Introduction

This is the reference guide for PyQt5 v5.15.2. PyQt5 is a set of Python bindings for v5 of the Qt application framework from The Qt Company.

Qt is a set of C++ libraries and development tools that includes platform independent abstractions for graphical user interfaces, networking, threads, regular expressions, SQL databases, SVG, OpenGL, XML, user and application settings, positioning and location services, short range communications (NFC and Bluetooth), web browsing, 3D animation, charts, 3D data visualisation and interfacing with app stores. PyQt5 implements over 1000 of these classes as a set of Python modules.

PyQt5 supports the Windows, Linux, UNIX, Android, macOS and iOS platforms.

PyQt does not include a copy of Qt. You must obtain a correctly licensed copy of Qt yourself. However, binary wheels of the GPL version of PyQt5 are provided and these include a copy of the appropriate parts of the LGPL version of Qt.

The homepage for PyQt5 is https://www.riverbankcomputing.com/software/pyqt/. Here you will always find the latest stable version, current development previews, and the latest version of this documentation.

PyQt5 is built using the SIP bindings generator. SIP must be installed in order to build and use PyQt5.

Earlier versions of Qt are supported by PyQt4.

License

PyQt5 is dual licensed on all platforms under the Riverbank Commercial License and the GPL v3. Your PyQt5 license must be compatible with your Qt license. If you use the GPL version then your own code must also use a compatible license.

PyQt5, unlike Qt, is not available under the LGPL.

You can purchase a commercial PyQt5 license here.

PyQt5 Components

PyQt5 comprises a number of different components. First of all there are a number of Python extension modules. These are all installed in the `PyQt5` Python package and are described in the list of modules.

PyQt5 is distributed as a number of source packages and corresponding binary wheels each of which implement one or more logically related extension modules.

PyQt5 contains plugins that enable Qt Designer and **qmlscene** to be extended using Python code. See Writing Qt Designer Plugins and Integrating Python and QML respectively for the details.

PyQt5 also contains a number of utility programs.

- **pyuic5** corresponds to the Qt **uic** utility. It converts QtWidgets based GUIs created using Qt Designer to Python code.
- **pyrcc5** corresponds to the Qt **rcc** utility. It embeds arbitrary resources (eg. icons, images, translation files) described by a resource collection file in a Python module.
- **pylupdate5** corresponds to the Qt **lupdate** utility. It extracts all of the translatable strings from Python code and creates or updates `.ts` translation files. These are then used by Qt Linguist to manage the translation of those strings.

The DBus support module is installed as dbus.mainloop.pyqt5. This module provides support for the Qt event loop in the same way that the dbus.mainloop.glib included with the standard `dbus-python` bindings package provides support for the GLib event loop. The API is described in DBus Support. It is only available if the `dbus-python` v0.80 (or later) bindings package is installed. The QtDBus module provides a more Qt-like interface to DBus.

When PyQt5 is configured a file called `PyQt5.api` is generated. This can be used by the QScintilla editor component to enable the use of auto-completion and call tips when editing PyQt5 code. The API file is installed automatically if QScintilla is already installed.

PyQt5 includes a large number of examples. These are ports to Python of many of the C++ examples provided with Qt. They can be found in the `examples` directory.

Finally, PyQt5 contains the `.sip` files used by SIP to generate PyQt5 itself. These can be used by developers of bindings of other Qt based class libraries.

Contributing to this Documentation

The reference section of this documentation describes each element of the PyQt5 API. It is based on the original Qt documentation which, of course, contains many references to C++. The intention is that, over time, the documentation will be updated to replace all of the C++ idioms with their Python equivalents. However, given the size of the API, it is unlikely that this task will ever be complete.

The system used to create the documentation has been designed to make it easy for users to contribute patches converting it from C++ to Python a bit at a time. This is done in such as a way as to ensure that the documentation can be updated with new releases of both PyQt5 and Qt without losing any user-contributed modifications.

The documentation itself is written as reStructuredText and generated using Sphinx.

The documentation has its own public Mercurial repository here. The repository can be cloned using the following command:

```
hg clone https://www.riverbankcomputing.com/hg/PyQt5Docs
```

The latest version will always be on the `default` branch.

Repository Structure

The `docs` directory contains the handwritten overview documentation.

The `docs/api` directory contains the stuctured skeleton of the API documentation. It is automatically generated from the PyQt5 `.sip` files and are updated with every new release of PyQt5. They include information on all elements of the API, including method arguments and types, but do not contain any descriptions of those elements. They must not be modified by hand.

The `descriptions` directory contains a file for every individual element of the PyQt5 API - even down to individual enum members. Amongst other things, a file contains the reStructuredText describing the API element and a `:status:` field describing the status of the description. It is this `:status:` field than ensures that any user contributed modifications cannot be subsequently overwritten. Description files are initially created when a new release of PyQt5 introduces new elements to the API. Those description files that haven't been modified will be overwritten with every new release of Qt.

The `images` directory contains the images that are referered to in description files. Originally they were copied from the Qt documentation and may be replaced by more Python-centric alternatives.

The `snippets` directory contains the code snippets that are referered to in description files. Originally they were copied from the Qt documentation but with every line of C++ code turned into a Python comment.

The `sphinx` directory contains a Sphinx extension and theme that implements the documentation system.

The `sip2rst.py` script is run whenever a new release of PyQt5 is made. It updates the `docs/api` and `descriptions` directories.

The `webxml2rst.py` script is run whenever a new release of Qt is made. It updates the `descriptions`, `images` and `snippets` directories.

> **Note:** The naming convention used for description files requires that the repository is cloned to a case sensitive filesystem.

Description Files

Most contributions to the documentation will be patches to description files. The description files for each module are placed in a module-specific sub-directory of `descriptions`. The name of a description file is derived from the fully qualified name of the API element being described, a type tag, an optional unique identifier, and a `.rst` extension.

For example the description file for the QObject class is `descriptions/QtCore/QObject-c.rst`. Here the type tag `c` denotes a class. The complete set of type tags is shown in the table below.

a	an attribute
c	a class
e	an enum
f	a function or method
m	a module
s	a signal
v	an enum member

A function, method or signal may have overloads. Each overload is described in a separate file. In these cases the name of each file also includes a unique numerical identifer. You must look at the `:realsig:` field within the description file to determine which of the overloads the file describes.

Apart from the reST description itself, the only part of the description file that should be modified is the `:status:` field. The possible values of this field are described below.

todo
: The description is that extracted from the last release of Qt (or a stub if nothing was extracted) and has not been subsequently modified. It will be replaced when the next release of Qt is made.

done
: The description has been modified and will not be overwritten by the next release of Qt.

review
: The description has been modified. However the original description in the Qt document has itself been updated since the modifications were made. Therefore the changes to the Qt documentation should be reviewed to see if corresponding changes should be made to the description.

It follows from the above that any contributed change to a description file should set the `:status:` field to `done`.

Any other fields in a description file must not be modified.

The description itself may use any of the normal Sphinx and docutils domains, directives and roles. The only exception is that all cross-references to any element of the PyQt5 API should use the `:sip:ref` role. For example, a reference to the QObject class should be specified as `:sip:ref:`~PyQt5.QtCore.QObject`.`

Contributing Patches

User contributed patches can cover any of the following:

- `descriptions`
- `docs`
- `images`
- `snippets`
- `sphinx/riverbank/static/riverbank.css`.

A patch is created by using the `hg diff` command. Patches should be emailed to support@riverbankcomputing.com.

Support for Old Versions of Python

Begining with PyQt v5.13 a formal policy for the support for older versions of Python v3 has been adopted.

When a Python version reaches it’s end-of-life, support for it will be removed in the next minor release of PyQt. For example, if the current version of PyQt is v5.x.y then the support will be removed in v5.x+1.0. Specifically, PyQt v5.13 will remove support for Python v3.0, v3.1, v3.2, v3.3 and v3.4.

Support for Python v2 is handled slightly differently. Support for Python v2 is determined by the version of SIP being used. PyQt will no longer support Python v2 when SIP v6 is released (at which point SIP v4 will become unsupported).

Deprecated Features and Behaviours

The are currently no deprecated features or behaviours.

Incompatibilities with Earlier Versions

PyQt v5.12

Overflow Checking when Converting Integers

In previous versions PyQt5 did not perform overflow checking when converting Python `int` objects to C++ integer types. The behaviour when overflow occurred was undefined.

This version performs overflow checking and will raise an appropriate exception if overflow is detected.

The previous behaviour can be restored by calling **`PyQt5.sip.enableoverflowchecking(False)()`**.

PyQt v5.11

Importing the sip Module

In previous versions PyQt5 used the copy of the **`sip`** module usually installed in the `site-packages` directory and applications accessed it using as follows:

```
import sip
```

This version includes a private copy of the module. Applications should access it as follows:

```
from PyQt5 import sip
```

As an aid to backwards compatibility the module can still be imported as before but this will only work if another **`PyQt5`** module is imported first. For example the following will work:

```
from PyQt5 import QtCore
import sip
```

However it will not work if the order of the `import` statements is reversed.

PyQt v5.6

Enforcement of pyqtSlot() Signatures

In previous versions if a signal was connected to a method that was decorated by pyqtSlot(), and the signatures of the signal and slot were incompatible, then the connection was made anyway as if the method had not been decorated. This behaviour was a bug and not a feature.

This version ensures that the signatures are compatible and will raise an exception if they are not.

PyQt v5.5

Conversion of Latin-1 Strings to QByteArray

This version removes the automatic conversion of a Latin-1 encoded string when a QByteArray is expected. It was deprecated in PyQt v5.4.

Unhandled Python Exceptions

There are a number of situations where Python code is executed from C++. Python reimplementations of C++ virtual methods is probably the most common example. In previous versions, if the Python code raised an exception then PyQt would call Python's `PyErr_Print()` function which would then call sys.excepthook. The default exception hook would then display the exception and any traceback to `stderr`. There are number of disadvantages to this behaviour:

- the application does not terminate, meaning the behaviour is different to when exceptions are raised in other situations
- the output written to `stderr` may not be seen by the developer or user (particularly if it is a GUI application) thereby hiding the fact that the application is trying to report a potential bug.

This behaviour was deprecated in PyQt v5.4. In PyQt v5.5 an unhandled Python exception will result in a call to Qt's `qFatal()` function. By default this will call `abort()` and the application will terminate. Note that an application installed exception hook will still take precedence.

PyQt v5.3

Execution of Python Slots

In previous versions, when a signal was emitted to a Python slot that was not decorated with pyqtSlot(), it would not check that the underlying C++ receiver instance still existed. This matched the PyQt v4 behaviour at the time that PyQt v5.0 was released, but doesn't reflect the standard C++ behaviour.

The lack of a check meant that an object could connect its destroyed signal to itself so that it could monitor when its underlying C++ instance was destroyed. Unfortunately this turned out to be a potential source of obscure bugs for more common code.

In this version the check has been introduced - hence creating an incompatibility for any code that relies on the earlier behaviour. As a workaround for this the `no_receiver_check` argument has been added to connect which allows the check to be suppressed on a per connection basis.

Qt Signals with Default Arguments

In previous versions Qt signals with default arguments were exposed as multiple signals each with one additional default argument. For example `QAbstractButton::clicked(bool checked = false)` was exposed as `QAbstractButton::clicked(bool checked)` and `QAbstractButton::clicked()` where the former

was the default signal. It was therefore possible to index the latter by using an empty tuple as the key - although there was no benefit in doing so.

In this version only the signal with all arguments supplied is exposed. However the signal's `emit()` method still supports the default argument, i.e. when used normally the change should not be noticed.

Installing PyQt5

Both the GPL and commercial versions of PyQt5 can be built from source packages or installed from binary wheels. Although this section concentrates on PyQt5 itself it applies equally to the related projects (i.e. PyQtWebEngine, PyQt3D, PyQtChart, PyQtDataVisualization and PyQtPurchasing).

Understanding the Correct Version to Install

Historically the version number of PyQt bears no relation to the version of Qt supported. For example it wasn't even true that PyQt4 required Qt v4 as it would also build against Qt v5. People sometimes mistakenly believe that, for example, PyQt5 v5.13 is needed when building against Qt v5.13.

