PIC 16, Winter 2018 – Preparation 5M

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

Intended Learning Outcomes

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

- create a simple GUI window and modify its size, location on the screen, and title;
- specify absolute positions and sizes for "widgets", including buttons, in a GUI;
- make a program respond to a button click;
- outline and fill shapes on a widget, prescribing the color, style, and/or pattern, and
- use a QTimer to run a function at a prescribed frequency.

Tasks

- ☐ Now let's start with Qt! When you follow the tutorials, I highly recommend re-typing the tutorial code yourself rather than copying and pasting. This encourages you to slow down and try to think through what might be going on before it is explained to you in the text. Comment lines in and out and play with parameters. Sometimes it's hard to see why GUIs work the way they do or are organized the way they are, and to be honest there's not much need to. However, with practice (like toying around with these example), you'll begin to see patterns in how things work, and that's what you need to become effective at programming GUIs.
- ☐ First, read the introduction. While these tutorials are not super well written, I think that the progression – the order in and pace at which elements are introduced - is very helpful. Note that the tutorial is written for Python 3; if there's anything that's difficult to convert to Python 2 I'll point it out.
- ☐ Next, visit the First Programs section and read "Simple Example". Note the use of the `if __name__ == '__main__':` idiom.
- ☐ Retype the code and run it once. Note two things, the second of which may be particularly important:
    - When you click the button to close the window, `app.exec_()` finishes and `sys.exit` causes an exception to be raised just before the code terminates. This is intentional, and I know that this can cause some cleanup code to run, but I'm not convinced of its necessity, and I think the exception is annoying. So I will always omit the `sys.exit` and just call `app.exec_()` by itself. You're welcome to do the same for all your PyQt applications in this class.
    - When I try to run the code a second time, no window appears, and my Python interpreter (kernel) crashes. You can read a bit more about this problem here. The solution is to put all the code in a `main` function, and `if __name__ == '__main__'`, call `main`. (Yup, just like in the last preparation. Here's my code, for example.) I suggest you do this, too, for all your PyQt programs.
- ☐ Spend some time playing with the "Simple Example" code. Please answer (for yourself) the following questions:
    - How big is the window (approximately) if we don't specify a size?
    - Where does the window show up if we don't specify a location?
    - What is the default window title?
    - Where is the origin (0,0) for coordinates, which is x and which is y, and which direction is positive for each?
    - What commands can we leave out and still have some sort of window show up (i.e. what is the minimal PyQt GUI program)? (Be prepared for your IPython to crash while you

investigate this. If it happens, just start a new kernel in Spyder. Then again, maybe it's more convenient to use the command line for this part.)

