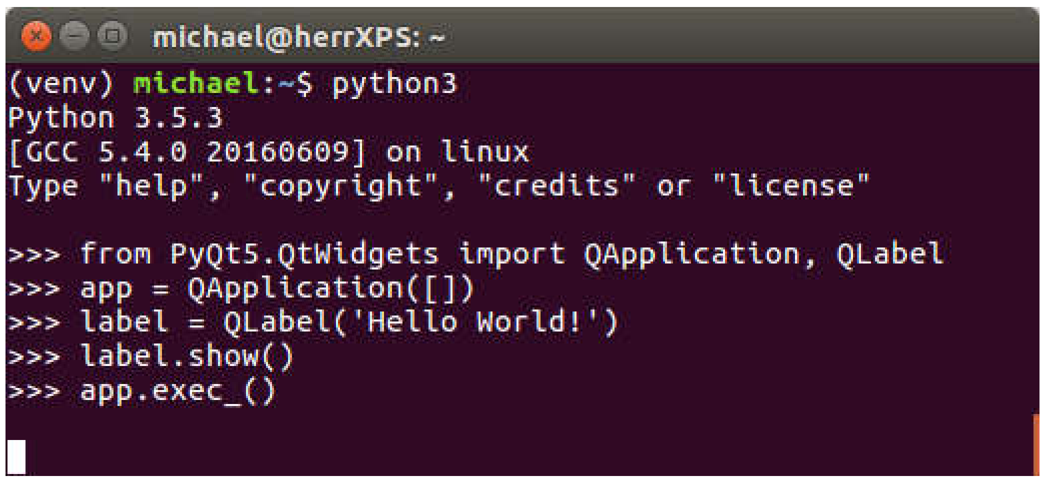
To now install PyQt, issue the following command:

```
pip install PyQt5==5.9.2
```

The reason why we're using version `5.9.2` is that not all (Py)Qt releases are equally stable. This version is guaranteed to work. Besides this subtlety – Congratulations! You've successfully set up PyQt5.

Create a GUI

Time to write our very first GUI app! With the virtual environment still active, start Python. We will execute the following commands:



```
michael@herrXPS: ~
(venv) michael:~$ python3
Python 3.5.3
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license"

>>> from PyQt5.QtWidgets import QApplication, QLabel
>>> app = QApplication([])
>>> label = QLabel('Hello World!')
>>> label.show()
>>> app.exec_()
```

First, we tell Python to load PyQt via the import statement:

```
from PyQt5.QtWidgets import QApplication, QLabel
```

Next, we create a `QApplication` with the command:

```
app = QApplication([])
```

This is a requirement of Qt: Every GUI app must have exactly one instance of `QApplication`. Many parts of Qt don't work until you have executed the above line. You will therefore need it in virtually every (Py)Qt app you write.

The brackets `[]` in the above line represent the command line arguments passed to the application. Because our app doesn't use any parameters, we leave the brackets empty.

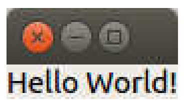
Now, to actually see something, we create a simple label:

```
label = QLabel('Hello World!')
```

Then, we tell Qt to show the label on the screen:

```
label.show()
```

Depending on your operating system, this already opens a tiny little window:



Hello World!

The last step is to hand control over to Qt and ask it to "run the application until the user closes it". This is done via the command:

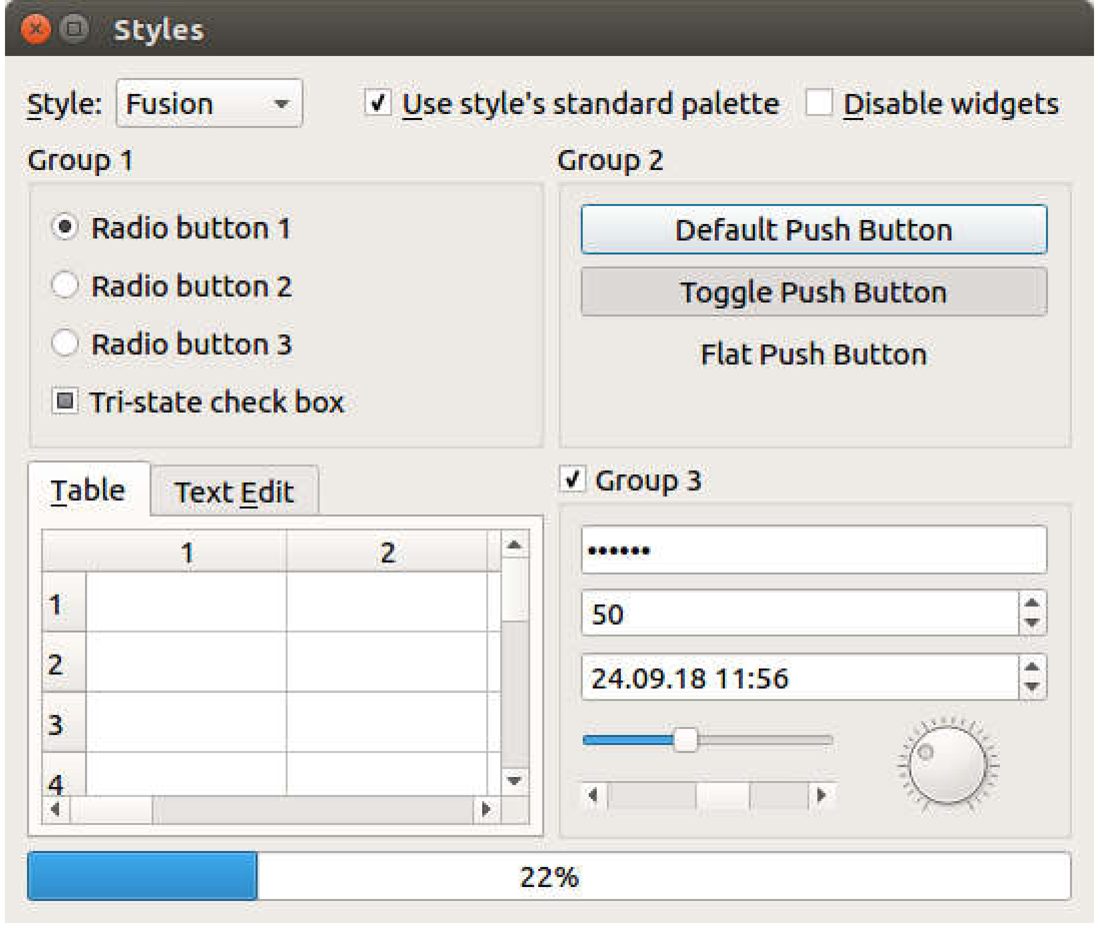
```
app.exec_()
```

If all this worked as expected then well done! You've just built your first GUI app with Python and Qt.

Widgets

Everything you see in a (Py)Qt app is a *widget*: Buttons, labels, windows, dialogs, progress bars etc. Like HTML elements, widgets are often nested. For example, a window can contain a button, which in turn contains a label.

The following screenshot shows the most common Qt widgets:



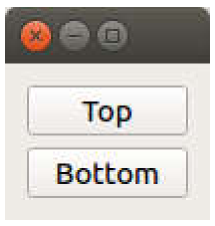
Top-to-bottom, left-to-right, they are:

- QLabel
- QComboBox
- QCheckBox
- QRadioButton
- QPushButton
- QTableWidget
- QLineEdit
- QSlider
- QProgressBar

You can download the code for the app shown in the screenshot here, if you are interested.

Layouts

Like the example above, your GUI will most likely consist of multiple widgets. In this case, you need to tell Qt how to position them. For instance, you can use QVBoxLayout to stack widgets vertically:



The code for this screenshot is:

```
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QVBoxLayout
app = QApplication([])
window = QWidget()
layout = QVBoxLayout()
layout.addWidget(QPushButton('Top'))
layout.addWidget(QPushButton('Bottom'))
window.setLayout(layout)
window.show()
app.exec_()
```

As before, we instantiate a `QApplication`. Then, we create a `window`. We use the most basic type `QWidget` for it because it merely acts as a container and we don't want it to have any special behavior. Next, we create the `layout` and add two `QPushButton`s to it. Finally, we tell the window to use this layout (and thus its contents). As in our first application, we end with calls to `.show()` and `app.exec_()`.

