Python Qt tutorial Documentation

Release 0.0

Thomas P. Robitaille

Jun 11, 2018

Contents

This is a short tutorial on using Qt from Python. There are two main versions of Qt in use (Qt4 and Qt5) and several Python libraries to use Qt from Python (PyQt and PySide), but rather than picking one of these, this tutorial makes use of the QtPy package which provides a way to use whatever Python Qt package is available.

This is not meant to be a completely exhaustive tutorial but just a place to start if you've never done Qt development before, and it will be expanded over time.

CHAPTER 1

Installing

1.1 conda

If you use conda to manage your Python environment (for example as part of Anaconda or Miniconda), you can easily install Qt, PyQt5, and QtPy (a common interface to all Python Qt bindings) using:

```
conda install pyqt qtpy
```

1.2 pip

If you don't make use of conda, an easy way to install Qt, PyQt5, and QtPy is to do:

```
pip install pyqt5 qtpy
```

CHAPTER 2

Part 1 - Hello, World!

To start off, we need to create a `QApplication` object, which represents the overall application:

```
from qtpy.QtWidgets import QApplication
app = QApplication([])
```

You will always need to ensure that a `QApplication` object exists, otherwise your Python script will terminate with an error if you try and use any other Qt objects. The above won't open any window - the term *application* here doesn't refer to a specific window that will open, but instead to a windowless object that forms the basis for anything else we will build.

Building blocks in Qt (and other similar frameworks) are called *widgets*. One of the simplest widgets is a text label. We can create a label widget by doing:

```
from qtpy.QtWidgets import QLabel
label = QLabel('Hello, world!')
label.show()
```

Note that this should be done after creating a `QApplication` object. We need to explicitly tell Qt that we want to show the widget as a graphical window using `.show()`, otherwise it will not be shown (this will come in handy once we want to use widgets as building blocks for windows).

If you were to run the two code blocks above, the label would appear and disappear straight away because the script would finish running. The final step is to start the event loop. In graphical user interface (GUI) programming, an *event loop* is a common concept - the idea is to basically have an infinite loop that will continuously check for user interaction with the interface, such as clicking on buttons, entering text, and moving or closing windows. The infinite loop can be terminated in various ways depending on the configuration in `QApplication` - by default it will terminate once the visible windows are closed. To start the event loop, we can do:

```
app.exec_()
```

This is equivalent to doing:

```
while True:
    app.processEvents()
```

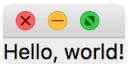
Let's take a look at the complete example:

```
from qtpy.QtWidgets import QApplication, QLabel

# Initialize application
app = QApplication([])

# Create label widget
label = QLabel('Hello, world!')
label.show()

# Start 'event loop'
app.exec_()
```

Copy this into a Python script, and run it with e.g.:

```
python 1.hello.py
```

You should see a small window pop up with 'Hello, world!' inside:



Close the window, and the Python script should stop running.

Congratulations, you've written your first Qt application!

CHAPTER 3

Part 2 - Buttons and events

Let's be honest - the application we wrote in *Part 1 - Hello, World!* wasn't the most exciting one ever. In this section, we're going to look at how to use buttons, and we will learn about events in Qt.

First, let's create a button (remember that as mentioned in *Part 1 - Hello, World!*, you should always set up a `QApplication` first):

```
from qtpy.QtWidgets import QPushButton
button = QPushButton('Say hello')
```

Qt widgets often have different kinds of *events* you can connect to. An event is just something that happens for example (but not limited to) something the user does. For instance, `button` has an event called `clicked` that gets triggered when the user clicks on the button. The normal way to use events is to write functions that can then do something as a result of the event being triggered.

Let's write a function that will get executed when the user clicks on the button:

```
def say_hello(event):
    print('Hello, world!')
```

Such a function typically takes the event as its only positional argument.

We can then connect this function to the `clicked` event:

```
button.clicked.connect(say_hello)
```

Finally, we can show the button with:

```
button.show()
```

Let's take a look at the full example:

```
from qtpy.QtWidgets import QApplication, QPushButton

# Initialize application
app = QApplication([])
```

(continues on next page)

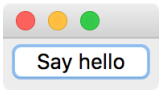
(continued from previous page)

```
# Define a function to connect to the button
def say_hello(event):
    print('Hello, world!')

# Create button widget and connect to 'say_hello'
button = QPushButton('Say hello')
button.clicked.connect(say_hello)
button.show()

# Start 'event loop'
app.exec_()
```

Copy this into a Python script, and run it with e.g.:

```
python 2.button.py
```

You should see a small window pop up with a button inside:



Now click on the button and you should see the following message appear in the terminal:

```
Hello, world!
```

Such fun!

