PyQt_Primitive_3_Interacting_Threads_V03

January 9, 2024

0.1 Purpose: Testing the use of background threads by a PyQt application.

Basic ideas: A PyQt App that starts a worker and a receiver thread in the background. The worker thread creates plot and text which are supplemented in the receiver thread. The signal/event mechanism of PyQt is used to create respective events which are handled by slot-callbacks in the main thread. The PyQt app then updates a Figure widget and a QTextEdit widget.

Documents

Basics : https://stackoverflow.com/questions/21071448/redirecting-stdout-and-stderr-to-a-pyqt4-qtextedit-from-a-secondary-thread

Redirect simple: https://codereview.stackexchange.com/questions/208766/capturing-stdout-in-a-qthread-and-update-gui

Stopping Qthreads : https://realpython.com/python-pyqt-qthread/

0.2 Imports

[1]:

```python
import time
#import gc  # need some garbage collection
import sys # for PyQt5
import math
import numpy as np
import queue
# a useful module to redirect print-output
from contextlib import redirect_stdout

# For plotting
import matplotlib
import matplotlib.backends
import matplotlib.pyplot as plt
from matplotlib.figure import Figure
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as NavigationToolbar

# PyQt
from PyQt5 import QtWidgets, QtCore
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import *
```

0.3 Activate QtAgg-backend !!! Do NOT forget !!!

[2]:

```
matplotlib.use('QtAgg')
```

0.4 Some helper classes

0.4.1 A stream object to redirect stdout

[3]:

```
# The Stream Object which replaces the default stream
# associated with sys.stdout
# This object just puts data into a Python queue!
# Any object with write method for a text str is working
class WriteStream(object):
    def __init__(self, queue_msgs):
        self.queue_msgs = queue_msgs

    # When texts come from print statements
    # two (!) objects are added to the queue "text" + "\n"
    def write(self, text):
        self.queue_msgs.put(text)
```

0.4.2 Object to send sinus data together with signal

This is dome this way just for demonstration purposes

[4]:

```
class SinObj():
    def __init__(self, pi_fact=1, col='red'):
        self.pi_fact = pi_fact
        # col (= color) will later be overwritten
        self.col = col
        self.pi = np.pi
        self.make_sins()
    def make_sins(self):
        self.sinx = np.arange(0, self.pi_fact*self.pi, 0.1)
        self.siny = np.sin(self.sinx)

```

0.4.3 An object to define private signal types

Used to show that we can combine signals /slots between any threads

[5]:

```
class Communicate(QObject):
    sig_sinus1 = pyqtSignal(object) # you can use Python types here!
    sig_sinus2 = pyqtSignal(object) # you can use Python types here!
```

0.5 Objects for background jobs

0.5.1 Main Object for Worker Thread

This class is for a worker object in the "worker thread".

It peridically creates an object with sinus-data and puts it into a queue for a receiver object. It also sends a msg in form of a signal to the main window.

[6]:

```
# Worker Object [derived from QObject]
# (It will later be run in a QThread)
class MyWorker(QObject):

    # Static variables
    # ~~~~~~~~~~~~~~~~
    # Signals MUST be defined as static variables
    # Note: Signals could also be defined in the app's MainWindow
    #       We would then emit them by using a reference to the window

    # Signal at start and regular end of the object's action
    # (= while loop) => will be send to qMainWin
    signal_start_end = pyqtSignal(str)
    # Intermdiate msg-signals -  will be sent directly to MainWindow
    signal_msg = pyqtSignal(str)

    # Constructor
    # ~~~~~~~~~~~
    def __init__(self, qMainWin, thrd
                 , num_iterations=20, time_sleep=1.0):