- ☐ Read "An application icon". Note that this example has been converted to an object-oriented program, and `super` is used to explicitly call the superclass `__init__` method within the subclass `__init__` method to ensure that the superclass part of the object is properly initialized.
- ☐ Retype the code. Before you run it, note that `super` requires arguments in Python 2, whereas `super` is shown without inputs, as the tutorial is written for Python 3. Based on the previous preparation, you should be able to determine what the inputs should be. If not, take a look at the PyQt4 tutorial. Once this is fixed, you can run the code. Note that following:
  - Spyder might complain that `ex` from the line `ex = Example()` is not used, but in more complex programs we will want to call the methods on our widget (an instance of `Example`, a subclass of `QWidget`) using its reference. So leave it alone.
  - There is no need to memorize the `setWindowIcon(QtGui.QIcon('web.png'))` command; you'll always be able to look it up if you need it. BTW, note that since you didn't download or create a file `web.png`, the icon file is not available to your program, so won't see `web.png` as the icon for the window when you run the code. Of course, if you'd like to create an image `web.png` and place it in the same directory as the program, you should see it when the program runs.
- ☐ Get comfortable with the code in "An application icon". This (minus the `setWindowIcon` command) will form the basis of all of our future GUIs.
- ☐ "Showing a tooltip" is good for seeing how to create a button and position it in the window. We'll be making a lot of these! This example doesn't really explain the call: `btn = QPushButton('Button', self)`. What do you think the arguments are for? The first is pretty obvious, but why is the second needed? The next example will explain.
- ☐ The "Closing a Window" tutorial is useful for two things: Explaining the call `btn = QPushButton('Button', self)` from the previous example, and seeing how to make a button *do* something. Read the description of signals and slots carefully. Along with events (next item), the signals and slots mechanism is one of the two ways we will make GUIs respond to user actions. We'll see much more of this next time.
- ☐ See if you can modify this example to make the button print something to the console. Just write a method `my_method` of your example class that prints something, and provide the reference to your method `self.my_method` in place of `QCoreApplication.instance().quit`.
- ☐ "Message Box" introduces the idea of overriding methods of GUI components. Our `Example` subclasses `QWidget`, which already has a `closeEvent` method defined. We're overriding it - replacing it - to change its behavior. This example also introduces the concept of "events". Events, like the signals and slots mechanism, is the second of the two ways Qt allows our GUIs to respond to user input. The `event` parameter of the `closeEvent` method contains information about what happened to trigger the method being called (the user tried to close the window) and has some useful methods, including an `accept()` method to actually close the window and an `ignore()` method to prevent the window from closing. You don't need to memorize these methods in particular, but we'll be overriding and using events quite a bit in GUIs.
- ☐ "Centering window on the screen" introduces a concept that comes up frequently in GUIs: often, in order to place or size things, we query the size/location of the container to generate the size/coordinates we need. The tutorial doesn't explain very well how this is working; perhaps you'll find my code more intuitive.
- ☐ Read Painting. Note the essentials of painting: you must override the widget's `paintEvent` method, you must create a `QPainter`, call its `begin` and `end` methods, and call other `QPainter` methods to draw in between.

    - There is nothing special about the `drawText` method; this is just where the `QPainter`'s color and font are set and the text is drawn. Those commands could have been put directly between `qp.begin(self)` and `qp.end()`; but putting them in a separate method will keep the code cleaner, especially when there are many things to draw.
    - The `paintEvent` method *is* special; it's inherited from `QWidget` and we're overriding it. Just as `closeEvent` in a previous example was called in response to the user closing the window, the `paintEvent` method is called automatically whenever Qt decides that the widget needs to be repainted. *You do not call* `paintEvent` *yourself, Qt calls it for you whenever it chooses.*
- ☐ In the assignment you'll get some practice with this material. But before you're done, try referring to the `QPainter` documentation and figuring out how to draw/fill a line or ellipse. If you ever need to look up details of a Qt class or method, the PyQt documentation can be a good starting point, or you can usually find specific class documentation by searching online for "PyQt5 <classname>" or similar. Unfortunately, most of the time you will just get redirected to the appropriate C++ documentation. It would be nice if there were solid Python-specific documentation, but most of the time almost everything is equivalent. I suggest looking at the documentation for every class we use and figuring out how the documentation presents the information you already know. This will help you understand how the documentation will present information that is new to you.
- ☐ Watch Creating a GUI Programmatically. I made the video for PyQt4, but everything is the same except the name of the module – everything gets imported from `PyQt5.QtWidgets` instead of `PyQt4.QtGui`. Also, please disregard the bit about the reason for having a `main` function; it's it's erroneous. You can invoke the script from the command line without a `main` function. The real reason for using that Python idiom here is explained at the beginning of this preparation.
- ☐ Almost done: read about using a QTimer to run a function at a particular frequency.
- ☐ Use the second example to create a `QTimer` that prints "`Hello` *i*`!`" to the console once per second, where *i* is the number of times the function has been executed already.
- ☐ Modify your program to accomplish the same task using the `try-finally` technique. This might help (although you are not typically encouraged to use global variables).
- ☐ If you're having trouble, watch this. The point is that the `QTimer` needs to be `start`ed after a `QApplication` has been created but before it has been `exec_`uted. Since I use `QTimers` mostly for GUI animations, I usually `start` mine when I'm initializing my subclass of `QWidget`, either within the `__init__` method itself or a method that is called by `__init__`. If you do this, keep two things in mind:
    - If you assign the `QTimer` to a *local* variable, that variable will disappear as soon as `__init__` is done; the `QTimer` will be eligible for garbage collection and will not run.
    - If you `def f(self):` in a class, you don't refer to it as `f` alone. It's part of the object…