Jo Plaete

Visual Effects – 3D – Interactive Design

Threading with PyQt4

with 21 comments

Small post showing some simple examples on how to deal with threading in PyQt4 which would at least have saved me a bit of time when I was first looking into it.

As you start developing ui's within this cool framework you'll probably quickly notice that it is valuable to be able to run processes in separate threads and as such keep your ui unlocked while doing things in the background. Tasks like data retrieval and such, which may possibly take up some time, are better done in a sort of worker thread which on completion updates your ui. You can achieve this using the standard python threads – but if you happen to be working with PyQt4 I'd suggest you make use of their threading libraries as they are nicely integrated ensuring signal/slot communication to be thread safe. Both are cross-platform and I found them very useful so far.

So here's an example on how you can make that happen. To start we'll set up a very simpel ui containing a list widget which we will add some items to by clicking a button – fancy!

```
 1  import sys, time
 2  from PyQt4 import QtCore, QtGui
 3
 4  class MyApp(QtGui.QWidget):
 5   def __init__(self, parent=None):
 6    QtGui.QWidget.__init__(self, parent)
 7
 8    self.setGeometry(300, 300, 280, 600)
 9    self.setWindowTitle('threads')
10
11    self.layout = QtGui.QVBoxLayout(self)
12
13    self.testButton = QtGui.QPushButton("test")
14    self.connect(self.testButton, QtCore.SIGNAL("released()"), sel
15    self.listwidget = QtGui.QListWidget(self)
16
17    self.layout.addWidget(self.testButton)
18    self.layout.addWidget(self.listwidget)
19
20   def add(self, text):
21    """ Add item to list widget """
22    print "Add: " + text
23    self.listwidget.addItem(text)
```

```
    self.listwidget.sortItems()

  def addBatch(self,text="test",iters=6,delay=0.3):
   """ Add several items to list widget """
   for i in range(iters):
    time.sleep(delay) # artificial time delay
    self.add(text+" "+str(i))

  def test(self):
   self.listwidget.clear()
   # adding entries just from main application: locks ui
   self.addBatch("_non_thread",iters=6,delay=0.3)
```

If we were to run this code (which you'll need to add the following for)

```
# run
app = QtGui.QApplication(sys.argv)
test = MyApp()
test.show()
app.exec_()
```

we'll notice that after displaying our ui and clicking the test button – the ui will hang for a bit whilst our addBatch method is adding some items to the list widget. To make this apparent a slight artificial delay is introduced by time.sleep() before adding each element. Now this is exactly the problem we want to address here as if your ui's grows bigger and you have waiting times for looking up data you really don't want to hang your ui each time frustrating your user.

Let's imagine time.sleep() is the time it takes to retrieve a certain piece of data from a database which has to result in an item being added to our list. Here's how we could let this be dealt with in the background. We will make use of qt's singal/slot communication mechanism as that is a thread safe way to communicate from our work thread back to the main application. First we need to create another object which will represent our new thread.

```
class WorkThread(QtCore.QThread):
 def __init__(self):
  QtCore.QThread.__init__(self)

 def run(self):
  for i in range(6):
   time.sleep(0.3) # artificial time delay
   self.emit( QtCore.SIGNAL('update(QString)'), "from work thread
  return
```

This is pretty much the easiest it gets (beware you may run into some trouble with this bare version as discussed below). As you can see we are inheriting from QtCore.QThread, that's where all the Qt threading magic will come from but we don't have to worry to much about that as long as we call its __init__() method to set it up and implement the right methods. Further on you find the run method which is what will be called when we start the thread. Just remember the method to implement is run() but starting the thread itself is done using start() ! What we currently have in there is something similar to our addBatch method only instead of calling the add method we will emit a signal to the main application passing on some data as an argument.

