pytest-qt Documentation

Bruno Oliveira

May 15, 2018

Contents

Repository GitHub

Version 2.3.2

License MIT

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 2.7 or later, including Python 3.4+.

Requires pytest version 2.7 or later.

Works with either `PyQt5`, `PyQt4`, `PySide` or `PySide2`, picking whichever is available on the system giving preference to the first one installed in this order:

- `PySide2`
- `PyQt5`
- `PySide`
- `PyQt4`

To force a particular API, set the configuration variable `qt_api` in your `pytest.ini` file to `pyqt5`, `pyside`, `pyside2`, `pyqt4` or `pyqt4v2`. `pyqt4v2` sets the `PyQt4` API to version 2.

```
[pytest]
qt_api=pyqt5
```

Alternatively, you can set the `PYTEST_QT_API` environment variable to the same values described above (the environment variable wins over the configuration if both are set).

From `pytest-qt` version 2 the behaviour of `pyqt4v2` has changed, as explained in *A note about pyqt4v2*.

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 `pytest`.

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
pip install -e .  # or python setup.py develop
pip install pyside # or pyqt4/pyqt5
```

If you also intend to build the documentation locally, you can make sure to have all the needed dependences executing:

```
pip install -e .[doc]
```

1.4 Versioning

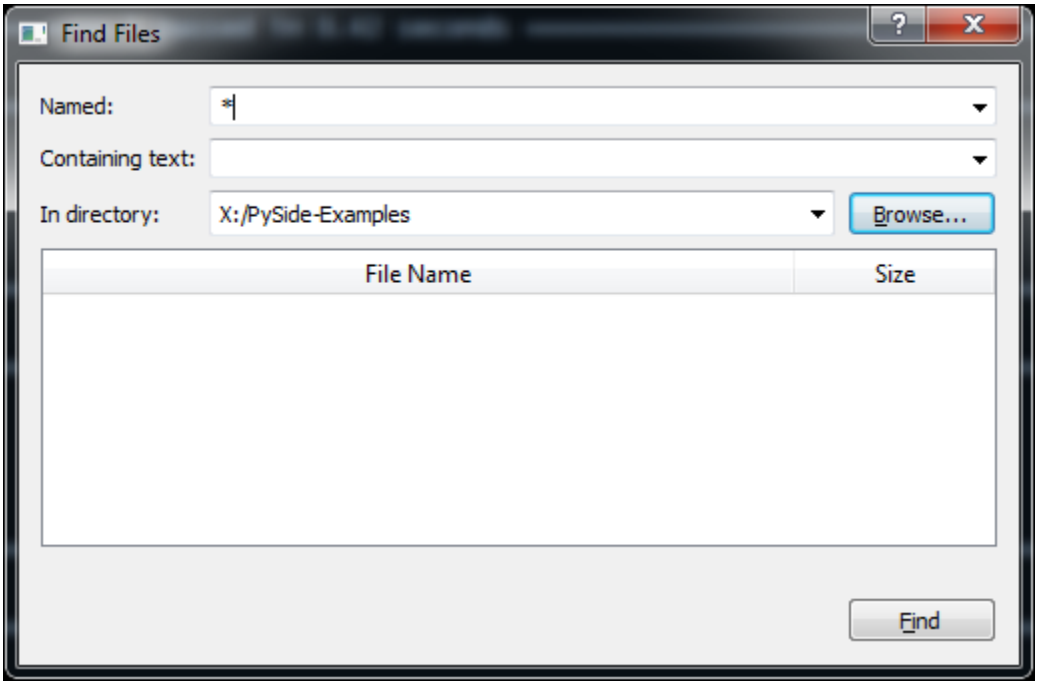
This projects follows semantic versioning.

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 pytest's standard 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 `qInstallMessageHandler` (`qInstallMsgHandler` on Qt4) 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():
    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
```

3.1 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
```

3.2 qtlog fixture

`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.

3.3 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}
```

3.4 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 the `qt_log_level_fail` setting and extend `qt_log_ignore` patterns 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()
```

If you would like to override the list of ignored patterns instead, pass `extend=False` to the `qt_log_ignore` mark:

```
@pytest.mark.qt_log_ignore('WM_DESTROY.*sent', extend=False)
def test_foo(qtlog):
    do_something()
```

CHAPTER 4

waitSignal: 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 either the "finished" or the
        # "failed" signal is emitted. If 10 seconds passed without a signal,
        # SignalTimeoutError will be raised.

    assert_application_results(app)
```

4.1 raising parameter

New in version 1.4.

Changed in version 2.0.

