PyOtherSide Documentation

Release 1.5.1

Thomas Perl

Apr 13, 2017

Contents

PyOtherSide is a Qt 5 QML Plugin that provides access to a Python 3 interpreter from QML. It was designed with mobile devices in mind, where high-framerate touch interfaces are common, and where the user usually interfaces only with one application at a time via a touchscreen. As such, it is important to never block the UI thread, so that the user can always continue to use the interface, even when the backend is processing, downloading or calculating something in the background.

At its core, PyOtherSide is basically a simple layer that converts Qt (QML) objects to Python objects and vice versa, with focus on asynchronous events and continuation-passing style function calls.

While legacy versions of PyOtherSide worked with Qt 4.x and Python 2.x, its focus now lies on Python 3.x and Qt 5. Python 3 has been out for several years, and offers some nice language features and clean-ups, while Qt 5 supports most mobile platforms well, and has an improved QML engine and a faster renderer (Qt Scene Graph) compared to Qt 4.

CHAPTER 1

QML API

This section describes the QML API exposed by the *PyOtherSide* QML Plugin.

Import Versions

The current QML API version of PyOtherSide is 1.5. When new features are introduced, or behavior is changed, the API version will be bumped and documented here.

io.thp.pyotherside 1.0

- Initial API release.

io.thp.pyotherside 1.2

- *importModule()* now behaves like the `import` statement in Python for names with dots. This means that `importModule('x.y.z', ...)` now works like `import x.y.z` in Python.
- If a JavaScript exception occurs in the callback passed to *importModule()* or *call()*, the signal *error()* is emitted with the exception information (filename, line, message) as `traceback`.

io.thp.pyotherside 1.3

- *addImportPath()* now also accepts `qrc:/` URLs. This is useful if your Python files are embedded as Qt Resources, relative to your QML files (use `Qt.resolvedUrl()` from the QML file).

io.thp.pyotherside 1.4

- Added *getattr()*

- *call()* and *call_sync()* now accept a Python callable object for the first parameter (previously, only strings were supported)
- If *error()* doesn't have a handler defined, error messages will be printed to the console as warnings

io.thp.pyotherside 1.5

- Added PyGLArea and PyFBO for OpenGL rendering, see *OpenGL rendering in Python*
- Added *importNames()* and *importNames_sync()* to mirror Python's from foo import bar, baz import mechanism

QML Python Element

The Python element exposes a Python interpreter in a QML file. In PyOtherSide 1.0, if multiple Python elements are instantiated, they will share the same underlying Python interpreter, so Python module-global state will be shared between all Python elements.

To use the Python element in a QML file, you have to import the plugin using:

```
import io.thp.pyotherside 1.5
```

Signals

received(*var data*)
Default event handler for *pyotherside.send()* if no other event handler was set.

error(*string traceback*)
Error handler for errors from Python.

Changed in version 1.4.0: If the error signal is not connected, PyOtherSide will print the error as QWarning on the console (previously, error messages were only shown if the signal was connected and printed there). To avoid printing the error, just define a no-op handler.

Methods

To configure event handlers for events from Python, you can use the *setHandler()* method:

setHandler(*string event*, *callable callback*)
Set the handler for events sent with *pyotherside.send()*.

Importing modules is then done by optionally adding an import path and then importing the module asynchronously:

addImportPath(*string path*)
Add a path to Python's sys.path.

Changed in version 1.1.0: *addImportPath()* will automatically strip a leading file:// from the path, so you can use Qt.resolvedUrl() without having to manually strip the leading file:// in QML.

Changed in version 1.3.0: Starting with QML API version 1.3 (import io.thp.pyotherside 1.3), *addImportPath()* now also accepts qrc:/ URLs. The first time a qrc:/ path is added, a new import handler will be installed, which will enable Python to transparently import modules from it.

importModule(*string name*, *function callback(success) {}*)
Import a Python module.

Changed in version 1.2.0: Previously, this function didn't work correctly for importing modules with dots in their name. Starting with the API version 1.2 (`import io.thp.pyotherside 1.2`), this behavior is now fixed, and `importModule('x.y.z', ...)` behaves like `import x.y.z`.

Changed in version 1.2.0: If a JavaScript exception occurs in the callback, the *error()* signal is emitted with `traceback` containing the exception info (QML API version 1.2 and newer).

importNames (*string module*, *array object_names*, *function callback(success) {}*)
Import a list of names from a given modules, like Python's `from foo import bar, baz` syntax – the equivalent call would be `importNames('module', ['bar', 'baz'], ...);`

New in version 1.5.0.

Once modules are imported, Python function can be called on the imported modules using:

call (*var func*, *args=[]*, *function callback(result) {}*)
Call the Python function `func` with `args` asynchronously. If `args` is omitted, `func` will be called without arguments. If `callback` is a callable, it will be called with the Python function result as single argument when the call has succeeded.

Changed in version 1.2.0: If a JavaScript exception occurs in the callback, the *error()* signal is emitted with `traceback` containing the exception info (QML API version 1.2 and newer).

Changed in version 1.4.0: `func` can also be a Python callable object, not just a string.

Attributes on Python objects can be accessed using *getattr()*:

getattr (*obj*, *string attr*) → var
Get the attribute `attr` of the Python object `obj`.

New in version 1.4.0.

For some of these methods, there also exist synchronous variants, but it is highly recommended to use the asynchronous variants instead to avoid blocking the QML UI thread:

evaluate (*string expr*) → var
Evaluate a Python expression synchronously.

importModule_sync (*string name*) → bool
Import a Python module. Returns `true` on success, `false` otherwise.

importNames_sync (*string module*, *array names*) → bool
Import names from a Python modules. Returns `true` on success, `false` otherwise.

call_sync (*var func*, *var args=[]*) → var
Call a Python function. Returns the return value of the Python function.

Changed in version 1.4.0: `func` can also be a Python callable object, not just a string.

The following functions allow access to the version of the running PyOtherSide plugin and Python interpreter.

pluginVersion () → string
Get the version of the PyOtherSide plugin that is currently used.

Note: This is not necessarily the same as the QML API version currently in use. The QML API version is decided by the QML import statement, so even if *pluginVersion()* returns 1.2.0, if the plugin has been imported as `import io.thp.pyotherside 1.0`, the API version used would be 1.0.

New in version 1.1.0.

pythonVersion () → string
Get the version of the Python interpreter that is currently used.

New in version 1.1.0.

Changed in version 1.5.0: Previously, *pythonVersion()* returned the compile-time version of Python against which PyOtherSide was built. Starting with version 1.5.0, the run-time version of Python is returned (e.g. PyOtherSide compiled against Python 3.4.0 and running with Python 3.4.1 returned "3.4.0" before, but returns "3.4.1" in PyOtherSide after and including 1.5.0).

QML `PyGLArea` Element

New in version 1.5.0.

The PyGLArea allows rendering arbitrary OpenGL content from Python into the QML scene.

Properties

`PyObject renderer`
: Python object that implements the IRenderer interface, see *OpenGL rendering in Python* for details.

`bool before`
: `true` to render before (= below) the rest of the QML scene, `false` to render after (= above) the rest of the QML scene. Default: `true`

QML `PyFBO` Element

New in version 1.5.0.

The PyFBO allows offscreen rendering of arbitrary OpenGL content from Python into the QML scene.

Properties

`PyObject renderer`
: Python object that implements the IRenderer interface, see *OpenGL rendering in Python* for details

CHAPTER 2

Python API

PyOtherSide uses a normal Python 3.x interpreter for running your Python code.

The `pyotherside` module

When a module is imported in PyOtherSide, it will have access to a special module called `pyotherside` in addition to all Python Standard Library modules and Python modules in `sys.path`:

```
import pyotherside
```

The module can be used to send events asynchronously (even from different threads) to the QML layer, register a callback for doing clean-ups at application exit and integrate with other QML-specific features of PyOtherSide.

Methods

`pyotherside.`**`send`**(*event*, **args*)
: Send an asynchronous event with name `event` with optional arguments `args` to QML.

`pyotherside.`**`atexit`**(*callback*)
: Register a `callback` to be called when the application is closing.

`pyotherside.`**`set_image_provider`**(*provider*)
: Set the QML *image provider* (`image://python/`).

New in version 1.1.0.

`pyotherside.`**`qrc_is_file`**(*filename*)
: Check if `filename` is an existing file in the Qt Resource System.

  Returns `True` if `filename` is a file, `False` otherwise.

New in version 1.3.0.

pyotherside.**qrc_is_dir**(*dirname*)
Check if `dirname` is an existing directory in the Qt Resource System.

Returns `True` if `dirname` is a directory, `False` otherwise.

New in version 1.3.0.

pyotherside.**qrc_get_file_contents**(*filename*)
Get the file contents of a file in the Qt Resource System.

Raises **ValueError** – If `filename` does not denote a valid file.

Returns The file contents as Python `bytearray` object.

New in version 1.3.0.

pyotherside.**qrc_list_dir**(*dirname*)
Get the entry list of a directory in the Qt Resource System.

Raises **ValueError** – If `dirname` does not denote a valid directory.

Returns The directory entries as list of strings.

New in version 1.3.0.

Constants

New in version 1.1.0.

These constants are used in the return value of a *image provider* function:

pyotherside.format_mono Mono pixel format (`QImage::Format_Mono`).

pyotherside.format_mono_lsb Mono pixel format, LSB alignment (`QImage::Format_MonoLSB`).

pyotherside.format_rgb32 32-bit RGB format (`QImage::Format_RGB32`).

pyotherside.format_argb32 32-bit ARGB format (`QImage::Format_ARGB32`).

pyotherside.format_rgb16 16-bit RGB format (`QImage::Format_RGB16`).

pyotherside.format_rgb666 18bpp RGB666 format (`QImage::Format_RGB666`).

pyotherside.format_rgb555 15bpp RGB555 format (`QImage::Format_RGB555`).

pyotherside.format_rgb888 24-bit RGB format (`QImage::Format_RGB888`).

pyotherside.format_rgb444 12bpp RGB format (`QImage::Format_RGB444`).

pyotherside.format_data Encoded image file data (e.g. PNG/JPEG data).

New in version 1.3.0.

The following constants have been added in PyOtherSide 1.3:

pyotherside.version Version of PyOtherSide as string.

New in version 1.5.0.

The following constants have been added in PyOtherSide 1.5:

pyotherside.format_svg_data SVG image XML data

CHAPTER 3

Data Type Mapping

PyOtherSide will automatically convert Python data types to Qt data types (which in turn will be converted to QML data types by the QML engine). The following data types are supported and can be used to pass data between Python and QML (and vice versa):

Python	QML	Remarks
bool	bool	
int	int	
float	double	
str	string	
list	JS Array	JS Arrays are always converted to Python lists.
tuple	JS Array	
dict	JS Object	Keys must be strings
datetime.date	QML date	since PyOtherSide 1.2.0
datetime.time	QML time	since PyOtherSide 1.2.0
datetime.datetime	JS Date	since PyOtherSide 1.2.0
set	JS Array	since PyOtherSide 1.3.0
iterable	JS Array	since PyOtherSide 1.3.0
object	(opaque)	since PyOtherSide 1.4.0
pyotherside.QObject	QObject	since PyOtherSide 1.4.0

Trying to pass in other types than the ones listed here is undefined behavior and will usually result in an error.

CHAPTER 4

Image Provider

New in version 1.1.0.

A QML Image Provider can be registered from Python to load image data (e.g. map tiles, diagrams, graphs or generated images) in QML `Image` elements without resorting to saving/loading files.

An image provider has the following argument list and return values:

```
def image_provider(image_id, requested_size):
    ...
    return bytearray(pixels), (width, height), format
```

The parameters to the image provider functions are:

image_id The ID of the image URL (`image://python/<image_id>`).

requested_size The source size of the QML `Image` as tuple: `(width, height)`. `(-1, -1)` if the source size is not set.

The image provider must return a tuple `(data, size, format)`:

data A `bytearray` object containing the pixel data for the given size and the given format.

size A tuple `(width, height)` describing the size of the pixel data in pixels.

format The pixel format of `data` (see *constants*), `pyotherside.format_data` if `data` contains an encoded (PNG/JPEG) image instead of raw pixel data or `pyotherside.format_svg_data` if `data` contains SVG image XML data.

In order to register the image provider with PyOtherSide for use as provider for `image://python/` URLs, the image provider function needs to be passed to PyOtherSide:

```
import pyotherside

def image_provider(image_id, requested_size):
    ...

pyotherside.set_image_provider(image_provider)
```

Because Python modules are usually imported asynchronously, the image provider will only be registered once the module registering the image provider is successfully imported. You have to make sure that setting the `source` property on a QML `Image` element only happens *after* the image provider has been set (e.g. by setting the `source` property in the callback function passed to *`importModule()`*).

CHAPTER 5

Qt Resource Access

New in version 1.3.0.

If you are using PyOtherSide in combination with an application binary compiled from C++ code with Qt Resources (see Qt Resource System), you can inspect and access the resources from Python. This example demonstrates the API by walking the whole resource tree, printing out directory names and file sizes:

```
import pyotherside
import os.path

def walk(root):
    for entry in pyotherside.qrc_list_dir(root):
        name = os.path.join(root, entry)
        if pyotherside.qrc_is_dir(name):
            print('Directory:', name)
            walk(name)
        else:
            data = pyotherside.qrc_get_file_contents(name)
            print('File:', name, 'has', len(data), 'bytes')

walk('/')
```

Importing Python modules from Qt Resources also works starting with QML API 1.3 using `Qt.resolvedUrl()` from within a QML file in Qt Resources. As an alternative, `addImportPath('qrc:/')` will add the root directory of the Qt Resources to Python's module search path.

CHAPTER 6

Accessing QObjects from Python

New in version 1.4.0.

Since version 1.4, PyOtherSide allows passing QObjects from QML to Python, and accessing (setting / getting) properties and calling slots and dynamic methods. References to QObjects passed to Python can be passed back to QML transparently:

```
# Assume func will be called with a QObject as sole argument
def func(qobject):
    # Getting properties
    print(qobject.x)

    # Setting properties
    qobject.x = 123

    # Calling slots and dynamic functions
    print(qobject.someFunction(123, 'b'))

    # Returning a QObject reference to the caller
    return qobject
```

It is possible to store a reference (bound method) to a method of a QObject. Such references cannot be passed to QML, and can only be used in Python for the lifetime of the QObject. If you need to pass such a bound method to QML, you can wrap it into a Python object (or even just a lambda) and pass that instead:

```
def func(qobject):
    # Can store a reference to a bound method
    bound_method = qobject.someFunction

    # Calling the bound method
    bound_method(123, 'b')

    # If you need to return the bound method, you must wrap it
    # in a lambda (or any other Python object), the bound method
    # cannot be returned as-is for now
    return lambda a, b: bound_method(a, b)
```

It's not possible to instantiate new QObjects from within Python, and it's not possible to subclass QObject from within Python. Also, be aware that a reference to a QObject in Python will become invalid when the QObject is deleted (there's no way for PyOtherSide to prevent referenced QObjects from being deleted, but PyOtherSide tries hard to detect the deletion of objects and give meaningful error messages in case the reference is accessed).

CHAPTER 7

OpenGL rendering in Python

New in version 1.5.0.

You can render directly to a QML application's OpenGL context in your Python code (i.e. via PyOpenGL or vispy.gloo) by using a `PyGLArea` or `PyFBO` item.

The `IRenderer` interface that needs to be implemented in Python and set as the `renderer` property of `PyGLArea` or `PyFBO` needs to provide the following functions:

`IRenderer.`**`init`**`()`
: Initialize OpenGL resources required for rendering. This method is optional.

`IRenderer.`**`reshape`**`(`*x*, *y*, *width*, *height*`)`
: Called when the geometry has changed.

  `(x, y)` is the position of the bottom left corner of the area, in window coordinates, e.g. (0, 0) is the bottom left corner of the window.

`IRenderer.`**`render`**`()`
: Render to the OpenGL context.

  It is the renderer's responsibility to unbind any used resources to leave the context in a clean state.

`IRenderer.`**`cleanup`**`()`
: Free any resources allocated by *`IRenderer.init()`*. This method is optional.

See *Rendering with PyOpenGL* for an example implementation.

Note that you might to use a recent version of PyOpenGL (>= 3.1.0) for some of the examples to work, earlier versions had problems. If your distribution does not provide new versions, you can install the most recent version of PyOpenGL to your `$HOME` using:

```
pip3 install --user --upgrade PyOpenGL PyOpenGL_accelerate
```

CHAPTER 8

Cookbook

This section contains code examples and best practices for combining Python and QML.

Importing modules and calling functions asynchronously

In this example, we import the Python Standard Library module `os` and - when the module is imported - call the `os.getcwd()` function on it. The result of the `os.getcwd()` function is then printed to the console and `os.chdir()` is called with a single argument (`'/'`) - again, after the `os.chdir()` function has returned, a message will be printed.

In this example, importing modules and calling functions are both done in an asynchronous way - the QML/GUI thread will not block while these functions execute. In fact, the `Component.onCompleted` code block will probably finish before the `os` module has been imported in Python.

```
Python {
    Component.onCompleted: {
        importModule('os', function() {
            call('os.getcwd', [], function (result) {
                console.log('Working directory: ' + result);
                call('os.chdir', ['/'], function (result) {
                    console.log('Working directory changed.');
                });});
            });
        });
    }
}
```

While this continuation-passing style might look a like a little pyramid due all the nesting and indentation at first, it makes sure your application's UI is always responsive. The user will be able to interact with the GUI (e.g. scroll and move around in the UI) while the Python code can process requests.

To avoid what's called callback hell in JavaScript, you can pull out the anonymous functions you give as callbacks, give them names and pass them to the API functions via name, e.g. the above example would turn into a shallow

structure (of course, in this example, splitting everything out does not make too much sense, as the functions are very simple to begin with, but it's here to demonstrate how splitting a callback hell pyramid basically works):

```
Python {
    Component.onCompleted: {
        function changedCwd(result) {
            console.log('Working directory changed.');
        }

        function gotCwd(result) {
            console.log('Working directory: ' + result);
            call('os.chdir', ['/'], changedCwd);
        }

        function withOs() {
            call('os.getcwd', [], gotCwd);
        }

        importModule('os', withOs);
    }
}
```

Evaluating Python expressions in QML

The *evaluate()* method on the Python object can be used to evaluate a simple Python expression and return its result as JavaScript object:

```
Python {
    Component.onCompleted: {
        console.log('Squares: ' + evaluate('[x for x in range(10)]'));
    }
}
```

Evaluating expressions is done synchronously, so make sure you only use it for expressions that are not long-running calculations / operations.

Error handling in QML

If an error happens in Python while calling functions, the traceback of the error (or an error message in case the error happens in the PyOtherSide layer) will be sent with the *error()* signal of the Python element. During early development, it's probably enough to just log the error to the console:

```
Python {
    // ...

    onError: console.log('Error: ' + traceback)
}
```

Once your application grows, it might make sense to maybe show the error to the user in a dialog box, message or notification in addition to or instead of using console.log() to print the error.

Handling asynchronous events from Python in QML

Your Python code can send asynchronous events with optional data to the QML layer using the *pyotherside.send()* function. You can call this function from functions called from QML, but also from anywhere else - including threads that you created in Python. The first parameter is mandatory, and must be a string that identifies the event. Additional parameters are optional and can be of any data type that PyOtherSide supports:

```
import pyotherside

pyotherside.send('new-entries', 100, 123)
```

If you do not add a special handler on the `Python` object, such events would be handled by the *received()* signal handler in QML - its `data` parameter contains the event name and all arguments in a list:

```
Python {
    // ..

    onReceived: console.log('Event: ' + data)
}
```

Usually, you want to install a handler for such events. If you have e.g. the `'new-entries'` event like shown above (with two numeric parameters that we will call `first` and `last` for this example), you might want to define a simple handler function that will process this event:

```
Python {
    // ..

    Component.onCompleted: {
        setHandler('new-entries', function (first, last) {
            console.log('New entries from ' + first + ' to ' + last);
        });
    }
}
```

Once a handler for a given event is defined, the *received()* signal will not be emitted anymore. If you need to unset a handler for a given event, you can use `setHandler('event', undefined)` to do so.

In some cases, it might be useful to not install a handler function directly, but turn the *pyotherside.send()* call into a new signal on the `Python` object. As there is no easy way for PyOtherSide to determine the names of the arguments of the event, you have to define and hook up these signals manually. The upside of having to define the signals this way is that all signals will be nicely documented in your QML file for future reference:

```
Python {
    signal updated()
    signal newEntries(int first, int last)
    signal entryRenamed(int index, string name)

    Component.onCompleted: {
        setHandler('updated', updated);
        setHandler('new-entries', newEntries);
        setHandler('entry-renamed', entryRenamed);
    }
}
```

With this setup, you can now emit these signals from the `Python` object by using *pyotherside.send()* in your Python code:

```
pyotherside.send('updated')
pyotherside.send('new-entries', 20, 30)
pyotherside.send('entry-renamed', 11, 'Hello World')
```

Loading `ListModel` data from Python

Most of the time a PyOtherSide QML application will display some data stored somewhere and retrieved or generated with Python. The easiest way to do this is to return a list-of-dicts in your Python function:

listmodel.py

```
def get_data():
    return [
        {'name': 'Alpha', 'team': 'red'},
        {'name': 'Beta', 'team': 'blue'},
        {'name': 'Gamma', 'team': 'green'},
        {'name': 'Delta', 'team': 'yellow'},
        {'name': 'Epsilon', 'team': 'orange'},
    ]
```

Of course, the function could do other things (such as doing web requests, querying databases, etc..) - as long as it returns a list-olf-dicts, it will be fine (if you are using a generator that yields dicts, just wrap the generator with `list()`). Using this function from QML is straightforward:

listmodel.qml

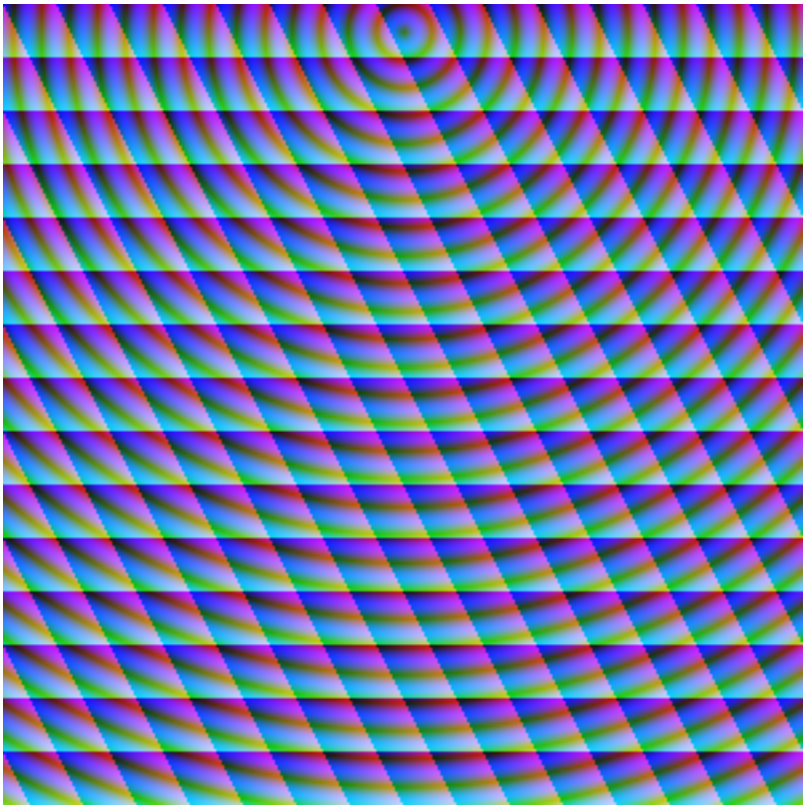
```
import QtQuick 2.0
import io.thp.pyotherside 1.5

Rectangle {
    color: 'black'
    width: 400
    height: 400

    ListView {
        anchors.fill: parent

        model: ListModel {
            id: listModel
        }

        delegate: Text {
            // Both "name" and "team" are taken from the model
            text: name
            color: team
        }
    }

    Python {
        id: py

        Component.onCompleted: {
            // Add the directory of this .qml file to the search path
            addImportPath(Qt.resolvedUrl('.'));

            // Import the main module and load the data
```

```
            importModule('listmodel', function () {
                py.call('listmodel.get_data', [], function(result) {
                    // Load the received data into the list model
                    for (var i=0; i<result.length; i++) {
                        listModel.append(result[i]);
                    }
                });
            });
        }
    }
}
```

Instead of passing a list-of-dicts, it is of course also possible to send new list items via *pyotherside.send()*, one item at a time, and append them to the list model that way.

Rendering RGBA image data in Python

New in version 1.1.0.



This example uses the *image provider* feature of PyOtherSide to render RGB image data in Python and display the rendered data in QML using a normal QtQuick 2.0 `Image` element:

imageprovider.py

```
import pyotherside
import math

def render(image_id, requested_size):
```

```
    print('image_id: "{image_id}", size: {requested_size}'.format(**locals()))

    # width and height will be -1 if not set in QML
    if requested_size == (-1, -1):
        requested_size = (300, 300)

    width, height = requested_size

    # center for circle
    cx, cy = width/2, 10

    pixels = []
    for y in range(height):
        for x in range(width):
            pixels.extend(reversed([
                255, # alpha
                int(10 + 10 * ((x - y * 0.5) % 20)), # red
                20 + 10 * (y % 20), # green
                int(255 * abs(math.sin(0.3*math.sqrt((cx-x)**2 + (cy-y)**2)))) # blue
            ]))
    return bytearray(pixels), (width, height), pyotherside.format_argb32

pyotherside.set_image_provider(render)
```

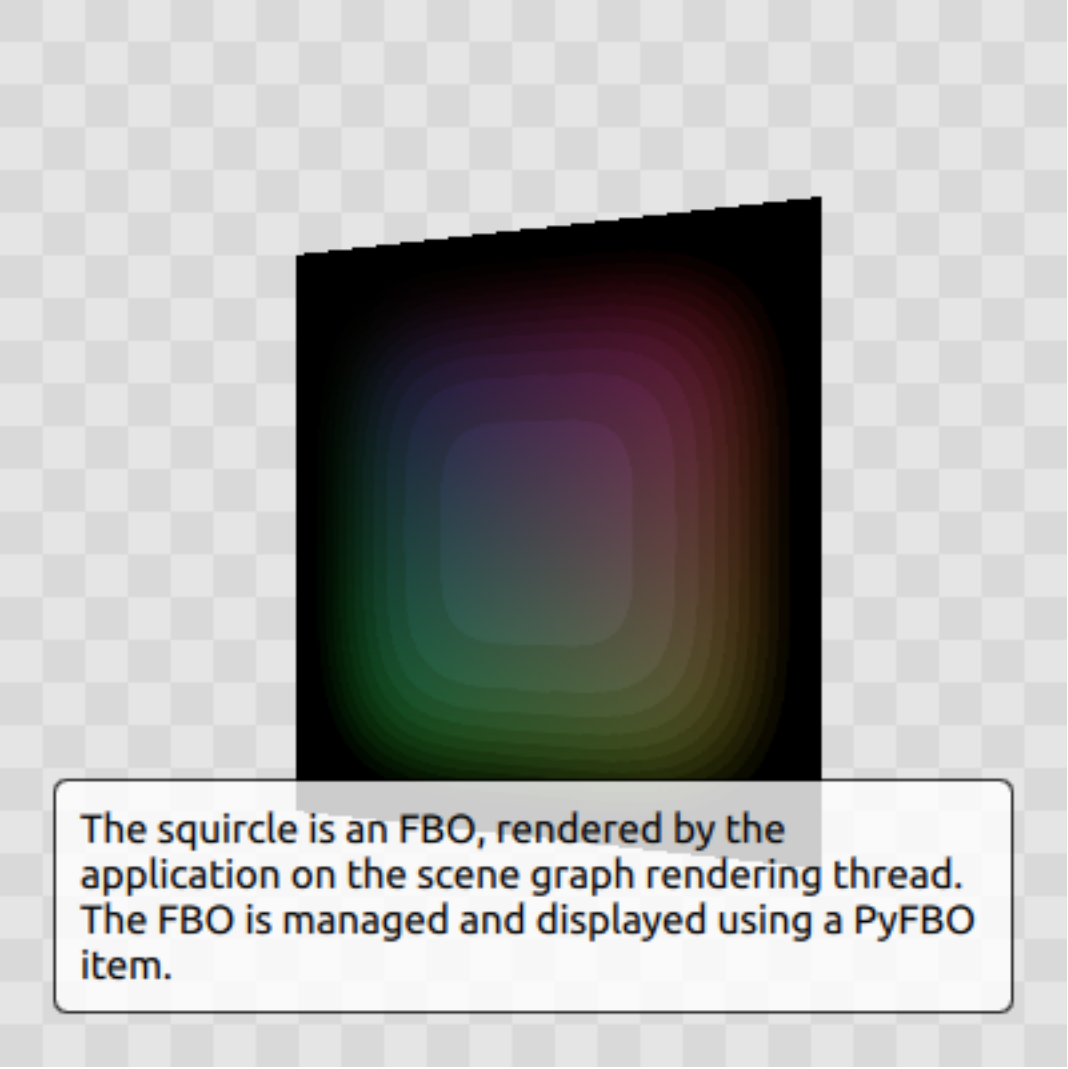
This module can now be imported in QML and used as `source` in the QML `Image` element:

imageprovider.qml

```
import QtQuick 2.0
import io.thp.pyotherside 1.5

Image {
    id: image
    width: 300
    height: 300

    Python {
        Component.onCompleted: {
            // Add the directory of this .qml file to the search path
            addImportPath(Qt.resolvedUrl('.'));

            importModule('imageprovider', function () {
                image.source = 'image://python/image-id-passed-from-qml';
            });
        }

        onError: console.log('Python error: ' + traceback)
    }
}
```

Rendering with PyOpenGL

New in version 1.5.0.



The example below shows how to do raw OpenGL rendering in PyOpenGL using `PyGLArea`. It has been adapted from the tutorial in the Qt documentation at http://qt-project.org/doc/qt-5/qtquick-scenegraph-openglunderqml-example.html.

renderer.py

```
import numpy

from OpenGL.GL import *
from OpenGL.GL.shaders import compileShader, compileProgram

VERTEX_SHADER = """#version 130
attribute highp vec4 vertices;
varying highp vec2 coords;

void main() {
    gl_Position = vertices;
    coords = vertices.xy;
}
"""

FRAGMENT_SHADER = """#version 130
```

```
uniform lowp float t;
varying highp vec2 coords;
void main() {
    lowp float i = 1. - (pow(abs(coords.x), 4.) + pow(abs(coords.y), 4.));
    i = smoothstep(t - 0.8, t + 0.8, i);
    i = floor(i * 20.) / 20.;
    gl_FragColor = vec4(coords * .5 + .5, i, i);
}
"""

class Renderer(object):

    def __init__(self):
        self.t = 0.0
        self.values = numpy.array([
            -1.0, -1.0,
            1.0, -1.0,
            -1.0, 1.0,
            1.0, 1.0
        ], dtype=numpy.float32)

    def set_t(self, t):
        self.t = t

    def init(self):
        self.vertexbuffer = glGenBuffers(1)
        vertex_shader = compileShader(VERTEX_SHADER, GL_VERTEX_SHADER)
        fragment_shader = compileShader(FRAGMENT_SHADER, GL_FRAGMENT_SHADER)
        self.program = compileProgram(vertex_shader, fragment_shader)
        self.vertices_attr = glGetAttribLocation(self.program, b'vertices')
        self.t_attr = glGetUniformLocation(self.program, b't')

    def reshape(self, x, y, width, height):
        glViewport(x, y, width, height)

    def render(self):
        glUseProgram(self.program)
        try:
            glDisable(GL_DEPTH_TEST)
            glClearColor(0, 0, 0, 1)
            glClear(GL_COLOR_BUFFER_BIT)
            glEnable(GL_BLEND)
            glBlendFunc(GL_SRC_ALPHA, GL_ONE)

            glBindBuffer(GL_ARRAY_BUFFER, self.vertexbuffer)
            glEnableVertexAttribArray(self.vertices_attr)
            glBufferData(GL_ARRAY_BUFFER, self.values, GL_STATIC_DRAW)
            glVertexAttribPointer(self.vertices_attr, 2, GL_FLOAT, GL_FALSE, 0, None)
            glUniform1f(self.t_attr, self.t)

            glDrawArrays(GL_TRIANGLE_STRIP, 0, 4)
        finally:
            glDisableVertexAttribArray(0)
            glBindBuffer(GL_ARRAY_BUFFER, 0)
            glUseProgram(0)

    def cleanup(self):
        glDeleteProgram(self.program)
```

```
        glDeleteBuffers(1, [self.vertexbuffer])
```

pyglarea.qml

```
import QtQuick 2.0
import io.thp.pyotherside 1.5

Item {
    width: 320
    height: 480

    PyGLArea {
        id: glArea
        anchors.fill: parent
        property var t: 0

        SequentialAnimation on t {
            NumberAnimation { to: 1; duration: 2500; easing.type: Easing.InQuad }
            NumberAnimation { to: 0; duration: 2500; easing.type: Easing.OutQuad }
            loops: Animation.Infinite
            running: true
        }

        onTChanged: {
            if (renderer) {
                py.call(py.getattr(renderer, 'set_t'), [t], update);
            }
        }
    }

    Rectangle {
        color: Qt.rgba(1, 1, 1, 0.7)
        radius: 10
        border.width: 1
        border.color: "white"
        anchors.fill: label
        anchors.margins: -10
    }

    Text {
        id: label
        color: "black"
        wrapMode: Text.WordWrap
        text: "The background here is a squircle rendered with raw OpenGL using a␣
↪PyGLArea. This text label and its border is rendered using QML"
        anchors.right: parent.right
        anchors.left: parent.left
        anchors.bottom: parent.bottom
        anchors.margins: 20
    }

    Python {
        id: py

        Component.onCompleted: {
            addImportPath(Qt.resolvedUrl('.'));
            importModule('renderer', function () {
                call('renderer', [], function (renderer) {
```

```
                    glArea.renderer = renderer;
                });
            });
        }

        onError: console.log(traceback);
    }
}
```

CHAPTER 9

Building PyOtherSide

The following build requirements have to be satisfied to build PyOtherSide:

- Qt 5.1.0 or newer
- Python 3.2.0 or newer

If you have the required build-dependencies installed, building and installing the PyOtherSide plugin should be as simple as:

```
qmake
make
make install
```

In case your system doesn't provide `python3-config`, you might have to pass a suitable `python-config` to `qmake` at configure time:

```
qmake PYTHON_CONFIG=python3.3-config
make
make install
```

Alternatively, you can edit `python.pri` manually and specify the compiler flags for compiling and linking against Python on your system.

As of version 1.3.0, PyOtherSide does not build against Python 2.x anymore.

Building for Blackberry 10

On Blackberry 10 (tested versions: 10.1, 10.2), Python 3.2.2 is already installed on-device. Qt 5 is not installed (only Qt 4), so if you are packaging a PyOtherSide application, you need to ship Qt 5 with it.

The approach we currently use is:

1. Build Qt 5 using the Native SDK
2. Get a set of matching Python 3.2.2 headers

3. Fetch the following files from the device's filesystem:

- `/usr/lib/libpython3.2m.so`
- `/usr/include/python3.2m/pyconfig.h`

4. Use `pyconfig.h` with the Python 3.2.2 headers and link against `libpython3.2m`

Modify `python.pri` to point to the fetched library and your Python 3.2.2 headers (with `pyconfig.h` from the device):

```
QMAKE_LIBS += -lpython3.2m -L/path/to/where/the/library/is
QMAKE_CXXFLAGS += -I/path/to/where/the/headers/are/include/python3.2m
```

After installing PyOtherSide in the locally-build Qt 5 (cross-compiled for BB10), the QML plugins folder can be deployed with the .bar file.

Building for Android

Unlike Blackberry there is no Python or Qt present by default and both need to be shipped with the application.

The current solution can be summarized like this:

1. Statically cross-compile Python 3 for Android using the Android NDK
2. Statically compile PyOtherSide against the Android Python build and bundle the Python standard library inside the PyOtherSide binary
3. Use the Qt 5 SDK to make a QtQuick application - the SDK will handle bundling of your application file and of the PyOtherSide binary automatically

A more detailed guide follows. It describes how to get from the source code of the relevant components to being able to run an Android application with a Qt Quick 2.0 GUI running on an Android device. The *gPodder* podcast aggregator serves as (full featured & fully functional!) example of such an application.

Performed in this environment:

- Fedora 20
- Qt 5.3.1 Android SDK
- latest Android SDK with API level 14 installed
- OpenJDK 1.7
- a few GB of harddrive space
- an Android 4.0+ device connected to the computer that is accessible over `adb` (eq. the debugging mode is enabled)

This is just one example environment where these build instructions have been tested to work. Reasonably similar environments should work just as well.

The build is going to be done in a folder called `build` in the users home directory, lets say that the use is named `user` (replace accordingly for your environment).

We start in the home directory:

```
mkdir build
cd build
```

Now clone the needed projects, load submodules and switch to correct branches.

```
git clone --branch fixes https://github.com/thp/python3-android
git clone https://github.com/thp/pyotherside
git clone --recursive https://github.com/gpodder/gpodder-android
```

Next we will build Python 3 for Android. This will first download the Android NDK, then Python 3 source code, followed by crosscompiling the Python 3 code for Android on ARM. *NOTE that this step alone can require multiple GB of harddisk space.*

```
cd python3-android
make all
```

As the next step we modify the `python.pri.android` file to point to our Python build. If should look like this as a result (remember to modify it for your environment):

```
QMAKE_LIBS += -L/home/user/build/python3-android/build/9d-14-arm-linux-androideabi-4.
↪8/lib -lpython3.3m -ldl -lm -lc -lssl -lcrypto
QMAKE_CXXFLAGS += -I/home/user/build/python3-android/build/9d-14-arm-linux-
↪androideabi-4.8/include/python3.3m/
```

Then copy the file over the python.pri file in the PyOtherSide project directory:

```
cd ..
cp python3-android/python.pri.android pyotherside/python.pri
```

PyOtherSide can also help us ship & load the Python standard library if we can provide it a suitable zip bundle, which can be created like this:

```
cd python3-android/build/9d-14-arm-linux-androideabi-4.8/lib/python3.3/
zip -r pythonlib.zip *
cd ../../../../..
```

For PyOtherSide to include the packed Python standard library it needs to be placed in its src subfolder:

```
mv python3-android/build/9d-14-arm-linux-androideabi-4.8/lib/python3.3/pythonlib.zip␣
↪pyotherside/src/
```

PyOtherSide will then use the qrc mechanism to compile the compressed standard library during inside it's own binary. This removes the need for us to handle its shipping & loading ourself.

Next you need to build PyOtherSide with QtCreator from the Qt 5.3 Android SDK, so make sure that the Qt 5.3 Android kit is using the exact same NDK that has been used to build Python 3 for Android. To do that go to *settings*, find the *kits* section, select the Android kit and make sure that the NDK path points to:

`/home/user/build/python3-android/sdk/android-ndk-r9d`

Next open the pyotherside/pyotherside.pro project file on QtCreator, select the Android kit and once the project loads go to the *project view* and make sure that under *run* the API level is set to 14 (this corresponds to Android 4.0 and later). The Android Python 3 build has been built for API level 14 and our PyOtherSide build should do the same to be compatible.

Also make sure that shadow build is disabled, just in case.

Once done with the configuration got to the *build* menu and select the *built pyotherside* option - this should build PyOtherSide for Android and statically compile in our Python build and also include the Python standard library zip file with qrc.

As the next step we need to move the PyOtherSide binary to the QML plugin folder for the Qt Android SDK, so that it can be fetched by the SDK when building gPodder.

Let's say we have the SDK installed in the `/opt` directory (default for the Qt SDK installer on Linux), giving us this path to the plugin folder:

`/opt/Qt5.3/5.3/android_armv7/qml`

First create the folder structure for the pyotherside plugin:

```
mkdir -p /opt/Qt5.3/5.3/android_armv7/qml/io/thp/pyotherside
```

Then copy the pyotherside binary and *qmldir* file to the folder:

```
cp pyotherside/src/libpyothersideplugin.so /opt/Qt5.3/5.3/android_armv7/qml/io/thp/
↪pyotherside/
cp pyotherside/src/qmldir /opt/Qt5.3/5.3/android_armv7/qml/io/thp/pyotherside/
```

Next open the gPodder project in QtCreator (gpodder-android/gpodder-android.pro) and again make sure the Android kit is selected, that the API level 14 is used and that *shadow build* is disabled. Then just press the *Run* button and the SDK should build an Android APK that includes the libpyotherside binary (it fetched automatically from the plugins directory because is referenced in the gPodder QML source code) and deploy it to the device where gPodder should be started.

Building for Windows

On Windows (tested versions: Windows 7), you need to download:

1. Qt 5 (VS 2010) from qt-project.org downloads (tested: 5.2.1)
2. Visual C++ 2010 Express with SP1
3. Python 3 from python.org Windows downloads (tested: 3.3.4)

We use VS 2010 instead of MinGW, because the MinGW version of Qt depends on working OpenGL driver, whereas the non-OpenGL version uses Direct3D via ANGLE. Also, Python is built with Visual C++ 2010 Express (see Compiling Python on Windows), so using the same toolchain when linking all three components (Qt, Python and PyOtherSide) together makes sense.

The necessary customizations for building PyOtherSide successfully on Windows have been integrated recently, and are available since PyOtherSide 1.3.0.

Once these pre-requisites are installed, you need to make some customizations to the build setup:

1. In `src/qmldir`: Change `plugin pyothersideplugin` to `plugin pyothersideplugin1`. This is needed, because on Windows, the library version gets encoded into the library name.
2. In `python.pri`: Modify it so that the Python 3 `libs/` folder is added to the linker path, and link against `-lpython33`. Also, modify it so that the Python 3 `include/` folder is added to the compiler flags.

Example `python.pri` file for a standard Python 3.3 installation on Windows:

```
QMAKE_LIBS += -LC:\Python33\libs -lpython33
QMAKE_CXXFLAGS += -IC:\Python33\include\
```

With the updated `qmldir` and `python.pri` files in place, simply open the `pyotherside.pro` project file in Qt Creator, and build the project. Configure a **Release Build**, and *disable* **Shadow Builds**.

To install PyOtherSide into your Qt installation, so that the QML import works from other projects:

1. Make sure the PyOtherSide project is opened in Qt Creator
2. In the left column, select **Projects**

3. Make sure the **Run** tab (Run Settings) of your project is selected
4. In **Deployment**, click **Add Deploy Step** and select **Make**
5. In the **Make arguments:** field, type `install`
6. Hit **Run** to install PyOtherSide in your local Qt folder
7. Dismiss the "Custom Executable" dialog that pops up

Known Problems:

- **Qt Resource System** importing might not fully work on Windows

CHAPTER 10

ChangeLog

Version 1.5.1 (2017-03-17)

- Fix *call_sync()* when used with parameters (fix by Robie Basak; issue #49)

Version 1.5.0 (2016-06-14)

- Support for *OpenGL rendering in Python* using PyOpenGL >= 3.1.0
- New QML components: `PyGLArea`, `PyFBO`
- *pythonVersion()* now returns the runtime Python version
- Add the library to `PYTHONPATH` for standard library appended as .zip (except on Windows)
- Call `PyDateTime_IMPORT` as often as necessary (Fixes #46)
- Added `pyotherside.format_svg_data` for using SVG data in the image provider
- Handle converting `QVariantHash` to Python `dict` type
- Added `.qmltypes` file to provide metadata information for Qt Creator
- New functions *importNames()* and *importNames_sync()* for from-imports

Version 1.4.0 (2015-02-19)

- Support for passing Python objects to QML and keeping references there
- Add *getattr()* to get an attribute from a Python object
- *call()* and *call_sync()* now also accept a Python callable as first argument
- Support for *Accessing QObjects from Python* (properties and slots)

- Print error messages to the console if *error()* doesn't have any handlers connected

Version 1.3.0 (2014-07-24)

- Access to the Qt Resource System from Python (see *Qt Resource Access*).
- QML API 1.3: Import from Qt Resources (*addImportPath()* with `qrc:/`).
- Add `pyotherside.version` constant to access version from Python as string.
- Support for building on Windows, build instructions for Windows builds.
- New data type conversions: Python `set` and iterable types (e.g. generator expressions and generators) are converted to JS `Array`.

Version 1.2.0 (2014-02-16)

- Introduced versioned QML imports for API change.
- QML API 1.2: Change *importModule()* behavior for imports with dots.
- QML API 1.2: Emit *error()* when JavaScript callbacks passed to *importModule()* and *call()* throw an exception.
- New data type conversions: Python `datetime.date`, `datetime.time` and `datetime.datetime` are converted to QML `date`, `time` and JS `Date` types, respectively.

Version 1.1.0 (2014-02-06)

- Add support for Python-based image providers (see *Image Provider*).
- Fix threading crashes and aborts due to assertions.
- *addImportPath()* will automatically strip a leading `file://`.
- Added *pluginVersion()* and *pythonVersion()* for runtime version detection.

Version 1.0.0 (2013-08-08)

- Initial QML plugin release.

Version 0.0.1 (2013-05-17)

- Proof-of-concept (based on a prototype from May 2011).

Index