Cómo usar Multithreading en QT (Python)

Autor: Salvador Jesús Megías Andreu

Este Notebook puede serte útil en caso de que necesites que tu GUI en QT mantenga Widgets que usen funciones con bucles infinitos o bucles condicionales, de los cuales no sabes cuándo van a acabar. En esos casos se necesitaría implementar **Multithreading** para evitar así que la GUI se quede congelada e inoperativa.

PYQT (Python QT) contiene su propio módulo (librería) para el manejo del concepto de MultiThreading en las GUI's QT, dicho módulo es **QThread**. Con QThread podremos asignar funciones y eventos en Python que hagan uso de bucles indefinidos sin llegar a congelar o bloquear nunca la GUI, lo cual es nuestro objetivo.

Para mostrar esto, voy a explicar un ejemplo sencillo que encontré en internet y modifiqué ligeramente para que sea más robusto (deshabilitando y habilitando botones correctamente) a la hora de fallas presionando los botones:

In [ ]:

```
from PyQt5 import QtCore, QtWidgets,QtGui
from PyQt5 import uic
import sys, time

# Clase de la GUI

class PyShine_THREADS_APP(QtWidgets.QMainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        self.ui = uic.loadUi('threads.ui',self) # Cargamos la GUI directamente del archivo .ui
        self.resize(888, 200)
        # Esto de abajo no hará nada puesto que no tenemos la imagen del logo dle ejemplo
        icon = QtGui.QIcon()
        icon.addPixmap(QtGui.QPixmap("PyShine.png"), QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.setWindowIcon(icon)

        # Deshabilito los botones de Stop inicialmente para evitar fallas en el programa
        self.pushButton_4.setEnabled(False)
        self.pushButton_5.setEnabled(False)
        self.pushButton_6.setEnabled(False)

        # Se define un conjunto vacío donde guardaremos los threads que se vayan a crear a lo largo del programa para manejarlos facilm
        self.thread={}
```

```
    # Se conectan todos los botones a las funciones correspondientes para que estas se ejecuten al pulsar el botón en la GUI (progr

    self.pushButton.clicked.connect(self.start_worker_1)
    self.pushButton_2.clicked.connect(self.start_worker_2)
    self.pushButton_3.clicked.connect(self.start_worker_3)
    self.pushButton_4.clicked.connect(self.stop_worker_1)
    self.pushButton_5.clicked.connect(self.stop_worker_2)
    self.pushButton_6.clicked.connect(self.stop_worker_3)


# Funciones encargadas de iniciar los threads

def start_worker_1(self):
    self.thread[1] = ThreadClass(parent=None,index=1) # Creamos en la posición 1 del conjunto thread un objeto de la clase ThreadCl
    self.thread[1].start() # Llamamos a la función run() de la clase ThreadClass para iniciar el Thread 1 (por defecto start() llam
    self.thread[1].any_signal.connect(self.my_function) # conforme van llegando las señales (datos del 0 al 99) se conecta con la f
    self.pushButton.setEnabled(False) # Cuando le doy al botón de start del Thread 1, deshabilito el botón de start del Thread 1
    self.pushButton_4.setEnabled(True) # Cuando le doy al botón de start del Thread 1, habilito el botón de stop del Thread 1

def start_worker_2(self):
    self.thread[2] = ThreadClass(parent=None,index=2) # Creamos en la posición 2 del conjunto thread un objeto de la clase ThreadCl
    self.thread[2].start() # Llamamos a la función run() de la clase ThreadClass para iniciar el Thread 2 (por defecto start() llam
    self.thread[2].any_signal.connect(self.my_function) # conforme van llegando las señales (datos del 0 al 99) se conecta con la f
    self.pushButton_2.setEnabled(False) # Cuando le doy al botón de start del Thread 2, deshabilito el botón de start del Thread 2
    self.pushButton_5.setEnabled(True) # Cuando le doy al botón de start del Thread 2, habilito el botón de stop del Thread 2

def start_worker_3(self):
    self.thread[3] = ThreadClass(parent=None,index=3) # Creamos en la posición 3 del conjunto thread un objeto de la clase ThreadCl
    self.thread[3].start() # Llamamos a la función run() de la clase ThreadClass para iniciar el Thread 3 (por defecto start() llam
    self.thread[3].any_signal.connect(self.my_function) # conforme van llegando las señales (datos del 0 al 99) se conecta con la f
    self.pushButton_3.setEnabled(False) # Cuando le doy al botón de start del Thread 3, deshabilito el botón de start del Thread 3
    self.pushButton_6.setEnabled(True) # Cuando le doy al botón de start del Thread 3, habilito el botón de stop del Thread 3

# Funciones encargadas de para los threads

def stop_worker_1(self):
    self.thread[1].stop() # Llamamos a la función stop() de la clase ThreadClass para parar el Thread 1
    self.pushButton.setEnabled(True) # Cuando le doy al botón de stop del Thread 1, habilito el botón de start del Thread 1
    self.pushButton_4.setEnabled(False) # Cuando le doy al botón de stop del Thread 1, deshabilito el botón de stop del Thread 1

def stop_worker_2(self):
    self.thread[2].stop() # Llamamos a la función stop() de la clase ThreadClass para parar el Thread 2
    self.pushButton_2.setEnabled(True) # Cuando le doy al botón de stop del Thread 2, habilito el botón de start del Thread 2
    self.pushButton_5.setEnabled(False) # Cuando le doy al botón de stop del Thread 2, deshabilito el botón de stop del Thread 2
```

```
    def stop_worker_3(self):
        self.thread[3].stop() # Llamamos a la función stop() de la clase ThreadClass para parar el Thread 3
        self.pushButton_3.setEnabled(True) # Cuando le doy al botón de stop del Thread 3, habilito el botón de start del Thread 3
        self.pushButton_6.setEnabled(False) # Cuando le doy al botón de stop del Thread 3, deshabilito el botón de stop del Thread 3


    # Función que va modificando el valor del progressBar correspondiente al id del thread asociado a dicho progressBar

    def my_function(self,counter):

        #cnt=counter
        index = self.sender().index
        if index==1:
            self.progressBar.setValue(counter)
        if index==2:
            self.progressBar_2.setValue(counter)
        if index==3:
            self.progressBar_3.setValue(counter)



# Clase para el control del Multithreading en la GUI

class ThreadClass(QtCore.QThread):

    any_signal = QtCore.pyqtSignal(int) # Creamos un Objeto de pyqtSignal el cual va a manejar enteros

    # Constructor para definir la variable index e is_running
    def __init__(self,parent=None,index=0):
        #super(ThreadClass, self).__init__(parent) # Carga el constructor del padre que en nuestro caso es QThread
        super(QtCore.QThread,self).__init__(parent) # la declaración de arriba y esta hacen exactamente lo mismo
        self.index=index
        self.is_running = True

    # Función (a la que recurre start()) con bucle infinito que va emitiendo los datos (del 0 al 99) una y otra vez mediante el objeto

    def run(self):
        print('Starting thread...',self.index) # Se muestra el index del Thread que llama a esta función
        cnt=0
        while (True):
            cnt+=1
            if cnt==99: cnt=0 # Si el valor de cnt a llegado a 99, se inicializa a 0 de nuevo
            time.sleep(0.01)
            self.any_signal.emit(cnt) # emit() ==> función que emite los datos
```

```
    # Función que se encarga de parar o finalizar el proceso del thread que llama esta función

    def stop(self):
        self.is_running = False
        print('Stopping thread...',self.index) # Se muestra el index del Thread que llama a esta función
        self.terminate() # terminate() ==> función que finaliza el proceso



app = QtWidgets.QApplication(sys.argv)
mainWindow = PyShine_THREADS_APP()
mainWindow.show()
sys.exit(app.exec_())
```

```
Starting thread... 1
Starting thread... 2
Starting thread... 3
Stopping thread... 2
Stopping thread... 1
Stopping thread... 3
Starting thread... 2
Stopping thread... 2
Starting thread... 1
Stopping thread... 1
```

