PIC 16, Winter 2018 – Preparation 5W

Assigned 2/5/2018. To be completed by class 2/7/2018.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- find widgets as needed and add them to a GUI;
- determine which signals can be sent by a given widget (using the documentation) and connect the desired signal to a slot (that is, a method that runs in response to the signal); and
- subclass widgets and override their default event methods in order to respond to events like mouse clicks and keyboard button presses.

Tasks

- ☐ Read "Events" and "Signals and Slots" from Events and Signals in PyQt5. You can ignore the "New API" section.
- ☐ Let's dig a little deeper into what's going on here to see if we can infer from this example how to respond to *any* signal from *any* widget. Download `slider.py`, which is the example code from the tutorial but with a few modifications:
  - I've eliminated the LCD so we can focus on the `QSlider` widget
  - I've eliminated the `QVBoxLayout` so we don't have to complicate things with a layout manager. In general, don't worry about layout yet; we're going to see how to lay things out with a graphical tool, QtDesigner, very soon.
  - I've made the slider an instance variable, i.e. `sld` is now `self.sld`. Before, `sld` (our *reference* to the slider object) was a local variable that would disappear when `initUI` was finished running. Now, we can manipulate the `QSlider` using `self.sld` even after `initUI` is finished running.
  - I've connected the `valueChanged` signal to a `printSlider` method, which I'll explain in detail below.

  Search for "pyqt5 QSlider" documentation on Google; in the future you may need to do this on your own for other classes. The first hit I get is "QSlider — PyQt 5.8.2 Reference Guide"; choose that and follow the link to the C++ documentation. This, *along with equivalent documentation for* `QSlider`'s *superclasses*, contains everything you might need to know about the `QSlider` class. (I repeat: some of the methods/signals you need might be in superclass documentation. For instance, `QPushButton`'s `pressed` signal is really only documented in its superclass, `QAbstractButton`. When you can't find the info you're looking for, check superclasses!)

  Search the page (`ctrl+f` in most browsers) for "signal". About a third of the way through the page is a very readable "Detailed Description" and list of the signals emitted by a `QSlider`, including the `valueChanged` signal we used in the code. See how we connected the `valueChanged` signal to our `printSlider` method in the code? You can connect any of these signals emitted by `QSlider` to any function or method using the same syntax, `<widgetReference>.<signalName>.connect(self.<methodName>)`.
  That makes the function/method run whenever the signal is emitted.

  Click the link to the `valueChanged` signal. This brings us to the documentation of the superclass of `QSlider`, `QAbstractSlider` (which has a lot of methods that are useful to subclasses like `QSlider` and `QScrollBar`, but isn't typically used on its own). Notice the `valueChanged`

documentation is really the documentation for a method. This `valueChanged` method has an `int` parameter. The documentation says "This signal is emitted when the slider value has changed, with the new slider *value* as argument." For us, that means when the signal is emitted, our `printSlider` method is called with the new slider value as an argument. We take that value in the parameter `val` and print it.

That's one way we can get the value of the slider – grab it when it's passed with the `valueChanged` signal. But what if you want to access the value of the slider elsewhere in the code, like when the slider value isn't changing? Note that the `QAbstractSlider` has a `value` method (see list of methods at the top). We can use that to access the value of the slider at any time. Uncomment the line of code after `print val` (and run) to see that `self.sld.value()` contains the slider value just like the `val` parameter does, but `self.sld.value()` can be called anywhere within the class.

Look at the other methods available. There's `setValue`. You can use that to change the value of the slider programmatically, such as if you wanted to keep the slider in sync with a text box. There are `minimum` and `maximum` for getting the lowest and highest possible values, and `setMinimum` and `setMaximum` for changing those. Looking for something that could change how wide the slider appears? I was too, but didn't find anything in this class, but notice that this class inherits from `QWidget`, which has a `setMinimumWidth` method. I tried using that, and indeed it changed the width that the slider appeared.