Qt uses semantic versioning when deciding on the version number of a release. In summary the major version is increased when a release includes incompatible changes, the minor version is increased when a release includes compatible changes, and the patch version is increased when a release includes no user-visible changes.

With PyQt5 the version number of PyQt5 is tied, to a certain extent, to the version of Qt v5 so that:

- The major version will always be **5**.
- For a particular minor version *n* it will build against any version of Qt v5, but will not support any new features introduced in Qt v5.*n+1* or later.
- It will support all the features of supported modules of Qt v5.*n* or earlier.
- Support for new modules may be added to PyQt5 at any time. This would result in a change of patch version only.
- The major and minor versions of the latest release of PyQt5 will be the same as the latest release of Qt v5.
- The patch versions of PyQt5 and Qt v5 are entirely unrelated to each other.

So, for example, PyQt5 v5.1 will build against Qt v5.2 but will not support any new features introduced in Qt v5.2. PyQt5 v5.1 will support all the features of supported modules of Qt v5.0 and those new features introduced in Qt v5.1.

In summary, you should always try and use the latest version of PyQt5 no matter what version of Qt v5 you are using.

Installing from Wheels

Wheels are the standard Python packaging format for pure Python or binary extension modules such as PyQt5. Only Python v3.5 and later are supported. Wheels are provide for 32- and 64-bit Windows, 64-bit macOS and 64-bit Linux. These correspond with the platforms for which The Qt Company provide binary installers.

Wheels are installed using the **pip** program that is included with current versions of Python.

Installing the GPL Version

To install the wheel for the GPL version of PyQt5, run:

```
pip install PyQt5
```

This will install the wheel for your platform and your version of Python (assuming both are supported). The wheel will be automatically downloaded from PyPI.

If you get an error message saying that no downloads could be found that satisfy the requirement then you are probably using an unsupported version of Python.

The PyQt5 wheel includes the necessary parts of the LGPL version of Qt. There is no need to install Qt yourself. You can use the **pyqt-bundle** program to create a new wheel with a different version of Qt bundled. See Bundling Qt Using pyqt-bundle for the full details of how to do this.

The sip module is packaged as a separate wheel which will be downloaded and installed automatically.

To uninstall the GPL version, run:

```
pip uninstall PyQt5
```

Note: Qt's support for TLS/SSL will not work on Windows when installing wheels that contain Qt v5.12.4 (or later) with Python v3.7.0 to v3.7.3. This is because of incompatibilities between the different versions of OpenSSL that these versions require. All other version combinations should be fine.

Installing the Commercial Version

It is not possible to provide wheels for the commercial version in the same way they are provided for the GPL version as it is not possible to distribute a copy of the commercial version of Qt. Therefore the **pyqt-bundle** program must be used to bundle your own copy of Qt with the provided commercial wheels.

Note: The old Riverbank Computing website provided *unlicensed* commercial wheels that required you to download and run the **pyqtlicense** program in order to create a *licensed* wheel. The new Riverbank Computing website provides pre-licensed wheels and there is no need to run **pyqtlicense**.

To uninstall the commercial version, run:

```
pip uninstall PyQt5-commercial
```

Building and Installing from Source

Starting with PyQt5 v5.14.0 **pip** can be used to download, build and install the GPL source packages from the PyQt5 project at PyPI. For this to work your **PATH** environment variable must contain your Qt installation's `bin` directory. If you do not do this then you will get a cryptic error message from **pip**.

However using **pip** to install from the source package is not recommended as it is not possible to configure the installation or to easily diagnose any problems. The rest of these instructions assume that you have downloaded the source package from PyPI and will used SIP's **sip-install** command line tool to do the build and installation.

If you are using the commercial version of PyQt5 then you should use the download instructions which were sent to you when you made your purchase. You must also download your `pyqt-commercial.sip` license file.

Installing Prerequisites

PyQt-builder extends the SIP build system and can be installed from PyPI by running:

```
pip install PyQt-builder
```

This will also automatically install SIP if required.

PyQt-builder extends the build system by adding options to SIP's command line tools.

PyQt5 further extends the build system by adding the following options to SIP's command line tools.

--confirm-license
Using this confirms that you accept the terms of the PyQt5 license. If it is omitted then you will be asked for confirmation during configuration.

--dbus `DIR`
The directory containing the `dbus/dbus-python.h` header file of the dbus-python package can be found in the directory `DIR`.

--license-dir `DIR`
The license files needed by the commercial version of PyQt5 can be found in the directory `DIR`.

--no-dbus-python
The Qt support for the dbus-python package will not be built.

--no-designer-plugin
The Qt Designer plugin will not be built.

--no-qml-plugin
The **qmlscene** plugin will not be built.

--no-tools
The **pyuic5**, **pyrcc5** and **pylupdate5** tools will not be built.

--qt-shared
Normally Qt is checked to see if it has been built as shared libraries. Some Linux distributions configure their Qt builds to make this check unreliable. This option ignores the result of the check and assumes that Qt has been built as shared libraries.

The Mercurial repository containing the latest development version of PyQt-builder can be found here.

Building the sip Module

It is not necessary to install the PyQt5.sip module before building PyQt5 but it must be installed before PyQt5 can be used.

The module is built using `setuptools` and is available from the PyQt5-sip project at PyPI. It uses `setuptools` as its build system and can be installed by **pip** or you can also unpack the sdist and install it by running its **setup.py** script.

Building PyQt5

Once you have downloaded the source package from PyPI, unpack it and change directory to its top level directory (i.e. the one containing the `pyproject.toml` file. To build and install PyQt5, run:

```
sip-install
```

In order to see all the available command line options, run:

```
sip-install -h
```

If you want to run **make** seperately then instead run:

```
sip-build --no-make
make
make install
```

Building PyQt5-related Projects

The additional PyQt5 projects (i.e. PyQtWebEngine, PyQt3D, PyQtChart, PyQtDataVisualization and PyQtPurchasing) are built and installed in exactly the same way as PyQt5 itself. PyQt5 must be built and installed first.

Bundling Qt Using pyqt-bundle

The wheels of the GPL version of PyQt5 and related projects on PyPI bundle a copy of the relevant parts of Qt. This is done so that users can install a complete PyQt environment with a single **pip** install.

The wheels of the commercial version of PyQt do not have a copy of Qt bundled because it is not possible to distribute a copy of the commercial version of Qt. Therefore a commercial user must bundle their own copy of Qt to create a complete wheel.

The **pyqt-bundle** program is provided as a means of bundling the relevant parts of a local Qt installation with a wheel, replacing any existing copy. You can also use it to produce a stripped down version of PyQt that contains only those modules you actually want to use. **pyqt-bundle** is part of PyQt-builder and is documented here.

Building PyQt5 with configure.py

Prior to the release of SIP v5 the only way to build PyQt5 (and related projects) was based on a **configure.py** script. This method is now deprecated and will be removed when SIP v6 is released (which is expected to be mid-2020).

configure.py supports both SIP v4 and SIP v5. Therefore you can move to SIP v5 without needing to change the way you build PyQt5 at the same time.

Installing Prerequisites

SIP

SIP v4 or SIP v5 must be installed before building and using PyQt5. If you are using SIP v5 you can simply install it using **pip**.

If you are using SIP v4 then you must build it from the source package from https://www.riverbankcomputing.com/software/sip/download.

> **Note:** When building PyQt5 v5.11 or later you must configure SIP v4 to create a private copy of the `sip` module using a command line similar to the following:
>
> ```
> python configure.py --sip-module PyQt5.sip
> ```
>
> If you already have SIP v4 installed and you just want to build and install the private copy of the module then add the `--no-tools` option.

Building PyQt5

Downloading

Starting with PyQt5 v5.14.0 the GPL source packages can be downloaded from the PyQt5 project at PyPI. You can download earlier releases from https://www.riverbankcomputing.com/software/pyqt/download5.

If you are using the commercial version of PyQt5 then you should use the download instructions which were sent to you when you made your purchase. You must also download your `pyqt-commercial.sip` license file.

Configuring

After unpacking the source package you should then check for any `README` files that relate to your platform.

If you are using the commercial version of PyQt5 then you must copy your `pyqt-commercial.sip` license file to the `sip` directory, or to the directory specified by the `--license-dir` option of **configure.py**.

You need to make sure your environment variables are set properly for your development environment.

In order to configure the build of PyQt5 you need to run the **configure.py** script as follows:

```
python3 configure.py
```

This assumes that the Python interpreter is on your path. Something like the following may be appropriate on Windows:

```
c:\Python38\python configure.py
```

If you have multiple versions of Python installed then make sure you use the interpreter for which you wish to build PyQt5 for.

The full set of command line options is:

-h, **--help**
Display a help message and exit.

--abi-version `VERSION`

New in version 5.12.3.

The sip module implements a versioned ABI and PyQt5 must be built to use a combatible version. The ABI version has a major number and a minor number separated by `.`. The ABI version used by PyQt5 must have the same major number and a minor number no larger than the minor number implemented by the sip module. By default PyQt5 will use the latest ABI version. The option is ignored unless SIP v5 is being used.

--allow-sip-warnings

New in version 5.9.1.

Normally any warning message generated by the SIP code generator is treated as an error. This option causes warning messages to be considered non-fatal. It is normally only required if a later version of the code generator is being used that has deprecated a feature used by this version of PyQt5.

--assume-shared
Normally Qt is checked to see if it has been built as shared libraries. Some Linux distributions configure their Qt builds to make this check unreliable. This option ignores the result of the check and assumes that Qt has been built as shared libraries.

--bindir `DIR`
The **pyuic5**, **pyrcc5** and **pylupdate5** utilities will be installed in the directory `DIR`.

--concatenate
The C++ source files for a Python module will be concatenated. This results in significantly reduced compilation times. Most, but not all, C++ compilers can handle the large files that result. See also the `--concatenate-split` option.

--concatenate-split N
If the `--concatenate` option is used to concatenate the C++ source files then this option determines how many files are created. The default is 1.

--configuration FILE
`FILE` contains the configuration of the PyQt5 build to be used instead of dynamically introspecting the system and is typically used when cross-compiling. See Configuring with Configuration Files.

--confirm-license
Using this confirms that you accept the terms of the PyQt5 license. If it is omitted then you will be asked for confirmation during configuration.

--dbus DIR
The `dbus-python.h` header file of the dbus-python package can be found in the directory `DIR/dbus`.

--debug
The PyQt5 modules will be built with debugging symbols. On Windows **configure.py** must be run using a debug version of Python.

--designer-plugindir DIR
The Python plugin for Qt Designer will be installed in the directory `DIR`.

--destdir DIR
The PyQt5 Python package will be installed in the directory `DIR`. The default is the Python installation's `site-packages` directory. If you use this option then the `PYTHONPATH` environment variable must include `DIR`.

--disable MODULE

New in version 5.5.1.

Normally all PyQt5 modules are enabled and are built if the corresponding Qt library can be found. This option will suppress the check for `MODULE`. The option may be specified any number of times.

--disable-feature FEATURE

New in version 5.10.1.

A PyQt5 module may be configured differently depending on the corresponding Qt configuration. This takes the form of a set of features that may be disabled. Normally this is determined automatically. This option will explicitly disable the `FEATURE` feature. The option may be specified any number of times.

--enable `MODULE`
Normally all PyQt5 modules are enabled and are built if the corresponding Qt library can be found. Using this option only those modules specifically enabled will be built. The option may be specified any number of times. Note that using this option suppresses the checks that are normally made to determine how the module should be configured, i.e. which features should be disabled.

--license-dir `DIR`
The license files needed by the commercial version of PyQt5 can be found in the directory `DIR`.

--link-full-dll

New in version 5.8.

On Windows the full Python API and the limited API (as used by PyQt) are implemented in different DLLs. Normally the limited DLL is linked (unless a debug version of the Python interpreter is being used to run **configure.py**). This option forces the full API DLL to be linked instead.

--no-designer-plugin
The Qt Designer plugin will not be built.

--no-dist-info

New in version 5.11.

This disables the creation of the PEP 376 `.dist-info` directory. Starting with this version a `.dist-info` directory is created. This contains meta-data about the installation including version information for dependent packages. It also means that **pip** can be used to uninstall the package.