Defición de run()

void QThread::run()

The starting point for the thread. After calling start(), the newly created thread calls this function.

Definición de super()

The idea of `super()` is that you don't have to bother calling both superclasses' `__init__()` methods separately -- `super()` will take care of it, provided you use it correctly -- see Raymond Hettinger's "Python's super() considered super!" for an explanation.

That said, I often find the disadvantages of `super()` for constructor calls outweighing the advantages. For example, all your constructors need to provide an additional `**kwargs` argument, all classes must collaborate, non-collaborating external classes need a wrapper, you have to take care that each constructor parameter name is unique across **all** your classes, etc.

So more often than not, I think it is easier to explicitly name the base class methods you want to call for constructor calls:

```
class Child(Parent1, Parent2):
    def __init__(self):
        Parent1.__init__(self)
        Parent2.__init__(self)
```

I do use `super()` for functions that have a guaranteed prototype, like `__getattr__()`, though. There are not disadvantages in these cases.

Aquí os dejo el vídeo tutorial donde encontré el ejemplo (a mi me sirvió muchísimo):

In [1]:

```
from IPython.display import YouTubeVideo

url = "https://youtu.be/k5tIk7w50L4"
idVideo = url.split("/")[-1] # sólo necesitamos la id del vídeo que es en nuestro caso "k5tIk7w50L4"

YouTubeVideo(idVideo, width=1000, height=700)
```

Out[1]:

Se ha producido un error.

Prueba a ver el vídeo en www.youtube.com o habilita JavaScript si está desactivado en tu navegador.

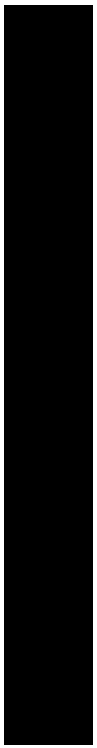
Demo del programa con los cambios:

In [1]:

```
from IPython.display import HTML

HTML("""
<div align="middle">
<video width="80%" controls>
      <source src="./multimedia/demo.mp4" type="video/mp4">
</video></div>""")
```

Out[1]:



In [ ]: