qtutils Documentation

Release 3.0.1.dev2+g9437142.d20210129

Christopher Billington, Philip Starkey

Jan 29, 2021

CONTENTS

QtUtils is a Python library that provides convenient tools for Python Developers creating applications using the PyQt/PySide widget library.

Utilities include those providing thread-safe access to Qt objects, simplified QSettings storage, and dynamic widget promotion when loading UI files. QtUtils also includes the Fugue icon set, free to use with attribution to Yusuke Kamiyamane.

Note: *qtutils* 3.0 dropped support for Python 2.7, PyQt4 and PySide. If you need to use these platforms, you may use *qtutils* 2.3.2 or earlier. *qtutils* 2.3.2 provides an abstraction layer for PySide/PyQt4 that matches the PyQt5 API.

CHAPTER

ONE

CONTENTS

1.1 Installation

These installation instructions assume you already have Python installed. If you do not already have a copy of Python, we recommend you install Anaconda Python.

1.1.1 PyPi

To install qtutils from the Python Package Index run:

```
pip install qtutils
```

Upgrading qtutils with PyPI

To upgrade to the latest version of qtutils, run:

```
pip install -U qtutils
```

To upgrade to a specific version of qtutils (or, alternatively, if you wish to downgrade), run:

```
pip install -U qtutils==<version>
```

where `<version>` is replaced by the version you wish (for example `pip install -U qtutils==2.3.2`).

1.1.2 Anaconda

To install qtutils using *conda* run:

```
conda install -c labscript-suite qtutils
```

Note: The qtutils library is published on the labscript-suite anaconda cloud channel as it was created by labscript-suite developers for use in that software suite.

Upgrading qtutils with conda

To upgrade to the latest version of qtutils, run:

```
conda update -c labscript-suite qtutils
```

To upgrade to a specific version of qtutils (or, alternatively, if you wish to downgrade), run:

```
conda update -c labscript-suite qtutils=<version>
```

where `<version>` is replaced by the version you wish (for example `conda update -c labscript-suite qtutils=3.0.0`).

1.1.3 Development Version

To install latest development version, clone the GitHub repository and run `pip install .` to install, or `pip install -e .` to install in 'editable' mode.

1.2 Convenience functions for using Qt safely from Python threads

QtUtils provides convenience functions for accessing Qt objects in a thread safe way. Qt requires that all GUI objects exist in the MainThread and that access to these objects is only made from the MainThread (see Qt documentation). This, while understandable, imposes significant limits on Python applications where threading is easy. While there are solutions using Qt signals, slots and a `QThread`, these require significant boiler plate code that we believe is unnecessary.

Note: There is some debate as to whether using Python threads with any part of the Qt library is safe, however this has been recently challenged. The QtUtils library only instantiates a `QEvent` and calls `QCoreApplication.postEvent()` from a Python thread. It seems likely that as long as the underlying Python threading implementation matches the underlying Qt threading implementation **for your particular platform**, that there is no issue with how we have written this library. While we have not observed any issues with our library (and we have used it extensively on Windows, OSX and Ubuntu), this does not mean all platforms will behave in the same way. If this matters to you, we suggest you confirm the underlying thread implementation for your build of Python and Qt.

1.2.1 Examples

We utilise the Qt event loop to execute arbitrary methods in the MainThread by posting a Qt event to the MainThread from a secondary thread. QtUtils provides a function called `inmain` which takes a reference to a method to execute in the MainThread, followed by any arguments to be passed to the method.

```
from qtutils import inmain

# This is equivalent to calling my_func(arg1, arg2, foo=7, bar='baz') in the␣
↪MainThread
# The calling thread will wait for the result to be returned before continuing
result = inmain(my_func, arg1, arg2, foo=7, bar='baz')
```