--no-docstrings
The PyQt5 modules will not contain automatically generated docstrings.

--no-python-dbus
The Qt support for the standard Python DBus bindings is disabled.

--no-qml-plugin
The **qmlscene** plugin will not be built.

--no-qsci-api
The `PyQt5.api` QScintilla API file is not installed even if QScintilla does appear to be installed.

--no-sip-files
The `.sip` files for the PyQt5 modules will not be installed.

--no-stubs

New in version 5.6.

The PEP 484 type hint stub files for the PyQt5 modules will not be installed. This option is ignored (and the stub files are not installed) for versions of Python earlier than v3.5.

--no-tools

New in version 5.3.

The **pyuic5**, **pyrcc5** and **pylupdate5** tools will not be built.

--no-timestamp
Normally the header comments of each generated C/C++ source file includes a timestamp corresponding to when the file was generated. This option suppresses the inclusion of the timestamp.

--protected-is-public
On certain platforms the size of PyQt5 modules can be significantly reduced by redefining the C++ `protected` keyword as `public` during compilation. This option enables this behaviour and is the default on Linux and macOS.

--protected-not-public
The default redefinition of `protected` to `public` during compilation on Linux and macOS is disabled.

--pyuic5-interpreter `FILE`
`FILE` is the name of the Python interpreter used in the **pyuic5** wrapper. The default is platform dependent.

--qmake `FILE`
Qt's **qmake** program is used to determine how your Qt installation is laid out. Normally **qmake** is found on your `PATH`. This option can be used to specify a particular instance of **qmake** to use.

--qml-debug

New in version 5.8.

Enable the QML debugging infrastructure. This should not be enabled in a production environment.

--qml-plugindir `DIR`
The Python plugin for **qmlscene** will be installed in the directory `DIR`.

--qsci-api
The `PyQt5.api` QScintilla API file is installed even if QScintilla does not appear to be installed. This option is implied if the `--qsci-api-destdir` option is specified.

--qsci-api-destdir `DIR`
The QScintilla API file will be installed in the `python` subdirectory of the `api` subdirectory of the directory `DIR`.

--qtconf-prefix `DIR`

New in version 5.6.

A `qt.conf` file is embedded in the PyQt5.QtCore module with `Prefix` set to `DIR` which is assumed to be relative to the directory that the PyQt5.QtCore module will be installed in.

--sip `FILE`
The SIP code generator is used to generate PyQt5's C++ source code. Normally the code genertor is found on your **PATH**. This option can be used to specify a particular instance of the code genertor to use.

--sip-incdir `DIR`
The `sip.h` header file can be found in the directory `DIR`.

--sipdir `DIR`
The `.sip` files for the PyQt5 modules will be installed in the directory `DIR`.

--spec `SPEC`
The argument `-spec SPEC` will be passed to **qmake**. The default behaviour is platform specific. On Windows **configure.py** will choose the value that is correct for the version of Python that is being used. (However if you have built Python yourself then you may need to explicitly specify `SPEC`.) On macOS **configure.py** will try and avoid `macx-xcode` if possible.

--static
The PyQt5 modules will be built as static libraries. This is useful when building a custom interpreter with the PyQt5 modules built in to the interpreter.

--stubsdir `DIR`

New in version 5.6.

The PEP 484 type hint stub files for the PyQt5 modules will be installed in the directory `DIR`. By default they will be stored in the same directory where (by default) the corresponding extension modules would be installed. This option is ignored (and the stub files are not installed) for versions of Python earlier than v3.5.

--sysroot `DIR`

New in version 5.3.

`DIR` is the name of an optional directory that replaces `sys.prefix` in the names of other directories (specifically those specifying where the various PyQt5 components will be installed and where the Python include and library directories can be found). It is typically used when cross-compiling or when building a static version of PyQt5. See Configuring with Configuration Files.

--target-py-version `VERSION`

New in version 5.3.

`VERSION` is the major and minor version (e.g. `3.4`) of the version of Python being targetted. By default the version of Python being used to run the **configure.py** script is used. It is typically

used when cross-compiling. See Configuring with Configuration Files.

--trace
The generated PyQt5 modules contain additional tracing code that is enabled using SIP's `sip.settracemask()` function.

--verbose
Compiler commands and any output issued during configuration is displayed instead of being suppressed. Use this if **configure.py** is having problems to see what exactly is going wrong.

--version
Display the version number and exit.

Any remaining command line arguments are expected to be in the form `name=value` or `name+=value`. Such arguments are added to any **qmake** `.pro` file created by **configure.py**.

Building and Installing

The next step is to build PyQt5 by running your platform's **make** command. For example:

```
make
```

The final step is to install PyQt5 by running the following command:

```
make install
```

(Depending on your system you may require root or administrator privileges.)

This will install the various PyQt5 components.

Configuring with Configuration Files

The **configure.py** script normally introspects the Python installation of the interpreter running it in order to determine the names of the various files and directories it needs. This is fine for a native build of PyQt5 but isn't appropriate when cross-compiling. In this case it is possible to supply a configuration file, specified using the `--configuration` option, which contains definitions of all the required values.

A configuration file is made up of a number of named sections each of which contains a number of configuration items. The format of a configuration file is as follows:

- a section name is a single line with the name enclosed between `[` and `]`
- a configuration item is a single line containing a name/value pair separated by `=`
- values may be extended to lines immediately following if they are indented by at least one space
- a value may include another value by embedding the name of that value enclosed between `%(` and `)`
- comments begin with `#` and continue to the end of the line
- blank lines are ignored.

Those configuration items that appear before the first section name are automatically added to all sections.

A configuration file defines a section for each version of Qt that requires a different configuration. **configure.py** will choose the most appropriate section according to the version of Qt you are actually using. For example, if a configuration file contains sections for Qt v5.3 and Qt v5.1 and you are using Qt v5.2.1 then the section for Qt v5.1 will be chosen.

configure.py provides the following preset values for a configuration:

`py_major`
: is the major version number of the target Python installation.

`py_minor`
: is the minor version number of the target Python installation.

`sysroot`
: is the name of the system root directory. This is specified with the `--sysroot` option.

The following is an example configuration file:

```
# The target Python installation.
py_platform = linux
py_inc_dir = %(sysroot)/usr/include/python%(py_major).%(py_minor)
py_pylib_dir = %(sysroot)/usr/lib/python%(py_major).%(py_minor)/config
py_pylib_lib = python%(py_major).%(py_minor)mu

# The target PyQt installation.
pyqt_module_dir = %(sysroot)/usr/lib/python%(py_major)/dist-packages
pyqt_bin_dir = %(sysroot)/usr/bin
pyqt_sip_dir = %(sysroot)/usr/share/sip/PyQt5
pyuic_interpreter = /usr/bin/python%(py_major).%(py_minor)
pyqt_disabled_features = PyQt_Desktop_OpenGL PyQt_qreal_double

# Qt configuration common to all versions.
qt_shared = True

[Qt 5.1]
pyqt_modules = QtCore QtDBus QtDesigner QtGui QtHelp QtMultimedia
    QtMultimediaWidgets QtNetwork QtOpenGL QtPrintSupport QtQml QtQuick
    QtSensors QtSerialPort QtSql QtSvg QtTest QtWebKit QtWebKitWidgets
    QtWidgets QtXmlPatterns _QOpenGLFunctions_ES2
```

This example contains a section for Qt v5.1. We have defined a number of values before the start of the section as they are not specific to any particular version of Qt. Note that if you use this configuration with a version of Qt earlier than v5.1 then you will get an error.

The following values can be specified in the configuration file:

`qt_shared`
: is set if Qt has been built as shared libraries. The default value is `False`.

`py_platform`
: is the target Python platform.

`py_debug`
: is set if a debug version of the target Python is being used.

`py_inc_dir`
: is the target Python include directory, i.e. the directory containing the `Python.h` file.

`py_pylib_dir`
: is the target Python library directory.

`py_pylib_lib`
: is the target Python interpreter library. It should not include any platform-specific prefix or suffix.

`pyqt_disabled_features`
: is the space separated list of features (as defined by SIP's `%Feature` directive) that should be disabled.

`pyqt_module_dir`
: is the target directory where the PyQt5 modules will be installed. It can be overridden by the `--destdir` option.

`pyqt_modules`
: is the space separated list of PyQt5 modules that will be built. It can be overridden by the `--enable` option.

`pyqt_bin_dir`
: is the name of the target directory where the PyQt5 related executables will be installed. It can be overridden by the `--bindir` option.

`pyqt_sip_dir`
: is the name of the target directory where the PyQt5 `.sip` files will be installed. It can be overridden by the `--sipdir` option.

`pyuic_interpreter`
: is the name of the Python interpreter (as it would be called from the target system) that will be used to run **pyuic5**. It can be overridden by the `--pyuic5-interpreter` option.

Building PyQt5-related Projects

The additional PyQt5 projects (i.e. PyQtWebEngine, PyQt3D, PyQtChart, PyQtDataVisualization and PyQtPurchasing) are built and installed in exactly the same way as PyQt5 itself. In other words the source packages contain a `configure.py` script.

Support for Signals and Slots

One of the key features of Qt is its use of signals and slots to communicate between objects. Their use encourages the development of reusable components.

A signal is emitted when something of potential interest happens. A slot is a Python callable. If a signal is connected to a slot then the slot is called when the signal is emitted. If a signal isn't connected then nothing happens. The code (or component) that emits the signal does not know or care if the signal is being used.

The signal/slot mechanism has the following features.

- A signal may be connected to many slots.
- A signal may also be connected to another signal.
- Signal arguments may be any Python type.
- A slot may be connected to many signals.
- Connections may be direct (ie. synchronous) or queued (ie. asynchronous).
- Connections may be made across threads.
- Signals may be disconnected.

Unbound and Bound Signals

A signal (specifically an unbound signal) is a class attribute. When a signal is referenced as an attribute of an instance of the class then PyQt5 automatically binds the instance to the signal in order to create a *bound signal*. This is the same mechanism that Python itself uses to create bound methods from class functions.

A bound signal has `connect()`, `disconnect()` and `emit()` methods that implement the associated functionality. It also has a `signal` attribute that is the signature of the signal that would be returned by Qt's `SIGNAL()` macro.

A signal may be overloaded, ie. a signal with a particular name may support more than one signature. A signal may be indexed with a signature in order to select the one required. A signature is a sequence of types. A type is either a Python type object or a string that is the name of a C++ type. The name of a C++ type is automatically normalised so that, for example, `QVariant` can be used instead of the non-normalised `const QVariant &`.

If a signal is overloaded then it will have a default that will be used if no index is given.

When a signal is emitted then any arguments are converted to C++ types if possible. If an argument doesn't have a corresponding C++ type then it is wrapped in a special C++ type that allows it to be passed around Qt's meta-type system while ensuring that its reference count is properly maintained.

Defining New Signals with pyqtSignal

PyQt5 automatically defines signals for all Qt's built-in signals. New signals can be defined as class attributes using the pyqtSignal factory.

PyQt5.QtCore.**pyqtSignal**(*types*[, *name*[, *revision=0*[, *arguments=[]*]]])

Create one or more overloaded unbound signals as a class attribute.

Parameters:
- **types** – the types that define the C++ signature of the signal. Each type may be a Python type object or a string that is the name of a C++ type. Alternatively each may be a sequence of type arguments. In this case each sequence defines the signature of a different signal overload. The first overload will be the default.
- **name** – the name of the signal. If it is omitted then the name of the class attribute is used. This may only be given as a keyword argument.
- **revision** – the revision of the signal that is exported to QML. This may only be given as a keyword argument.
- **arguments** – the sequence of the names of the signal's arguments that is exported to QML. This may only be given as a keyword argument.

Return type: an unbound signal

The following example shows the definition of a number of new signals:

```
from PyQt5.QtCore import QObject, pyqtSignal

class Foo(QObject):

    # This defines a signal called 'closed' that takes no arguments.
    closed = pyqtSignal()

    # This defines a signal called 'rangeChanged' that takes two
    # integer arguments.
    range_changed = pyqtSignal(int, int, name='rangeChanged')

    # This defines a signal called 'valueChanged' that has two overloads,
    # one that takes an integer argument and one that takes a QString
    # argument.  Note that because we use a string to specify the type of
    # the QString argument then this code will run under Python v2 and v3.
    valueChanged = pyqtSignal([int], ['QString'])
```

New signals should only be defined in sub-classes of QObject. They must be part of the class definition and cannot be dynamically added as class attributes after the class has been defined.

