pytest-qt Documentation

Release 1.5.0

Bruno Oliveira

June 29, 2015

Contents

Repository GitHub

Version 1.5.0

License LGPL

Author Bruno Oliveira

CHAPTER 1

Introduction

pytest-qt is a pytest plugin that provides fixtures to help programmers write tests for PySide and PyQt.

The main usage is to use the `qtbot` fixture, which provides methods to simulate user interaction, like key presses and mouse clicks:

```
def test_hello(qtbot):
    widget = HelloWidget()
    qtbot.addWidget(widget)

    # click in the Greet button and make sure it updates the appropriate label
    qtbot.mouseClick(window.button_greet, QtCore.Qt.LeftButton)

    assert window.greet_label.text() == 'Hello!'
```

1.1 Requirements

Python 2.6 or later, including Python 3+.

Tested with pytest version 2.5.2.

Works with either `PySide`, `PyQt4` or `PyQt5`, picking whichever is available on the system giving preference to the first one installed in this order:

- `PySide`
- `PyQt4`
- `PyQt5`

To force a particular API, set the environment variable `PYTEST_QT_API` to `pyside`, `pyqt4` or `pyqt5`.

1.2 Installation

The package may be installed by running:

```
pip install pytest-qt
```

Or alternatively, download the package from pypi, extract and execute:

```
python setup.py install
```

Both methods will automatically register it for usage in `py.test`.

1.3 Development

If you intend to develop `pytest-qt` itself, use virtualenv to activate a new fresh environment and execute:

```
git clone https://github.com/pytest-dev/pytest-qt.git
cd pytest-qt
python setup.py develop
pip install pyside # or pyqt4/pyqt5
```

1.4 Versioning

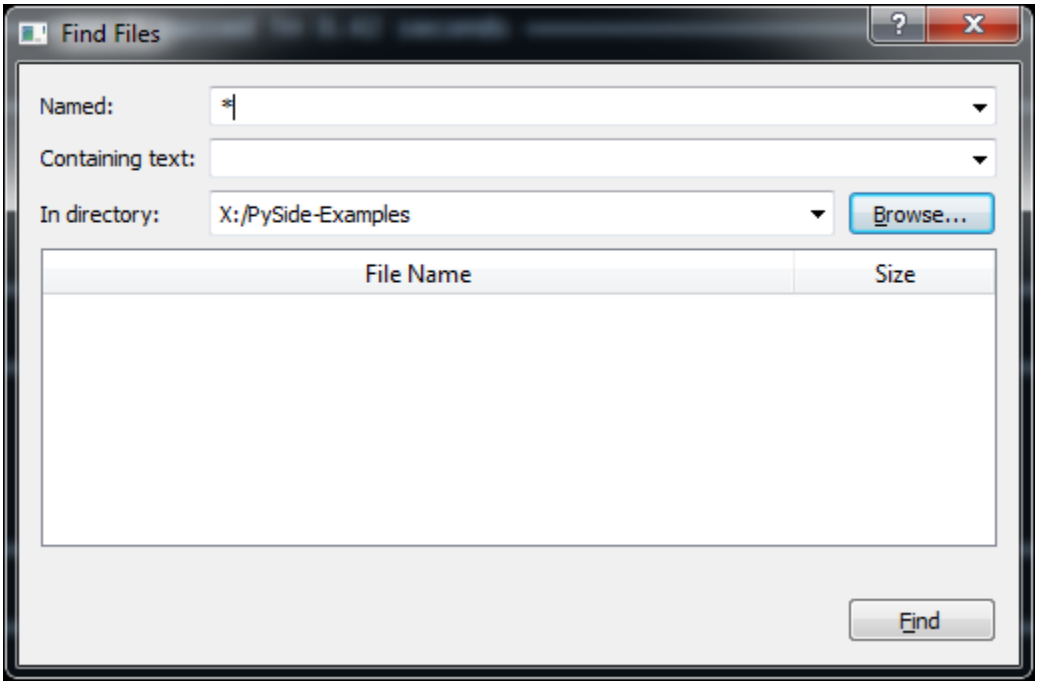
This projects follows semantic versioning.

CHAPTER 2

Tutorial

`pytest-qt` registers a new fixture named `qtbot`, which acts as *bot* in the sense that it can send keyboard and mouse events to any widgets being tested. This way, the programmer can simulate user interaction while checking if GUI controls are behaving in the expected manner.

To illustrate that, consider a widget constructed to allow the user to find files in a given directory inside an application.



It is a very simple dialog, where the user enters a standard file mask, optionally enters file text to search for and a button to browse for the desired directory. Its source code is available here,

To test this widget's basic functionality, create a test function:

```
def test_basic_search(qtbot, tmpdir):
    '''
    test to ensure basic find files functionality is working.
    '''
    tmpdir.join('video1.avi').ensure()
    tmpdir.join('video1.srt').ensure()

    tmpdir.join('video2.avi').ensure()
    tmpdir.join('video2.srt').ensure()
```

Here the first parameter indicates that we will be using a `qtbot` fixture to control our widget. The other parameter is py.test standard's tmpdir that we use to create some files that will be used during our test.

Now we create the widget to test and register it:

```
window = Window()
window.show()
qtbot.addWidget(window)
```

Tip: Registering widgets is not required, but recommended because it will ensure those widgets get properly closed after each test is done.

Now we use `qtbot` methods to simulate user interaction with the dialog:

```
window.fileComboBox.clear()
qtbot.keyClicks(window.fileComboBox, '*.avi')

window.directoryComboBox.clear()
qtbot.keyClicks(window.directoryComboBox, str(tmpdir))
```

The method `keyClicks` is used to enter text in the editable combo box, selecting the desired mask and directory.

We then simulate a user clicking the button with the `mouseClick` method:

```
qtbot.mouseClick(window.findButton, QtCore.Qt.LeftButton)
```

Once this is done, we inspect the results widget to ensure that it contains the expected files we created earlier:

```
assert window.filesTable.rowCount() == 2
assert window.filesTable.item(0, 0).text() == 'video1.avi'
assert window.filesTable.item(1, 0).text() == 'video2.avi'
```

CHAPTER 3

Qt Logging Capture

New in version 1.4.

Qt features its own logging mechanism through `qInstallMsgHandler` (`qInstallMessageHandler` on Qt5) and `qDebug`, `qWarning`, `qCritical` functions. These are used by Qt to print warning messages when internal errors occur.

`pytest-qt` automatically captures these messages and displays them when a test fails, similar to what `pytest` does for `stderr` and `stdout` and the pytest-catchlog plugin. For example:

```
from pytestqt.qt_compat import qWarning

def do_something():
    qWarning('this is a WARNING message')

def test_foo(qtlog):
    do_something()
    assert 0
```

```
$ py.test test.py -q
F
================================== FAILURES ===================================
_____________________________________ test_types ______________________________________

    def test_foo():
        do_something()
>       assert 0
E       assert 0

test.py:8: AssertionError
---------------------------- Captured Qt messages -----------------------------
QtWarningMsg: this is a WARNING message
1 failed in 0.01 seconds
```

Disabling Logging Capture

Qt logging capture can be disabled altogether by passing the `--no-qt-log` to the command line, which will fallback to the default Qt bahavior of printing emitted messages directly to `stderr`:

```
py.test test.py -q --no-qt-log
F
================================== FAILURES ===================================
_____________________________________ test_types ______________________________________

    def test_foo():
```

```
        do_something()
>       assert 0
E       assert 0

test.py:8: AssertionError
---------------------------- Captured stderr call -----------------------------
this is a WARNING message
```

`pytest-qt` also provides a `qtlog` fixture that can used to check if certain messages were emitted during a test:

```
def do_something():
    qWarning('this is a WARNING message')

def test_foo(qtlog):
    do_something()
    emitted = [(m.type, m.message.strip()) for m in qtlog.records]
    assert emitted == [(QtWarningMsg, 'this is a WARNING message')]
```

`qtlog.records` is a list of *Record* instances.

Logging can also be disabled on a block of code using the `qtlog.disabled()` context manager, or with the `pytest.mark.no_qt_log` mark:

```
def test_foo(qtlog):
    with qtlog.disabled():
        # logging is disabled within the context manager
        do_something()

@pytest.mark.no_qt_log
def test_bar():
    # logging is disabled for the entire test
    do_something()
```

Keep in mind that when logging is disabled, `qtlog.records` will always be an empty list.