```

```
        # Parameters:
        # ~~~~~~~~~~~
        # qMainWin: A reference to the App's MainWindow
        #           derived from QMainWindow
        # thrd: A reference to the thread which the object gets affine to
        # num_iterations: max num of iterations of the while loop
        # time_sleep: sleep time between iterations
        #             required here for demonstration purposes
        #             Normally extensive operations consume the time

        # Constructor of parent class
        QObject.__init__(self)

        # Main App window and threadd
        self.qMainWin = qMainWin
        self.thrd = thrd

        # Maximum number of iterations / sleep time
        self.num = num_iterations
        self.time_sleep = time_sleep

        # Number of elements in a "batch"
        # Here: Just used to send intermediate msg
        #       Normally we would operate with real batches
        #       of data, e.g. in ML scenarios
        self.batch_size = qMainWin.batch_size_worker
        # print("Worker: Batch size = ", self.batch_size)

        # factor for sine period - will be raised
        self.pi_fact = 0
        self.pi = np.pi

        # Queues - will be read by Receiver object
        # ~~~~~~~~
        # Queue for messages to Receiver - stdout-redirect
        self.queue_msgs = qMainWin.queue_worker_msgs
        # Queue for sine data
        self.queue_sins = qMainWin.queue_sins

        # Stream object to capture stdout
        self.streamObj = WriteStream(self.queue_msgs)

        # Connect signals to callbacks
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # connect sart/end signals to callback in main window
        self.signal_start_end.connect(qMainWin.callback_worker_start_end)
        # connect intermediate msg signal to callback in main window
```

```
        self.signal_msg.connect(qMainWin.callback_for_worker_msgs)

        # get a time-reference + send start time
        # ~~~~~~~~~~~~~~~~~~~~
        self.time_ref = qMainWin.time_thrds_start
        self.start_time = time.perf_counter()
        st_w = round( (self.start_time - self.time_ref), 5)
        msg = "\nWORKER: Started at " + str(st_w)
        self.signal_start_end.emit(msg)


    # Method to stop Worker regularly (by stopping while loop)
    # ~~~~~~~~~~~~~~~~~~~~
    # Note : This method will be called directly; not via signal
    def ende(self):
        print("WORKER: stopping ... ")
        # this stops the while loop and leads indirectly
        # to the invocation of othee methods
        self.num = 0

    # Method to print final msg and send a signal
    # ~~~~~~~~~~~~~~~~~~~~~~~
    # Will be called directly - not via signal
    def end_msg(self):
        print("WORKER: finished !")
        end_time = round( (time.perf_counter() - self.time_ref), 5)
        msg = "\nWORKER: Finished at " + str(end_time)
        self.signal_start_end.emit(msg)


    # Worker's main function. Gets started via signal from thread
    # Note: This method will be connected to a strat signal from the
    #       (affine) thread => Should be marked as a SLOT in Python
    @QtCore.pyqtSlot()
    def worker_run(self):
        i = 0
        n_worker_batch = 0
        # Need a while loop as self.num will be changed dynamically
        while i < self.num:
            # Print option to a notebook cell
            # print("Worker:  i=", i)

            # Create new sine-data with growing period number
            self.pi_fact += 1

            # We create a full object for data transmission
            # (recommended; but in real world apps we may
```

```
            #  need to trigger garbage collection sometimes)
            sin_obj = SinObj(pi_fact=self.pi_fact)

            # put obj into queue for receiver
            self.queue_sins.put(sin_obj)

            # Capture print() -> put text into queue for receiver
            # In parallel: After each "batch" send a msg to qMainWin
            if i%self.batch_size == 0:
                n_worker_batch += 1
                print_text = "Worker i = " + str(i) + \
                             " :: w-batch = " + str(n_worker_batch)
                print_text2 = "Worker To Rec.: " + print_text

                # Print something uncaptured to stdout
                # print(print_text)

                # ! Note: Capturing will always print a "\n" ahead
                # ! This leads to 2 entries in the queue:
                #   "\n" and print-"text"
                with redirect_stdout(self.streamObj):
                    print(print_text2)

            # Send signal to main window with msg-text
            time_pt = round( (time.perf_counter() - self.time_ref), 5)
            msg = "\nWorker: Sine obj " + str(i) + \
                  " to queue (at " + str(time_pt) + ", batch: " + \
                  str(n_worker_batch) + ")"
            self.signal_msg.emit(msg)

            # Pause during which other threads can work.
            # In real life cases we have ongoing data production
            # operations, which should be done by libs/operations
            # bypassing the GIL (NUMPY, OpenBLAS, TF2, I/O)
            time.sleep(self.time_sleep)
            i += 1

        # Regular end of Worker
        # ~~~~~~~~~~~~~~~~~~~~~
        # We directly set the status variable for a running worker
        # to False. This is harmless as fully controlled and no
        # conflicting events can occur
        self.qMainWin.worker_is_running = False

        # Sequence of required steps to shutdown object AND thread
        self.end_msg()
        self.deleteLater()
```

```
        self.thrd.quit()
```