You can pass `raising=False` to avoid raising 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=False) as blocker:
```

(continues on next page)

(continued from previous page)

```
        app.worker.start()

    assert_application_results(app)

    # qtbot.SignalTimeoutError is not raised, but you can still manually
    # check whether the signal was triggered:
    assert blocker.signal_triggered, "process timed-out"
```

4.2 qt_wait_signal_raising ini option

New in version 1.11.

Changed in version 2.0.

The `qt_wait_signal_raising` ini option can be used to override the default value of the `raising` parameter of the `qtbot.waitSignal` and `qtbot.waitSignals` functions when omitted:

```
[pytest]
qt_wait_signal_raising = false
```

Calls which explicitly pass the `raising` parameter are not affected.

4.3 check_params_cb parameter

New in version 2.0.

If the signal has parameters you want to compare with expected values, you can pass `check_params_cb=some_callable` that compares the provided signal parameters to some expected parameters. It has to match the signature of `signal` (just like a slot function would) and return `True` if parameters match, `False` otherwise.

```
def test_status_100(status):
    """Return true if status has reached 100%."""
    return status == 100

def test_status_complete(qtbot):
    app = Application()

    # the following raises if the worker's status signal (which has an int parameter)␣
↪wasn't raised
    # with value=100 within the default timeout
    with qtbot.waitSignal(app.worker.status, raising=True, check_params_cb=test_
↪status_100) as blocker:
        app.worker.start()
```

4.4 Getting arguments of the emitted signal

New in version 1.10.

The arguments emitted with the signal are available as the `args` attribute of the blocker:

```
def test_signal(qtbot):
    ...
    with qtbot.waitSignal(app.got_cmd) as blocker:
        app.listen()
    assert blocker.args == ['test']
```

Signals without arguments will set `args` to an empty list. If the time out is reached instead, `args` will be `None`.

4.4.1 Getting all arguments of non-matching arguments

New in version 2.1.

When using the `check_params_cb` parameter, it may happen that the provided signal is received multiple times with different parameter values, which may or may not match the requirements of the callback. `all_args` then contains the list of signal parameters (as tuple) in the order they were received.

4.5 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 `raising` parameter:

```
def test_workers(qtbot):
    workers = spawn_workers()
    with qtbot.waitSignals([w.finished for w in workers]):
        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)
```

4.5.1 check_params_cbs parameter

New in version 2.0.

Corresponding to the `check_params_cb` parameter of `waitSignal` you can use the `check_params_cbs` parameter to check whether one or more of the provided signals are emitted with expected parameters. Provide a list of callables, each matching the signature of the corresponding signal in `signals` (just like a slot function would). Like for `waitSignal`, each callable has to return `True` if parameters match, `False` otherwise. Instead of a specific callable, `None` can be provided, to disable parameter checking for the corresponding signal. If the number of callbacks doesn't match the number of signals `ValueError` will be raised.

The following example shows that the `app.worker.status` signal has to be emitted with values 50 and 100, and the `app.worker.finished` signal has to be emitted too (for which no signal parameter evaluation takes place).

```
def test_status_100(status):
    """Return true if status has reached 100%."""
    return status == 100

def test_status_50(status):
    """Return true if status has reached 50%."""
```

(continues on next page)

(continued from previous page)

```
    return status == 50

def test_status_complete(qtbot):
    app = Application()

    signals = [app.worker.status, app.worker.status, app.worker.finished]
    callbacks = [test_status_50, test_status_100, None]
    with qtbot.waitSignals(signals, raising=True, check_params_cbs=callbacks) as␣
↪blocker:
        app.worker.start()
```

4.5.2 order parameter

New in version 2.0.

By default a test using `qtbot.waitSignals` completes successfully if *all* signals in `signals` are emitted, irrespective of their exact order. The `order` parameter can be set to `"strict"` to enforce strict signal order. Exemplary, this means that `blocker.signal_triggered` will be `False` if `waitSignals` expects the signals `[a, b]` but the sender emitted signals `[a, a, b]`.

Note: The tested component can still emit signals unknown to the blocker. E.g. `blocker.waitSignals([a, b], raising=True, order="strict")` won't raise if the signal-sender emits signals `[a, c, b]`, as `c` is not part of the observed signals.

A third option is to set `order="simple"` which is like "strict", but signals may be emitted in-between the provided ones, e.g. if the expected signals are `[a, b, c]` and the sender actually emits `[a, a, b, a, c]`, the test completes successfully (it would fail with `order="strict"`).

4.5.3 Getting emitted signals and arguments

New in version 2.1.

To determine which of the expected signals were emitted during a `wait()` you can use `blocker.all_signals_and_args` which contains a list of `wait_signal.SignalAndArgs` objects, indicating the signals (and their arguments) in the order they were received.

4.6 Making sure a given signal is not emitted

New in version 1.11.

If you want to ensure a signal is **not** emitted in a given block of code, use the `qtbot.assertNotEmitted` context manager:

```
def test_no_error(qtbot):
    ...
    with qtbot.assertNotEmitted(app.worker.error):
        app.worker.start()
```

CHAPTER 5

waitUntil: Waiting for arbitrary conditions

New in version 2.0.

Sometimes your tests need to wait a certain condition which does not trigger a signal, for example that a certain control gained focus or a `QListView` has been populated with all items.

For those situations you can use `qtbot.waitUntil` to wait until a certain condition has been met or a timeout is reached. This is specially important in X window systems due to their asynchronous nature, where you can't rely on the fact that the result of an action will be immediately available.

For example:

```
def test_validate(qtbot):
    window = MyWindow()
    window.edit.setText('not a number')
    # after focusing, should update status label
    window.edit.setFocus()
    assert window.status.text() == 'Please input a number'
```

The `window.edit.setFocus()` may not be processed immediately, only in a future event loop, which might lead to this test to work sometimes and fail in others (a *flaky* test).

A better approach in situations like this is to use `qtbot.waitUntil` with a callback with your assertion:

```
def test_validate(qtbot):
    window = MyWindow()
    window.edit.setText('not a number')
    # after focusing, should update status label
    window.edit.setFocus()
    def check_label():
        assert window.status.text() == 'Please input a number'
    qtbot.waitUntil(check_label)
```

`qtbot.waitUntil` will periodically call `check_label` until it no longer raises `AssertionError` or a timeout is reached. If a timeout is reached, the last assertion error re-raised and the test will fail:

```
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
    def check_label():
>       assert window.status.text() == 'Please input a number'
E       assert 'OK' == 'Please input a number'
E         - OK
E         + Please input a number
```

A second way to use `qtbot.waitUntil` is to pass a callback which returns `True` when the condition is met or `False` otherwise. It is usually terser than using a separate callback with `assert` statement, but it produces a generic message when it fails because it can't make use of `pytest`'s assertion rewriting:

```
def test_validate(qtbot):
    window = MyWindow()
    window.edit.setText('not a number')
    # after focusing, should update status label
    window.edit.setFocus()
    qtbot.waitUntil(lambda: window.edit.hasFocus())
    assert window.status.text() == 'Please input a number'
```

CHAPTER 6

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 `PyQt4` and `PySide` will just print the exception traceback to standard error, since this method is called deep within Qt's event loop handling and exceptions are not allowed at that point. In `PyQt5.5+`, exceptions in virtual methods will by default call `abort()`, which will crash the interpreter.

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           ______________________________________________________________________
 ↪_______
```

(continues on next page)

(continued from previous page)

```
E               File "x:\pytest-qt\pytestqt\_tests\test_exceptions.py", line 14, in␣
↪event
E                 raise RuntimeError('unexpected error')
E
E             RuntimeError: unexpected error
```

6.1 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.

Note: Starting with `PyQt5.5`, exceptions raised during virtual methods will actually trigger an `abort()`, crashing the Python interpreter. For this reason, disabling exception capture in `PyQt5.5+` is not recommended unless you install your own exception hook.

CHAPTER 7

Model Tester

New in version 2.0.

`pytest-qt` includes a fixture that helps testing QAbstractItemModel implementations. The implementation is copied from the C++ code as described on the Qt Wiki, and it continuously checks a model as it changes, helping to verify the state and catching many common errors the moment they show up.

Some of the conditions caught include:

- Verifying X number of rows have been inserted in the correct place after the signal `rowsAboutToBeInserted()` says X rows will be inserted.
- The parent of the first index of the first row is a `QModelIndex()`
- Calling `index()` twice in a row with the same values will return the same `QModelIndex`
- If `rowCount()` says there are X number of rows, model test will verify that is true.
- Many possible off by one bugs
- `hasChildren()` returns true if `rowCount()` is greater then zero.
- and many more...

To use it, create an instance of your model implementation, fill it with some items and call `qtmodeltester.check`:

```
def test_standard_item_model(qtmodeltester):
    model = QStandardItemModel()
    items = [QStandardItem(str(i)) for i in range(4)]
    model.setItem(0, 0, items[0])
    model.setItem(0, 1, items[1])
    model.setItem(1, 0, items[2])
    model.setItem(1, 1, items[3])
    qtmodeltester.check(model)
```

If the tester finds a problem the test will fail with an assert pinpointing the issue.