Now the only thing we have to do in our main application is to make an instance of this and connect to the signal it emits, adding this to our test method

```
def test(self):
 self.listwidget.clear()
 # adding in main application: locks ui
 self.addBatch("_non_thread",iters=6,delay=0.3)

 # adding by emitting signal in different thread
 self.workThread = WorkThread()
 self.connect( self.workThread, QtCore.SIGNAL("update(QString)"),
 self.workThread.start()
```

If we run this we should find that after clicking our button our ui still freezes for about a second whilst running our original addBatch method but afterwards it unlocks and as the workThread gets started we can see item per item being added without the ui being stuck. This is thanks to the work thread signaling back to the main app which gets then updated accordingly – all the rest is done inside the thread away from the main app. Because we have matched the emit signal signature to our add method we can just connect to this method to the signal call.

An important thing to be aware that of is that if the object which holds the thread gets cleaned up, your thread will die with it and most likely give you some kind of segmentation fault. As we have stored it in an object variable this won't happen here although it is recommended to override the destructor as follows

```
class WorkThread(QtCore.QThread):
 def __init__(self):
  QtCore.QThread.__init__(self)

 def __del__(self):
  self.wait()

 def run(self):
  for i in range(6):
   time.sleep(0.3) # artificial time delay
   self.emit( QtCore.SIGNAL('update(QString)'), "from work threa
  return
```

This will (should) ensure that the thread stops processing before it gets destroyed. That will do the job in some cases but (at least for me) it may still go wrong. If you hammer the test button a few times (and take out the first addBatch call for that), you will notice you get: *The thread is waiting on itself* – after which it will get destroyed and the app gets reset or crashes. This is where it gets a bit tricky. As for me, and I am very open to suggestions/explanations on this one, the best cure for this is to terminate the (waiting) thread after your run code has been executed. This makes it (in this scenario at least) more stable.

```
class WorkThread(QtCore.QThread):
 def __init__(self):
  QtCore.QThread.__init__(self)

 def __del__(self):
  self.wait()

```

```
  def run(self):
   for i in range(6):
    time.sleep(0.3) # artificial time delay
    self.emit( QtCore.SIGNAL('update(QString)'), "from work threa

   self.terminate()
```

However, terminate() is not encouraged by the docs and overwriting this variable over and over again is not the best thing to do. It is better to design your code so it avoids this from happening altogether. If you happen to be spawning lots of threads, there is a more stable way to get around this problem by using for example a thread pool. This will just be a simple list to store all your threads

```
# add to __init__()
self.threadPool = []

# replace in test()
self.threadPool.append( WorkThread() )
self.connect( self.threadPool[len(self.threadPool)-1], QtCore.SIG
self.threadPool[len(self.threadPool)-1].start()
```

Which makes it behave stable without the need to call terminate().

Furthermore something I found convenient is to have a sort of generic thread which you can send a certain method to. That way you can keep your app specific code inside your main class and just dispatch a certain function to the thread. For that we can create a thread object as follows

```
class GenericThread(QtCore.QThread):
 def __init__(self, function, *args, **kwargs):
  QtCore.QThread.__init__(self)
  self.function = function
  self.args = args
  self.kwargs = kwargs

 def __del__(self):
  self.wait()

 def run(self):
  self.function(*self.args,**self.kwargs)
  return
```

As you can see this thread takes a function and its args and kwargs. In the run() method it will then just call this. In our test() method we can add

```
# generic thread
self.genericThread = GenericThread(self.addBatch,"from generic th
self.genericThread.start()
```

Tough it is better/safer to communicate through signals so we could change the addBatch method to emit a signal itself

```
def addBatch2(self,text="test",iters=6,delay=0.3):
 for i in range(iters):
  time.sleep(delay) # artificial time delay
```

```
4       self.emit( QtCore.SIGNAL('add(QString)'), text+" "+str(i) )
```

And then connect to it as follows

```
1  # generic thread using signal
2  self.genericThread2 = GenericThread(self.addBatch2,"from generic
3  self.disconnect( self, QtCore.SIGNAL("add(QString)"), self.add )
4  self.connect( self, QtCore.SIGNAL("add(QString)"), self.add )
5  self.genericThread2.start()
```

Disconnecting the signal first in this example to avoid registering multiple times to it.

Be careful when you start doing more complicated things with this involving access to data structures and such. Sometimes if you really need to lock an object while you're working on it is worth looking into the QMutex functionality to enforce access serialization between threads. Something else that ties very well into it is the QEventLoop but I'll leave those up to you to have a play with!