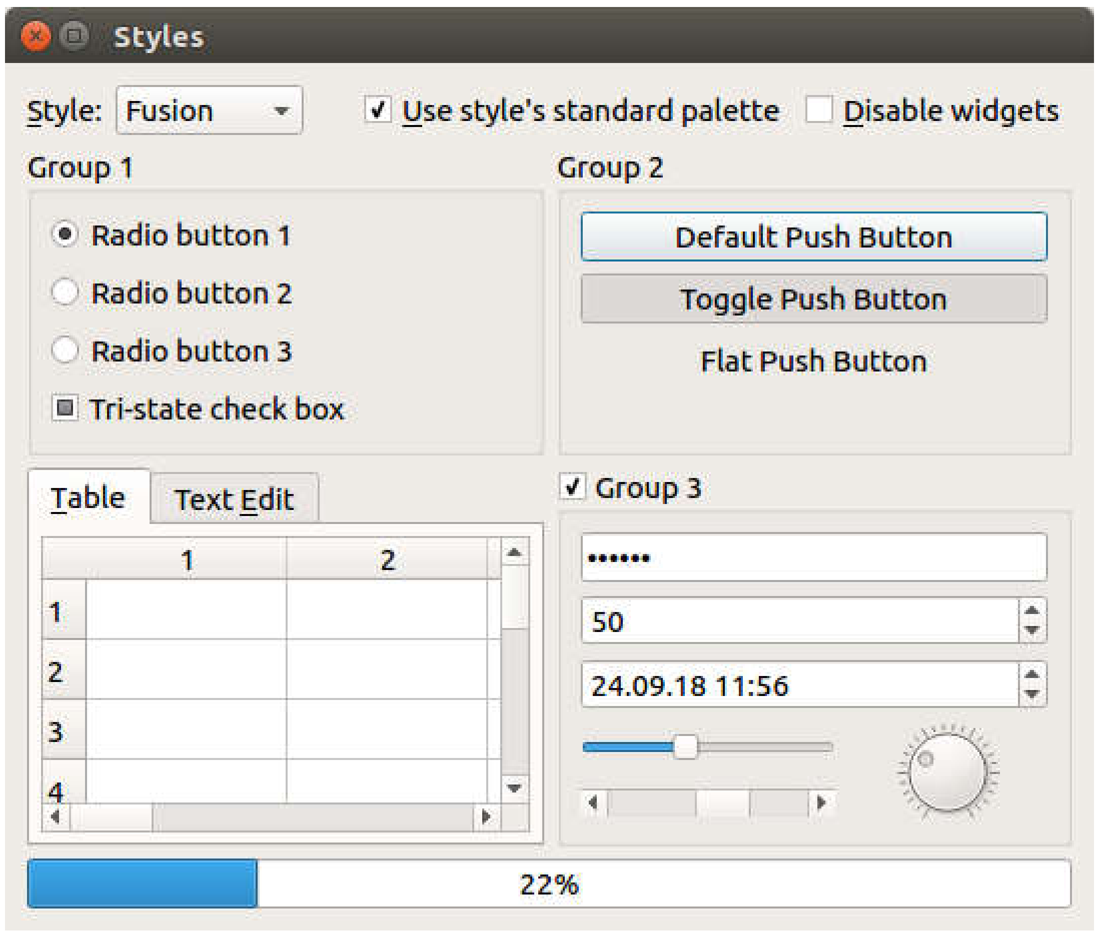
There are of course many other kinds of layouts (eg. QHBoxLayout to lay out items in a row). See Qt's documentation for an overview.