New signals defined in this way will be automatically added to the class's QMetaObject. This means that they will appear in Qt Designer and can be introspected using the QMetaObject API.

Overloaded signals should be used with care when an argument has a Python type that has no corresponding C++ type. PyQt5 uses the same internal C++ class to represent such objects and so it is possible to have overloaded signals with different Python signatures that are implemented with identical C++ signatures with unexpected results. The following is an example of this:

```
class Foo(QObject):

    # This will cause problems because each has the same C++ signature.
    valueChanged = pyqtSignal([dict], [list])
```

Connecting, Disconnecting and Emitting Signals

Signals are connected to slots using the `connect()` method of a bound signal.

connect(*slot*[, *type=PyQt5.QtCore.Qt.AutoConnection*[, *no_receiver_check=False*]]) → PyQt5.QtCore.QMetaObject.Connection

Connect a signal to a slot. An exception will be raised if the connection failed.

Parameters:
- **slot** – the slot to connect to, either a Python callable or another bound signal.
- **type** – the type of the connection to make.
- **no_receiver_check** – suppress the check that the underlying C++ receiver instance still exists and deliver the signal anyway.

Returns: a Connection object which can be passed to `disconnect()`. This is the only way to disconnect a connection to a lambda function.

Signals are disconnected from slots using the `disconnect()` method of a bound signal.

disconnect([*slot*])

Disconnect one or more slots from a signal. An exception will be raised if the slot is not connected to the signal or if the signal has no connections at all.

Parameters: **slot** – the optional slot to disconnect from, either a Connection object returned by `connect()`, a Python callable or another bound signal. If it is omitted then all slots connected to the signal are disconnected.

Signals are emitted from using the `emit()` method of a bound signal.

emit(**args*)

Emit a signal.

Parameters: **args** – the optional sequence of arguments to pass to any connected slots.

The following code demonstrates the definition, connection and emit of a signal without arguments:

```
from PyQt5.QtCore import QObject, pyqtSignal

class Foo(QObject):

    # Define a new signal called 'trigger' that has no arguments.
    trigger = pyqtSignal()

    def connect_and_emit_trigger(self):
        # Connect the trigger signal to a slot.
        self.trigger.connect(self.handle_trigger)

        # Emit the signal.
        self.trigger.emit()

    def handle_trigger(self):
        # Show that the slot has been called.

        print "trigger signal received"
```

The following code demonstrates the connection of overloaded signals:

```
from PyQt5.QtWidgets import QComboBox

class Bar(QComboBox):

    def connect_activated(self):
        # The PyQt5 documentation will define what the default overload is.
        # In this case it is the overload with the single integer argument.
        self.activated.connect(self.handle_int)

        # For non-default overloads we have to specify which we want to
        # connect.  In this case the one with the single string argument.
        # (Note that we could also explicitly specify the default if we
        # wanted to.)
        self.activated[str].connect(self.handle_string)

    def handle_int(self, index):
        print "activated signal passed integer", index

    def handle_string(self, text):
        print "activated signal passed QString", text
```

Connecting Signals Using Keyword Arguments

It is also possible to connect signals by passing a slot as a keyword argument corresponding to the name of the signal when creating an object, or using the `pyqtConfigure()` method. For example the following three fragments are equivalent:

```
act = QAction("Action", self)
act.triggered.connect(self.on_triggered)

act = QAction("Action", self, triggered=self.on_triggered)

act = QAction("Action", self)
act.pyqtConfigure(triggered=self.on_triggered)
```

The pyqtSlot() Decorator

Although PyQt5 allows any Python callable to be used as a slot when connecting signals, it is sometimes necessary to explicitly mark a Python method as being a Qt slot and to provide a C++ signature for it. PyQt5 provides the pyqtSlot() function decorator to do this.

PyQt5.QtCore.**pyqtSlot**(*types*[, *name*[, *result*[, *revision=0*]]])

Decorate a Python method to create a Qt slot.

Parameters:

- **types** – the types that define the C++ signature of the slot. Each type may be a Python type object or a string that is the name of a C++ type.
- **name** – the name of the slot that will be seen by C++. If omitted the name of the Python method being decorated will be used. This may only be given as a keyword argument.
- **revision** – the revision of the slot that is exported to QML. This may only be given as a keyword argument.
- **result** – the type of the result and may be a Python type object or a string that specifies a C++ type. This may only be given as a keyword argument.

Connecting a signal to a decorated Python method also has the advantage of reducing the amount of memory used and is slightly faster.

For example:

```
from PyQt5.QtCore import QObject, pyqtSlot

class Foo(QObject):

    @pyqtSlot()
    def foo(self):
        """ C++: void foo() """

    @pyqtSlot(int, str)
    def foo(self, arg1, arg2):
        """ C++: void foo(int, QString) """

    @pyqtSlot(int, name='bar')
    def foo(self, arg1):
        """ C++: void bar(int) """

    @pyqtSlot(int, result=int)
    def foo(self, arg1):
        """ C++: int foo(int) """

    @pyqtSlot(int, QObject)
    def foo(self, arg1):
        """ C++: int foo(int, QObject *) """
```

It is also possible to chain the decorators in order to define a Python method several times with different signatures. For example:

```
from PyQt5.QtCore import QObject, pyqtSlot

class Foo(QObject):

    @pyqtSlot(int)
    @pyqtSlot('QString')
    def valueChanged(self, value):
        """ Two slots will be defined in the QMetaObject. """
```

The `PyQt_PyObject` Signal Argument Type

It is possible to pass any Python object as a signal argument by specifying `PyQt_PyObject` as the type of the argument in the signature. For example:

```
finished = pyqtSignal('PyQt_PyObject')
```

This would normally be used for passing objects where the actual Python type isn't known. It can also be used to pass an integer, for example, so that the normal conversions from a Python object to a C++ integer and back again are not required.

The reference count of the object being passed is maintained automatically. There is no need for the emitter of a signal to keep a reference to the object after the call to `finished.emit()`, even if a connection is queued.

Connecting Slots By Name

PyQt5 supports the `connectSlotsByName()` function that is most commonly used by **pyuic5** generated Python code to automatically connect signals to slots that conform to a simple naming convention. However, where a class has overloaded Qt signals (ie. with the same name but with different arguments) PyQt5 needs additional information in order to automatically connect the correct signal.

For example the QSpinBox class has the following signals:

```
void valueChanged(int i);
void valueChanged(const QString &text);
```

When the value of the spin box changes both of these signals will be emitted. If you have implemented a slot called `on_spinbox_valueChanged` (which assumes that you have given the QSpinBox instance the name `spinbox`) then it will be connected to both variations of the signal. Therefore, when the user changes the value, your slot will be called twice - once with an integer argument, and once with a string argument.

The pyqtSlot() decorator can be used to specify which of the signals should be connected to the slot.

For example, if you were only interested in the integer variant of the signal then your slot definition would look like the following:

```
@pyqtSlot(int)
def on_spinbox_valueChanged(self, i):
    # i will be an integer.
    pass
```

If you wanted to handle both variants of the signal, but with different Python methods, then your slot definitions might look like the following:

```
@pyqtSlot(int, name='on_spinbox_valueChanged')
def spinbox_int_value(self, i):
    # i will be an integer.
    pass

@pyqtSlot(str, name='on_spinbox_valueChanged')
def spinbox_qstring_value(self, s):
    # s will be a Python string object (or a QString if they are enabled).
    pass
```

Support for Qt Properties

PyQt5 does not support the setting and getting of Qt properties as if they were normal instance attributes. This is because the name of a property often conflicts with the name of the property's getter method.

However, PyQt5 does support the initial setting of properties using keyword arguments passed when an instance is created. For example:

```
act = QAction("&Save", self, shortcut=QKeySequence.Save,
        statusTip="Save the document to disk", triggered=self.save)
```

The example also demonstrates the use of a keyword argument to connect a signal to a slot.

PyQt5 also supports setting the values of properties (and connecting a signal to a slot) using the **`pyqtConfigure()`** method. For example, the following gives the same results as above:

```
act = QAction("&Save", self)
act.pyqtConfigure(shortcut=QKeySequence.Save,
        statusTip="Save the document to disk", triggered=self.save)
```

Defining New Qt Properties

A new Qt property may be defined using the pyqtProperty function. It is used in the same way as the standard Python `property()` function. In fact, Qt properties defined in this way also behave as Python properties.

`PyQt5.QtCore.` **`pyqtProperty`**(*type*[, *fget=None*[, *fset=None*[, *freset=None*[, *fdel=None*[, *doc=None*[, *designable=True*[, *scriptable=True*[, *stored=True*[, *user=False*[, *constant=False*[, *final=False*[, *notify=None*[, *revision=0*]]]]]]]]]]]]])

Create a property that behaves as both a Python property and a Qt property.

Parameters:
- **type** – the type of the property. It is either a Python type object or a string that is the name of a C++ type.
- **fget** – the optional callable used to get the value of the property.
- **fset** – the optional callable used to set the value of the property.
- **freset** – the optional callable used to reset the value of the property to its default value.
- **fdel** – the optional callable used to delete the property.
- **doc** – the optional docstring of the property.
- **designable** – optionally sets the Qt `DESIGNABLE` flag.
- **scriptable** – optionally sets the Qt `SCRIPTABLE` flag.
- **stored** – optionally sets the Qt `STORED` flag.
- **user** – optionally sets the Qt `USER` flag.
- **constant** – optionally sets the Qt `CONSTANT` flag.
- **final** – optionally sets the Qt `FINAL` flag.
- **notify** – the optional unbound notify signal.

- **revision** – the revision exported to QML.

Return type: the property object.

It is also possible to use pyqtProperty as a decorator in the same way as the standard Python `property()` function. The following example shows how to define an `int` property with a getter and setter:

```
from PyQt5.QtCore import QObject, pyqtProperty

class Foo(QObject):

    def __init__(self):
        QObject.__init__(self)

        self._total = 0

    @pyqtProperty(int)
    def total(self):
        return self._total

    @total.setter
    def total(self, value):
        self._total = value
```

If you prefer the Qt terminology you may also use `write` instead of `setter` (and `read` instead of `getter`).

Other Support for Dynamic Meta-objects

PyQt5 creates a QMetaObject instance for any Python sub-class of QObject without the need for the equivalent of Qt's `Q_OBJECT` macro. Most of a QMetaObject is populated automatically by defining signals, slots and properties as described in previous sections. In this section we cover the ways in which the remaining parts of a QMetaObject are populated.

> **Note:** Q_ENUM(), Q_FLAG() and Q_CLASSINFO() are not available when PyQt5 is built for PyPy.

Q_ENUM() and Q_FLAG()

New in version 5.11.

The Q_ENUM() and Q_FLAG() functions declare enumerated types and flag types respectively that are published in the QMetaObject. The typical use in PyQt5 is to declare symbolic constants that can be used by QML, and as type of properties that can be set in Qt Designer.

Each function takes a Python type object or an `Enum` object that implements the enumerated or flag type. For example:

```
from enum import Enum

from PyQt5.QtCore import Q_ENUM, Q_FLAG, QObject


class Instruction(QObject):

    class Direction(Enum):
        Up, Down, Left, Right = range(4)

    Q_ENUM(Direction)

    class Status:
        Null = 0x00
        Urgent = 0x01
        Acknowledged = 0x02
        Completed = 0x04

    Q_FLAG(Status)
```

New in version 5.2.

The (now deprecated) Q_ENUMS() and Q_FLAGS() functions are also provided. These differ from the above in that they can define multiple types in one invocation.

Q_CLASSINFO()

The Q_CLASSINFO() function is used in the same way as Qt's macro of the same name, i.e. it is called from a class's definition in order to specify a name/value pair that is placed in the class's QMetaObject.