The following attribute may influence the outcome of the check depending on your model implementation:

- `data_display_may_return_none` (default: `False`): While you can technically return `None` (or an invalid `QVariant`) from `data()` for `QtCore.Qt.DisplayRole`, this usually is a sign of a bug in your implementation. Set this variable to `True` if this really is OK in your model.

The source code was ported from modeltest.cpp by Florian Bruhin, many thanks!

CHAPTER 8

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 9

A note about pyqt4v2

Starting with pytest-qt version 2.0, PyQt or PySide are lazily loaded when first needed instead of at pytest startup. This usually means pytest-qt will import PyQt or PySide when the tests actually start running, well after conftest.py files and other plugins have been imported. This can lead to some unexpected behaviour if pyqt4v2 is set.

If the conftest.py files, either directly or indirectly, set the API version to 2 and import PyQt4, one of the following cases can happen:

- if all the available types are set to version 2, then using pyqt4 or pyqt4v2 is equivalent
- if only some of the types set to version 2, using pyqt4v2 will make pytest to fail with an error similar to:

```
INTERNALERROR> sip.setapi("QDate", 2)
INTERNALERROR> ValueError: API 'QDate' has already been set to version 1
```

  If this is the case, use pyqt4.

If the API is set in the test functions or in the code imported by them, then the new behaviour is indistinguishable from the old one and pyqt4v2 must be used to avoid errors if version 2 is used.

CHAPTER 10

A note about Modal Dialogs

10.1 Simple Dialogs

For QMessageBox.question one approach is to mock the function using the monkeypatch fixture:

```
def test_Qt(qtbot, monkeypatch):
    simple = Simple()
    qtbot.addWidget(simple)

    monkeypatch.setattr(QMessageBox, 'question', lambda *args: QMessageBox.Yes)
    simple.query()
    assert simple.answer
```

10.2 Custom Dialogs

Suppose you have a custom dialog that asks the user for their name and age, and a form that uses it. One approach is to add a convenience function that also has the nice benefit of making testing easier, like this:

```
class AskNameAndAgeDialog(QDialog):
...
    @classmethod
    def ask(cls, text, parent):
        dialog = cls(parent)
        dialog.text.setText(text)
        if dialog.exec_() == QDialog.Accepted:
            return dialog.getName(), dialog.getAge()
        else:
            return None, None
```

This allows clients of the dialog to use it this way:

```
name, age = AskNameAndAgeDialog.ask("Enter name and age because of bananas:", parent)
if name is not None:
    # use name and age for bananas
```

And now it is also easy to mock `AskNameAndAgeDialog.ask` when testing the form:

```
def test_form_registration(qtbot, monkeypatch):
    form = RegistrationForm()

    monkeypatch.setattr(AskNameAndAgeDialog, 'ask', classmethod(lambda *args: ('Jonh',
↪ 30)))
    qtbot.click(form.enter_info())
    # calls AskNameAndAgeDialog.ask
    # test that the rest of the form correctly behaves as if
    # user entered "Jonh" and 30 as name and age
```

CHAPTER 11

Troubleshooting

11.1 tox: `InvocationError` without further information

It might happen that your `tox` run finishes abruptly without any useful information, e.g.:

```
ERROR: InvocationError:
'/path/to/project/.tox/py36/bin/python setup.py test --addopts --doctest-modules'
___ summary _____
ERROR: py36: commands failed
```

`pytest-qt` needs a `DISPLAY` to run, otherwise `Qt` calls `abort()` and the process crashes immediately.

One solution is to use the pytest-xvfb plugin which takes care of the grifty details automatically, starting up a virtual framebuffer service, initializing variables, etc. This is the recommended solution if you are running in CI servers without a GUI, for example in Travis or CircleCI.

Alternatively, `tox` users may edit `tox.ini` to allow the relevant variables to be passed to the underlying `pytest` invocation:

```
[testenv]
passenv = DISPLAY XAUTHORITY
```

Note that this solution will only work in boxes with a GUI.

More details can be found in issue #170.

CHAPTER 12

Reference

12.1 QtBot

class pytestqt.qtbot.**QtBot**(*request*)

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.

Note: This method is also available as `add_widget` (pep-8 alias)

---

captureExceptions(***kwds*)

New in version 2.1.

Context manager that captures Qt virtual method exceptions that happen in block inside context.

```
with qtbot.capture_exceptions() as exceptions:
    qtbot.click(button)

# exception is a list of sys.exc_info tuples
assert len(exceptions) == 1
```

Note: This method is also available as `capture_exceptions` (pep-8 alias)

waitActive(*widget*, *timeout=1000*)

Context manager that waits for `timeout` milliseconds or until the window is active. If window is not exposed within `timeout` milliseconds, raise `TimeoutError`.