Log Formatting

The output format of the messages can also be controlled by using the `--qt-log-format` command line option, which accepts a string with standard `{}` formatting which can make use of attribute interpolation of the record objects:

```
$ py.test test.py --qt-log-format="{rec.when} {rec.type_name}: {rec.message}"
```

Keep in mind that you can make any of the options above the default for your project by using pytest's standard `addopts` option in you `pytest.ini` file:

```
[pytest]
qt_log_format = {rec.when} {rec.type_name}: {rec.message}
```

3.1 Automatically failing tests when logging messages are emitted

Printing messages to `stderr` is not the best solution to notice that something might not be working as expected, specially when running in a continuous integration server where errors in logs are rarely noticed.

You can configure `pytest-qt` to automatically fail a test if it emits a message of a certain level or above using the `qt_log_level_fail` ini option:

```
[pytest]
qt_log_level_fail = CRITICAL
```

With this configuration, any test which emits a CRITICAL message or above will fail, even if no actual asserts fail within the test:

```
from pytestqt.qt_compat import qCritical

def do_something():
    qCritical('WM_PAINT failed')

def test_foo(qtlog):
    do_something()
```

```
>py.test test.py --color=no -q
F
================================== FAILURES ===================================
_____________________________________ test_foo _____________________________________
test.py:5: Failure: Qt messages with level CRITICAL or above emitted
---------------------------- Captured Qt messages -----------------------------
QtCriticalMsg: WM_PAINT failed
```

The possible values for `qt_log_level_fail` are:

- `NO`: disables test failure by log messages.
- `DEBUG`: messages emitted by `qDebug` function or above.
- `WARNING`: messages emitted by `qWarning` function or above.
- `CRITICAL`: messages emitted by `qCritical` function only.

If some failures are known to happen and considered harmless, they can be ignored by using the `qt_log_ignore` ini option, which is a list of regular expressions matched using `re.search`:

```
[pytest]
qt_log_level_fail = CRITICAL
qt_log_ignore =
    WM_DESTROY.*sent
    WM_PAINT failed
```

```
py.test test.py --color=no -q
.
1 passed in 0.01 seconds
```

Messages which do not match any of the regular expressions defined by `qt_log_ignore` make tests fail as usual:

```
def do_something():
    qCritical('WM_PAINT not handled')
    qCritical('QObject: widget destroyed in another thread')

def test_foo(qtlog):
    do_something()
```

```
py.test test.py --color=no -q
F
================================== FAILURES ===================================
_____________________________________ test_foo _____________________________________
test.py:6: Failure: Qt messages with level CRITICAL or above emitted
---------------------------- Captured Qt messages -----------------------------
QtCriticalMsg: WM_PAINT not handled  (IGNORED)
QtCriticalMsg: QObject: widget destroyed in another thread
```

You can also override `qt_log_level_fail` and `qt_log_ignore` settins from `pytest.ini` in some tests by using a mark with the same name:

```
def do_something():
    qCritical('WM_PAINT not handled')
    qCritical('QObject: widget destroyed in another thread')

@pytest.mark.qt_log_level_fail('CRITICAL')
@pytest.mark.qt_log_ignore('WM_DESTROY.*sent', 'WM_PAINT failed')
def test_foo(qtlog):
    do_something()
```

CHAPTER 4

Waiting for threads, processes, etc.

New in version 1.2.

If your program has long running computations running in other threads or processes, you can use *qtbot.waitSignal* to block a test until a signal is emitted (such as `QThread.finished`) or a timeout is reached. This makes it easy to write tests that wait until a computation running in another thread or process is completed before ensuring the results are correct:

```
def test_long_computation(qtbot):
    app = Application()

    # Watch for the app.worker.finished signal, then start the worker.
    with qtbot.waitSignal(app.worker.finished, timeout=10000) as blocker:
        blocker.connect(app.worker.failed)  # Can add other signals to blocker
        app.worker.start()
        # Test will block at this point until signal is emitted or
        # 10 seconds has elapsed

    assert blocker.signal_triggered, "process timed-out"
    assert_application_results(app)
```

raising parameter

New in version 1.4.

You can pass `raising=True` to raise a `qtbot.SignalTimeoutError` if the timeout is reached before the signal is triggered:

```
def test_long_computation(qtbot):
    ...
    with qtbot.waitSignal(app.worker.finished, raising=True) as blocker:
        app.worker.start()
    # if timeout is reached, qtbot.SignalTimeoutError will be raised at this point
    assert_application_results(app)
```

waitSignals

New in version 1.4.

