pytest-qt Documentation

Release 1.3.0

Bruno Oliveira

April 03, 2015

Repository GitHub

Version 1.3.0

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!'
```

CHAPTER 2

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`.

CHAPTER 3

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`.

3.1 Development

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

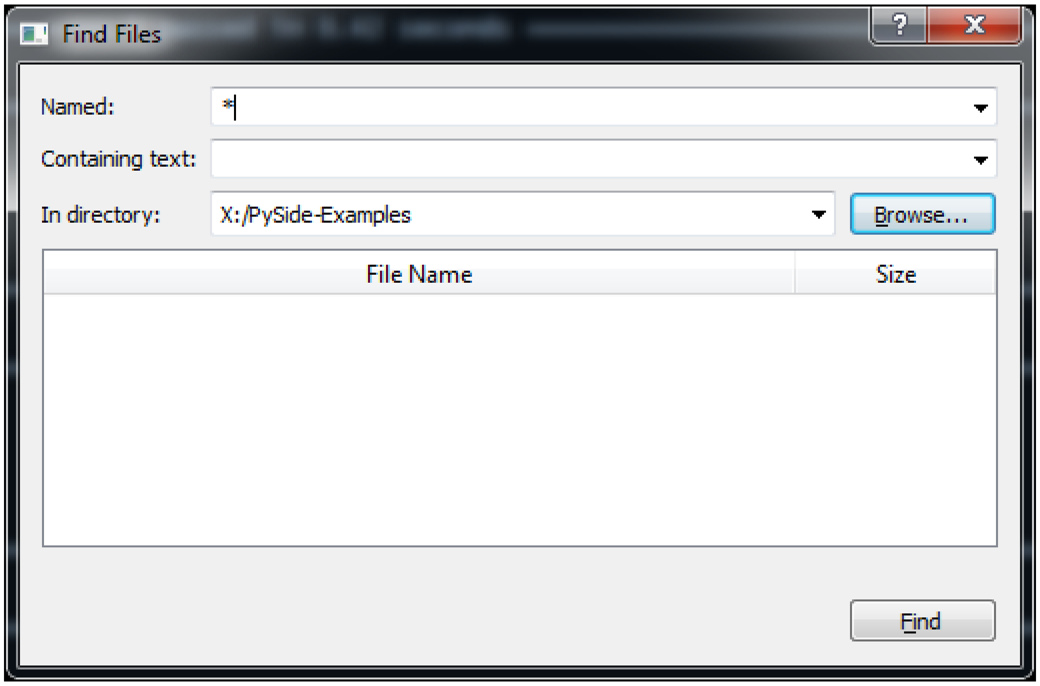
```
git clone https://github.com/pytest-dev/pytest-qt.git
cd pytest-qt
python setup.py develop
pip install pyside
```

CHAPTER 4

Quick 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 5

Waiting for threads, processes, etc.

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:

```python
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)
```

Exceptions in virtual methods

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
```

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.

CHAPTER 7

QtBot

class `pytestqt.plugin.`**`QtBot`**(*app*)

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*)

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(other_signal)
long_function_that_calls_signal()
blocker.wait()
```

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.

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.

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).

7.1 SignalBlocker

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

Returned by `QtBot.waitSignal()` method.

`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.

Parameters **signal** – QtCore.Signal

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 was triggered, or `False` if timeout was reached instead. Until `wait()` is called, this is set to `None`.

CHAPTER 8

Versioning

This projects follows semantic versioning.

Python Module Index

Index