A call to `inmain` blocks the calling thread until the Qt event loop can process our message, execute the specified method and return the result. For situations where you don't wait to wait for the result, or you wish to do some other processing while waiting for the result, QtUtils provides the `inmain_later` function. This works in the same way

as `inmain`, but returns a reference to a Python `Queue` object immediately. The result can be retrieved from this queue at any time, as shown in the following example:

```
from qtutils import inmain_later, get_inmain_result

# This is equivalent to calling my_func(arg1, arg2, foo=7, bar='baz') in the␣
↪MainThread
# The calling thread will immediately continue execution, and the result of the␣
↪function
# will be placed in the queue once the Qt event loop has processed the request
queue = inmain_later(my_func, arg1, arg2, foo=7, bar='baz')
# You can get the result (or raise any caught exceptions) by calling
# Note that any exception will have already been raised in the MainThread
result = get_inmain_result(queue)
```

This of course works directly with Qt methods as well as user defined functions/methods. For example:

```
from qtutils import inmain_later, get_inmain_result, inthread

def run_in_thread(a_line_edit, ignore=True):
    # set the text of the line edit, and wait for it to be set before continuing
    inmain(a_line_edit.setText, 'foobar')

    # Get the text of the line edit, and wait for it to be returned
    current_text = inmain(a_line_edit.text)

    # queue up a call to deselect() and don't wait for a result to be returned
    inmain_later(a_line_edit.deselect)

    # request the text of the line edit, but don't wait for it to be returned
    # However, this call is guaranteed to run AFTER the above inmain_later call
    queue = inmain_later(a_line_edit.text)

    # do some intensive calculations here

    # now get the text
    current_text = get_inmain_result(queue)
    print(current_text)

# instantiate a QLineEdit
# This object should only be accessed from the MainThread
my_line_edit = QLineEdit()

# starts a Python thread (in daemon mode)
# with target run_in_thread(my_line_edit, ignore=False)
thread = inthread(run_in_thread,my_line_edit,ignore=False)
```

As you can see, the change between a direct call to a Qt method, and doing it in a thread safe way, is very simple:

```
       a_line_edit.setText('foobar')
inmain(a_line_edit.setText,'foobar')
```

We also provide decorators so that you can ensure the decorated method always runs in the MainThread regardless of the calling thread. This is particularly useful when combined with Python properties.

```
# This function will always run in the MainThread, regardless of which thread calls␣
↪it.
# The calling thread will block until the function is run in the MainThread, and the␣
↪result returned.
```

(continues on next page)

(continued from previous page)

```
# If called from the MainThread, the function is executed immediately as if you had␣
↪called a_function()
@inmain_decorator(wait_for_return=True)
def a_function(a_line_edit):
    a_line_edit.setText('bar')
    return a_line_edit.text()

# This function will always run in the MainThread, regardless of which thread calls␣
↪it.
# A call to this function will return immediately, and the function will be run at a
# later time. A call to this function returns a python Queue.Queue() in which the␣
↪result of
# the decorated function will eventually be placed (or any exception raised)
@inmain_decorator(wait_for_return=False)
def another_function(a_line_edit):
    a_line_edit.setText('baz')
```

QtUtils also provides a convenience function for launching a Python thread in daemon mode. `inthread(target_method, arg1, arg2, ... kwarg1=False, kwargs2=7, ...)`

Exception handling

Typically, exceptions are raised in the calling thread. However, `inmain_later` and the associated decorator will also raise the exception in the MainThread as there is no guarantee that the results will ever be read from the calling thread.

Using QtUtils from the MainThread

When using `inmain`, or the associated decorator, QtUtils will bypass the Qt Event loop as just immediately execute the specified method. This avoids the obvious deadlock where the calling code is being executed by the Qt event loop, and is now waiting for the Qt event loop to execute the next event (which won't ever happen because it is blocked waiting for the next event by the calling code). `inmain_later` still posts an event to the Qt event loop when used from the MainThread. This is useful if you want to execute something asynchronously from the MainThread (for example, asynchronously update the text of a label) but we recommend you do not attempt to read the result of such a call as you risk creating a deadlock.

What if I want to wait for user input in a thread?

If you want your thread to wait for user input, then this is not the library for you! We suggest you check out how to Wait in thread for user input from GUI for a Qt solution and/or Python threading events for a Python solution.