If you have to wait until **all** signals in a list are triggered, use *qtbot.waitSignals*, which receives a list of signals instead of a single signal. As with *qtbot.waitSignal*, it also supports the new `raising` parameter:

```
def test_workers(qtbot):
    workers = spawn_workers()
    with qtbot.waitSignal([w.finished for w in workers], raising=True):
        for w in workers:
```

```
            w.start()

    # this will be reached after all workers emit their "finished"
    # signal or a qtbot.SignalTimeoutError will be raised
    assert_application_results(app)
```

CHAPTER 5

Exceptions in virtual methods

New in version 1.1.

It is common in Qt programming to override virtual C++ methods to customize behavior, like listening for mouse events, implement drawing routines, etc.

Fortunately, both `PyQt` and `PySide` support overriding this virtual methods naturally in your python code:

```
class MyWidget(QWidget):

    # mouseReleaseEvent
    def mouseReleaseEvent(self, ev):
        print('mouse released at: %s' % ev.pos())
```

This works fine, but if python code in Qt virtual methods raise an exception `PyQt` and `PySide` will just print the exception traceback to standard error, since this method is called deep within Qt's even loop handling and exceptions are not allowed at that point.

This might be surprising for python users which are used to exceptions being raised at the calling point: for example, the following code will just print a stack trace without raising any exception:

```
class MyWidget(QWidget):

    def mouseReleaseEvent(self, ev):
        raise RuntimeError('unexpected error')

w = MyWidget()
QTest.mouseClick(w, QtCore.Qt.LeftButton)
```

To make testing Qt code less surprising, `pytest-qt` automatically installs an exception hook which captures errors and fails tests when exceptions are raised inside virtual methods, like this:

```
E           Failed: Qt exceptions in virtual methods:
E           ________________________________________________________________________________
E             File "x:\pytest-qt\pytestqt\_tests\test_exceptions.py", line 14, in event
E               raise RuntimeError('unexpected error')
E
E           RuntimeError: unexpected error
```

Disabling the automatic exception hook

You can disable the automatic exception hook on individual tests by using a `qt_no_exception_capture` marker:

```
@pytest.mark.qt_no_exception_capture
def test_buttons(qtbot):
    ...
```

Or even disable it for your entire project in your `pytest.ini` file:

```
[pytest]
qt_no_exception_capture = 1
```

This might be desirable if you plan to install a custom exception hook.

CHAPTER 6

A note about QApplication.exit()

Some `pytest-qt` features, most notably `waitSignal` and `waitSignals`, depend on the Qt event loop being active. Calling `QApplication.exit()` from a test will cause the main event loop and auxiliary event loops to exit and all subsequent event loops to fail to start. This is a problem if some of your tests call an application functionality that calls `QApplication.exit()`.

One solution is to *monkeypatch* `QApplication.exit()` in such tests to ensure it was called by the application code but without effectively calling it.

For example:

```
def test_exit_button(qtbot, monkeypatch):
    exit_calls = []
    monkeypatch.setattr(QApplication, 'exit', lambda: exit_calls.append(1))
    button = get_app_exit_button()
    qtbot.click(button)
    assert exit_calls == [1]
```

Or using the `mock` package:

```
def test_exit_button(qtbot):
    with mock.patch.object(QApplication, 'exit'):
        button = get_app_exit_button()
        qtbot.click(button)
        assert QApplication.exit.call_count == 1
```

CHAPTER 7

Reference

7.1 QtBot

class pytestqt.plugin.**QtBot**

Instances of this class are responsible for sending events to *Qt* objects (usually widgets), simulating user input.

Important: Instances of this class should be accessed only by using a `qtbot` fixture, never instantiated directly.

Widgets

addWidget(*widget*)

Adds a widget to be tracked by this bot. This is not required, but will ensure that the widget gets closed by the end of the test, so it is highly recommended.

Parameters **widget** (*QWidget*) – Widget to keep track of.

waitForWindowShown(*widget*)

Waits until the window is shown in the screen. This is mainly useful for asynchronous systems like X11, where a window will be mapped to screen some time after being asked to show itself on the screen.

Parameters **widget** (*QWidget*) – Widget to wait on.

Note: In Qt5, the actual method called is qWaitForWindowExposed, but this name is kept for backward compatibility

---

stopForInteraction()

Stops the current test flow, letting the user interact with any visible widget.

This is mainly useful so that you can verify the current state of the program while writing tests.

Closing the windows should resume the test run, with `qtbot` attempting to restore visibility of the widgets as they were before this call.

Note: As a convenience, it is also aliased as *stop*.