Custom styles

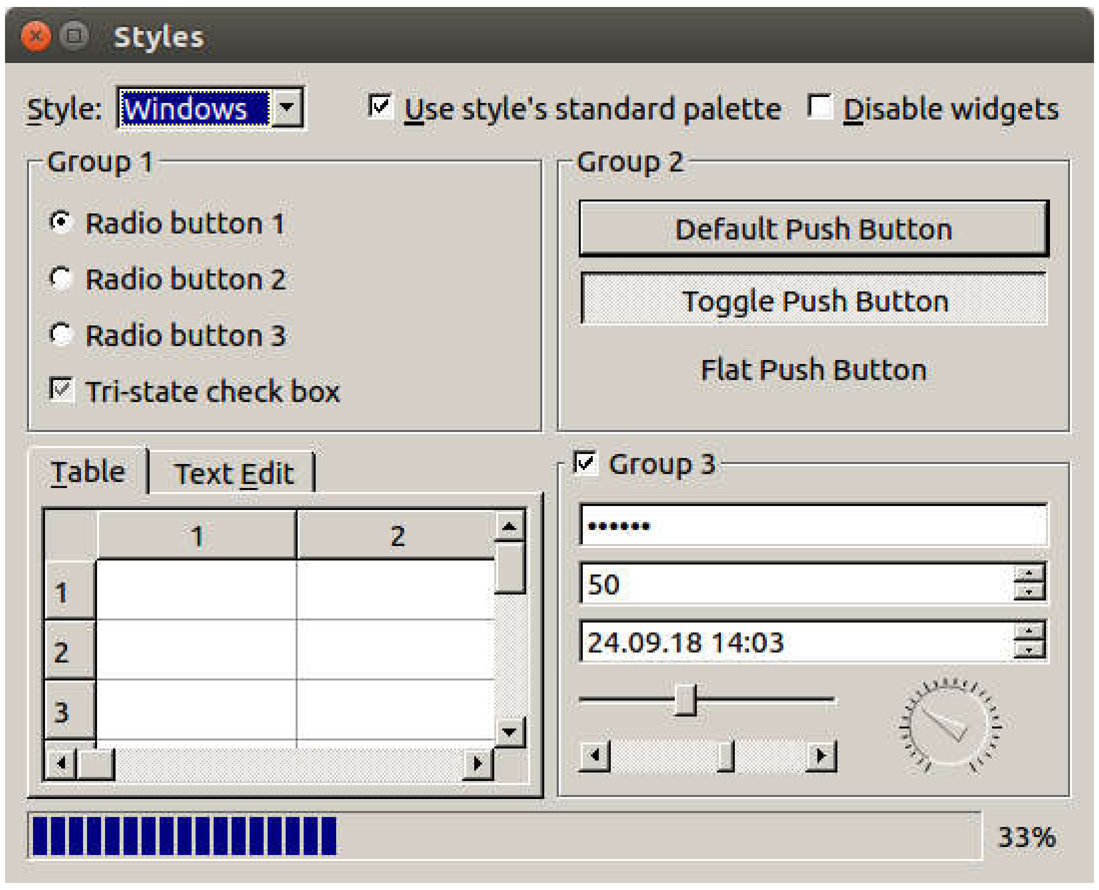
One of Qt's strengths is its support for custom styles. There are many mechanisms that let you customize the look and feel of your application. This section outlines a few.

Built-in styles

The coarsest way to change the appearance of your application is to set the global Style. Recall the widgets screenshot above:



This uses a style called `Fusion`. If you use the `Windows` style instead, then it looks as follows:



To apply a style, use `app.setStyle(...)`:

```
from PyQt5.QtWidgets import *
app = QApplication([])
app.setStyle('Fusion')
...
```

The available styles depend on your platform but are usually `'Fusion'`, `'Windows'`, `'WindowsVista'` (Windows only) and `'Macintosh'` (Mac only).

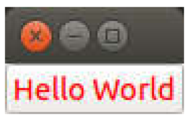
Custom colors

If you like a style, but want to change its colors (eg. to a dark theme), then you can use QPalette and `app.setPalette(...)`. For example:

```
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QPalette
from PyQt5.QtWidgets import QApplication, QPushButton

app = QApplication([])
app.setStyle('Fusion')
palette = QPalette()
palette.setColor(QPalette.ButtonText, Qt.red)
app.setPalette(palette)
button = QPushButton('Hello World')
button.show()
app.exec_()
```

This changes the text color in buttons to red:

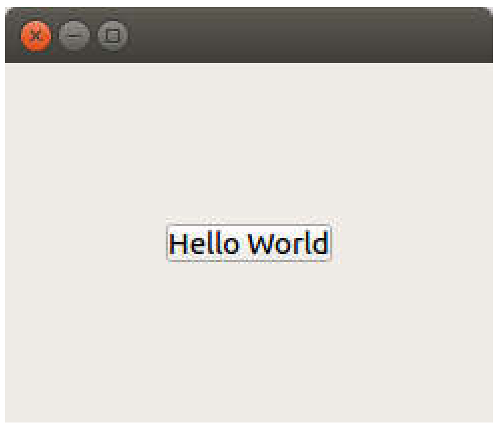


For a dark theme of the Fusion style, see here.