0.5.2 Main Object for Receiver Thread

This class is for a receiver object in the “receiver thread”.

It periodically reads out sine objects and msgs from two queues (filled by the Worker). It adds a color to the sinus-data. It sends a signal with the data (including a msg) to the main window.The Receiver is assumed to work faster than the Worker.

[7]:

```
# Receiver Object [derived from QObject]
# to be run in a QThread
class MyReceiver(QObject):
    # Signals at start and regular end of the receiver object
    # will be send to qMainWin
    signal_start_end = pyqtSignal(str)
    signal_finished = pyqtSignal()
    # Intermediate signals to emit - with sine data in object form
    signal_data = QtCore.pyqtSignal(object)
    signal_msg  = QtCore.pyqtSignal(str)

    # Constructor
    def __init__(self, qMainWin, thrd
                 , num_iterations=50, time_sleep=0.05):

        # Parameters:
        # ~~~~~~~~~~
        # qMainWin: a reference to the Main Application Window
        # thrd: a reference to the thread which the object is affine to
        # num_iterations: max num of iterations of while loop
        # time_sleep: sleep time between iterations

        QObject.__init__(self)

        # Main App window and thrd
        self.qMainWin = qMainWin
        self.thrd = thrd

        # Color list
        self.li_col = ['blue', 'red', 'orange', 'green', 'darkgreen'
                       , 'darkred', 'magenta', 'black']


        # Queues
        # ~~~~~~~~
        # Queue for messages from Worker - stdout-redirect
```

```
        self.queue_msgs = qMainWin.queue_worker_msgs
        # Queue for sine data
        self.queue_sins = qMainWin.queue_sins

        # maximum number of iterations and sleep time
        self.num = num_iterations
        self.time_sleep = time_sleep

        # Worker batch size
        self.worker_batch_size = qMainWin.batch_size_worker
        # Receiever batch size
        self.receiver_batch_size = qMainWin.batch_size_receiver

        # Connect signals to callbacks
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # connect msg signal to callback in the main window
        self.signal_msg.connect(qMainWin.callback_for_receiver_msgs)
        # connect data signal to callback in the main window
        self.signal_data.connect(qMainWin.callback_for_receiver_data)
        # connect signal for regular end of Receiver object
        self.signal_start_end.connect(qMainWin.callback_receiver_start_end)
        # Special signal at end of the Receiver to stop Worker, too
        self.signal_finished.connect(qMainWin.callback_receiver_finish)

        # get a time-reference
        self.time_ref = qMainWin.time_thrds_start
        self.start_time = time.perf_counter()
        st_r = round( (self.start_time - self.time_ref), 5)
        msg = "\nRECEIVER: Started at " + str(st_r)
        self.signal_start_end.emit(msg)


    # Method to stop Receiver (by stopping while loop)
    # ~~~~~~~~~~~~~~~~~~~~~~
    def ende(self):
        print("RECEIVER: Stopping ...")
        # this stops the while loop
        self.num = 0

    # Method to print final msg + send signal to qMainWin
    # ~~~~~~~~~~~~~~~~~~~~~~~
    def end_msg(self):
        print("RECEIVER: finished !")
        end_time = round( (time.perf_counter() - self.time_ref), 5)
        msg = "\nRECEIVER: Finished at " + str(end_time)
        self.signal_start_end.emit(msg)
        self.signal_finished.emit()
```

```
    @QtCore.pyqtSlot() # gets started signal from thread
    def receiver_run(self):
        i = 0
        n_worker_batches = 0
        n_receiver_batches = 0
        n_sine_objects = 0
        col_rand = 1
        while i < self.num:
            #if i%10 == 0:
                #print("Receiver: loop i = ", i)

            # Receiver works faster than Worker
            # gets data from 2 queues

            # Data from Worker msg queue
            text_worker = ''
            if self.queue_msgs.qsize() > 0:
                text_worker = self.queue_msgs.get()
                slash_n = self.queue_msgs.get()
                n_worker_batches += 1
                # print("Receiver: i = ", i, " :: n_w_batch = ",␣
↪n_worker_batches)
                # print("Receiver: i = ", i, " :: text_worker = ", text_worker)
                msg = "\nRECEIVER: Worker msg =  " + \
                      text_worker
                self.signal_msg.emit(msg)


            # print(self.queue.qsize())
            if self.queue_sins.qsize() > 0:
                n_sine_objects += 1
                # print("From Receiver: n_sine = ", n_sine_objects)
                sine_obj = self.queue_sins.get()
                # add color
                sine_obj.col = self.li_col[col_rand]
                # send signal with dtaa obj to qMainWin
                self.signal_data.emit(sine_obj)

                if n_sine_objects%self.receiver_batch_size == 0:
                    col_rand = np.random.randint(0, len(self.li_col))
                    n_receiver_batches += 1
                    msg_batch = "\nRECEIVER: Rec-batch Nr " + \
                                str(n_receiver_batches) + "\n"
                    self.signal_msg.emit(msg_batch)

            time.sleep(self.time_sleep)
            i += 1

        # Regular end of Receiver
        # ~~~~~~~~~~~~~~~~~~~~~~~
        # We directly set the status of the running Receiver to False
        self.qMainWin.receiver_is_running = False
        # Sequence of steps to shutdown object and thread
        self.end_msg()
        self.deleteLater()
        self.thrd.quit()
```