Signals

waitSignal(*signal=None*, *timeout=1000*, *raising=False*)

New in version 1.2.

Stops current test until a signal is triggered.

Used to stop the control flow of a test until a signal is emitted, or a number of milliseconds, specified by `timeout`, has elapsed.

Best used as a context manager:

```
with qtbot.waitSignal(signal, timeout=1000):
    long_function_that_calls_signal()
```

Also, you can use the *SignalBlocker* directly if the context manager form is not convenient:

```
blocker = qtbot.waitSignal(signal, timeout=1000)
blocker.connect(another_signal)
long_function_that_calls_signal()
blocker.wait()
```

Any additional signal, when triggered, will make `wait()` return.

New in version 1.4: The *raising* parameter.

Parameters

- **signal** (*Signal*) – A signal to wait for. Set to `None` to just use timeout.
- **timeout** (*int*) – How many milliseconds to wait before resuming control flow.
- **raising** (*bool*) – If *QtBot.SignalTimeoutError* should be raised if a timeout occurred.

Returns `SignalBlocker` object. Call `SignalBlocker.wait()` to wait.

Note: Cannot have both `signals` and `timeout` equal `None`, or else you will block indefinitely. We throw an error if this occurs.

---

waitSignals(*signals=None*, *timeout=1000*, *raising=False*)

New in version 1.4.

Stops current test until all given signals are triggered.

Used to stop the control flow of a test until all (and only all) signals are emitted or the number of milliseconds specified by `timeout` has elapsed.

Best used as a context manager:

```
with qtbot.waitSignals([signal1, signal2], timeout=1000):
    long_function_that_calls_signals()
```

Also, you can use the *MultiSignalBlocker* directly if the context manager form is not convenient:

```
blocker = qtbot.waitSignals(signals, timeout=1000)
long_function_that_calls_signal()
blocker.wait()
```

Parameters

- **signals** (*list*) – A list of `Signal`s to wait for. Set to ``None`` to just use timeout.
- **timeout** (*int*) – How many milliseconds to wait before resuming control flow.
- **raising** (*bool*) – If *QtBot.SignalTimeoutError* should be raised if a timeout occurred.

Returns `MultiSignalBlocker` object. Call `MultiSignalBlocker.wait()` to wait.

Note: Cannot have both `signals` and `timeout` equal `None`, or else you will block indefinitely. We throw an error if this occurs.

Raw QTest API

Methods below provide very low level functions, as sending a single mouse click or a key event. Those methods are just forwarded directly to the QTest API. Consult the documentation for more information.

—

Below are methods used to simulate sending key events to widgets:

static `keyPress` (*widget*, *key*[, *modifier=Qt.NoModifier*[, *delay=-1*]])

static `keyClick` (*widget*, *key*[, *modifier=Qt.NoModifier*[, *delay=-1*]])

static `keyClicks` (*widget*, *key sequence*[, *modifier=Qt.NoModifier*[, *delay=-1*]])

static `keyEvent` (*action*, *widget*, *key*[, *modifier=Qt.NoModifier*[, *delay=-1*]])

static `keyPress` (*widget*, *key*[, *modifier=Qt.NoModifier*[, *delay=-1*]])

static `keyRelease` (*widget*, *key*[, *modifier=Qt.NoModifier*[, *delay=-1*]])

Sends one or more keyword events to a widget.

Parameters

- **`widget`** (*QWidget*) – the widget that will receive the event
- **`key`** (*str|int*) – key to send, it can be either a Qt.Key_* constant or a single character string.

Parameters

- **`modifier`** (*Qt.KeyboardModifier*) – flags OR'ed together representing other modifier keys also pressed. Possible flags are:
  - `Qt.NoModifier`: No modifier key is pressed.
  - `Qt.ShiftModifier`: A Shift key on the keyboard is pressed.
  - `Qt.ControlModifier`: A Ctrl key on the keyboard is pressed.
  - `Qt.AltModifier`: An Alt key on the keyboard is pressed.
  - `Qt.MetaModifier`: A Meta key on the keyboard is pressed.
  - `Qt.KeypadModifier`: A keypad button is pressed.
  - `Qt.GroupSwitchModifier`: X11 only. A Mode_switch key on the keyboard is pressed.
- **`delay`** (*int*) – after the event, delay the test for this miliseconds (if > 0).

static `keyToAscii` (*key*)

Auxilliary method that converts the given constant ot its equivalent ascii.

Parameters **`key`** (*Qt.Key_**) – one of the constants for keys in the Qt namespace.