Style sheets

In addition to the above, you can change the appearance of your application via *style sheets*. This is Qt's analogue of CSS. We can use this for example to add some spacing:

```
from PyQt5.QtWidgets import QApplication, QPushButton
app = QApplication([])
app.setStyleSheet("QPushButton { margin: 10ex; }")
button = QPushButton('Hello World')
button.show()
app.exec_()
```



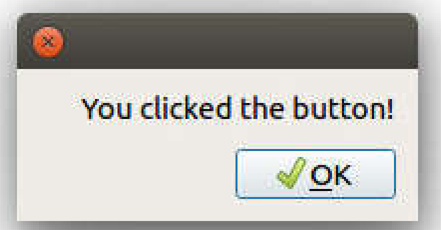
For more information about style sheets, please see Qt's documentation.

Signals / slots

Qt uses a mechanism called *signals* to let you react to events such as the user clicking a button. The following example illustrates this. It contains a button that, when clicked, shows a message box:

```
from PyQt5.QtWidgets import *
app = QApplication([])
button = QPushButton('Click')
def on_button_clicked():
    alert = QMessageBox()
    alert.setText('You clicked the button!')
    alert.exec_()

button.clicked.connect(on_button_clicked)
button.show()
app.exec_()
```



The interesting line is highlighted above: `button.clicked` is a signal, `.connect(...)` lets us install a so-called *slot* on it. This is simply a function that gets called when the signal occurs. In the above example, our slot shows a message box.

The term slot is important when using Qt from C++, because slots must be declared in a special way in C++. In Python however, any function can be a slot – we saw this above. For this reason, the distinction between slots and "normal" functions has little relevance for us.

Signals are ubiquitous in Qt. And of course, you can also define your own. This however is beyond the scope of this tutorial.

Compile your app

You now have the basic knowledge for creating a GUI that responds to user input. Say you've written an app. It runs on your computer. How do you give it to other people, so they can run it as well?

You could ask the users of your app to install Python and PyQt like we did above, then give them your source code. But that is very tedious (and usually impractical). What we want instead is a *standalone* version of your app. That is, a binary executable that other people can run on their systems without having to install anything.

In the Python world, the process of turning source code into a self-contained executable is called *freezing*. Although there are many libraries that address this issue – such as PyInstaller, py2exe, cx_Freeze, bbfreze, py2app, ... – freezing PyQt apps has traditionally been a surprisingly hard problem.

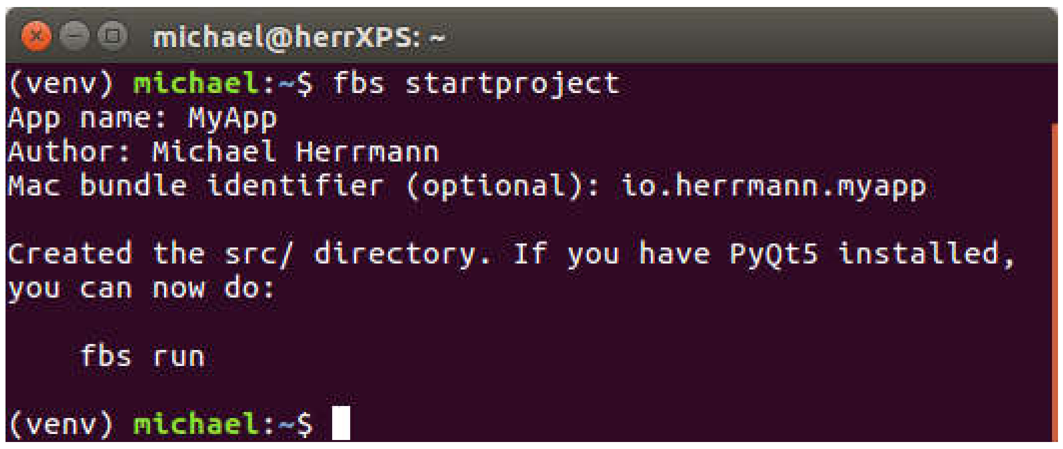
We will use a new library called fbs that lets you create standalone executables for PyQt apps. To install it, enter the command:

```
pip install fbs PyInstaller==3.4
```

Then, execute the following:

```
fbs startproject
```

This prompts you for a few values:



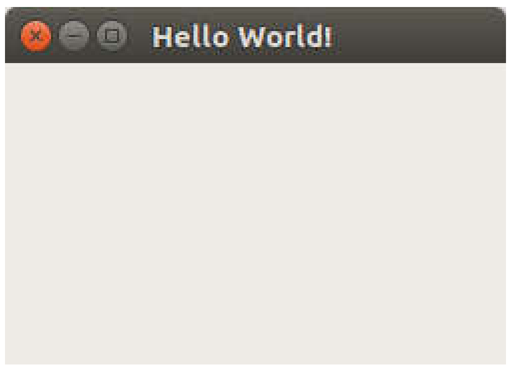
```
(venv) michael:~$ fbs startproject
App name: MyApp
Author: Michael Herrmann
Mac bundle identifier (optional): io.herrmann.myapp

Created the src/ directory. If you have PyQt5 installed,
you can now do:

    fbs run

(venv) michael:~$
```

When you type in the suggested `run` command, an empty window should open:



This is a PyQt5 app just like the ones we have seen before. Its source code is in `src/main/python/main.py` in your current directory. But here's the cool part: We can use fbs to turn it into a standalone executable!

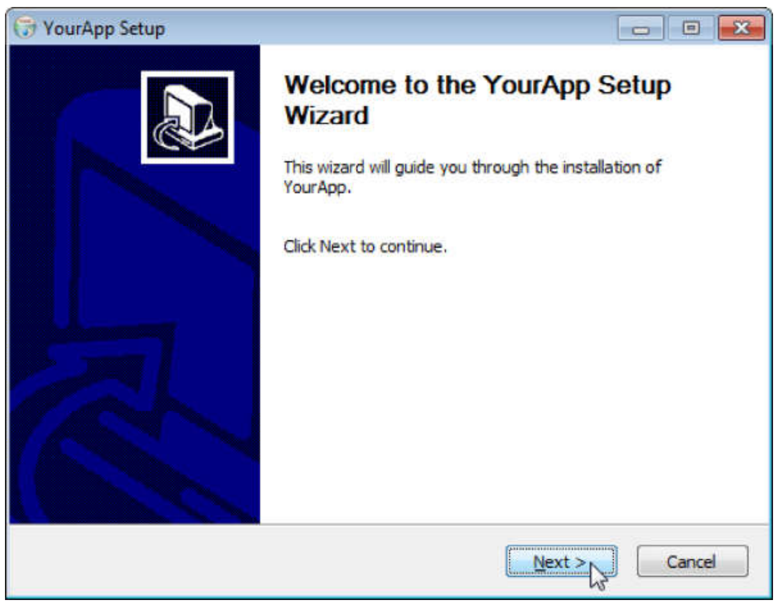
```
fbs freeze
```

This places a self-contained binary in the `target/MyApp/` folder of your current directory. You can send it to your friends (with the same OS as yours) and they will be able to run your app!

(Please note that fbs currently targets Python 3.5 or 3.6. If you have a different version and the above does not work, please install Python 3.6 and try again. On macOS, you can also install Python 3.5 with Homebrew.)

Bonus: Create an installer

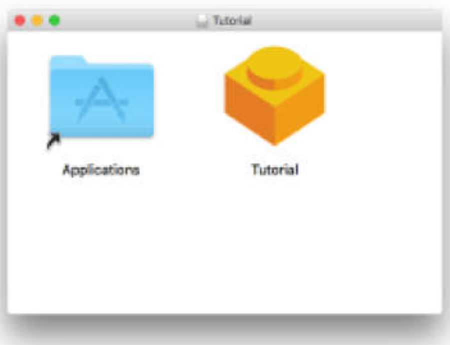
fbs also lets you create an installer for your app via the command `fbs installer`:


YourApp Setup

Welcome to the YourApp Setup Wizard

This wizard will guide you through the installation of YourApp.

Click Next to continue.



(If you are on Windows, you first need to install NSIS and place it on your `PATH`.)

For more information on how you can use fbs for your existing application, please see this article. Or fbs's tutorial.

Summary

If you have made it this far, then big congratulations. Hopefully, you now have a good idea of how PyQt (and its various parts) can be used to write a desktop application with Python. We also saw how fbs lets you create standalone executables and installers.

If you have any questions or feedback on this tutorial, feel free to email me. Have fun writing your own apps!

Michael has been working with PyQt5 since 2016, when he started fman, a cross-platform file manager. Frustrated with the many challenges of creating a desktop application, Michael open sourced fman's build system (fbs). It lets you create PyQt GUIs in in minutes instead of months!