0.6 The Main application window

We always stop a thread by stopping the while loop of its assigned object AND triggering final actions to deleteLater() the object and event loop (if started).

[8]:

```
# A Man Window for our example application
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# An instance will produce a Qt-window on the screen
class MyApp(QtWidgets.QMainWindow):

    # Constructor
    def __init__(self, max_worker_iters=20, max_receiver_iters=50):

        # initialization of parent class
        QtWidgets.QMainWindow.__init__(self)

        self.setWindowTitle("PyQt My-Threader")

        # Initial size of Qt window
        #self.setMinimumSize(QSize(300, 300))
        self.resize(960, 800)

        # some useful colors
        self.col_red = QColor('red')
        self.col_darkred = QColor(125, 0, 0)
        self.col_darkblue = QColor(0, 0, 125)
        self.col_darkgreen = QColor(0, 125, 0)
        self.col_black = QColor(3, 3, 3)


        # Queues for data and msgs
        # ~~~~~~~~~~~~~~~~~~~~~~~~
        # Both queues will be read by the Receiver object
        # in the receiver thread
```

```
        # Queue for msgs from the worker thread
        self.queue_worker_msgs = queue.Queue()
        # Queue for sinx/siny data from worker
        self.queue_sins = queue.Queue()

        # Design of the Main Window
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~
        # Central widget
        self.mainWidget = QtWidgets.QWidget()
        self.setCentralWidget(self.mainWidget)

        # VBOX-Layout
        self.mainLayout = QVBoxLayout(self)
        self.mainWidget.setLayout(self.mainLayout)
        self.mainLayout.setContentsMargins(0, 0, 0, 0)
        self.mainLayout.setSpacing(12)


        # Groupbox1 = Multiple Buttons / fixed height
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        self.groupbox1 = QGroupBox("   Buttons for Thread Control")
        self.groupbox1.setStyleSheet('font-weight:bold;')
        self.groupbox1.setFixedHeight(100)
        self.mainLayout.addWidget(self.groupbox1)
        self.vbox1 = QHBoxLayout()
        self.groupbox1.setLayout(self.vbox1)

        # 1st button: start threads
        self.but_start_threads = QPushButton('start\nthreads', self)
        self.but_start_threads.setMinimumSize(QSize(150, 50))
        sizePolicy_but = QSizePolicy(QSizePolicy.Maximum, QSizePolicy.Maximum)
        self.but_start_threads.setSizePolicy(sizePolicy_but)
        self.vbox1.addWidget(self.but_start_threads)

        # Stretch element
        self.vbox1.insertStretch(1)

        # 2nd button: stop threads
        # 1st button: start threads
        self.but_stop_threads = QPushButton('stop\nthreads', self)
        self.but_stop_threads.setMinimumSize(QSize(150, 50))
        sizePolicy_but = QSizePolicy(QSizePolicy.Maximum, QSizePolicy.Maximum)
        self.but_stop_threads.setSizePolicy(sizePolicy_but)
        self.vbox1.addWidget(self.but_stop_threads)
```

```
        # Groupbox 2 = Figure Canvas for Matplotlib Figure
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # WARNING: Create matplotlib figure inside this class.
        # ~~~~~~
        # Otherwise you MUST destroy figure separatly
        # after closing the window via x on GUI-window
        # or win.close in Jupyterlab
        # Alternative: catch the close event and close then
        self.groupbox2 = QGroupBox("   Qt-Canvas for MPL-plot")
        self.groupbox2.setStyleSheet('font-weight:bold;')
        self.mainLayout.addWidget(self.groupbox2)
        self.vbox2 = QVBoxLayout()
        self.groupbox2.setLayout(self.vbox2)

        # Create Matplotlib figure
