Graphics with PyQt5!!

Preparing for graphics

- Install PyQt

Go into the Terminal application on your personal machine and installing the graphics framework by typing

conda install pyqt

This will install or update the graphics libraries for Python as well as the necessary system libraries specific to your platform. After this command, you should have PyQt5 installed.

Note that the documentation links provided here link to PyQt4. You can access the version docs here: PyQt5, but they are slightly less complete than the older documentation. That's okay; the items used in this lab are pretty much the same as they were in PyQt4.

- Download the graphics skeleton and example files

These were included in the zip archive with this notebook. The skeleton is called **click.py** and the example is **click_example.py**.

A drawing example

Now that we know what a class is, how to create new objects of a class, how to create child classes, and how to invoke methods, we can make use of Python modules to draw graphics!

Download click.py and navigate with cd in Terminal to the folder containing the file. From that folder, type:

python click.py

A window should appear. When you click inside the window, the terminal will print the coordinates of the click.

The PyQt5 module provides classes to create graphical user interfaces (GUIs). Let's take a closer look at the contents of the click.py file you just executed:

NOTE: If you run this in the notebook it will likely crash your kernel, run the .py files from the command line

```
import sys
from PyQt5.QtGui import QPainter, QPen, QBrush
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtCore import Qt

class Click(QWidget):

  def __init__(self):
    super().__init__()
    self.setGeometry(300, 300, 550, 600)
    self.setWindowTitle('Click')
    self.show()

  def mousePressEvent(self, event):
    print('You clicked at ' + str(event.x()) + ',' + str(event.y()) + '.')

  def paintEvent(self, event):
    qp = QPainter()
    qp.begin(self)
    #draw stuff
    qp.end()

if __name__ == '__main__':
  app = QApplication(sys.argv)
  ex = Click()
  sys.exit(app.exec_())
```

We're creating a new class called `Click` which inherits from (is a child of) `QWidget` which is a class defined in PyQt5. Widgets are user interface (UI) items (e.g. scroll bars, buttons, windows, etc.) - in this case our widget is a window. We use inheritance here so that we can take advantage of code that is already written. It is complicated to create graphical objects - by creating a child class of an existing object, we can just add any specific attributes we need without having to start from scratch.

Let's look at the constructor:

```
class Click(QWidget):

  def __init__(self):
    super().__init__()
    self.setGeometry(300, 300, 550, 600)
    self.setWindowTitle('Click')
        self.show()
```

It calls the constructor for the parent class QWidget which creates a new widget we can use. This default constructor creates a widget that is a window. The call to setGeometry establishes the window size in terms of a rectangle, and setWindowTitle creates the title for the window. Even though we've created our widget, it doesn't show up on the screen until we explicitly tell it to - this is what self.show() does.

To make our widget behave the way we want, we overwrite some of the methods it inherits from QWidget (http://pyqt.sourceforge.net/Docs/PyQt4/qwidget.html). Widgets are set up to capture mouse clicks, releases, and movements. The method mousePressEvent is called when the user presses the mouse button anywhere inside the widget. We can overwrite it to do whatever we want (within reason!) when the user presses the mouse button. Here, we want to know the coordinates for where the user clicked - these are captured in a QMouseEvent object that is generated by the mouse click and passed to the method as the event parameter:

```
  def mousePressEvent(self, event):
        print('You clicked at ' + str(event.x()) + ',' + str(event.y()) + '.')
```

We have the method print the coordinates out so that we can see them in standard output.

Remember that it is very important to return from these *callback* methods as quickly as possible; Python cannot process any other UI (user interface) events until mousePressEvent returns.

Widgets render themselves in a processing known as **painting**. This method is called automatically by Python whenever the window needs to be redrawn, and contains **all** of the code necessary to render the window's contents - you should put all of your drawing code in this method. The default implementation of paintEvent in QWidget does

nothing (hence the blank window). We can override the default paintEvent method to have the widget render a drawing of our choosing. Let's start with the implementation above:

```
def paintEvent(self, event):
    qp = QPainter()
    qp.begin(self)
    #draw stuff
        qp.end()
```

First, we create a new QPainter() object. This object actually does the drawing. When we call the method begin(self) we pass our widget to the QPainter so it can draw on it. After this, notice the comment *#draw stuff*. We would put our drawing code here.