For example it is used by QML to define the default property of a class:

```
from PyQt5.QtCore import Q_CLASSINFO, QObject


class BirthdayParty(QObject):

    Q_CLASSINFO('DefaultProperty', 'guests')
```

Support for OpenGL

When compiled against Qt v5.1 or later, PyQt5 implements a set of either desktop QOpenGL bindings or OpenGL ES v2 bindings depending on how Qt was configured. This removes the dependency on any third-party OpenGL bindings such as PyOpenGL.

At the moment the desktop bindings are for OpenGL v2.0 and are mostly complete. Other versions will be added in later releases. If there are calls which you need, but are currently unsupported, then please ask for the support to be added.

Obtaining an object that implements the bindings for a particular OpenGL version and profile is done in the same way as it is done from C++, i.e. by calling `versionFunctions()`. In addition, the bindings object also contains attributes corresponding to all of the OpenGL constants.

Support for Qt Interfaces

PyQt5 does not, generally, support defining a class that inherits from more than one Qt class. The exception is when inheriting from classes that Qt defines as *interfaces*, for example QTextObjectInterface.

A Qt interface is an abstract class contains only pure virtual methods and is used as a mixin with (normally) a QObject sub-class. It is often used to define the interface that a plugin must implement.

Note that PyQt5 does not need an equivalent of Qt's `Q_INTERFACES` macro in order to use an interface class.

The `textobject.py` example includedd with PyQt5 demonstrates the use of an interface.

Support for QVariant

PyQt4 implements two APIs for QVariant. v1 (the default for Python v2) exposes the QVariant class to Python and requires applications to explicitly convert a QVariant to the actual value. v2 (the default for Python v3) does not expose the QVariant class to Python and automatically converts a QVariant to the actual value. While this is usually the best thing to do, it does raise problems of its own:

- Information is lost when converting between a C++ QVariant and the corresponding Python object. For example a QVariant distinguishes between signed and unsigned integers but Python doesn't. Normally this doesn't matter but some applications may need to make the distinction.
- There is no obvious way to represent a null QVariant as a standard Python object. PyQt4 introduced the `QPyNullVariant` class to address this problem.

Multiple APIs are intended to help manage an application's use of an old API to a newer, incompatible API. They cannot be used to temporarily change the behaviour - modules that rely on different API versions cannot be used in the same application.

In PyQt5 the implementation of QVariant is different to those of PyQt4. By default the behaviour is the same as PyQt4's v2 API. However it is possible to temporarily suppress the automatic conversion of a C++ QVariant to a Python object and to return a wrapped Python QVariant instead - behaviour similar to PyQt4's v1 API - by calling the **`sip.enableautoconversion()`** function.

The actual value of a wrapped Python QVariant is obtained by calling its **`value()`** method. (Note that in PyQt4's v1 API this method is called `toPyObject()`.)

An invalid QVariant is automatically converted to `None` and vice versa.

PyQt5 does not support the `QPyNullVariant` class.

Support for QSettings

Qt provies the QSettings class as a platform independent API for the persistent storage and retrieval of application settings. Settings are retrieved using the **value()** method. However the type of the value returned may not be what is expected. Some platforms only ever store string values which means that the type of the original value is lost. Therefore a setting with an integer value of `42` may be retrieved (on some platforms) as a string value of `'42'`.

As a solution to this problem PyQt5's implementation of **value()** takes an optional third argument called `type`. This is either a Python type object, e.g. `int`, or a string that is the name of a C++ type, e.g. `'QStringList'`. The value returned will be an object of the requested type.

For example:

```
from PyQt5.QtCore import QSettings, QPoint

settings = QSettings('foo', 'foo')

settings.setValue('int_value', 42)
settings.setValue('point_value', QPoint(10, 12))

# This will write the setting to the platform specific storage.
del settings

settings = QSettings('foo', 'foo')

int_value = settings.value('int_value', type=int)
print("int_value: %s" % repr(int_value))

point_value = settings.value('point_value', type=QPoint)
print("point_value: %s" % repr(point_value))
```

When this is executed then the following will be displayed for all platforms:

```
int_value: 42
point_value: PyQt5.QtCore.QPoint(10, 20)
```

If the value of the setting is a container (corresponding to either `QVariantList`, `QVariantMap` or `QVariantHash`) then the type is applied to the contents of the container.

For example:

```
from PyQt5.QtCore import QSettings

settings = QSettings('foo', 'foo')

settings.setValue('list_value', [1, 2, 3])
settings.setValue('dict_value', {'one': 1, 'two': 2})

# This will write the setting to the platform specific storage.
del settings

settings = QSettings('foo', 'foo')

list_value = settings.value('list_value', type=int)
print("list_value: %s" % repr(list_value))
```

```
dict_value = settings.value('dict_value', type=int)
print("dict_value: %s" % repr(dict_value))
```

When this is executed then the following will be displayed for all platforms:

```
list_value: [1, 2, 3]
dict_value: {'one': 1, 'two': 2}
```

Integrating Python and QML

Qt includes QML as a means of declaratively describing a user interface and using JavaScript as a scripting language within it. It is possible to write complete standalone QML applications, or to combine them with C++. PyQt5 allows QML to be integrated with Python in exactly the same way. In particular:

- Python types that are sub-classed from QObject can be registered with QML.
- Instances of registered Python types can be created and made available to QML scripts.
- Instances of registered Python types can be created by QML scripts.
- Singleton instances of registered Python types can be created automatically by a QML engine and made available to QML scripts.
- QML scripts interact with Python objects through their properties, signals and slots.
- Python properties, signals and slots can be given revision numbers that only those implemented by a specific version are made available to QML.

Note: The PyQt support for QML requires knowledge of the internals of the C++ code that implements QML. This can (and does) change between Qt versions and may mean that some features only work with specific Qt versions and may not work at all with some future version of Qt.

It is recommended that, in an MVC architecture, QML should only be used to implement the view. The model and controller should be implemented in Python.

Registering Python Types

Registering Python types with QML is done in the same way is it is done with C++ classes, i.e. using the qmlRegisterType(), qmlRegisterSingletonType(), qmlRegisterUncreatableType() and qmlRegisterRevision() functions.

In C++ these are template based functions that take the C++ class, and sometimes a revision, as template arguments. In the Python implementation these are simply passed as the first arguments to the respective functions.

A Simple Example

The following simple example demonstates the implementation of a Python class that is registered with QML. The class defines two properties. A QML script is executed which creates an instance of the class and sets the values of the properties. That instance is then returned to Python which then prints the values of those properties.

Hopefully the comments are self explanatory:

```
import sys

from PyQt5.QtCore import pyqtProperty, QCoreApplication, QObject, QUrl
from PyQt5.QtQml import qmlRegisterType, QQmlComponent, QQmlEngine
```

```
# This is the type that will be registered with QML.  It must be a
# sub-class of QObject.
class Person(QObject):
    def __init__(self, parent=None):
        super().__init__(parent)

        # Initialise the value of the properties.
        self._name = ''
        self._shoeSize = 0

    # Define the getter of the 'name' property.  The C++ type of the
    # property is QString which Python will convert to and from a string.
    @pyqtProperty('QString')
    def name(self):
        return self._name

    # Define the setter of the 'name' property.
    @name.setter
    def name(self, name):
        self._name = name

    # Define the getter of the 'shoeSize' property.  The C++ type and
    # Python type of the property is int.
    @pyqtProperty(int)
    def shoeSize(self):
        return self._shoeSize

    # Define the setter of the 'shoeSize' property.
    @shoeSize.setter
    def shoeSize(self, shoeSize):
        self._shoeSize = shoeSize


# Create the application instance.
app = QCoreApplication(sys.argv)

# Register the Python type.  Its URI is 'People', it's v1.0 and the type
# will be called 'Person' in QML.
qmlRegisterType(Person, 'People', 1, 0, 'Person')

# Create a QML engine.
engine = QQmlEngine()

# Create a component factory and load the QML script.
component = QQmlComponent(engine)
component.loadUrl(QUrl('example.qml'))

# Create an instance of the component.
person = component.create()

if person is not None:
    # Print the value of the properties.
    print("The person's name is %s." % person.name)
    print("They wear a size %d shoe." % person.shoeSize)
else:
    # Print all errors that occurred.
    for error in component.errors():
        print(error.toString())
```

The following is the `example.qml` QML script that is executed:

```
import People 1.0
```

```
Person {
    name: "Bob Jones"
    shoeSize: 12
}
```

Using QQmlListProperty

Defining list-based properties in Python that can be updated from QML is done using the QQmlListProperty class. However the way it is used in Python is slightly different to the way it is used in C++.

In the simple case QQmlListProperty wraps a Python list that is usually an instance sttribute, for example:

```
class BirthdayParty(QObject):

    def __init__(self, parent=None):
        super().__init__(parent)

        # The list which will be accessible from QML.
        self._guests = []

    @pyqtProperty(QQmlListProperty)
    def guests(self):
        return QQmlListProperty(Person, self, self._guests)
```

QML can now manipulate the Python list of `Person` instances. QQmlListProperty also acts as a proxy for the Python list so that the following can be written:

```
for guest in party.guests:
    print("Guest:", guest.name)
```

QQmlListProperty can also be used to wrap a *virtual* list. The following code fragment is taken from the `chapter5-listproperties.py` example included with PyQt5:

```
class PieChart(QQuickItem):

    @pyqtProperty(QQmlListProperty)
    def slices(self):
        return QQmlListProperty(PieSlice, self,
                append=lambda pie_ch, pie_sl: pie_sl.setParentItem(pie_ch))
```

`PieChart` and `PieSlice` are Quick items that are registered using qmlRegisterType(). Instances of both can be created from QML. `slices` is a property of `PieChart` that, as far as QML is concerned, is a list of `PieSlice` instances.

The pyqtProperty decorator specifies that the property is a QQmlListProperty, that its name is `slices` and that the `slices()` function is its getter.

The getter returns an instance of QQmlListProperty. This specifies that elements of the list should be of type `PieSlice`, that the `PieChart` instance (i.e. `self`) has the property, and defines the callable that will be invoked in order to append a new element to the list.

The `append` callable is passed two arguments: the object whose property is to be updated (i.e. the `PyChart` instance), and the element to be appended (i.e. a `PieSlice` instance). Here we simply set the chart as the slice's parent item. Note that there isn't actually a list anywhere - this is because, in this particular example, one isn't needed.

The signature of the `append` callable is slightly different to that of the corresponding C++ function. In C++ the first argument is the QQmlListProperty instance rather than the `PyChart` instance. The signatures of the `at`, `clear` and `count` callables are different in the same way.

Using Attached Properties

In order to use attached properties in C++, three steps need to be taken.

- A type that has attached properties must implement a static function called `qmlAttachedProperties`. This is a factory that creates an instance of the properties object to attach.
- A type that has attached properties needs to be defined as such using the `QML_DECLARE_TYPEINFO` macro with the `QML_HAS_ATTACHED_PROPERTIES` argument.
- The instance of an attached properties object is retrieved using the `qmlAttachedPropertiesObject()` template function. The template type is the type that has the attached properties.

PyQt5 uses similar, but slightly simpler steps to achieve the same thing.

- When calling qmlRegisterType() to register a type that has attached properties the type of the properties object is passed as the `attachedProperties` argument. This type will be used as the factory for creating an instance of the properties object.
- The instance of an attached properties object is retrieved using the qmlAttachedPropertiesObject() function in the same way that you would from C++. Just like qmlRegisterType(), qmlAttachedPropertiesObject() takes an additional first argument that is the type that, in C++, would be the template argument.

See the `attach.py` example included with PyQt5 for a complete example showing the use of attached properties.

Using Property Value Sources

Property values sources are implemented in PyQt5 in the same way as they are implemented in C++. Simply sub-class from both QObject and QQmlPropertyValueSource and provide an implementation of the setTarget() method.

Using QQmlParserStatus

Monitoring the QML parser status is implemented in PyQt5 in the same way as it is implemented in C++. Simply sub-class from both QObject and QQmlParserStatus and provide implementations of the classBegin() and componentComplete() methods.