1.2.2 API reference

class qtutils.invoke_in_main.**CallEvent**(*queue*, *exceptions_in_main*, *fn*, **args*, ***kwargs*)

An event containing a request for a function call.

class qtutils.invoke_in_main.**Caller**

An event handler which calls the function held within a CallEvent.

event(*self*, *QEvent*) → bool

qtutils.invoke_in_main.**get_inmain_result**(*queue*)
Processes the result of *qtutils.invoke_in_main.inmain_later()*.

This function takes the queue returned by `inmain_later` and blocks until a result is obtained. If an exception occurred when executing the function in the MainThread, it is raised again here (it is also raised in the MainThread). If no exception was raised, the result from the execution of the function is returned.

Parameters **queue** – The Python Queue object returned by `inmain_later`

Returns The result from executing the function specified in the call to `inmain_later`

qtutils.invoke_in_main.**inmain**(*fn*, **args*, ***kwargs*)
Execute a function in the main thread. Wait for it to complete and return its return value.

This function queues up a custom `QEvent` to the Qt event loop. This event executes the specified function `fn` in the Python MainThread with the specified arguments and keyword arguments, and returns the result to the calling thread.

This function can be used from the MainThread, but such use will just directly call the function, bypassing the Qt event loop.

Parameters

- **fn** – A reference to the function or method to run in the MainThread.
- ***args** – Any arguments to pass to `fn` when it is called from the MainThread.
- ****kwargs** – Any keyword arguments to pass to `fn` when it is called from the MainThread

Returns The result of executing `fn(*args, **kwargs)`

qtutils.invoke_in_main.**inmain_decorator**(*wait_for_return=True*, *exceptions_in_main=True*)
A decorator which enforces the execution of the decorated thread to occur in the MainThread.

This decorator wraps the decorated function or method in either *qtutils.invoke_in_main.inmain()* or *qtutils.invoke_in_main.inmain_later()*.

Keyword Arguments

- **wait_for_return** – Specifies whether to use `inmain` (if `True`) or `inmain_later` (if `False`).
- **exceptions_in_main** – Specifies whether the exceptions should be raised in the main thread or not. This is ignored if `wait_for_return=True`. If this is `False`, then exceptions may be silenced if you do not explicitly use *qtutils.invoke_in_main.get_inmain_result()*.

Returns

The decorator returns a function that has wrapped the decorated function in the appropriate call to `inmain` or `inmain_later` (if you are unfamiliar with how decorators work, please see the Python documentation).

When calling the decorated function, the result is either the result of the function executed in the MainThread (if `wait_for_return=True`) or a Python Queue to be used with *qtutils.invoke_in_main.get_inmain_result()* at a later time.

qtutils.invoke_in_main.**inmain_later**(*fn*, **args*, ***kwargs*)
Queue up the executing of a function in the main thread and return immediately.

This function queues up a custom `QEvent` to the Qt event loop. This event executes the specified function `fn` in the Python MainThread with the specified arguments and keyword arguments, and returns a Python Queue which will eventually hold the result from the executing of `fn`. To access the result, use *qtutils.invoke_in_main.get_inmain_result()*.

This function can be used from the MainThread, but such use will just directly call the function, bypassing the Qt event loop.

Parameters

- **fn** – A reference to the function or method to run in the MainThread.
- ***args** – Any arguments to pass to `fn` when it is called from the MainThread.
- ****kwargs** – Any keyword arguments to pass to `fn` when it is called from the MainThread

Returns A Python Queue which will eventually hold the result `(fn(*args, **kwargs), exception)` where `exception=[type,value,traceback]`.

`qtutils.invoke_in_main.`**`inthread`**(*f*, **args*, ***kwargs*)

A convenience function for starting a Python thread.

This function launches a Python thread in Daemon mode, and returns a reference to the running thread object.

Parameters

- **f** – A reference to the target function to be executed in the Python thread.
- ***args** – Any arguments to pass to `f` when it is executed in the new thread.
- ****kwargs** – Any keyword arguments to pass to `f` when it is executed in the new thread.

Returns A reference to the (already running) Python thread object

PYTHON MODULE INDEX

q

INDEX