        self.fig1 = Figure(figsize=[5., 3.], dpi=96)
        # Create ax inside
        self.ax11 = self.fig1.add_subplot(111)
        # Assign Qt FigureCanvas widget to fig1-variable
        self.canvas1 = FigureCanvas(self.fig1) # !important
        # Create interactive navigation toolbar widget
        self.nav1 = NavigationToolbar(self.canvas1
                                      , self.mainWidget)

        # Add figure and toolbar widgets to vbox_fig1
        self.vbox2.addWidget(self.nav1)
        self.vbox2.addWidget(self.canvas1)

        # !!! Important Otherwise the nav-bar will crash
        # It needs a drawn ax => Else mismatch with
        # figure.canvas and navi bar (x-position)
        self.canvas1.draw()

        # Set vertical Stretchfactor
        self.mainLayout.setStretchFactor(self.groupbox2, 1)


        # Groupbox 3 = Multiple QTextEdits
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        self.groupbox3 = QGroupBox("   Messages from thread-affine objects")
        self.groupbox3.setStyleSheet('font-weight:bold;')
        #self.groupbox3.setFixedHeight(250)
        self.mainLayout.addWidget(self.groupbox3)
        self.hbox3 = QHBoxLayout()
        self.groupbox3.setLayout(self.hbox3)
```

```
        # Display Ctrl messages
        self.groupbox3_1 = QGroupBox(" Ctrl-Msgs")
        self.groupbox3_1.setStyleSheet('font-weight:bold;')
        self.vbox3_1 = QVBoxLayout()
        self.groupbox3_1.setLayout(self.vbox3_1)
        self.qTextEdit_1 = QTextEdit()  # For control msgs
        self.qTextEdit_1.setReadOnly(True)
        self.hbox3.addWidget(self.groupbox3_1)
        self.vbox3_1.addWidget(self.qTextEdit_1)

        # Display Worker messages
        self.groupbox3_2 = QGroupBox(" Worker-Msgs")
        self.groupbox3_2.setStyleSheet('font-weight:bold;')
        self.vbox3_2 = QVBoxLayout()
        self.groupbox3_2.setLayout(self.vbox3_2)
        self.qTextEdit_2 = QTextEdit()  # For Worker msgs
        self.qTextEdit_2.setReadOnly(True)
        self.hbox3.addWidget(self.groupbox3_2)
        self.vbox3_2.addWidget(self.qTextEdit_2)

        # Display Receiver messages
        self.groupbox3_3 = QGroupBox(" Receiver-Msgs")
        self.groupbox3_3.setStyleSheet('font-weight:bold;')
        self.vbox3_3 = QVBoxLayout()
        self.groupbox3_3.setLayout(self.vbox3_3)
        self.qTextEdit_3 = QTextEdit()  # For Reveiver msgs
        self.qTextEdit_3.setReadOnly(True)
        self.hbox3.addWidget(self.groupbox3_3)
        self.vbox3_3.addWidget(self.qTextEdit_3)

        # equal horizontal stretch factor
        self.hbox3.setStretchFactor(self.groupbox3_1, 1)
        self.hbox3.setStretchFactor(self.groupbox3_2, 1)
        self.hbox3.setStretchFactor(self.groupbox3_3, 1)

        # Set vertical Stretchfactor - realtive to
        # previous plot figure
        self.mainLayout.setStretchFactor(self.groupbox3, 1)


        # Connect buttons to callbacks
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        self.but_start_threads.clicked.connect(self.start_threads)
        self.but_stop_threads.clicked.connect(self.stop_threads)


        # Setting thread and worker parameters
```

```
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        self.threads = []

        # Max nums of iterations for Worker / Receiver
        self.max_num_worker_iters = max_worker_iters
        self.max_num_receiver_iters = max_receiver_iters

        # Batch sizes Worker / Receiever (here just for intermediate msgs)
        self.batch_size_worker = 5
        self.batch_size_receiver = 5

        # Sleep times for Worker / Receiever [secs]
        self.time_sleep_worker = 0.1