CHAPTER 4

Part 3 - Laying out widgets

4.1 Vertical layout

Let's now take a look at how to construct more interesting application windows by combining different widgets together. Qt widgets can all include a 'layout', which defines combinations of children widgets. A simple example is the `QVBoxLayout` class, which allows widgets to be stacked vertically. Let's use a label and a button widget and stack these vertically into a layout:

```
from qtpy.QtWidgets import QLabel, QVBoxLayout, QPushButton

# Create label and button
label = QLabel('Hello, world!')
button = QPushButton('test')

# Create layout and add widgets
layout = QVBoxLayout()
layout.addWidget(label)
layout.addWidget(button)
```

However, note that a layout is not a widget, so we can't simply do `layout.show()` at this point. Instead, we need to add this layout to a parent widget. For this, we'll use `QWidget`, which is the most basic kind of a widget - it contains nothing by default:

```
widget = QWidget()
widget.setLayout(layout)
```

We can now show this widget using:

```
widget.show()
```

Note that in this kind of situation, it's clear why we don't want widgets to be shown by default and to have to specify `widget.show()`. If we were to instead run `button.show()`, only the button widget would be shown.

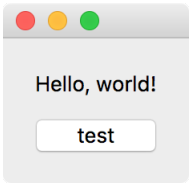
Here is the complete example:

```
from qtpy.QtWidgets import (QApplication, QLabel, QWidget,
                            QVBoxLayout, QPushButton)

# Initialize application
app = QApplication([])

# Create label and button
label = QLabel('Hello, world!')
button = QPushButton('test')

# Create layout and add widgets
layout = QVBoxLayout()
layout.addWidget(label)
layout.addWidget(button)

# Apply layout to widget
widget = QWidget()
widget.setLayout(layout)

# Show widget
widget.show()

# Start event loop
app.exec_()
```

Copy this into a Python script, and run it with e.g.:

```
python 3.vertical_layout.py
```

You should see a small window pop up with the two widgets:

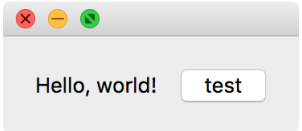


4.2 Other layouts

There are other types of layout besides vertical stacking. As you might expect, there is a corresponding `QHBoxLayout` class that will stack widgets horizontally:

```
from qtpy.QtWidgets import QHBoxLayout

layout = QHBoxLayout()
layout.addWidget(label)
layout.addWidget(button)
```

which will look like:



Another example is `QGridLayout` which allows widgets to be places in a grid. To add widgets to this kind of layout, you need to specify the row and column:

```
# Create label and button
label1 = QLabel('Label 1')
label2 = QLabel('Label 2')
label3 = QLabel('Label 3')
button = QPushButton('Press me!')

# Create layout and add widgets
layout = QGridLayout()
layout.addWidget(label1, 0, 0)
layout.addWidget(label2, 1, 0)
layout.addWidget(label3, 0, 1)
layout.addWidget(button, 1, 1)
```

which will look like:

4.3 Nesting layouts

In practice, you may need to use more complex layouts - for this, you can start to nest layouts. You can do this by adding a layout to a widget which is itself in a layout. Let's take a look at the following example:

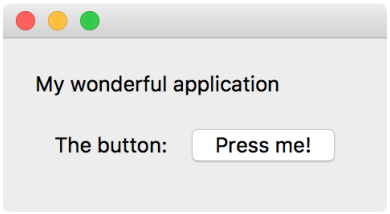
```
from qtpy.QtWidgets import (QApplication, QLabel, QWidget,
                            QHBoxLayout, QVBoxLayout, QPushButton)

# Initialize application
app = QApplication([])

# Set up individual widgets
label1 = QLabel('My wonderful application')
label2 = QLabel('The button:')
button = QPushButton('Press me!')

# Combine label2 and button into a single widget
bottom_widget = QWidget()
bottom_layout = QHBoxLayout()
bottom_layout.addWidget(label2)
bottom_layout.addWidget(button)
bottom_widget.setLayout(bottom_layout)

# Combine this widget with label1 to form the main window
widget = QWidget()
layout = QVBoxLayout()
layout.addWidget(label1)
layout.addWidget(bottom_widget)
widget.setLayout(layout)

# Show main widget
```

(continues on next page)

(continued from previous page)

```
widget.show()

# Start event loop
app.exec_()
```

This will result in the following window:



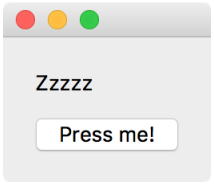
CHAPTER 5

Part 4 - Dynamically updating widgets

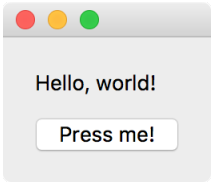
Now that we know how to show multiple widgets in a single window, we can make use of what we learned in 2.button.py to connect different widgets together. For example, we can make it so that pressing a button changes the text in one of the widgets:

```
from qtpy.QtWidgets import (QApplication, QLabel, QWidget, QVBoxLayout,
                            QPushButton)

# Initialize application
app = QApplication([])

# Create label
label = QLabel('Zzzzz')

def say_hello(event):
    label.setText('Hello, world!')

# Create button
button = QPushButton('Press me!')
button.clicked.connect(say_hello)

# Create layout and add widgets
layout = QVBoxLayout()
layout.addWidget(label)
layout.addWidget(button)

# Apply layout to widget
widget = QWidget()
widget.setLayout(layout)

widget.show()

app.exec_()
```

Try running this script, and you should see something like:



Now press the button, and the label text should change:



If you are interested in helping maintain these pages and/or add new pages, head over to the GitHub repository!