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.

```
with qtbot.waitActive(widget, timeout=500):
    show_action()
```

Parameters

- **widget** (*QWidget*) – Widget to wait for.
- **timeout** (*int | None*) – How many milliseconds to wait for.

Note: This function is only available in PyQt5, raising a `RuntimeError` if called from `PyQt4` or `PySide`.

Note: This method is also available as `wait_active` (pep-8 alias)

waitExposed(*widget*, *timeout=1000*)

Context manager that waits for `timeout` milliseconds or until the window is exposed. If the window is not exposed within `timeout` milliseconds, raise `TimeoutError`.

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.

```
with qtbot.waitExposed(splash, timeout=500):
    startup()
```

Parameters

- **widget** (*QWidget*) – Widget to wait for.
- **timeout** (*int | None*) – How many milliseconds to wait for.

Note: This function is only available in PyQt5, raising a `RuntimeError` if called from `PyQt4` or `PySide`.

Note: This method is also available as `wait_exposed` (pep-8 alias)

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 `PyQt5` this function is considered deprecated in favor of *`waitExposed()`*.

Note: This method is also available as `wait_for_window_shown` (pep-8 alias)

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*.

wait(*ms*)
New in version 1.9.

Waits for `ms` milliseconds.

While waiting, events will be processed and your test will stay responsive to user interface events or network communication.

Signals and Events

waitSignal(*signal=None*, *timeout=1000*, *raising=None*, *check_params_cb=None*)
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.

New in version 2.0: The *check_params_cb* parameter.

Parameters

- **signal** (*Signal*) – A signal to wait for, or a tuple `(signal, signal_name_as_str)` to improve the error message that is part of `TimeoutError`. Set to `None` to just use timeout.
- **timeout** (*int*) – How many milliseconds to wait before resuming control flow.

- **raising** (*bool*) – If `QtBot.TimeoutError` should be raised if a timeout occurred. This defaults to `True` unless `qt_wait_signal_raising = false` is set in the config.
- **check_params_cb** (*Callable*) – Optional `callable` that compares the provided signal parameters to some expected parameters. It has to match the signature of `signal` (just like a slot function would) and return `True` if parameters match, `False` otherwise.

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.

Note: This method is also available as `wait_signal` (pep-8 alias)

---

waitSignals(*signals=None*, *timeout=1000*, *raising=None*, *check_params_cbs=None*, *order='none'*)

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` objects to wait for. Alternatively: a list of `(Signal, str)` tuples of the form `(signal, signal_name_as_str)` to improve the error message that is part of `TimeoutError`. Set to `None` to just use timeout.
- **timeout** (*int*) – How many milliseconds to wait before resuming control flow.
- **raising** (*bool*) – If `QtBot.TimeoutError` should be raised if a timeout occurred. This defaults to `True` unless `qt_wait_signal_raising = false` is set in the config.
- **check_params_cbs** (*list*) – optional list of callables that compare the provided signal parameters to some expected parameters. Each callable has to match the signature of the corresponding signal in `signals` (just like a slot function would) and return `True` if parameters match, `False` otherwise. Instead of a specific callable, `None` can be provided, to disable parameter checking for the corresponding signal. If the number of callbacks doesn't match the number of signals `ValueError` will be raised.
- **order** (*str*) – Determines the order in which to expect signals:
  - `"none"`: no order is enforced

- `"strict"`: signals have to be emitted strictly in the provided order (e.g. fails when expecting signals [a, b] and [a, a, b] is emitted)
- `"simple"`: like "strict", but signals may be emitted in-between the provided ones, e.g. expected `signals == [a, b, c]` and actually emitted `signals = [a, a, b, a, c]` works (would fail with `"strict"`).

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.

Note: This method is also available as `wait_signals` (pep-8 alias)

---

assertNotEmitted(***kwds*)

New in version 1.11.

Make sure the given `signal` doesn't get emitted.

This is intended to be used as a context manager.

Note: This method is also available as `assert_not_emitted` (pep-8 alias)

---

waitUntil(*callback*, *timeout=1000*)

New in version 2.0.

Wait in a busy loop, calling the given callback periodically until timeout is reached.

`callback()` should raise `AssertionError` to indicate that the desired condition has not yet been reached, or just return `None` when it does. Useful to `assert` until some condition is satisfied:

```
def view_updated():
    assert view_model.count() > 10
qtbot.waitUntil(view_updated)
```

Another possibility is for `callback()` to return `True` when the desired condition is met, `False` otherwise. Useful specially with `lambda` for terser code, but keep in mind that the error message in those cases is usually not very useful because it is not using an `assert` expression.

```
qtbot.waitUntil(lambda: view_model.count() > 10)
```

Note that this usage only accepts returning actual `True` and `False` values, so returning an empty list to express "falseness" raises a `ValueError`.