        self.time_sleep_receiver = 0.05

        # Status of threads
        self.worker_is_running = False
        self.receiver_is_running = False

        # display Qt-window on screen
        self.show()


    # Function to start the two background threads
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def start_threads(self):

        # Clear some objects
        self.qTextEdit_1.clear()   # For control msgs
        self.qTextEdit_1.setFontWeight(QFont.Normal)
        self.qTextEdit_1.setTextColor(self.col_black)

        self.qTextEdit_2.clear()   # For msgsfrom Worker Thread
        self.qTextEdit_2.setFontWeight(QFont.Normal)
        self.qTextEdit_2.setTextColor(self.col_black)

        self.qTextEdit_3.clear()   # For msgs from Receiver Thread
        self.qTextEdit_3.setFontWeight(QFont.Normal)
        self.qTextEdit_3.setTextColor(self.col_black)

        self.queue_worker_msgs.queue.clear()
        self.queue_sins.queue.clear()

        # A list for the opened threads
        self.threads = []
```

```
        # Time reference
        self.time_thrds_start = time.perf_counter()


        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # Prepare Worker thread and start it
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        self.worker_thread = QThread()

        # Set up a worker object - we submit also the ref. to the main win
        self.worker_obj = MyWorker(self
                                   , self.worker_thread
                                   , num_iterations=self.max_num_worker_iters
                                   , time_sleep=self.time_sleep_worker)

        # Change thread affinity of worker object
        self.worker_obj.moveToThread(self.worker_thread)

        # Start the objects function "worker_run"
        # We use an automatic start signal from the thread for this purpose
        self.worker_thread.started.connect(self.worker_obj.worker_run)


        # End worker object and worker thread
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        # Situation 1:   Regular stop via the worker object
        # ~~~~~~~~~~~    The while loop in the worker obj is forced to stop =>
        #                The thread should stop, too
        # All the required action is done in the worker object.

        # Situation 2:  Stop of thread by external command
        # ~~~~~~~~~~~   e.g. by some brutal intervention
        # E.g. the user closes the main window =>
        # Relevant is a close event which can be captured.

        # We always turn such situations into regular stops.
        # But we also need to delete the thread control objects
        # after the threads are stopped.
        # We also display a final message.
        # We use an automatic signal at the end of the threads operations to
        # trigger these actions.

        self.worker_thread.finished.connect(self.worker_thrd_finished)
        # last direct print related to worker
        # self.worker_thread.finished.connect(
        #     lambda: print("Finished Worker Thread")
```

```
        # )

        # Start the worker thread
        # ~~~~~~~~~~~~~~~~~~~~~~~~~
        # # triggers the thread's run function, if existent
        self.threads.append(self.worker_thread)
        self.worker_thread.start()
        self.worker_is_running = True



        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # Prepare Receiver thread and start it
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        self.receiver_thread = QThread()

        # Set up a receiver object
        self.receiver_obj = MyReceiver(self
                                      , self.receiver_thread
