

---

# PyOtherSide Documentation

*Release 1.3.0*

**Thomas Perl**

July 24, 2014



<b>1</b>	<b>QML API</b>	<b>3</b>
1.1	Import Versions . . . . .	3
1.2	QML Python Element . . . . .	3
<b>2</b>	<b>Python API</b>	<b>7</b>
2.1	The pyotherside module . . . . .	7
<b>3</b>	<b>Data Type Mapping</b>	<b>9</b>
<b>4</b>	<b>Image Provider</b>	<b>11</b>
<b>5</b>	<b>Qt Resource Access</b>	<b>13</b>
<b>6</b>	<b>Cookbook</b>	<b>15</b>
6.1	Importing modules and calling functions asynchronously . . . . .	15
6.2	Error handling in QML . . . . .	16
6.3	Handling asynchronous events from Python in QML . . . . .	16
6.4	Loading ListModel data from Python . . . . .	17
6.5	Rendering RGBA image data in Python . . . . .	19
<b>7</b>	<b>Building PyOtherSide</b>	<b>21</b>
7.1	Building for Blackberry 10 . . . . .	21
7.2	Building for Windows . . . . .	22
<b>8</b>	<b>ChangeLog</b>	<b>23</b>
8.1	Version 1.3.0 (2014-07-24) . . . . .	23
8.2	Version 1.2.0 (2014-02-16) . . . . .	23
8.3	Version 1.1.0 (2014-02-06) . . . . .	23
8.4	Version 1.0.0 (2013-08-08) . . . . .	23
8.5	Version 0.0.1 (2013-05-17) . . . . .	24



*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.



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

## 1.1 Import Versions

The current QML API version of *PyOtherSide* is 1.3. When new features are introduced, or behavior is changed, the API version will be bumped and documented here.

### 1.1.1 `io.thp.pyotherside 1.0`

- Initial API release.

### 1.1.2 `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`.

### 1.1.3 `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).

## 1.2 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 1.3
```

### 1.2.1 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.

### 1.2.2 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).

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

**call** (*string 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).

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*)  $\rightarrow$  var

Evaluate a Python expression synchronously.

**importModule\_sync** (*string name*)  $\rightarrow$  bool

Import a Python module. Returns `true` on success, `false` otherwise.

**call\_sync** (*string func, var args=[]*)  $\rightarrow$  var

Call a Python function. Returns the return value of the Python function.

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.



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

## 2.1 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.

### 2.1.1 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.

## 2.1.2 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.

---

## 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	JS Arrays are always converted to Python lists.
int	int	
float	double	
str	string	
list	JS Array	
tuple	JS Array	Keys must be strings since PyOtherSide 1.2.0
dict	JS Object	
datetime.date	QML date	
datetime.time	QML time	
datetime.datetime	JS Date	
set	JS Array	since PyOtherSide 1.3.0
iterable	JS Array	since PyOtherSide 1.3.0

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



---

## 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](#)), or pyotherside.format\_data if data contains an encoded (PNG/JPEG) image instead of raw pixel 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()`).





---

## 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.



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

## 6.1 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 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);
}
```

### 6.1.1 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.

## 6.2 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.

## 6.3 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')
```

## 6.4 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-of-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 1.3

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

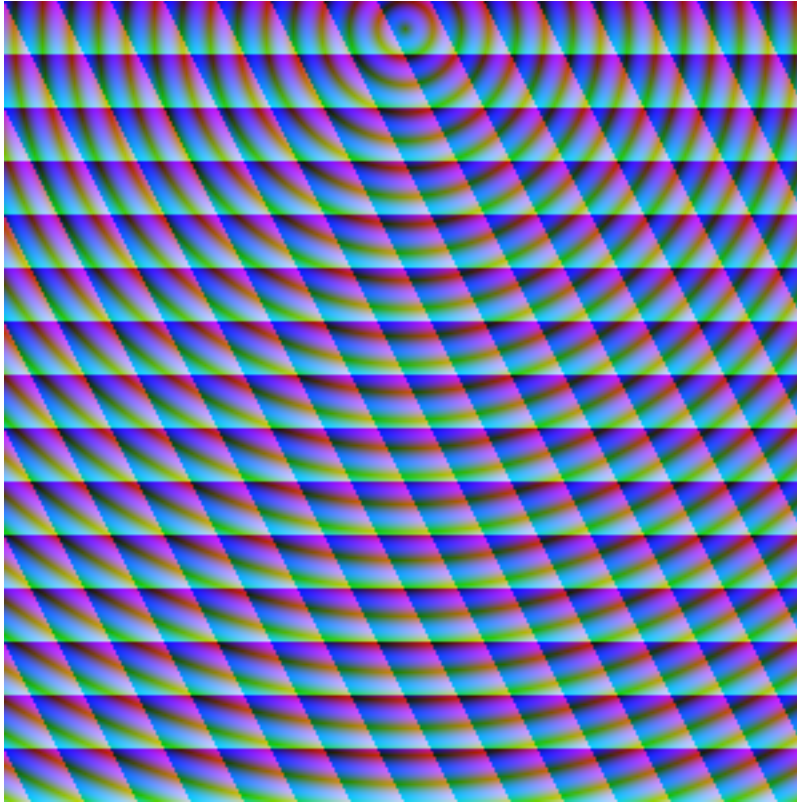
    Component.onCompleted: {
        // Add the directory of this .qml file to the search path
        addImportPath(Qt.resolvedUrl('.'));

        // 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.

## 6.5 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 1.3

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)
    }
}
```



---

## 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.

### 7.1 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.

## 7.2 Building for Windows

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

1. Qt 5 (VS 2010) from [qt-project.org downloads](http://qt-project.org/downloads) (tested: 5.2.1)
2. [Visual C++ 2010 Express](#)
3. Python 3 from [python.org Windows downloads](http://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

---

## ChangeLog

---

### 8.1 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`.

### 8.2 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.

### 8.3 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.

### 8.4 Version 1.0.0 (2013-08-08)

- Initial QML plugin release.

## 8.5 Version 0.0.1 (2013-05-17)

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