Parameters

- **callback** – callable that will be called periodically.
- **timeout** – timeout value in ms.

Raises **ValueError** – if the return value from the callback is anything other than `None`, `True` or `False`.

Note: This method is also available as `wait_until` (pep-8 alias)

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 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).

12.2 TimeoutError

class pytestqt.qtbot.**TimeoutError**

New in version 2.1.

Exception thrown by *pytestqt.qtbot.QtBot* methods.

Note: In versions prior to `2.1`, this exception was called `SignalTimeoutError`. An alias is kept for backward compatibility.

12.3 SignalBlocker

class pytestqt.wait_signal.**SignalBlocker**(*timeout=1000*, *raising=True*, *check_params_cb=None*)

Returned by *pytestqt.qtbot.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 `TimeoutError` should be raised if a timeout occurred.

  Note: contrary to the parameter of same name in *pytestqt.qtbot.QtBot.waitSignal()*, this parameter does not consider the *qt_wait_signal_raising ini option*.

- **args** (*list*) – The arguments which were emitted by the signal, or None if the signal wasn't emitted at all.

New in version 1.10: The *args* attribute.

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 or tuple (QtCore.Signal, str)

12.4 MultiSignalBlocker

class pytestqt.wait_signal.**MultiSignalBlocker**(*timeout=1000*, *raising=True*, *check_params_cbs=None*, *order='none'*)
Returned by *pytestqt.qtbot.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.

12.5 SignalEmittedError

class pytestqt.wait_signal.**SignalEmittedError**
New in version 1.11.

The exception thrown by *pytestqt.qtbot.QtBot.assertNotEmitted()* if a signal was emitted unexpectedly.

12.6 Record

class pytestqt.logging.**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.

12.7 qapp fixture

pytestqt.plugin.**qapp**(*qapp_args*)

Fixture that instantiates the QApplication instance that will be used by the tests.

You can use the `qapp` fixture in tests which require a `QApplication` to run, but where you don't need full `qtbot` functionality.

pytestqt.plugin.**qapp_args**()

Fixture that provides QApplication arguments to use.

You can override this fixture to pass different arguments to `QApplication`:

```
@pytest.fixture(scope='session')
def qapp_args():
    return ['--arg']
```

CHAPTER 13

Changelog

13.1 2.3.2

- Fix `QStringListModel` import when using `PySide2` (#209). Thanks @rth for the PR.

13.2 2.3.1

- `PYTEST_QT_API` environment variable correctly wins over `qt_api` ini variable if both are set at the same time (#196). Thanks @mochick for the PR.

13.3 2.3.0

- New `qapp_args` fixture which can be used to pass custom arguments to `QApplication`. Thanks @The-Compiler for the PR.

13.4 2.2.1

- `modeltester` now accepts `QBrush` for `BackgroundColorRole` and `TextColorRole` (#189). Thanks @p0las for the PR.

13.5 2.2.0

- `pytest-qt` now supports PySide2 thanks to @rth!

13.6 2.1.2

- Fix issue where `pytestqt` was hiding the information when there's an exception raised from another exception on Python 3.

13.7 2.1.1

- Fixed tests on Python 3.6.

13.8 2.1

- `waitSignal` and `waitSignals` now provide much more detailed messages when expected signals are not emitted. Many thanks to @MShekow for the PR (#153).
- `qtbot` fixture now can capture Qt virtual method exceptions in a block using `captureExceptions` (#154). Thanks to @fogo for the PR.
- New qtbot.waitActive and qtbot.waitExposed methods for PyQt5. Thanks @The-Compiler for the request (#158).
- `SignalTimeoutError` has been renamed to `TimeoutError`. `SignalTimeoutError` is kept as a backward compatibility alias.

13.9 2.0

13.9.1 Breaking Changes

With `pytest-qt` 2.0, we changed some defaults to values we think are much better, however this required some backwards-incompatible changes:

- `pytest-qt` now defaults to using `PyQt5` if `PYTEST_QT_API` is not set. Before, it preferred `PySide` which is using the discontinued Qt4.
- Python 3 versions prior to 3.4 are no longer supported.
- The `@pytest.mark.qt_log_ignore` mark now defaults to `extend=True`, i.e. extends the patterns defined in the config file rather than overriding them. You can pass `extend=False` to get the old behaviour of overriding the patterns.
- `qtbot.waitSignal` now defaults to `raising=True` and raises an exception on timeouts. You can set `qt_wait_signal_raising = false` in your config to get back the old behaviour.
- `PYTEST_QT_FORCE_PYQT` environment variable is no longer supported. Set `PYTEST_QT_API` to the appropriate value instead or the new `qt_api` configuration option in your `pytest.ini` file.

13.9.2 New Features

- From this version onward, `pytest-qt` is licensed under the MIT license (#134).
- New `qtmodeltester` fixture to test `QAbstractItemModel` subclasses. Thanks @The-Compiler for the initiative and port of the original C++ code for ModelTester (#63).

- New `qtbot.waitUntil` method, which continuously calls a callback until a condition is met or a timeout is reached. Useful for testing asynchronous features (like in X window environments for example).
- `waitSignal` and `waitSignals` can receive an optional callback (or list of callbacks) that can evaluate if the arguments of emitted signals should resume execution or not. Additionally `waitSignals` has a new `order` parameter that allows to expect signals and their arguments in a strict, semi-strict or no specific order. Thanks @MShekow for the PR (#141).
- Now which Qt binding `pytest-qt` will use can be configured by the `qt_api` config option. Thanks @The-Compiler for the request (#129).
- While `pytestqt.qt_compat` is an internal module and shouldn't be imported directly, it is known that some test suites did import it. This module now uses a lazy-load mechanism to load Qt classes and objects, so the old symbols (`QtCore`, `QApplication`, etc.) are no longer available from it.

13.9.3 Other Changes

- Exceptions caught by `pytest-qt` in `sys.excepthook` are now also printed to `stderr`, making debugging them easier from within an IDE. Thanks @fabioz for the PR (126)!

13.10 1.11.0

Note: The default value for `raising` is planned to change to `True` starting in pytest-qt version `1.12`. Users wishing to preserve the current behavior (`raising` is `False` by default) should make use of the new `qt_wait_signal_raising` ini option below.

- New `qt_wait_signal_raising` ini option can be used to override the default value of the `raising` parameter of the `qtbot.waitSignal` and `qtbot.waitSignals` functions when omitted:

```
[pytest]
qt_wait_signal_raising = true
```

  Calls which explicitly pass the `raising` parameter are not affected. Thanks @The-Compiler for idea and initial work on a PR (120).

- `qtbot` now has a new `assertNotEmitted` context manager which can be used to ensure the given signal is not emitted (92). Thanks @The-Compiler for the PR!

13.11 1.10.0

- `SignalBlocker` now has a `args` attribute with the arguments of the signal that triggered it, or `None` on a time out (115). Thanks @billyshambrook for the request and @The-Compiler for the PR.
- `MultiSignalBlocker` is now properly disconnects from signals upon exit.

13.12 1.9.0

- Exception capturing now happens as early/late as possible in order to catch all possible exceptions (including fixtures)(105). Thanks @The-Compiler for the request.

- Widgets registered by `qtbot.addWidget` are now closed before all other fixtures are tear down (106). Thanks @The-Compiler for request.
- `qtbot` now has a new `wait` method which does a blocking wait while the event loop continues to run, similar to `QTest::qWait`. Thanks @The-Compiler for the PR (closes 107)!
- raise `RuntimeError` instead of `ImportError` when failing to import any Qt binding: raising the latter causes *pluggy* in *pytest-2.8* to generate a subtle warning instead of a full blown error. Thanks @Sheeo for bringing this problem to attention (closes 109).

13.13 1.8.0

- `pytest.mark.qt_log_ignore` now supports an `extend` parameter that will extend the list of regexes used to ignore Qt messages (defaults to False). Thanks @The-Compiler for the PR (99).
- Fixed internal error when interacting with other plugins that raise an error, hiding the original exception (98). Thanks @The-Compiler for the PR!
- Now `pytest-qt` is properly tested with PyQt5 on Travis-CI. Many thanks to @The-Compiler for the PR!

13.14 1.7.0

- `PYTEST_QT_API` can now be set to `pyqt4v2` in order to use version 2 of the PyQt4 API. Thanks @montefra for the PR (93)!

13.15 1.6.0

- Reduced verbosity when exceptions are captured in virtual methods (77, thanks @The-Compiler).
- `pytestqt.plugin` has been split in several files (74) and tests have been moved out of the `pytestqt` package. This should not affect users, but it is worth mentioning nonetheless.
- `QApplication.processEvents()` is now called before and after other fixtures and teardown hooks, to better try to avoid non-processed events from leaking from one test to the next. (67, thanks @The-Compiler).
- Show Qt/PyQt/PySide versions in pytest header (68, thanks @The-Compiler!).
- Disconnect SignalBlocker functions after its loop exits to ensure second emissions that call the internal functions on the now-garbage-collected SignalBlocker instance (#69, thanks @The-Compiler for the PR).

13.16 1.5.1

- Exceptions are now captured also during test tear down, as delayed events will get processed then and might raise exceptions in virtual methods; this is specially problematic in `PyQt5.5`, which changed the behavior to call `abort` by default, which will crash the interpreter. (65, thanks @The-Compiler).

13.17 1.5.0

- Fixed log line number in messages, and provide better contextual information in Qt5 (55, thanks @The-Compiler);

- Fixed issue where exceptions inside a `waitSignals` or `waitSignal` with-statement block would be swallowed and a `SignalTimeoutError` would be raised instead. (59, thanks @The-Compiler for bringing up the issue and providing a test case);
- Fixed issue where the first usage of `qapp` fixture would return `None`. Thanks to @gqmelo for noticing and providing a PR;
- New `qtlog` now sports a context manager method, `disabled` (58). Thanks @The-Compiler for the idea and testing;

13.18 1.4.0

- Messages sent by `qDebug`, `qWarning`, `qCritical` are captured and displayed when tests fail, similar to pytest-catchlog. Also, tests can be configured to automatically fail if an unexpected message is generated.
- New method `waitSignals`: will block untill **all** signals given are triggered (thanks @The-Compiler for idea and complete PR).
- New parameter `raising` to `waitSignals` and `waitSignals`: when `True` will raise a `qtbot.SignalTimeoutError` exception when timeout is reached (defaults to `False`). (thanks again to @The-Compiler for idea and complete PR).
- `pytest-qt` now requires `pytest` version >= 2.7.

13.18.1 Internal changes to improve memory management

- `QApplication.exit()` is no longer called at the end of the test session and the `QApplication` instance is not garbage collected anymore;
- `QtBot` no longer receives a QApplication as a parameter in the constructor, always referencing `QApplication.instance()` now; this avoids keeping an extra reference in the `qtbot` instances.
- `deleteLater` is called on widgets added in `QtBot.addWidget` at the end of each test;
- `QApplication.processEvents()` is called at the end of each test to make sure widgets are cleaned up;

13.19 1.3.0

- pytest-qt now supports PyQt5!

  Which Qt api will be used is still detected automatically, but you can choose one using the `PYTEST_QT_API` environment variable (the old `PYTEST_QT_FORCE_PYQT` is still supported for backward compatibility).

  Many thanks to @jdreaver for helping to test this release!

13.20 1.2.3

- Now the module ``qt_compat`` no longer sets `QString` and `QVariant` APIs to `2` for PyQt, making it compatible for those still using version `1` of the API.

13.21 1.2.2

- Now it is possible to disable automatic exception capture by using markers or a `pytest.ini` option. Consult the documentation for more information. (26, thanks @datalyze-solutions for bringing this up).
- `QApplication` instance is created only if it wasn't created yet (21, thanks @fabioz!)
- `addWidget` now keeps a weak reference its widgets (20, thanks @fabioz)

13.22 1.2.1

- Fixed 16: a signal emitted immediately inside a `waitSignal` block now works as expected (thanks @baudren).

13.23 1.2.0

This version include the new `waitSignal` function, which makes it easy to write tests for long running computations that happen in other threads or processes:

```
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 wait here until either signal is emitted, or 10 seconds has␣
↪elapsed

    assert blocker.signal_triggered  # Assuming the work took less than 10 seconds
    assert_application_results(app)
```

Many thanks to @jdreaver for discussion and complete PR! (12, 13)

13.24 1.1.1

- Added `stop` as an alias for `stopForInteraction` (10, thanks @itghisi)
- Now exceptions raised in virtual methods make tests fail, instead of silently passing (11). If an exception is raised, the test will fail and it exceptions that happened inside virtual calls will be printed as such:

```
E           Failed: Qt exceptions in virtual methods:
E           ______________________________________________________________________
↪_____________
E             File "x:\pytest-qt\pytestqt\_tests\test_exceptions.py", line 14, in␣
↪event
E               raise ValueError('mistakes were made')
E
E           ValueError: mistakes were made
E           ______________________________________________________________________
↪_____________
```

(continues on next page)

(continued from previous page)

```
E               File "x:\pytest-qt\pytestqt\_tests\test_exceptions.py", line 14, in␣
↪event
E                 raise ValueError('mistakes were made')
E
E             ValueError: mistakes were made
E             ___________________________________________________________________␣
↪___________
```

Thanks to @jdreaver for request and sample code!

- Fixed documentation for `QtBot`: it was not being rendered in the docs due to an import error.

13.25 1.1.0

Python 3 support.

13.26 1.0.2

Minor documentation fixes.

13.27 1.0.1

Small bug fix release.

13.28 1.0.0

First working version.

Python Module Index

p

Index