                                      , num_iterations=self.
↪max_num_receiver_iters
                                      , time_sleep= self.time_sleep_receiver)

        self.receiver_obj.moveToThread(self.receiver_thread)
        # Start the objects function "receiver_run" when thread starts
        self.receiver_thread.started.connect(self.receiver_obj.receiver_run)

        # finished
        self.receiver_thread.finished.connect(self.receiver_thrd_finished)
        self.receiver_thread.finished.connect(
            lambda: print("Finished Thread Receiver")
        )


        # Start the receiver thread
        # ~~~~~~~~~~~~~~~~~~~~~~~~~
        # # triggers the thread's run function, if existent
        self.threads.append(self.receiver_thread)
        self.receiver_thread.start()
        self.receiver_is_running = True



    # Callbacks for worker and receiver threads
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    # Method for regular start/end signal of Worker
```

```
    # msg will be written to 1st QTextEdit
    @QtCore.pyqtSlot(str)
    def callback_worker_start_end(self, text):
        # We write a msg to the Ctrl QTExtEdit_1
        self.qTextEdit_1.moveCursor(QTextCursor.End)
        self.qTextEdit_1.setFontWeight(QFont.Normal)
        self.qTextEdit_1.setTextColor(self.col_black)
        self.qTextEdit_1.insertPlainText(text)
        QtWidgets.QApplication.processEvents() #update gui for pyqt

    # Method to handle Worker signals with msgs
    # will be written to the 2nd QTextEdit
    @QtCore.pyqtSlot(str)
    def callback_for_worker_msgs(self, text):
        self.qTextEdit_2.moveCursor(QTextCursor.End)
        self.qTextEdit_2.setFontWeight(QFont.Normal)
        self.qTextEdit_2.setTextColor(self.col_black)
        self.qTextEdit_2.insertPlainText(text)
        QtWidgets.QApplication.processEvents() #update gui for pyqt


    # Method for regular start/end signal of Receiver
    # msg will be written to 1st QTextEdit
    @QtCore.pyqtSlot(str)
    def callback_receiver_start_end(self, text):
        # We write a msg to the Ctrl QTExtEdit_1
        self.qTextEdit_1.moveCursor(QTextCursor.End)
        self.qTextEdit_1.setFontWeight(QFont.Normal)
        self.qTextEdit_1.setTextColor(self.col_black)
        self.qTextEdit_1.insertPlainText(text)
        QtWidgets.QApplication.processEvents() #update gui for pyqt

    # Method for regular end signal of Reciever
    # => Stop the Worker, too
    @QtCore.pyqtSlot()
    def callback_receiver_finish(self):
        # Stop worker - if still running
        if self.worker_is_running:
            self.worker_obj.ende()
        # We write a msg to the Ctrl QTExtEdit_1
        msg = "\nStopping Worker due to end of Receiver"
        self.qTextEdit_1.moveCursor(QTextCursor.End)
        self.qTextEdit_1.insertPlainText(msg)
        QtWidgets.QApplication.processEvents() #update gui for pyqt


    # Method to handle Receiver signals with msgs
    # will be written to the 3rd QTextEdit
    @QtCore.pyqtSlot(str)
    def callback_for_receiver_msgs(self, text):
        self.qTextEdit_3.moveCursor(QTextCursor.End)
        self.qTextEdit_3.setFontWeight(QFont.Normal)
        self.qTextEdit_3.setTextColor(self.col_black)
        self.qTextEdit_3.insertPlainText(text)
        QtWidgets.QApplication.processEvents() #update gui for pyqt


    # Method for Receiver signals with sine data
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    @QtCore.pyqtSlot(object)
    def callback_for_receiver_data(self, sine_obj):
        sin_x = sine_obj.sinx
        sin_y = sine_obj.siny
        sine_col = sine_obj.col