Return type str

Returns the equivalent character string.

Note: this method is not available in PyQt.

—

Below are methods used to simulate sending mouse events to widgets.

static **mouseClick**(*widget*, *button*[, *stateKey=0*[, *pos=QPoint()*[, *delay=-1*]]])

static **mouseDClick**(*widget*, *button*[, *stateKey=0*[, *pos=QPoint()*[, *delay=-1*]]])

static **mouseEvent**(*action*, *widget*, *button*, *stateKey*, *pos*[, *delay=-1*])

static **mouseMove**(*widget*[, *pos=QPoint()*[, *delay=-1*]])

static **mousePress**(*widget*, *button*[, *stateKey=0*[, *pos=QPoint()*[, *delay=-1*]]])

static **mouseRelease**(*widget*, *button*[, *stateKey=0*[, *pos=QPoint()*[, *delay=-1*]]])

Sends a mouse moves and clicks to a widget.

Parameters

- **widget** (*QWidget*) – the widget that will receive the event
- **button** (*Qt.MouseButton*) – flags OR'ed together representing the button pressed. Possible flags are:
  - `Qt.NoButton`: The button state does not refer to any button (see QMouseEvent.button()).
  - `Qt.LeftButton`: The left button is pressed, or an event refers to the left button. (The left button may be the right button on left-handed mice.)
  - `Qt.RightButton`: The right button.
  - `Qt.MidButton`: The middle button.
  - `Qt.MiddleButton`: The middle button.
  - `Qt.XButton1`: The first X button.
  - `Qt.XButton2`: The second X button.
- **modifier** (*Qt.KeyboardModifier*) – flags OR'ed together representing other modifier keys also pressed. See *keyboard modifiers*.
- **position** (*QPoint*) – position of the mouse pointer.
- **delay** (*int*) – after the event, delay the test for this miliseconds (if > 0).

7.2 SignalBlocker

class `pytestqt.plugin.`**SignalBlocker**(*timeout=1000*, *raising=False*)

Returned by *`QtBot.waitSignal()`* method.

Variables

- **timeout** (*int*) – maximum time to wait for a signal to be triggered. Can be changed before *`wait()`* is called.
- **signal_triggered** (*bool*) – set to `True` if a signal (or all signals in case of `MultipleSignalBlocker`) was triggered, or `False` if timeout was reached instead. Until *`wait()`* is called, this is set to `None`.
- **raising** (*bool*) – If *`SignalTimeoutError`* should be raised if a timeout occurred.

wait()

Waits until either a connected signal is triggered or timeout is reached.

Raises ValueError if no signals are connected and timeout is None; in this case it would wait forever.

connect(*signal*)
Connects to the given signal, making *wait()* return once this signal is emitted.

More than one signal can be connected, in which case **any** one of them will make `wait()` return.

Parameters **signal** – QtCore.Signal

7.3 MultiSignalBlocker

class pytestqt.plugin.**MultiSignalBlocker**(*timeout=1000*, *raising=False*)
Returned by *QtBot.waitSignals()* method, blocks until all signals connected to it are triggered or the timeout is reached.

Variables identical to *SignalBlocker*:

- `timeout`
- `signal_triggered`
- `raising`

wait()
Waits until either a connected signal is triggered or timeout is reached.

Raises ValueError if no signals are connected and timeout is None; in this case it would wait forever.

7.4 SignalTimeoutError

class pytestqt.plugin.**SignalTimeoutError**
New in version 1.4.

The exception thrown by *QtBot.waitSignal()* if the *raising* parameter has been given and there was a timeout.

7.5 Record

class pytestqt.plugin.**Record**(*msg_type*, *message*, *ignored*, *context*)
Hold information about a message sent by one of Qt log functions.

Variables

- **message** (*str*) – message contents.
- **type** (*Qt.QtMsgType*) – enum that identifies message type
- **type_name** (*str*) – `type` as string: `"QtDebugMsg"`, `"QtWarningMsg"` or `"QtCriticalMsg"`.
- **log_type_name** (*str*) – type name similar to the logging package: `DEBUG`, `WARNING` and `CRITICAL`.
- **when** (*datetime.datetime*) – when the message was captured

- **ignored** (*bool*) – If this record matches a regex from the "qt_log_ignore" option.
- **context** – a namedtuple containing the attributes `file`, `function`, `line`. Only available in Qt5, otherwise is None.

p