The file click_example.py has an example which draws a picture and some text. Run this file from the command line to see what it does:

python click_example.py

Let's take a closer look at the paintEvent method for this example:

```
def paintEvent(self, event):
    qp = QPainter()
    qp.begin(self)
    greenPen = QPen(QBrush(Qt.green),10)
    qp.setPen(greenPen)
    qp.drawChord(100,100,200,200,(30*16),(120*16))
    smallGreen = QPen(QBrush(Qt.green),4)
    redPen = QPen(QBrush(Qt.red),4)
    qp.setPen(smallGreen)
    qp.drawRect(145,149,20,20)
    qp.drawRect(251,149,20,20)
    qp.drawChord(288,119,50,50,(60*16),(120*16))
    qp.setPen(redPen)
    qp.drawEllipse(302,122,5,5)
    textPen = QPen(3)
    qp.setPen(textPen)
    qp.drawText(150,200,"TURTLE POWER!")
    qp.end()
```

Whoa! That's a lot of stuff. Let's look at it piece by piece.

In order to draw, we need a pen, more specifically a QPen. The default constructor for QPen creates a pen that draws with a black solid line with a width of 0 - not much good! Here, we pass two arguments, a QBrush object that specifies a color for our pen, and also a type of brush stroke (we're using the default, so we don't pass that as a parameter). We also specify a width for our brush, 4. We create 3 pens in total in this code - how does each one draw? How do we create a pen that is blue with a width of 16?

When we want to use one of our pens, we need to tell the QPainter to use it:

```
qp.setPen(redPen)
```

Any drawing commands that follow this will be done with the specified pen. We can change pens by calling this method again.

There are many different things we can draw and ways to draw them. To draw a rectangle, we use the drawRect method. There are a few varities, but the way we will use it for this class is by passing in four arguments, an x-coordinate, a y-coordinate, a width, and a height. So:

```
qp.drawRect(145,149,20,20)
```

will draw a rectangle with the current pen starting at the position (145,149) in the widget. The rectangle will have a width of 20 and a height of 20 (it is a square).

We can draw ellipses and circles using drawEllipse:

```
qp.drawEllipse(302,122,5,5)
```

and text using drawText:

```
qp.drawText(150,200,"TURTLE POWER!")
```

Who's calling our methods?!

We have derived Click from the PyQt5 class QWidget. Widgets have quite a few methods. The one we use is mousePressEvent(). This method is called when a mouse button is pushed. There is another one called mouseReleaseEvent(), which is called when the mouse is released.

But who calls mousePressEvent()? In PyQt5 there is another, unseen part of the program that is monitoring mouse events. It is running independently of our code in a separate **thread**. You can think of a **thread** as part of our program that is running concurrently and independently of other parts of the program, but with the ability to communicate with other parts of the program.

When the mouse event monitor detects a mouse button being pressed, it calls Click.mousePressEvent(). It passes some information about the mouse event as a QMouseEvent which is the argument event that is passed to the method. This is itself an object, and we retrieve the x and y coordinates of the click by calling accessor methods. This allows us to determine the position of the cursor when the mouse event occurred.

Exercises

1) Using the click.py template file, insert code to draw a yellow circle and the text "One ring to bind them" written in red below it. You can position the circle and text anywhere you like in the window, and use your discretion as to the brush size. Save this as ring.py and run it from the command line.

2) Using the click.py template file, modify the paintEvent method so that it draws two hollow rectangles somewhere in the window, one green and one gold. Once you have modified the code, launching click.py (or you can rename it if you choose) should display your spirited rectangles and the printing of click coordinates anywhere (inside or outside of the rectangles) should still work.

3) Explore the code in the file double_click.py. Notice that use of a Boolean to aid in drawing. The logic to react to a mouse click is written in mousePressEvent but ALL of the drawing occurs in paintEvent. Modify the double_click.py file so that the text only displays when the user clicks OUTSIDE the rectangle.