Writing Python Plugins for **qmlscene**

Qt allows plugins that implement QML modules to be written that can be dynamically loaded by a C++ application (e.g. **qmlscene**). These plugins are sub-classes of QQmlExtensionPlugin. PyQt5 supports exactly the same thing and allows those plugin to be written in Python. In other words it is possible to provide QML extensions written in Python to a C++ application, and to provide QML extensions written in C++ to a Python application.

PyQt5 provides a QML plugin called `pyqt5qmlplugin`. This acts as a wrapper around the Python code that implements the plugin. It handles the loading of the Python interpreter, locating and importing the Python module that contains the implementation of QQmlExtensionPlugin, creating an instance of that class, and calling the instance's registerTypes() method. By default the `pyqt5qmlplugin` is installed in the `PyQt5` sub-directory of your Qt installation's `plugin` directory.

> **Note:** `pyqt5qmlplugin` is the name of the plugin as seen by QML. Its actual filename will be different and operating system dependent.

A QML extension module is a directory containing a file called `qmldir`. The file contains the name of the module and the name of the plugin that implements the module. It may also specify the directory containing the plugin. Usually this isn't needed because the plugin is installed in the same directory.

Therefore, for a QML extension module called `Charts`, the contents of the `qmldir` file might be:

```
module Charts
plugin pyqt5qmlplugin /path/to/qt/plugins/PyQt5
```

The `pyqt5qmlplugin` expects to find a Python module in the same directory with a filename ending with `plugin.py` or `plugin.pyw`. In this case the name `chartsplugin.py` would be a sensible choice. Before importing this module `pyqt5qmlplugin` first places the name of the directory at the start of **`sys.path`**.

> **Note:** `pyqt5qmlplugin` has to locate the directory containing the `qmldir` file itself. It does this using the same algorithm used by QML, i.e. it searches some standard locations and locations specified by the **`QML2_IMPORT_PATH`** environment variable. When using **qmlscene**, `pyqt5qmlplugin` will not know about any additional locations specified by its `-I` option. Therefore, **`QML2_IMPORT_PATH`** should always be used to specify additional locations to search.

Due to a limitation in QML it is not possible for multiple QML modules to use the same C++ plugin. In C++ this is not a problem as there is a one-to-one relationship between a module and the plugin. However, when using Python, `pyqt5qmlplugin` is used by every module. There are two solutions to this:

- on operating systems that support it, place a symbolic link in the directory containing the `qmldir` file that points to the actual `pyqt5qmlplugin`
- make a copy of `pyqt5qmlplugin` in the directory containing the `qmldir` file.

In both cases the contents of the `qmldir` file can be simplifed to:

```
module Charts
plugin pyqt5qmlplugin
```

PyQt5 provides an example that can be run as follows:

```
cd /path/to/examples/quick/tutorials/extending/chapter6-plugins
QML2_IMPORT_PATH=. /path/to/qmlscene app.qml
```

On Linux you may also need to set a value for the **LD_LIBRARY_PATH** environment variable.

Support for Cooperative Multi-inheritance

> **Note:** This section is not about sub-classing from more that one Qt class.

Cooperative multi-inheritance is a technique for implementing classes that inherit multiple super-classes - typically a main super-class and one or more mixin classes that add additional behaviour. It makes it easy to add new mixins at a later date to further extend the behavior, without needing to change either the implementation of the class or any existing code that creates an instance of the class.

The technique requires that all the super-class's `__init__` methods follow the same pattern in the way that they handle unrecognised keyword arguments and use `super()` to invoke their own super-class's `__init__` methods.

PyQt5's classes follow this pattern.

See Raymond Hettinger's Python's super() considered super! blog post for some more background on the subject.

As an example, let's say we have a class that represents a person, and that a person has a name. The following might be an initial implementation:

```
class Person(QObject):
    def __init__(self, name, parent=None)
        QObject.__init__(self, parent)

        self.name = name
```

An instance would normally be created in one of the following ways:

```
person = Person("Joe")
person = Person("Joe", some_parent)
```

This approach has some limitations:

- Only a sub-set of the QObject API is exposed. For example you cannot set the value of a Qt property or connect a signal by passing appropriate keyword arguments to `Person.__init__`.
- Adding another class to `Person`'s list of super-classes means that its `__init__` implementation needs to be changed. If the new mixin takes non-optional arguments then every call to create a `Person` instance will need changing.

Consider this alternative implementation:

```
class Person(QObject):
    def __init__(self, name, **kwds):
        super().__init__(**kwds)

        self.name = name
```

The difference is that we only handle arguments that are used by the `Person` class itself and we punt all the other arguments to the super-classes by calling `super()`.

With this implementation an instance would normally be created in one of the following ways:

```
person = Person("Joe")
person = Person("Joe", parent=some_parent)
```

Here the difference is that we are using keyword arguments to specify any arguments that are not handled by the `Person` class itself. Note that we could use keyword arguments for all arguments - whether or not you do so is down to personal choice.

The limitations of the first implementation no longer apply. For example, without any further changes we can also do this:

```
person = Person("Joe", destroyed=some_callable)
```

Let's say we now want to extend the behaviour of the `Person` class by adding a mixin that handles a person's age. The implementation of the mixin would be as follows:

```
class Age(object):
    def __init__(self, age=0, **kwds):
        super().__init__(**kwds)

        self.age = age
```

This follows a similar pattern to our `Person` implementation, but notice that we have provided the `age` argument with a default value.

The following is our new `Person` implementation:

```
class Person(QObject, Age):
    def __init__(self, name, **kwds):
        super().__init__(**kwds)

        self.name = name
```

The only change we have had to make is to add `Age` to `Person`'s list of super-classes. More importantly we do not need to change any call to create a `Person` instance.

If we do want to create a `Person` instance with a non-default age then we simply pass it as a keyword argument as follows:

```
person = Person("Joe", age=38)
```

This technique increases the use of keyword arguments - while this means a bit more typing, it significantly increases the readability of application code.

Things to be Aware Of

TLS Support

Support for Transport Layer Security (TLS) is increasingly important, particularly on mobile platforms where an application is typically a front end to a cloud-based server. As both Python and Qt implement different APIs that support TLS, a PyQt application has a choice as to which to use. This is particularly important when deploying an application as the support may have to be included with, or built into, the application itself.

Ideally the TLS implementation provided by the target would be used (e.g. CryptoAPI on Windows, Secure Transport on macOS and iOS). This would mean that security updates, including certificate updates, would be handled by the vendor of the target operating system and could be ignored by the application. Unfortunately there is no common TLS API. The resolution to this problem is the subject of PEP 543 but that has yet to be implemented.

Python uses OpenSSL as its TLS implementation. Python v3.7.4 and later use OpenSSL v1.1.1. Python v3.7.0 to v3.7.3 use OpenSSL v1.1.0. Earlier versions of Python use OpenSSL v1.0.2. On Windows and macOS the standard Python binary installers include copies of the corresponding OpenSSL libraries.

Qt has support for the native TLS implementation on macOS and iOS but on other platforms (except for Linux) a deployed application must include it’s own OpenSSL implementaion.

enums

New in version 5.11.

PyQt (or rather SIP) wraps C/C++ enums using a dedicated Python type. Members of the enum are visible at the same scope as the enum itself.

Qt is making increasing use of C++11 scoped enums and support for them was added to SIP v4.19.4. Scoped enums are implemented using the standard `enum.Enum` Python type. In this case members of the enum are only visible within the scope of the enum.

The difference in visibility is unfortunate as it requires the Python programmer to be aware of the nature of the underlying C++ enum.

With SIP v4.19.9 members of traditional C/C++ enums are now also visible within the scope of the enum. It is strongly recommended that enum members are always referenced by specifying the scope of the enum. PyQt6 will not allow any other method of access.

Crashes On Exit

When the Python interpreter leaves a *scope* (for example when it returns from a function) it will potentially garbage collect all objects local to that scope. The order in which it is done is, in effect, random. Theoretically this can cause problems because it may mean that the C++ destructors of

any wrapped Qt instances are called in an order that Qt isn't expecting and may result in a crash. However, in practice, this is only likely to be a problem when the application is terminating.

As a way of mitigating this possiblity PyQt5 ensures that the C++ destructors of any QObject instances owned by Python are invoked before the destructor of any QCoreApplication instance is invoked. Note however that the order in which the QObject destructors are invoked is still random.

Keyword Arguments

PyQt5 supports the use of keyword arguments for optional arguments. Although the PyQt5 and Qt documentation may indicate that an argument has a particular name, you may find that PyQt5 actually uses a different name. This is because the name of an argument is not part of the Qt API and there is some inconsistency in the way that similar arguments are named. Different versions of Qt may use a different name for an argument which wouldn't affect the C++ API but would break the Python API.

The docstrings that PyQt5 generates for all classes, functions and methods will contain the correct argument names. In a future version of PyQt5 the documentation will also be guaranteed to contain the correct argument names.

Python Strings, Qt Strings and Unicode

Qt uses the `QString` class to represent Unicode strings, and the `QByteArray` to represent byte arrays or strings. In Python v3 the corresponding native object types are `str` and `bytes`. In Python v2 the corresponding native object types are `unicode` and `str`.

PyQt5 does its best to automatically convert between objects of the various types. Explicit conversions can be easily made where necessary.

In some cases PyQt5 will not perform automatic conversions where it is necessary to distinguish between different overloaded methods.

For Python v3 the following conversions are done by default.

- If Qt expects a `char *` (or a `const` version) then PyQt5 will accept a `str` that contains only ASCII characters, a `bytes`, a `QByteArray`, or a Python object that implements the buffer protocol.
- If Qt expects a `char` (or a `const` version) then PyQt5 will accept the same types as for `char *` and also require that a single character is provided.
- If Qt expects a `signed char *` or an `unsigned char *` (or a `const` version) then PyQt5 will accept a `bytes`.
- If Qt expects a `signed char` or an `unsigned char` (or a `const` version) then PyQt5 will accept a `bytes` of length 1.
- If Qt expects a `QString` then PyQt5 will accept a `str`, a `bytes` that contains only ASCII characters, a `QByteArray` or `None`.
- If Qt expects a `QByteArray` then PyQt5 will also accept a `bytes`.
- If Qt expects a `QByteArray` then PyQt5 will also accept a `str` that contains only Latin-1 characters.

For Python v2 the following conversions are done by default.

- If Qt expects a `char *`, `signed char *` or an `unsigned char *` (or a `const` version) then PyQt5 will accept a `unicode` that contains only ASCII characters, a `str`, a `QByteArray`, or a Python object that implements the buffer protocol.
- If Qt expects a `char`, `signed char` or an `unsigned char` (or a `const` version) then PyQt5 will accept the same types as for `char *`, `signed char *` and `unsigned char *` and also require that a single character is provided.
- If Qt expects a `QString` then PyQt5 will accept a `unicode`, a `str` that contains only ASCII characters, a `QByteArray` or `None`.
- If Qt expects a `QByteArray` then PyQt5 will accept a `str`.
- If Qt expects a `QByteArray` then PyQt5 will accept a `unicode` that contains only Latin-1 characters.

Note that the different behaviour between Python v2 and v3 is due to v3's reduced support for the buffer protocol.

Historically `QString` distinguishes between empty strings and null strings. Current versions of Qt treat null strings as empty strings but there may be other C++ code that PyQt5 applications call that maintains the distinction. Consequently PyQt5 will convert `None` to a null `QString`. The reverse conversion is not done and both a null and an empty `QString` will be converted to an empty (i.e. zero length) Python string.

Garbage Collection

C++ does not garbage collect unreferenced class instances, whereas Python does. In the following C++ fragment both colours exist even though the first can no longer be referenced from within the program:

```
col = new QColor();
col = new QColor();
```

In the corresponding Python fragment, the first colour is destroyed when the second is assigned to `col`:

```
col = QColor()
col = QColor()
```

In Python, each colour must be assigned to different names. Typically this is done within class definitions, so the code fragment would be something like:

```
self.col1 = QColor()
self.col2 = QColor()
```

Sometimes a Qt class instance will maintain a pointer to another instance and will eventually call the destructor of that second instance. The most common example is that a QObject (and any of its sub-classes) keeps pointers to its children and will automatically call their destructors. In these cases, the corresponding Python object will also keep a reference to the corresponding child objects.