That's about it, please let me know if you have any remarks or issues. Here's the whole thing again in one piece.

```
1  import sys, time
2  from PyQt4 import QtCore, QtGui
3
4  class MyApp(QtGui.QWidget):
5   def __init__(self, parent=None):
6    QtGui.QWidget.__init__(self, parent)
7
8    self.setGeometry(300, 300, 280, 600)
9    self.setWindowTitle('threads')
10
11   self.layout = QtGui.QVBoxLayout(self)
12
13   self.testButton = QtGui.QPushButton("test")
14   self.connect(self.testButton, QtCore.SIGNAL("released()"), sel
15   self.listwidget = QtGui.QListWidget(self)
16
17   self.layout.addWidget(self.testButton)
18   self.layout.addWidget(self.listwidget)
19
20   self.threadPool = []
21
22  def add(self, text):
23   """ Add item to list widget """
24   print "Add: " + text
25   self.listwidget.addItem(text)
26   self.listwidget.sortItems()
27
28  def addBatch(self,text="test",iters=6,delay=0.3):
29   """ Add several items to list widget """
30   for i in range(iters):
31    time.sleep(delay) # artificial time delay
32    self.add(text+" "+str(i))
33
34  def addBatch2(self,text="test",iters=6,delay=0.3):
35   for i in range(iters):
```

```
36      time.sleep(delay) # artificial time delay
37      self.emit( QtCore.SIGNAL('add(QString)'), text+" "+str(i) )
38
39    def test(self):
40     self.listwidget.clear()
41     # adding in main application: locks ui
42     #self.addBatch("_non_thread",iters=6,delay=0.3)
43
44     # adding by emitting signal in different thread
45     self.threadPool.append( WorkThread() )
46     self.connect( self.threadPool[len(self.threadPool)-1], QtCore.
47     self.threadPool[len(self.threadPool)-1].start()
48
49     # generic thread using signal
50     self.threadPool.append( GenericThread(self.addBatch2,"from gen
51     self.disconnect( self, QtCore.SIGNAL("add(QString)"), self.add
52     self.connect( self, QtCore.SIGNAL("add(QString)"), self.add )
53     self.threadPool[len(self.threadPool)-1].start()
54
55  class WorkThread(QtCore.QThread):
56    def __init__(self):
57     QtCore.QThread.__init__(self)
58
59    def __del__(self):
60     self.wait()
61
62    def run(self):
63     for i in range(6):
64      time.sleep(0.3) # artificial time delay
65      self.emit( QtCore.SIGNAL('update(QString)'), "from work threa
66     return
67
68  class GenericThread(QtCore.QThread):
69    def __init__(self, function, *args, **kwargs):
70     QtCore.QThread.__init__(self)
71     self.function = function
72     self.args = args
73     self.kwargs = kwargs
74
75    def __del__(self):
76     self.wait()
77
78    def run(self):
79     self.function(*self.args,**self.kwargs)
80     return
81
82  # run
83  app = QtGui.QApplication(sys.argv)
84  test = MyApp()
85  test.show()
86  app.exec_()
```

And some more docs and links on the topic:
http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/qthread.html
http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/qeventloop.html
http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/qmutex.html

http://diotavelli.net/PyQtWiki/Threading,_Signals_and_Slots
http://stackoverflow.com/questions/1595649/threading-in-a-pyqt-application-use-qt-threads-or-python-threads

Jo

Written by Jo Plaete

July 21, 2010 at 9:30 am

Posted in interactivity, interfaces, pyqt, qt, scripting, TD, threading, ui

21 Responses

Subscribe to comments with RSS.

Thanks Jo, I used this straight away.

Gerard

August 5, 2010 at 6:10 pm

Reply

cool

Jo Plaete

August 6, 2010 at 2:14 pm

Reply

>Tough it is better/safer to communicate through signals so we could change the addBatch method to emit a signal itself

Could you explain why it is better to communicate through signals?

jack minardi

August 13, 2011 at 7:45 am

Reply

Sweet site , I hadn't noticed joplaete.wordpress.com before till my friend told me about it.
Keep up the great work! I will be posting more at joplaete.wordpress.com

Looking forward to some good experiences here at joplaete.wordpress.com

dtngwyfr

September 20, 2010 at 5:18 pm

Reply

Hi,

Thank you very much for this post. I was tearing my hair off trying to keep my gui using responsive and got nowhere until I came here. With your generic thread class, I used a queue to keep the data from the run method and used it in my gui thread for displaying.