        # Hier weitermachen XXXX
        self.ax11.clear()
        self.ax11.plot(sin_x, sin_y, color=sine_col)
        self.fig1.canvas.draw()
        self.fig1.canvas.flush_events()


    # Stop threads - and related msgs
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    # Reaction to finished Worker thread
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    @QtCore.pyqtSlot()
    def worker_thrd_finished(self):
        if self.worker_is_running:
            print("STRANGE END of WORKER THREAD!")

        text = "\nWORKER: Thread finalized"
        print(text)
        self.qTextEdit_1.moveCursor(QTextCursor.End)
        self.qTextEdit_1.insertPlainText( text )
        QtWidgets.QApplication.processEvents() #update gui for pyqt
        self.worker_thread.deleteLater()
        self.threads.pop(0)


    # Reaction to finished Receiver thread
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    @QtCore.pyqtSlot()
    def receiver_thrd_finished(self):
```

```
        if self.worker_is_running:
            print("STRANGE END of RECEIVER THREAD!")

        text = "\nRECEIVER: Thread finalized"
        print(text)
        self.qTextEdit_1.moveCursor(QTextCursor.End)
        self.qTextEdit_1.insertPlainText( text )
        QtWidgets.QApplication.processEvents() #update gui for pyqt
        self.threads.pop(0)


    # Actively stop worker obj and thread
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def stop_worker(self):
        if self.worker_is_running:
            self.worker_obj.ende()

    # Actively stop receiver obj and thread
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def stop_receiver(self):
        if self.receiver_is_running:
            self.receiver_obj.ende()


    # Actively stop threads and worker/receiver objects
    # ~~~~~~~~~~~~~~~~~~~~~
    @QtCore.pyqtSlot()
    def stop_threads(self, b_how=0):
        print("Start Finishing Threads")
        # An end of the Receiver will stop the Worker, too
        self.stop_receiver()
        # The threads should come to an automatic end, too
        if b_how == 0:
            text = "\nInitialized end of threads (via button)"
        else:
            text = "\nInitialized end of threads"
        self.qTextEdit_1.moveCursor(QTextCursor.End)
        self.qTextEdit_1.insertPlainText( text )
        QtWidgets.QApplication.processEvents() #update gui for pyqt


    # Closing main window by pressing X
    # ~~~~~~~~~~~~~~~~~~~~~~
    # This event must lead to a controlled end of
    # both the threads and their affine objects
    @QtCore.pyqtSlot()
    def closeEvent(self, event):
```

```
        # An end of the Receiver will stop the Worker, too
        text = "\nUser is closing Main Win "
        print(text)
        self.qTextEdit_1.moveCursor(QTextCursor.End)
        self.qTextEdit_1.insertPlainText( text )
        QtWidgets.QApplication.processEvents() #update gui for pyqt

        self.stop_threads(b_how=1)
        # close figure
        self.canvas1.deleteLater()

        # Wait a bit
        time_sleep = 1.0
        time.sleep(time_sleep)
        # accept event
        event.accept()
```

0.7 Execution code

[9]:

```
max_worker_iters = 60
max_receiver_iters = 100
```

[10]:

```
# Create QApplication and QWidget
app = MyApp(max_worker_iters=max_worker_iters
            , max_receiver_iters=max_receiver_iters)
```

```
Start Finishing Threads
RECEIVER: Stopping …
RECEIVER: finished !
WORKER: stopping …
STRANGE END of RECEIVER THREAD!

RECEIVER: Thread finalized
Finished Thread Receiver
WORKER: finished !

WORKER: Thread finalized

User is closing Main Win
Start Finishing Threads
```

[ ]:

```
```