So, in the following Python fragment, the first QLabel is not destroyed when the second is assigned to `lab` because the parent QWidget still has a reference to it:

```
parent = QWidget()
lab = QLabel("First label", parent)
lab = QLabel("Second label", parent)
```

Multiple Inheritance

It is not possible to define a new Python class that sub-classes from more than one Qt class. The exception is classes specifically intended to act as mixin classes such as those (like QQmlParserStatus) that implement Qt interfaces.

Access to Protected Member Functions

When an instance of a C++ class is not created from Python it is not possible to access the protected member functions of that instance. Attempts to do so will raise a Python exception. Also, any Python methods corresponding to the instance's virtual member functions will never be called.

`None` and `NULL`

Throughout PyQt5, the `None` value can be specified wherever `NULL` is acceptable to the underlying C++ code.

Equally, `NULL` is converted to `None` whenever it is returned by the underlying C++ code.

Support for `void *`

PyQt5 (actually SIP) represents `void *` values as objects of type sip.voidptr. Such values are often used to pass the addresses of external objects between different Python modules. To make this easier, a Python integer (or anything that Python can convert to an integer) can be used whenever a sip.voidptr is expected.

A sip.voidptr may be converted to a Python integer by using the `int()` builtin function.

A sip.voidptr may be converted to a Python string by using its **`asstring()`** method. The **`asstring()`** method takes an optional integer argument which is the length of the data in bytes.

A sip.voidptr may also be given a size (ie. the size of the block of memory that is pointed to) by calling its **`setsize()`** method. If it has a size then it is also able to support Python's buffer protocol and behaves like a Python `memoryview` object so that the block of memory can be treated as a mutable list of bytes. It also means that the Python struct module can be used to unpack and pack binary data structures in memory, memory mapped files or shared memory.

Using Qt Designer

Qt Designer is the Qt tool for designing and building graphical user interfaces. It allows you to design widgets, dialogs or complete main windows using on-screen forms and a simple drag-and-drop interface. It has the ability to preview your designs to ensure they work as you intended, and to allow you to prototype them with your users, before you have to write any code.

Qt Designer uses XML `.ui` files to store designs and does not generate any code itself. Qt includes the `uic` utility that generates the C++ code that creates the user interface. Qt also includes the `QUiLoader` class that allows an application to load a `.ui` file and to create the corresponding user interface dynamically.

PyQt5 does not wrap the `QUiLoader` class but instead includes the uic Python module. Like `QUiLoader` this module can load `.ui` files to create a user interface dynamically. Like the **uic** utility it can also generate the Python code that will create the user interface. PyQt5's **pyuic5** utility is a command line interface to the uic module. Both are described in detail in the following sections.

Using the Generated Code

The code that is generated has an identical structure to that generated by Qt's `uic` and can be used in the same way.

The code is structured as a single class that is derived from the Python `object` type. The name of the class is the name of the toplevel object set in Designer with `Ui_` prepended. (In the C++ version the class is defined in the `Ui` namespace.) We refer to this class as the *form class*.

The class contains a method called `setupUi()`. This takes a single argument which is the widget in which the user interface is created. The type of this argument (typically QDialog, QWidget or QMainWindow) is set in Designer. We refer to this type as the *Qt base class*.

In the following examples we assume that a `.ui` file has been created containing a dialog and the name of the QDialog object is `ImageDialog`. We also assume that the name of the file containing the generated Python code is `ui_imagedialog.py`. The generated code can then be used in a number of ways.

The first example shows the direct approach where we simply create a simple application to create the dialog:

```
import sys
from PyQt5.QtWidgets import QApplication, QDialog
from ui_imagedialog import Ui_ImageDialog

app = QApplication(sys.argv)
window = QDialog()
ui = Ui_ImageDialog()
ui.setupUi(window)

window.show()
sys.exit(app.exec_())
```

The second example shows the single inheritance approach where we sub-class QDialog and set up the user interface in the `__init__()` method:

```
from PyQt5.QtWidgets import QDialog
from ui_imagedialog import Ui_ImageDialog

class ImageDialog(QDialog):
    def __init__(self):
        super(ImageDialog, self).__init__()

        # Set up the user interface from Designer.
        self.ui = Ui_ImageDialog()
        self.ui.setupUi(self)

        # Make some local modifications.
        self.ui.colorDepthCombo.addItem("2 colors (1 bit per pixel)")

        # Connect up the buttons.
        self.ui.okButton.clicked.connect(self.accept)
        self.ui.cancelButton.clicked.connect(self.reject)
```

The final example shows the multiple inheritance approach:

```
from PyQt5.QtGui import QDialog
from ui_imagedialog import Ui_ImageDialog

class ImageDialog(QDialog, Ui_ImageDialog):
    def __init__(self):
        super(ImageDialog, self).__init__()

        # Set up the user interface from Designer.
        self.setupUi(self)

        # Make some local modifications.
        self.colorDepthCombo.addItem("2 colors (1 bit per pixel)")

        # Connect up the buttons.
        self.okButton.clicked.connect(self.accept)
        self.cancelButton.clicked.connect(self.reject)
```

For a full description see the Qt Designer Manual in the Qt Documentation.

pyuic5

The **pyuic5** utility is a command line interface to the uic module. The command has the following syntax:

```
pyuic5 [options] .ui-file
```

The full set of command line options is:

-h , --help
A help message is written to `stdout`.

--version
The version number is written to `stdout`.

-i <N>, **--indent** <N>
The Python code is generated using an indentation of <N> spaces. If <N> is 0 then a tab is used. The default is 4.

-o <FILE>, **--output** <FILE>
The Python code generated is written to the file <FILE>.

-p, **--preview**
The GUI is created dynamically and displayed. No Python code is generated.

-x, **--execute**
The generated Python code includes a small amount of additional code that creates and displays the GUI when it is executes as a standalone application.

--import-from <PACKAGE>

New in version 5.6.

Resource modules are imported using `from <PACKAGE> import ...` rather than a simple `import ...`.

--from-imports
This is the equivalent of specifying `--import-from ..`

--resource-suffix <SUFFIX>
The suffix <SUFFIX> is appended to the basename of any resource file specified in the `.ui` file to create the name of the Python module generated from the resource file by **pyrcc5**. The default is `_rc`. For example if the `.ui` file specified a resource file called `foo.qrc` then the corresponding Python module is `foo_rc`.

Note that code generated by **pyuic5** is not guaranteed to be compatible with earlier versions of PyQt5. However, it is guaranteed to be compatible with later versions. If you have no control over the version of PyQt5 the users of your application are using then you should run **pyuic5**, or call **`compileUi()`**, as part of your installation process. Another alternative would be to distribute the `.ui` files (perhaps as part of a resource file) and have your application load them dynamically.

Writing Qt Designer Plugins

Qt Designer can be extended by writing plugins. Normally this is done using C++ but PyQt5 also allows you to write plugins in Python. Most of the time a plugin is used to expose a custom widget to Designer so that it appears in Designer's widget box just like any other widget. It is possibe to change the widget's properties and to connect its signals and slots.

It is also possible to add new functionality to Designer. See the Qt documentation for the full details. Here we will concentrate on describing how to write custom widgets in Python.

The process of integrating Python custom widgets with Designer is very similar to that used with widget written using C++. However, there are particular issues that have to be addressed.

- Designer needs to have a C++ plugin that conforms to the interface defined by the `QDesignerCustomWidgetInterface` class. (If the plugin exposes more than one custom widget then it must conform to the interface defined by the `QDesignerCustomWidgetCollectionInterface` class.) In addition the plugin class must sub-class QObject as well as the interface class. PyQt5 does not allow Python classes to be sub-classed from more than one Qt class.
- Designer can only connect Qt signals and slots. It has no understanding of Python signals or callables.
- Designer can only edit Qt properties that represent C++ types. It has no understanding of Python attributes or Python types.

PyQt5 provides the following components and features to resolve these issues as simply as possible.

- PyQt5's QtDesigner module includes additional classes (all of which have a `QPy` prefix) that are already sub-classed from the necessary Qt classes. This avoids the need to sub-class from more than one Qt class in Python. For example, where a C++ custom widget plugin would sub-class from QObject and `QDesignerCustomWidgetInterface`, a Python custom widget plugin would instead sub-class from QPyDesignerCustomWidgetPlugin.

- PyQt5 installs a C++ plugin in Designer's plugin directory. It conforms to the interface defined by the `QDesignerCustomWidgetCollectionInterface` class. It searches a configurable set of directories looking for Python plugins that implement a class sub-classed from QPyDesignerCustomWidgetPlugin. Each class that is found is instantiated and the instance created is added to the custom widget collection.

  The **PYQTDESIGNERPATH** environment variable specifies the set of directories to search for plugins. Directory names are separated by a path separator (a semi-colon on Windows and a colon on other platforms). If a directory name is empty (ie. there are consecutive path separators or a leading or trailing path separator) then a set of default directories is automatically inserted at that point. The default directories are the `python` subdirectory of each directory that Designer searches for its own plugins. If the environment variable is not set then only the default directories are searched. If a file's basename does not end with `plugin` then it is ignored.

- A Python custom widget may define new Qt signals using pyqtSignal.

- A Python method may be defined as a new Qt slot by using the pyqtSlot() decorator.

- A new Qt property may be defined using the pyqtProperty function.

Note that the ability to define new Qt signals, slots and properties from Python is potentially useful to plugins conforming to any plugin interface and not just that used by Designer.

For a simple but complete and fully documented example of a custom widget that defines new Qt signals, slots and properties, and its plugin, look in the `examples/designer/plugins` directory of the PyQt5 source package. The `widgets` subdirectory contains the `pydemo.py` custom widget and the `python` subdirectory contains its `pydemoplugin.py` plugin.

The PyQt5 Resource System

PyQt5 supports Qt's resource system. This is a facility for embedding resources such as icons and translation files in an application. This makes the packaging and distribution of those resources much easier.

A `.qrc` resource collection file is an XML file used to specify which resource files are to be embedded. The application then refers to the resource files by their original names but preceded by a colon.

For a full description, including the format of the `.qrc` files, see the Qt Resource System in the Qt documentation.

pyrcc5

pyrcc5 is PyQt5's equivalent to Qt's **rcc** utility and is used in exactly the same way. **pyrcc5** reads the `.qrc` file, and the resource files, and generates a Python module that only needs to be `import`ed by the application in order for those resources to be made available just as if they were the original files.

Support for Pickling

The following PyQt5 classes may be pickled.

- QByteArray
- QColor
- QDate
- QDateTime
- QKeySequence
- QLine
- QLineF
- QPoint
- QPointF
- QPolygon
- QRect
- QRectF
- QSize
- QSizeF
- QTime

Also all named enums (PyQt5.QtCore.Qt.Key for example) may be pickled.

Using PyQt5 from the Python Shell

PyQt5 installs an input hook (using `PyOS_InputHook`) that processes events when an interactive interpreter is waiting for user input. This means that you can, for example, create widgets from the Python shell prompt, interact with them, and still being able to enter other Python commands.

For example, if you enter the following in the Python shell:

```
>>> from PyQt5.QtWidgets import QApplication, QWidget
>>> a = QApplication([])
>>> w = QWidget()
>>> w.show()
>>> w.hide()
>>>
```

The widget would be displayed when `w.show()` was entered and hidden as soon as `w.hide()` was entered.

The installation of an input hook can cause problems for certain applications (particularly those that implement a similar feature using different means). The QtCore module contains the pyqtRemoveInputHook() and pyqtRestoreInputHook() functions that remove and restore the input hook respectively.

Internationalisation of PyQt5 Applications

PyQt5 and Qt include a comprehensive set of tools for translating applications into local languages. For a full description, see the Qt Linguist Manual in the Qt documentation.

The process of internationalising an application comprises the following steps.