The point of all this is that you can use the documentation to get what you want from any widget. I can't teach you everything you'll need to know about every widget you'll ever need, but with some practice, you can learn to learn what you need as needed!

- Try adding a single line text box (search Google for "`single line text pyqt5`") to your GUI. The first hit for me is the documentation for a particular PyQt4 widget; the name suggests that it will do what we want, and I can tell you that PyQt4 and PyQt5 are very similar. So you can either use this PyQt4 documentation for the widget, or you can search again for "<widget name> pyqt5" to get the latest version of the documentation.
  Use its `move` method, inherited from `QWidget` (so all `QWidgets` have it! How convenient!), to position it below the slider. Connect the appropriate signal from the slider to a method (which you will write) called `syncText` that will sync the text of the text box to the value of the slider. Keep in mind that the text box method that sets its text is expecting a `str`, not an `int`. Now connect a signal from the text box to a method `syncSlider` that will sync the value of the slider to the text of the text box. You might have to try a few different signals before you find the one that behaves the way you want. Keep in in mind that the `QSlider` method that sets the value is expecting an `int`, not a `str`. How can you prevent your program from producing an error message or crashing when the text box is changed to something other than an integer within the limits of the slider? Your kernel might die a few times in the debugging process, and you won't always get helpful error messages.

- Now read "Reimplementing Event Handler" from Events and Signals in PyQt5.

- This is the other way of responding to things in a GUI. Rather than connecting the signal of a widget to a slot (method), subclass the widget and override a method called when an "event" occurs. How do you know what method to override? Look in the documentation of the widget for methods that have "event" in the name. For another example, download `MousePress.py`. Search google for "QWidget pyqt" to find the `QWidget` class documentation. Find

`mousePressEvent` in the documentation: "This event handler… can be reimplemented in a subclass to receive mouse press events for the widget." I have overridden it in the `MyWidget` class, which subclasses `QWidget`, so now when the mouse is pressed, "`Mouse Press`" is printed to the console. Click the argument type `QMouseEvent` in the documentation. This is an object passed into the `mousePressEvent` method that includes useful information about the mouse press, such as the coordinates within the `QWidget` at which the mouse was pressed (see `x` and `y` in the documentation). `button` is the number of the mouse button that was pressed (1 is left, 2 is right).

- Try overriding methods to detect a mouse double click, mouse release, mouse leaving the `QWidget`, mouse entering the `QWidget`, keyboard key press, and keyboard key release. Explore the object passed into the key press event method to find out which key was pressed.
- Explore "Widgets" and "Widgets II".
- Try adding a checkbox, toggle switch, and calendar to a GUI and printing relevant information (checked/unchecked, on/off, or date, as appropriate) when the widget's state changes. Where possible, explore multiple options for getting the relevant information: 1) the event object passed into the event handler function and 2) a method of the widget that returns the information you need. PyQt documentation for the latter option is not perfect; sometimes all that is listed is the (self-explanatory) method name. Note for example the `QCalendarWidget` method `selectedDate`: there is no information about what this method does. Calling it returns information about the date that is currently selected, as the name would suggest, but there is no detailed method description in the documentation. When in doubt, try it out!
- Finally, if you ever need a widget that we haven't seen, try searching Google for "pyqt widget <description>" where <description> is to be replaced by what the widget is supposed to do. Chances are good that if you've seen a widget on a web form or some other program, PyQt will have it.
- If you're interested, here is a video on learning to use a new widget. I had technical difficulties during recording, so the video is finished here. This video was recorded for PyQt4, but the only difference in PyQt5 is the submodule from which the classes are `import`ed.
  Given how many mistakes I make in the video, I think it's proof that it doesn't take a lot of memorization to create GUIs. Things can be figured out on the fly pretty quickly if you have a basic understanding of how Qt tends to work, know how to apply knowledge of components you've already used, skim through documentation for methods with names that sound right, and test things out.