Qt's async stuff is a real PITA, atleast for newbie's like me.

jack daniels

December 19, 2010 at 8:58 am

Reply

you're welcome Jack, glad it helped.

Jo Plaete

December 20, 2010 at 8:25 am

Reply

Thanks! This helped a lot !

Saurabh Joshi

May 3, 2012 at 1:41 pm

Reply

self.terminate() didn't work for me

Another solution found @ http://diotavelli.net/PyQtWiki/Threading,_Signals_and_Slots

```
class Worker(QThread):
def __init__(self, parent = None):
QThread.__init__(self, parent)
self.exiting = False
…….
```

def __del__(self):
self.exiting = True
self.wait()

lflejo

May 29, 2012 at 8:41 pm

Reply
[…] Reference: https://joplaete.wordpress.com/2010/07/21/threading-with-pyqt4/ […]

Threading in PyQt4 « Perspective

January 24, 2013 at 1:06 pm

Reply
Reblogged this on My Blabbery.

admin

February 18, 2013 at 1:38 am

Reply
[…] Used this as a reference https://joplaete.wordpress.com/2010/07/21/threading-with-pyqt4/ […]

using threading for a qt application | BlogoSfera

April 8, 2013 at 6:02 am

Reply
Nice post, thanks Jo. I usually grey out buttons after the user presses it whenever I need only one process/thread to run in the background. That avoids the user to be able to accidentally run more unwanted threads and me to connect/disconnect events.

ivanoras

May 16, 2013 at 11:37 am

Reply
Hi,Jo. i'm new to pyqt, do you know why i can't use genericThread a second time ?
http://pastebin.com/LS1yEPjf

basically i justed added another thread, and i wanted to run it after first thread finishes, but as soon as i add the second thread, i haven't even connected and emit signal yet.

the first thread.run() will call the function in the second thread ??? why is this happening ?
thanks !

oglops

July 12, 2013 at 11:01 am

Reply

i just found it’s a bug of the ide i’m using with QThread, wingide 4.1 , in the console it runs fine.

oglops

July 15, 2013 at 2:18 am

Reply

hey, thanks for this really informational post. I just tried it using PySide instead of PyQt – the final version using the generic thread and signal-approach crashes. (btw, also when using the signal thing without threads). Just researching what this could be – maybe some difference in Signal/Slot-mechanism. Or a bug in PySide (it just doesn’t seem to be that mature as PyQt yet)?

felix

October 24, 2013 at 10:50 am

Reply

Hi Felix,
I just tried running this changing the import statement to PySide and it seems to work fine for me..
My test was using python 2.6.4 and PySide version 1.1.2.. maybe you got different versions?
Cheers
Jo

Jo Plaete

October 24, 2013 at 3:28 pm

Reply

thanks for this post, it helped me a lot !

wud

February 25, 2014 at 10:53 pm

Reply

Excellent!
thank you for the helpful blog,well presented usable info.
Thank you for the time you took to present this so clearly.

mina mina

July 25, 2015 at 1:14 pm

Reply

[…] My advice is to try and even further simplify your example. Try to accomplish what you are doing w/ a single button / list / thread. Then once you got that working then build on it. Kind of like this article joplaete.wordpress.com/2010/07/21/threading-with-pyqt4 […]

Python:multiprocessing GUI schemas to combat the “Not Responding” blocking – IT Sprite

October 15, 2015 at 8:55 am

Reply
[…] 备忘录 下面的例子是Jo Plaete的 […]

伪”多”线程QThread的使用方法 | 一半君的备忘录

December 19, 2015 at 10:39 am

Reply
[…] 备忘录 下面的例子是Jo Plaete的 […]

伪”多”线程QThread的中古(subclass QThread)和近代(moveToThread)使用方法 | 一半君的备忘录

December 19, 2015 at 10:41 am

Reply