- The programmer uses **pylupdate5** to create or update a `.ts` translation file for each language that the application is to be translated into. A `.ts` file is an XML file that contains the strings to be translated and the corresponding translations that have already been made. **pylupdate5** can be run any number of times during development to update the `.ts` files with the latest strings for translation.
- The translator uses Qt Linguist to update the `.ts` files with translations of the strings.
- The release manager then uses Qt's **lrelease** utility to convert the `.ts` files to `.qm` files which are compact binary equivalents used by the application. If an application cannot find an appropriate `.qm` file, or a particular string hasn't been translated, then the strings used in the original source code are used instead.
- The release manage may optionally use **pyrcc5** to embed the `.qm` files, along with other application resources such as icons, in a Python module. This may make packaging and distribution of the application easier.

pylupdate5

pylupdate5 is PyQt5's equivalent to Qt's **lupdate** utility and is used in exactly the same way. A Qt `.pro` project file is read that specifies the Python source files and Qt Designer interface files from which the text that needs to be translated is extracted. The `.pro` file also specifies the `.ts` translation files that **pylupdate5** updates (or creates if necessary) and are subsequently used by Qt Linguist.

Differences Between PyQt5 and Qt

Qt implements internationalisation support through the QTranslator class, and the translate() and tr() methods. Usually tr() is used to obtain the correct translation of a message. The translation process uses a message context to allow the same message to be translated differently. In Qt tr() is actually generated by `moc` and uses the hardcoded class name as the context. On the other hand, translate allows the context to be specified explicitly.

Unfortunately, because of the way Qt implements tr() it is not possible for PyQt5 to exactly reproduce its behaviour. The PyQt5 implementation of tr() uses the class name of the instance as the context. The key difference, and the source of potential problems, is that the context is determined dynamically in PyQt5, but is hardcoded in Qt. In other words, the context of a translation may change depending on an instance's class hierarchy. For example:

```
class A(QObject):
    def hello(self):
        return self.tr("Hello")

class B(A):
```

```
    pass

a = A()
a.hello()

b = B()
b.hello()
```

In the above the message is translated by `a.hello()` using a context of `A`, and by `b.hello()` using a context of `B`. In the equivalent C++ version the context would be `A` in both cases.

The PyQt5 behaviour is unsatisfactory and may be changed in the future. It is recommended that translate() be used in preference to tr(). This is guaranteed to work with current and future versions of PyQt5 and makes it much easier to share message files between Python and C++ code. Below is the alternative implementation of `A` that uses translate():

```
class A(QObject):
    def hello(self):
        return QCoreApplication.translate('A', "Hello")
```

DBus Support

PyQt5 provides two different modules that implement support for DBus. The QtDBus module provides wrappers for the standard Qt DBus classes. The dbus.mainloop.pyqt5 module adds support for the Qt event loop to the standard `dbus-python` Python module.

QtDBus

The QtDBus module is used in a similar way to the C++ library it wraps. The main difference is in the way it supports the demarshalling of DBus structures. C++ relies on the template-based registration of types using `qDBusRegisterMetaType()` which isn't possible from Python. Instead a slot that accepts a DBus structure in an argument should specify a slot with a single QDBusMessage argument. The implementation of the slot should then extract the arguments from the message using its **`arguments()`** method.

For example, say we have a DBus method called `setColors()` that has a single argument that is an array of structures of three integers (red, green and blue). The DBus signature of the argument would then be `a(iii)`. In C++ you would typically define a class to hold the red, green and blue values and so your code would include the following (incomplete) fragments:

```
struct Color
{
    int red;
    int green;
    int blue;
};
Q_DECLARE_METATYPE(Color)

qDBusRegisterMetaType<Color>();

class ServerAdaptor : public QDBusAbstractAdaptor
{
    Q_OBJECT

public slots:
    void setColors(QList<const Color &> colors);
};
```

The Python version is, of course, much simpler:

```
class ServerAdaptor(QDBusAbstractAdaptor):

    @pyqtSlot(QDBusMessage)
    def setColors(self, message):
        # Get the single argument.
        colors = message.arguments()[0]

        # The argument will be a list of 3-tuples of ints.
        for red, green, blue in colors:
            print("RGB:", red, green, blue)
```

Note that this technique can be used for arguments of any type, it is only require if DBus structures are involved.

dbus.mainloop.pyqt5

The dbus.mainloop.pyqt5 module provides support for the Qt event loop to `dbus-python`. The module's API is almost identical to that of the dbus.mainloop.glib modules that provides support for the GLib event loop.

The dbus.mainloop.pyqt5 module contains the following function.

DBusQtMainLoop(*set_as_default=False*)

Create a `dbus.mainloop.NativeMainLoop` object that uses the the Qt event loop.

Parameters: **set_as_default** – is optionally set to make the main loop instance the default for all new Connection and Bus instances. It may only be specified as a keyword argument, and not as a positional argument.

The following code fragment is all that is normally needed to set up the standard `dbus-python` language bindings package to be used with PyQt5:

```
from dbus.mainloop.pyqt5 import DBusQtMainLoop

DBusQtMainLoop(set_as_default=True)
```

Deploying Commercial PyQt5 Applications

Deploying commercial PyQt5 applications can be a complicated process for a number of reasons:

- It is usually better not to rely on pre-requisite packages being already installed on the user's system. This means that as well as your application code, you also need to include the Python interpreter, the standard library, third-pary packages and extension modules, and Qt itself.
- Some target platforms (iOS for example) have restrictions on how an application is built in order for it to be included in app stores.
- It is necessary to discourage users from accessing the underlying PyQt5 modules for themselves. A user that used the modules shipped with your application to develop new applications would themselves be considered a developer and would need their own commercial PyQt5 license.

The recommended solution to all of these issues is to use pyqtdeploy.

The PyQt5 Extension API

An important feature of PyQt5 (and SIP generated modules in general) is the ability for other extension modules to build on top of it. QScintilla is such an example.

PyQt5 provides an extension API that can be used by other modules. This has the advantage of sharing code and also enforcing consistent behaviour. Part of the API is accessable from Python and part from C++.

Python API

The Python part of the API is accessible via the QtCore module and is typically used by an extension module's equivalent of PyQt5's **configure.py**.

The API consists of `PyQt5.QtCore.PYQT_CONFIGURATION` which is a dict that describes how PyQt5 was configured. At the moment it contains a single value called `sip_flags` which is a string containing the `-t` and `-x` flags that were passed to the **sip** executable by **configure.py**. Other extension modules must use the same flags in their configuration.

This information is also provided by SIP v4's sipconfig module. However this module will not be implemented by SIP v5.

C++ API

The C++ API is a set of functions. The addresses of each function is obtained by calling SIP's `sipImportSymbol()` function with the name of the function required.

Several of the functions are provided as a replacement for SIP v4 features (i.e. `SIP_ANYSLOT`, `SIP_QOBJECT`, `SIP_RXOBJ_CON`, `SIP_RXOBJ_DIS`, `SIP_SIGNAL`, `SIP_SLOT`, `SIP_SLOT_CON` and `SIP_SLOT_DIS`) that are not supported by SIP v5.

The functions exported by PyQt5 are as follows:

void **pyqt_cleanup_qobjects**()

New in version 5.13.1.

Call the C++ destructor of any QObject instance that is owned by Python (with the exception of any QCoreApplication instance).

void **pyqt_err_print**()

New in version 5.4.

A replacement for `PyErr_Print()`. In PyQt v5.4 it raises a deprecation warning and calls `PyErr_Print()`. In PyQt v5.5 and later it passes the text of the exception and traceback to `qFatal()`.

char **`pyqt5_from_argv_list`(PyObject *_argv_list_, int &_argc_)

Convert a Python list to a standard C array of command line arguments and an argument count.

Parameters:
- **argv_list** – is the Python list of arguments.
- **argc** – is updated with the number of arguments in the list.

Returns: an array of pointers to the arguments on the heap.

PyObject *`pyqt5_from_qvariant_by_type`(QVariant &_value_, PyObject *_type_)

Convert a QVariant to a Python object according to an optional Python type.

Parameters:
- **value** – is the value to convert.
- **type** – is the Python type.

Returns: the converted value. If it is `0` then a Python exception will have been raised.

sipErrorState `pyqt5_get_connection_parts`(PyObject *_slot_, QObject *_transmitter_, _const_ char *_signal_signature_, bool _single_shot_, QObject **_receiver_, QByteArray &_slot_signature_)

Get the receiver object and slot signature to allow a signal to be connected to an optional transmitter.

Parameters:
- **slot** – is the slot and should be a callable or a bound signal.
- **transmitter** – is the optional QObject transmitter.
- **signal_signature** – is the signature of the signal to be connected.
- **single_shot** – is `true` if the signal will only ever be emitted once.
- **receiver** – is updated with the QObject receiver. This may be a proxy if the slot requires it.
- **slot_signature** – is updated with the signature of the slot.

Returns: the error state. If this is `sipErrorFail` then a Python exception will have been raised.

const QMetaObject *`pyqt5_get_qmetaobject`(PyTypeObject *_type_)

Get the QMetaObject instance for a Python type. The Python type must be a sub-type of QObject's Python type.

Parameters: **type** – is the Python type object.

Returns: the QMetaObject.

sipErrorState `pyqt5_get_pyqtsignal_parts`(PyObject *_signal_, QObject **_transmitter_, QByteArray &_signal_signature_)

Get the transmitter object and signal signature from a bound signal.

Parameters:
- **signal** – is the bound signal.
- **transmitter** – is updated with the QObject transmitter.
- **signal_signature** – is updated with the signature of the signal.

Returns: the error state. If this is `sipErrorFail` then a Python exception will have been raised.

sipErrorState `pyqt5_get_pyqtslot_parts`(PyObject *_slot_, QObject **_receiver_, QByteArray &_slot_signature_)

Get the receiver object and slot signature from a callable decorated with pyqtSlot().

Parameters:
- **slot** – is the callable slot.
- **receiver** – is updated with the QObject receiver.
- **slot_signature** – is updated with the signature of the slot.

Returns: the error state. If this is `sipErrorFail` then a Python exception will have been raised.

sipErrorState **pyqt5_get_signal_signature**(PyObject *_signal_, _const_ QObject *_transmitter_, QByteArray &_signal_signature_)

Get the signature string for a bound or unbound signal. If the signal is bound, and the given transmitter is specified, then it must be bound to the transmitter.

Parameters:
- **signal** – is the signal.
- **transmitter** – is the optional QObject transmitter.
- **signal_signature** – is updated with the signature of the signal.

Returns: the error state. If this is `sipErrorFail` then a Python exception will have been raised.

void **pyqt5_register_from_qvariant_convertor**(bool (*_convertor_)(_const_ QVariant&, PyObject**))

Register a convertor function that converts a QVariant value to a Python object.

Parameters: **convertor** – is the convertor function. This takes two arguments. The first argument is the QVariant value to be converted. The second argument is updated with a reference to the result of the conversion and it will be `0`, and a Python exception raised, if there was an error. The convertor will return `true` if the value was handled so that no other convertor will be tried.

void **pyqt5_register_to_qvariant_convertor**(bool (*_convertor_)(PyObject*, QVariant&, bool*))

Register a convertor function that converts a Python object to a QVariant value.

Parameters: **convertor** – is the convertor function. This takes three arguments. The first argument is the Python object to be converted. The second argument is a pointer to QVariant value that is updated with the result of the conversion. The third argument is updated with an error flag which will be `false`, and a Python exception raised, if there was an error. The convertor will return `true` if the value was handled so that no other convertor will be tried.

void **pyqt5_register_to_qvariant_data_convertor**(bool (*_convertor_)(PyObject*, void*, int, bool*))

Register a convertor function that converts a Python object to the pre-allocated data of a QVariant value.

Parameters: **convertor** – is the convertor function. This takes four arguments. The first argument is the Python object to be converted. The second argument is a pointer to the pre-allocated data of a QVariant value that is updated with the result of the conversion. The third argument is the meta-type of the value. The

fourth argument is updated with an error flag which will be `false`, and a Python exception raised, if there was an error. The convertor will return `true` if the value was handled so that no other convertor will be tried.

void **pyqt5_update_argv_list**(PyObject **argv_list*, int *argc*, char ***argv*)

Update a Python list from a standard C array of command line arguments and an argument count. This is used in conjunction with `pyqt5_from_argv_list()` to handle the updating of argument lists after calling constructors of classes such as QCoreApplication.

Parameters:
- **argv_list** – is the Python list of arguments that will be updated.
- **argc** – is the number of command line arguments.
- **argv** – is the array of pointers to the arguments on the heap.